Software Carpentry
Volume 1: Basics

Software Carpentry
Volume 1: Basics
Edited by Greg Wilson

The full text of this book is available online at `http://www.aosabook.org/`.
All royalties from its sale will be donated to Amnesty International.

While every precaution has been taken in the preparation of this book, the editors and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

FIXME: cover photo credit

Revision Date: June 5, 2014

ISBN: FIXME

This book is dedicated to
Betty Jennings, Betty Snyder, Fran Bilas, Kay McNulty, Marlyn Wescoff, and Ruth Lichterman,
the original programmers of the ENIAC,
and to all of the people who have contributed to Software Carpentry over the years.

Contents

Chapter 1

Introduction

Here's the dream:

> Computers have revolutionized research, and that revolution is only beginning. Every day, scientists and engineers all over the world use them to study things that are too big, too small, too fast, too slow, too expensive, too dangerous, or just too hard to study any other way.

Now here's the reality:

> Every day, scientists and engineers all over the world waste time wrestling with computers. Tasks that should take a few moments take hours or days, and many things never work at all. And even when things *do* work, most scientists have no idea how reliable their results are.

Most of the pain that researchers feel stems from not knowing how to develop software systematically, how to tell if their programs are working correctly, how to share their work with others (except by mailing files to one another), or how to keep track of what they've done. This sorry state of affairs persists for four reasons:

- *No room, no time.* Everybody's curriculum is full—there's simply not space to add more about computing without dropping something else.
- *No standards.* Reviewers and granting agencies don't check whether software is correct, ask how long it took to write, or count it toward tenure, so there's no incentive for scientists to do better.
- *The blind leading the blind.* Senior researchers can't teach the next generation how to do things that they don't know how to do themselves.
- *The cult of big iron.* Attention and funding mostly goes to things that politicians and university presidents can brag about on opening day, rather than to the basic skills that almost everyone uses.

Our goal is to show scientists and engineers how to do more in less time and with less pain. Our lessons have been used by more than four thousand learners in over a hundred two-day workshops since the spring of 2010. Here's how they can help:

- If you've ever overwritten the wrong file, we'll show you how to use version control.

- If you’ve ever spent hours typing the same commands over and over again, we’ll show you how to automate those tasks using simple scripts.
- If you’ve ever spent an afternoon trying to figure out what the program you wrote last week actually does, we’ll show you how to break your code into modules that you can read, debug, and improve piece by piece.
- If you’ve ever spent days copying and pasting data in text files and spreadsheets, we’ll show you how a database can do the work for you.

About Us

Software Carpentry is an open source project. Our instructors are volunteers, and all of our lessons are freely available under the Creative Commons - Attribution License[1], so you can re-use and re-mix them however you want so long as you cite us as the original source.

Like all volunteer projects, Software Carpentry needs your help to grow. If you find a bug, please file a report in our GitHub repo[2]. If you would like to host a workshop, please get in touch; if you’d like to teach, we run an instructor training course[3]; and if you’d like to write lessons or exercises, please let us know.

To find out more, please visit the http://software-carpentry.org[4] or read these[5] papers[6] or our most popular blog posts[7].

[1] http://creativecommons.org/licenses/by/3.0/

[2] https://github.com/swcarpentry/bc/

[3] http://teaching.software-carpentry.org

[4] Software Carpentry web site

[5] http://www.plosbiology.org/article/info%3Adoi%2F10.1371%2Fjournal.pbio.1001745

[6] http://arxiv.org/abs/1307.5448

[7] http://software-carpentry.org/blog/index.html#popular

Acknowledgments

Software Carpentry has been made possible by the generous support of:

- Continuum Analytics[8]
- Indiana University[9]
- Lawrence Berkeley National Laboratory[10]
- Los Alamos National Laboratory[11]
- MathWorks[12]
- Michigan State University[13]
- Microsoft[14]
- MITACS[15]
- The Mozilla Foundation[16]
- The Python Software Foundation[17]
- Queen Mary University of London[18]
- Scimatic Software[19]
- SciNet[20]
- SHARCNET[21]
- The Alfred P. Sloan Foundation[22]
- The Space Telescope Science Institute[23]
- The UK Meteorological Office[24]
- The University of Alberta[25]
- The University Consortium for Atmospheric Research[26]
- The University of Toronto[27]

[8] http://continuum.io/
[9] http://www.indiana.edu
[10] http://www.lbl.gov
[11] http://www.lanl.gov
[12] http://www.mathworks.com
[13] http://www.msu.edu
[14] http://www.microsoft.com
[15] http://www.mitacs.ca
[16] http://mozillafoundation.org
[17] http://www.python.org/psf/
[18] http://www.qmul.ac.uk
[19] http://www.scimatic.com
[20] http://www.scinet.utoronto.ca
[21] http://www.sharcnet.ca
[22] http://www.sloan.org
[23] http://www.stsci.edu
[24] http://www.metoffice.gov.uk
[25] http://www.ualberta.ca
[26] http://www.ucar.edu
[27] http://www.utoronto.ca

Chapter 2

The Unix Shell

The Unix shell has been around longer than most of its users have been alive. It has survived so long because it's a power tool that allows people to do complex things with just a few keystrokes. More importantly, it helps them combine existing programs in new ways and automate repetitive tasks so that they don't have to type the same things over and over again.

2.1 Introducing the Shell

Objectives

- Explain how the shell relates to the keyboard, the screen, the operating system, and users' programs.
- Explain when and why command-line interfaces should be used instead of graphical interfaces.

Nelle Nemo, a marine biologist, has just returned from a six-month survey of the North Pacific Gyre[1], where she has been sampling gelatinous marine life in the Great Pacific Garbage Patch[2]. She has 300 samples in all, and now needs to:

1. Run each sample through an assay machine that will measure the relative abundance of 300 different proteins. The machine's output for a single sample is a file with one line for each protein.
2. Calculate statistics for each of the proteins separately using a program her supervisor wrote called `goostat`.
3. Compare the statistics for each protein with corresponding statistics for each other protein using a program one of the other graduate students wrote called `goodiff`.
4. Write up. Her supervisor would really like her to do this by the end of the month so that her paper can appear in an upcoming special issue of *Aquatic Goo Letters*.

It takes about half an hour for the assay machine to process each sample. The good news is, it only takes two minutes to set each one up. Since her lab has eight assay machines that she can use in parallel, this step will "only" take about two weeks.

The bad news is that if she has to run `goostat` and `goodiff` by hand, she'll have to enter filenames and click "OK" 45,150 times (300 runs of `goostat`, plus 300ÃŮ299/2 runs of `goodiff`). At 30

[1] http://en.wikipedia.org/wiki/North_Pacific_Gyre

[2] http://en.wikipedia.org/wiki/Great_Pacific_Garbage_Patch

seconds each, that will take more than two weeks. Not only would she miss her paper deadline, the chances of her typing all of those commands right are practically zero.

The next few lessons will explore what she should do instead. More specifically, they explain how she can use a command shell to automate the repetitive steps in her processing pipeline so that her computer can work 24 hours a day while she writes her paper. As a bonus, once she has put a processing pipeline together, she will be able to use it again whenever she collects more data.

What and Why

At a high level, computers do four things:

- run programs;
- store data;
- communicate with each other; and
- interact with us.

They can do the last of these in many different ways, including direct brain-computer links and speech interfaces. Since these are still in their infancy, most of us use windows, icons, mice, and pointers. These technologies didn't become widespread until the 1980s, but their roots go back to Doug Engelbart's work in the 1960s, which you can see in what has been called "The Mother of All Demos[3]".

Going back even further, the only way to interact with early computers was to rewire them. But in between, from the 1950s to the 1980s, most people used line printers. These devices only allowed input and output of the letters, numbers, and punctuation found on a standard keyboard, so programming languages and interfaces had to be designed around that constraint.

This kind of interface is called a *command-line interface*, or CLI, to distinguish it from the *graphical user interface*, or GUI, that most people now use. The heart of a CLI is a *read-evaluate-print loop*, or REPL: when the user types a command and then presses the enter (or return) key, the computer reads it, executes it, and prints its output. The user then types another command, and so on until the user logs off.

This description makes it sound as though the user sends commands directly to the computer, and the computer sends output directly to the user. In fact, there is usually a program in between called a *command shell*. What the user types goes into the shell; it figures out what commands to run and orders the computer to execute them.

A shell is a program like any other. What's special about it is that its job is to run other programs rather than to do calculations itself. The most popular Unix shell is Bash, the Bourne Again SHell (so-called because it's derived from a shell written by Stephen Bourne—this is what passes for wit among programmers). Bash is the default shell on most modern implementations of Unix, and in most packages that provide Unix-like tools for Windows.

Using Bash or any other shell sometimes feels more like programming than like using a mouse. Commands are terse (often only a couple of characters long), their names are frequently cryptic, and their output is lines of text rather than something visual like a graph. On the other hand, the shell allows us to combine existing tools in powerful ways with only a few keystrokes and to set up pipelines to handle large volumes of data automatically. In addition, the command line is often the easiest way to interact with remote machines. As clusters and cloud computing become more popular for scientific data crunching, being able to drive them is becoming a necessary skill.

[3] http://www.youtube.com/watch?v=a11JDLBXtPQ

Key Points

- A shell is a program whose primary purpose is to read commands and run other programs.
- The shell's main advantages are its high action-to-keystroke ratio, its support for automating repetitive tasks, and that it can be used to access networked machines.
- The shell's main disadvantages are its primarily textual nature and how cryptic its commands and operation can be.

2.2 Files and Directories

Objectives

- Explain the similarities and differences between a file and a directory.
- Translate an absolute path into a relative path and vice versa.
- Construct absolute and relative paths that identify specific files and directories.
- Explain the steps in the shell's read-run-print cycle.
- Identify the actual command, flags, and filenames in a command-line call.
- Demonstrate the use of tab completion, and explain its advantages.

The part of the operating system responsible for managing files and directories is called the *file system*. It organizes our data into files, which hold information, and directories (also called "folders"), which hold files or other directories.

Several commands are frequently used to create, inspect, rename, and delete files and directories. To start exploring them, let's open a shell window:

```
$
```

The dollar sign is a *prompt*, which shows us that the shell is waiting for input; your shell may show something more elaborate.

Type the command `whoami`, then press the Enter key (sometimes marked Return) to send the command to the shell. The command's output is the ID of the current user, i.e., it shows us who the shell thinks we are:

```
$ whoami
```

```
vlad
```

More specifically, when we type `whoami` the shell:

1. finds a program called `whoami`,
2. runs that program,
3. displays that program's output, then
4. displays a new prompt to tell us that it's ready for more commands.

Next, let's find out where we are by running a command called `pwd` (which stands for "print working directory"). At any moment, our *current working directory* is our current default directory, i.e., the directory that the computer assumes we want to run commands in unless we explicitly specify something else. Here, the computer's response is `/users/vlad`, which is Vlad's *home directory*:

```
$ pwd
```

```
/users/vlad
```

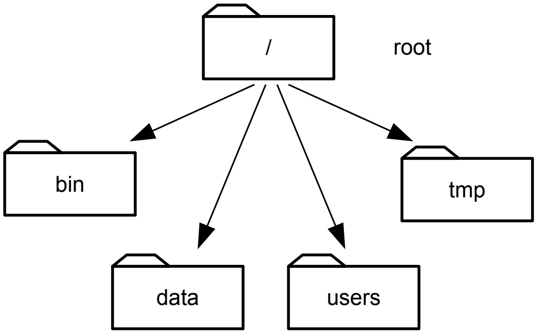


Figure 2.1: A Typical Filesystem

Alphabet Soup

If the command to find out who we are is `whoami`, the command to find out where we are ought to be called `whereami`, so why is it `pwd` instead? The usual answer is that in the early 1970s, when Unix was first being developed, every keystroke counted: the devices of the day were slow, and backspacing on a teletype was so painful that cutting the number of keystrokes in order to cut the number of typing mistakes was actually a win for usability. The reality is that commands were added to Unix one by one, without any master plan, by people who were immersed in its jargon. The result is as inconsistent as the roolz uv Inglish speling, but we're stuck with it now.

To understand what a "home directory" is, let's have a look at how the file system as a whole is organized. At the top is the *root directory* that holds everything else. We refer to it using a slash character `/` on its own; this is the leading slash in `/users/vlad`.

Inside that directory are several other directories: `bin` (which is where some built-in programs are stored), `data` (for miscellaneous data files), `users` (where users' personal directories are located), `tmp` (for temporary files that don't need to be stored long-term), and so on (Figure 2.1).

We know that our current working directory `/users/vlad` is stored inside `/users` because `/users` is the first part of its name. Similarly, we know that `/users` is stored inside the root directory `/` because its name begins with `/`.

Underneath `/users`, we find one directory for each user with an account on this machine (Figure 2.2). The Mummy's files are stored in `/users/imhotep`, Wolfman's in `/users/larry`, and ours in `/users/vlad`, which is why `vlad` is the last part of the directory's name.

> Notice that there are two meanings for the `/` character. When it appears at the front of a file or directory name, it refers to the root directory. When it appears *inside* a name, it's just a separator.

Let's see what's in Vlad's home directory by running `ls`, which stands for "listing" (Figure 2.3):

```
$ ls
```

```
bin         data      mail       music
notes.txt   papers    pizza.cfg  solar
solar.pdf   swc
```

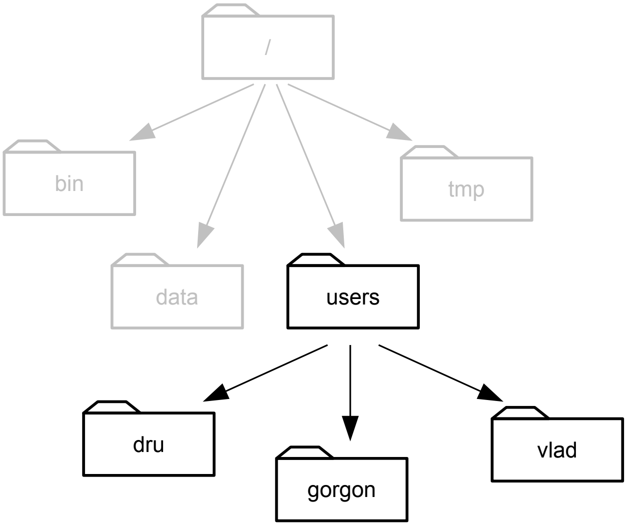


Figure 2.2: Home Directories

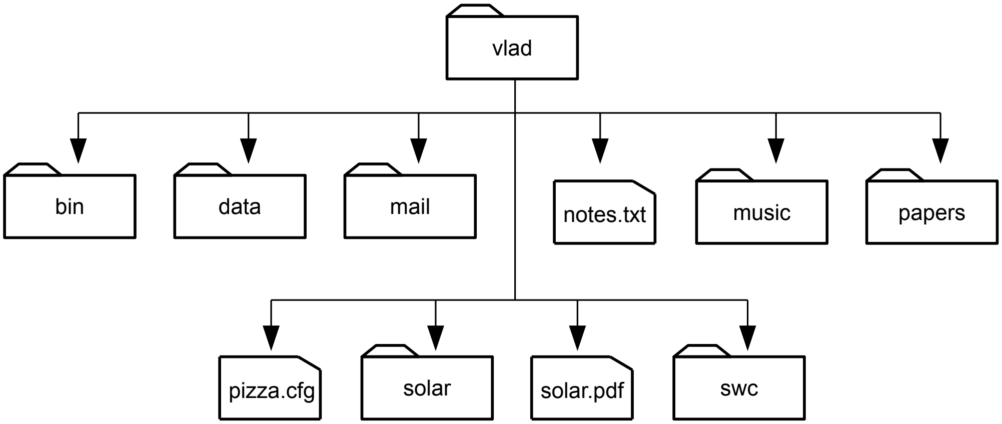


Figure 2.3: Vlad's Home Directory

`ls` prints the names of the files and directories in the current directory in alphabetical order, arranged neatly into columns. We can make its output more comprehensible by using the *flag* `-F`, which tells `ls` to add a trailing `/` to the names of directories:

```
$ ls -F

bin/        data/     mail/      music/
notes.txt   papers/   pizza.cfg  solar/
solar.pdf   swc/
```

Here, we can see that `/users/vlad` contains seven *sub-directories*. The names that don't have trailing slashes, like `notes.txt`, `pizza.cfg`, and `solar.pdf`, are plain old files. And note that there is a space between `ls` and `-F`: without it, the shell thinks we're trying to run a command called `ls-F`, which doesn't exist.

> **What's In A Name?**
>
> You may have noticed that all of Vlad's files' names are "something dot something". This is just a convention: we can call a file `mythesis` or almost anything else we want. However, most people use two-part names most of the time to help them (and their programs) tell different kinds of files apart. The second part of such a name is called the *filename extension*, and indicates what type of data the file holds: `.txt` signals a plain text file, `.pdf` indicates a PDF document, `.cfg` is a configuration file full of parameters for some program or other, and so on.
>
> This is just a convention, albeit an important one. Files contain bytes: it's up to us and our programs to interpret those bytes according to the rules for PDF documents, images, and so on.
>
> Naming a PNG image of a whale as `whale.mp3` doesn't somehow magically turn it into a recording of whalesong, though it *might* cause the operating system to try to open it with a music player when someone double-clicks it.

Now let's take a look at what's in Vlad's `data` directory by running `ls -F data`, i.e., the command `ls` with the parameters `-F` and `data`. The second parameter—the one *without* a leading dash—tells `ls` that we want a listing of something other than our current working directory:

```
$ ls -F data

amino-acids.txt   elements/     morse.txt
pdb/              planets.txt   sunspot.txt
```

The output shows us that there are four text files and two sub-sub-directories. Organizing things hierarchically in this way helps us keep track of our work: it's possible to put hundreds of files in our home directory, just as it's possible to pile hundreds of printed papers on our desk, but it's a self-defeating strategy.

Notice, by the way that we spelled the directory name `data`. It doesn't have a trailing slash: that's added to directory names by `ls` when we use the `-F` flag to help us tell things apart. And it doesn't begin with a slash because it's a *relative path*, i.e., it tells `ls` how to find something from where we are, rather than from the root of the file system.

If we run `ls -F /data` (*with* a leading slash) we get a different answer, because `/data` is an *absolute path*:

```
$ ls -F /data
```

```
access.log    backup/    hardware.cfg
network.cfg
```

The leading / tells the computer to follow the path from the root of the filesystem, so it always refers to exactly one directory, no matter where we are when we run the command.

What if we want to change our current working directory? Before we do this, pwd shows us that we're in /users/vlad, and ls without any parameters shows us that directory's contents:

```
$ pwd
```

```
/users/vlad
```

```
$ ls
```

```
bin/         data/     mail/      music/
notes.txt    papers/   pizza.cfg  solar/
solar.pdf    swc/
```

We can use cd followed by a directory name to change our working directory. cd stands for "change directory", which is a bit misleading: the command doesn't change the directory, it changes the shell's idea of what directory we are in.

```
$ cd data
```

cd doesn't print anything, but if we run pwd after it, we can see that we are now in /users/vlad/data. If we run ls without parameters now, it lists the contents of /users/vlad/data, because that's where we now are:

```
$ pwd
```

```
/users/vlad/data
```

```
$ ls
```

```
amino-acids.txt   elements/     morse.txt
pdb/              planets.txt   sunspot.txt
```

We now know how to go down the directory tree: how do we go up? We could use an absolute path:

```
$ cd /users/vlad
```

but it's almost always simpler to use cd .. to go up one level:

```
$ pwd
```

```
/users/vlad/data
```

```
$ cd ..
```

.. is a special directory name meaning "the directory containing this one", or more succinctly, the *parent* of the current directory. Sure enough, if we run pwd after running cd .., we're back in /users/vlad:

```
$ pwd
```

```
/users/vlad
```

The special directory `..` doesn't usually show up when we run `ls`. If we want to display it, we can give `ls` the `-a` flag:

```
$ ls -F -a
```

```
./          ../      bin/       data/
mail/       music/   notes.txt  papers/
pizza.cfg   solar/   solar.pdf    swc/
```

`-a` stands for "show all"; it forces `ls` to show us file and directory names that begin with `.`, such as `..` (which, if we're in `/users/vlad`, refers to the `/users` directory). As you can see, it also displays another special directory that's just called `.`, which means "the current working directory". It may seem redundant to have a name for it, but we'll see some uses for it soon.

> **Orthogonality**
>
> The special names `.` and `..` don't belong to `ls`; they are interpreted the same way by every program. For example, if we are in `/users/vlad/data`, the command `ls ..` will give us a listing of `/users/vlad`. When the meanings of the parts are the same no matter how they're combined, programmers say they are *orthogonal*: Orthogonal systems tend to be easier for people to learn because there are fewer special cases and exceptions to keep track of.

Nelle's Pipeline: Organizing Files

Knowing just this much about files and directories, Nelle is ready to organize the files that the protein assay machine will create. First, she creates a directory called `north-pacific-gyre` (to remind herself where the data came from). Inside that, she creates a directory called `2012-07-03`, which is the date she started processing the samples. She used to use names like `conference-paper` and `revised-results`, but she found them hard to understand after a couple of years. (The final straw was when she found herself creating a directory called `revised-revised-results-3`.)

> Nelle names her directories "year-month-day", with leading zeroes for months and days, because the shell displays file and directory names in alphabetical order. If she used month names, December would come before July; if she didn't use leading zeroes, November ('11') would come before July ('7').

Each of her physical samples is labelled according to her lab's convention with a unique ten-character ID, such as "NENE01729A". This is what she used in her collection log to record the location, time, depth, and other characteristics of the sample, so she decides to use it as part of each data file's name. Since the assay machine's output is plain text, she will call her files `NENE01729A.txt`, `NENE01812A.txt`, and so on. All 1520 files will go into the same directory.

If she is in her home directory, Nelle can see what files she has using the command:

```
$ ls north-pacific-gyre/2012-07-03/
```

This is a lot to type, but she can let the shell do most of the work. If she types:

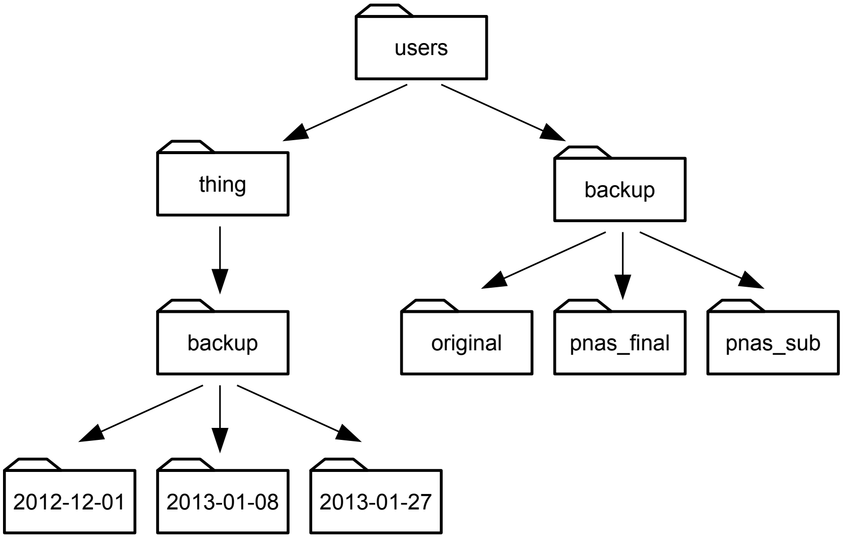


Figure 2.4: File System for Challenge Questions

```
$ ls no
```

and then presses tab, the shell automatically completes the directory name for her:

```
$ ls north-pacific-gyre/
```

If she presses tab again, Bash will add `2012-07-03/` to the command, since it's the only possible completion. Pressing tab again does nothing, since there are 1520 possibilities; pressing tab twice brings up a list of all the files, and so on. This is called *tab completion*, and we will see it in many other tools as we go on.

Key Points

- The file system is responsible for managing information on the disk.
- Information is stored in files, which are stored in directories (folders).
- Directories can also store other directories, which forms a directory tree.
- `/` on its own is the root directory of the whole filesystem.
- A relative path specifies a location starting from the current location.
- An absolute path specifies a location from the root of the filesystem.
- Directory names in a path are separated with '/' on Unix, but '\' on Windows.
- '..' means "the directory above the current one"; '.' on its own means "the current directory".
- Most files' names are `something.extension`. The extension isn't required, and doesn't guarantee anything, but is normally used to indicate the type of data in the file.
- Most commands take options (flags) which begin with a '-'.

Please refer to Figure 2.4 for the following challenges.

Challenge

If pwd displays /users/thing, what will ls ../backup display? 1. ../backup: No such file or directory 2. 2012-12-01 2013-01-08 2013-01-27 3. 2012-12-01/ 2013-01-08/ 2013-01-27/ 4. original pnas_final pnas_sub

Challenge

If pwd displays /users/backup, and -r tells ls to display things in reverse order, what command will display:

```
pnas-sub/ pnas-final/ original/
```

1. ls pwd
2. ls -r -F
3. ls -r -F /users/backup
4. Either #2 or #3 above, but not #1.

Challenge

What does the command cd without a directory name do? 1. It has no effect. 2. It changes the working directory to /. 3. It changes the working directory to the user's home directory. 4. It produces an error message.

Challenge

What does the command ls do when used with the -s and -h arguments?

2.3 Creating Things

Objectives

- Create a directory hierarchy that matches a given diagram.
- Create files in that hierarchy using an editor or by copying and renaming existing files.
- Display the contents of a directory using the command line.
- Delete specified files and/or directories.

We now know how to explore files and directories, but how do we create them in the first place? Let's go back to Vlad's home directory, /users/vlad, and use ls -F to see what it contains:

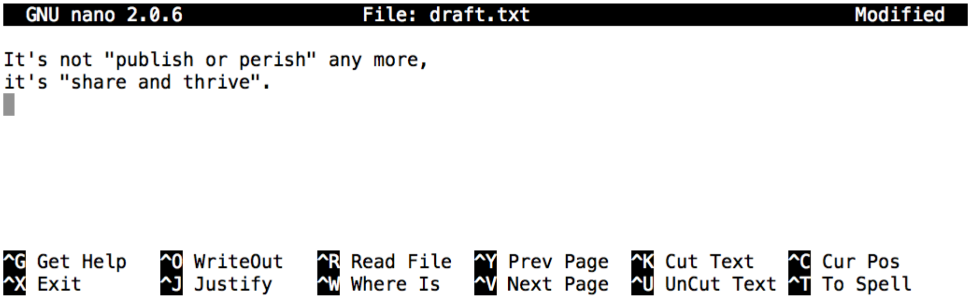
```
$ pwd
```

```
/users/vlad
```

```
$ ls -F
```

```
bin/         data/     mail/      music/
notes.txt    papers/   pizza.cfg  solar/
solar.pdf    swc/
```

Let's create a new directory called thesis using the command mkdir thesis (which has no output):



```
  GNU nano 2.0.6               File: draft.txt                         Modified

It's not "publish or perish" any more,
it's "share and thrive".
```

Figure 2.5: The Nano Editor

```
$ mkdir thesis
```

As you might (or might not) guess from its name, `mkdir` means “make directory”. Since `thesis` is a relative path (i.e., doesn’t have a leading slash), the new directory is made below the current working directory:

```
$ ls -F
```

```
bin/        data/     mail/      music/
notes.txt   papers/   pizza.cfg  solar/
solar.pdf   swc/      thesis/
```

However, there’s nothing in it yet:

```
$ ls -F thesis
```

Let’s change our working directory to `thesis` using `cd`, then run a text editor called Nano to create a file called `draft.txt` (Figure 2.5).

```
$ cd thesis
$ nano draft.txt
```

Which Editor?

When we say, “`nano` is a text editor” we really do mean “text”: it can only work with plain character data, not tables, images, or any other human-friendly media. We use it in examples because almost anyone can drive it anywhere without training, but please use something more powerful for real work. On Unix systems (such as Linux and Mac OS X), many programmers use Emacs[4] or Vim[5] (both of which are completely unintuitive, even by Unix standards), or a graphical editor such as Gedit[6]. On Windows, you may wish to use Notepad++[7].

No matter what editor you use, you will need to know where it searches for and saves files. If you start it from the shell, it will (probably) use your current working directory as its default location. If you use your computer’s start menu, it may want to save files in your desktop or documents directory instead. You can change this by navigating to another directory the first time you “Save As…”

[4] http://www.gnu.org/software/emacs/
[5] http://www.vim.org/
[6] http://projects.gnome.org/gedit/
[7] http://notepad-plus-plus.org/

Let’s type in a few lines of text, then use Control-O to write our data to disk:

Once our file is saved, we can use Control-X to quit the editor and return to the shell. (Unix documentation often uses the shorthand `^A` to mean “control-A”.) `nano` doesn’t leave any output on the screen after it exits, but `ls` now shows that we have created a file called `draft.txt`:

```
$ ls
```

```
draft.txt
```

Let’s tidy up by running `rm draft.txt`:

```
$ rm draft.txt
```

This command removes files (“rm” is short for “remove”). If we run `ls` again, its output is empty once more, which tells us that our file is gone:

```
$ ls
```

> **Deleting Is Forever**
>
> Unix doesn’t have a trash bin: when we delete files, they are unhooked from the file system so that their storage space on disk can be recycled. Tools for finding and recovering deleted files do exist, but there’s no guarantee they’ll work in any particular situation, since the computer may recycle the file’s disk space right away.

Let’s re-create that file and then move up one directory to `/users/vlad` using `cd ..`:

```
$ pwd
```

```
/users/vlad/thesis
```

```
$ nano draft.txt
$ ls
```

```
draft.txt
```

```
$ cd ..
```

If we try to remove the entire `thesis` directory using `rm thesis`, we get an error message:

```
$ rm thesis
```

```
rm: cannot remove `thesis': Is a directory
```

This happens because `rm` only works on files, not directories. The right command is `rmdir`, which is short for “remove directory”. It doesn’t work yet either, though, because the directory we’re trying to remove isn’t empty:

```
$ rmdir thesis
```

```
rmdir: failed to remove `thesis': Directory not empty
```

This little safety feature can save you a lot of grief, particularly if you are a bad typist. To really get rid of `thesis` we must first delete the file `draft.txt`:

```
$ rm thesis/draft.txt
```

The directory is now empty, so `rmdir` can delete it:

```
$ rmdir thesis
```

> **With Great Power Comes Great Responsibility**
>
> Removing the files in a directory just so that we can remove the directory quickly becomes tedious. Instead, we can use `rm` with the `-r` flag (which stands for "recursive"):
>
> ```
> $ rm -r thesis
> ```
>
> This removes everything in the directory, then the directory itself. If the directory contains sub-directories, `rm -r` does the same thing to them, and so on. It's very handy, but can do a lot of damage if used without care.

Let's create that directory and file one more time. (Note that this time we're running `nano` with the path `thesis/draft.txt`, rather than going into the `thesis` directory and running `nano` on `draft.txt` there.)

```
$ pwd
```

```
/users/vlad
```

```
$ mkdir thesis
```

```
$ nano thesis/draft.txt
$ ls thesis
```

```
draft.txt
```

`draft.txt` isn't a particularly informative name, so let's change the file's name using `mv`, which is short for "move":

```
$ mv thesis/draft.txt thesis/quotes.txt
```

The first parameter tells `mv` what we're "moving", while the second is where it's to go. In this case, we're moving `thesis/draft.txt` to `thesis/quotes.txt`, which has the same effect as renaming the file. Sure enough, `ls` shows us that `thesis` now contains one file called `quotes.txt`:

```
$ ls thesis
```

```
quotes.txt
```

Just for the sake of inconsistency, `mv` also works on directories—there is no separate `mvdir` command.

Let's move `quotes.txt` into the current working directory. We use `mv` once again, but this time we'll just use the name of a directory as the second parameter to tell `mv` that we want to keep the filename, but put the file somewhere new. (This is why the command is called "move".) In this case, the directory name we use is the special directory name `.` that we mentioned earlier.

```
$ mv thesis/quotes.txt .
```

The effect is to move the file from the directory it was in to the current working directory. `ls` now shows us that `thesis` is empty:

```
$ ls thesis
```

Further, `ls` with a filename or directory name as a parameter only lists that file or directory. We can use this to see that `quotes.txt` is still in our current directory:

```
$ ls quotes.txt
```

```
quotes.txt
```

The `cp` command works very much like `mv`, except it copies a file instead of moving it. We can check that it did the right thing using `ls` with two paths as parameters—like most Unix commands, `ls` can be given thousands of paths at once:

```
$ cp quotes.txt thesis/quotations.txt
$ ls quotes.txt thesis/quotations.txt
```

```
quotes.txt   thesis/quotations.txt
```

To prove that we made a copy, let's delete the `quotes.txt` file in the current directory and then run that same `ls` again. This time it tells us that it can't find `quotes.txt` in the current directory, but it does find the copy in `thesis` that we didn't delete:

```
$ ls quotes.txt thesis/quotations.txt
```

```
ls: cannot access quotes.txt: No such file or directory
```

```
thesis/quotations.txt
```

Another Useful Abbreviation

The shell interprets the character `~` (tilde) at the start of a path to mean "the current user's home directory". For example, if Vlad's home directory is `/home/vlad`, then `~/data` is equivalent to `/home/vlad/data`. This only works if it is the first character in the path: `here/there/~/elsewhere` is *not* `/home/vlad/elsewhere`.

Key Points

- Unix documentation uses '^A' to mean "control-A".
- The shell does not have a trash bin: once something is deleted, it's really gone.
- Nano is a very simple text editor—please use something else for real work.

Challenge

What is the output of the closing `ls` command in the sequence shown below?

```
$ pwd
```

```
/home/thing/data
```

```
$ ls
```

```
proteins.dat
```

```
$ mkdir recombine
$ mv proteins.dat recombine
$ cp recombine/proteins.dat ../proteins-saved.dat
$ ls
```

Challenge

Suppose that:

```
$ ls -F
```

```
analyzed/  fructose.dat    raw/   sucrose.dat
```

What command(s) could you run so that the commands below will produce the output shown?

```
$ ls
```

```
analyzed   raw
```

```
$ ls analyzed
```

```
fructose.dat    sucrose.dat
```

Challenge

What does `cp` do when given several filenames and a directory name, as in:

```
$ mkdir backup
$ cp thesis/citations.txt thesis/quotations.txt backup
```

What does it do when given three or more filenames, as in:

```
$ ls -F
```

```
intro.txt    methods.txt    survey.txt
```

```
$ cp intro.txt methods.txt survey.txt
```

Challenge

The command `ls -R` lists the contents of directories recursively, i.e., lists their sub-directories, sub-sub-directories, and so on in alphabetical order at each level. The command `ls -t` lists things by time of last change, with most recently changed files or directories first. In what order does `ls -R -t` display things?

2.4 Pipes and Filters

Objectives

- Redirect a command's output to a file.
- Process a file instead of keyboard input using redirection.
- Construct command pipelines with two or more stages.
- Explain what usually happens if a program or pipeline isn't given any input to process.
- Explain Unix's "small pieces, loosely joined" philosophy.

Now that we know a few basic commands, we can finally look at the shell's most powerful feature: the ease with which it lets us combine existing programs in new ways. We'll start with a directory called `molecules` that contains six files describing some simple organic molecules. The `.pdb` extension indicates that these files are in Protein Data Bank format, a simple text format that specifies the type and position of each atom in the molecule.

```
$ ls molecules

cubane.pdb    ethane.pdb    methane.pdb
octane.pdb    pentane.pdb   propane.pdb
```

Let's go into that directory with cd and run the command wc *.pdb. wc is the "word count" command: it counts the number of lines, words, and characters in files. The * in *.pdb matches zero or more characters, so the shell turns *.pdb into a complete list of .pdb files:

```
$ cd molecules
$ wc *.pdb

  20  156 1158 cubane.pdb
  12   84  622 ethane.pdb
   9   57  422 methane.pdb
  30  246 1828 octane.pdb
  21  165 1226 pentane.pdb
  15  111  825 propane.pdb
 107  819 6081 total
```

Wildcards

* is a *wildcard*. It matches zero or more characters, so *.pdb matches ethane.pdb, propane.pdb, and so on. On the other hand, p*.pdb only matches pentane.pdb and propane.pdb, because the 'p' at the front only matches itself.

? is also a wildcard, but it only matches a single character. This means that p?.pdb matches pi.pdb or p5.pdb, but not propane.pdb. We can use any number of wildcards at a time: for example, p*.p?* matches anything that starts with a 'p' and ends with '.', 'p', and at least one more character (since the '?' has to match one character, and the final '*' can match any number of characters). Thus, p*.p?* would match preferred.practice, and even p.pi (since the first '*' can match no characters at all), but not quality.practice (doesn't start with 'p') or preferred.p (there isn't at least one character after the '.p').

When the shell sees a wildcard, it expands the wildcard to create a list of matching filenames *before* running the command that was asked for. This means that commands like wc and ls never see the wildcard characters, just what those wildcards matched. This is another example of orthogonal design.

If we run wc -l instead of just wc, the output shows only the number of lines per file:

```
$ wc -l *.pdb

  20  cubane.pdb
  12  ethane.pdb
   9  methane.pdb
  30  octane.pdb
  21  pentane.pdb
  15  propane.pdb
 107  total
```

We can also use -w to get only the number of words, or -c to get only the number of characters.

Which of these files is shortest? It's an easy question to answer when there are only six files, but what if there were 6000? Our first step toward a solution is to run the command:

```
$ wc -l *.pdb > lengths
```

The > tells the shell to *redirect* the command's output to a file instead of printing it to the screen. The shell will create the file if it doesn't exist, or overwrite the contents of that file if it does. (This is why there is no screen output: everything that wc would have printed has gone into the file lengths instead.) ls lengths confirms that the file exists:

```
$ ls lengths
```

```
lengths
```

We can now send the content of lengths to the screen using cat lengths. cat stands for "concatenate": it prints the contents of files one after another. There's only one file in this case, so cat just shows us what it contains:

```
$ cat lengths
```

```
  20  cubane.pdb
  12  ethane.pdb
   9  methane.pdb
  30  octane.pdb
  21  pentane.pdb
  15  propane.pdb
 107  total
```

Now let's use the sort command to sort its contents. This does *not* change the file; instead, it sends the sorted result to the screen:

```
$ sort lengths
```

```
  9  methane.pdb
 12  ethane.pdb
 15  propane.pdb
 20  cubane.pdb
 21  pentane.pdb
 30  octane.pdb
107  total
```

We can put the sorted list of lines in another temporary file called sorted-lengths by putting > sorted-lengths after the command, just as we used > lengths to put the output of wc into lengths. Once we've done that, we can run another command called head to get the first few lines in sorted-lengths:

```
$ sort lengths > sorted-lengths
$ head -1 sorted-lengths
```

```
  9  methane.pdb
```

Using the parameter -1 with head tells it that we only want the first line of the file; -20 would get the first 20, and so on. Since sorted-lengths contains the lengths of our files ordered from least to greatest, the output of head must be the file with the fewest lines.

If you think this is confusing, you're in good company: even once you understand what wc, sort, and head do, all those intermediate files make it hard to follow what's going on. We can make it easier to understand by running sort and head together:

```
$ sort lengths | head -1
```

```
  9  methane.pdb
```

The vertical bar between the two commands is called a *pipe*. It tells the shell that we want to use the output of the command on the left as the input to the command on the right. The computer might create a temporary file if it needs to, or copy data from one program to the other in memory, or something else entirely; we don't have to know or care.

We can use another pipe to send the output of `wc` directly to `sort`, which then sends its output to `head`:

```
$ wc -l *.pdb | sort | head -1
```

```
  9  methane.pdb
```

This is exactly like a mathematician nesting functions like $sin(\pi x)^2$ and saying "the square of the sine of x times π". In our case, the calculation is "head of sort of word count of `*.pdb`".

Here's what actually happens behind the scenes when we create a pipe. When a computer runs a program—any program—it creates a *process* in memory to hold the program's software and its current state. Every process has an input channel called *standard input*. (By this point, you may be surprised that the name is so memorable, but don't worry: most Unix programmers call it "stdin". Every process also has a default output channel called *standard output* (or "stdout").

The shell is actually just another program. Under normal circumstances, whatever we type on the keyboard is sent to the shell on its standard input, and whatever it produces on standard output is displayed on our screen. When we tell the shell to run a program, it creates a new process and temporarily sends whatever we type on our keyboard to that process's standard input, and whatever the process sends to standard output to the screen.

Here's what happens when we run `wc -l *.pdb > lengths`. The shell starts by telling the computer to create a new process to run the `wc` program. Since we've provided some filenames as parameters, `wc` reads from them instead of from standard input. And since we've used `>` to redirect output to a file, the shell connects the process's standard output to that file.

If we run `wc -l *.pdb | sort` instead, the shell creates two processes (one for each process in the pipe) so that `wc` and `sort` run simultaneously. The standard output of `wc` is fed directly to the standard input of `sort`; since there's no redirection with `>`, `sort`'s output goes to the screen. And if we run `wc -l *.pdb | sort | head -1`, we get three processes with data flowing from the files, through `wc` to `sort`, and from `sort` through `head` to the screen.

This simple idea is why Unix has been so successful. Instead of creating enormous programs that try to do many different things, Unix programmers focus on creating lots of simple tools that each do one job well, and that work well with each other. This programming model is called *pipes and filters*. We've already seen pipes; a *filter* is a program like `wc` or `sort` that transforms a stream of input into a stream of output. Almost all of the standard Unix tools can work this way: unless told to do otherwise, they read from standard input, do something with what they've read, and write to standard output.

The key is that any program that reads lines of text from standard input and writes lines of text to standard output can be combined with every other program that behaves this way as well. You can *and should* write your programs this way so that you and other people can put those programs into pipes to multiply their power.

Redirecting Input

> As well as using > to redirect a program's output, we can use < to redirect its input, i.e., to read from a file instead of from standard input. For example, instead of writing wc ammonia.pdb, we could write wc < ammonia.pdb. In the first case, wc gets a command line parameter telling it what file to open. In the second, wc doesn't have any command line parameters, so it reads from standard input, but we have told the shell to send the contents of ammonia.pdb to wc's standard input.

Nelle's Pipeline: Checking Files

Nelle has run her samples through the assay machines and created 1520 files in the north-pacific-gyre/2012-07-03 directory described earlier. As a quick sanity check, she types:

```
$ cd north-pacific-gyre/2012-07-03
$ wc -l *.txt
```

The output is 1520 lines that look like this:

```
300 NENE01729A.txt
300 NENE01729B.txt
300 NENE01736A.txt
300 NENE01751A.txt
300 NENE01751B.txt
300 NENE01812A.txt
... ...
```

Now she types this:

```
$ wc -l *.txt | sort | head -5
```

```
 240 NENE02018B.txt
 300 NENE01729A.txt
 300 NENE01729B.txt
 300 NENE01736A.txt
 300 NENE01751A.txt
```

Whoops: one of the files is 60 lines shorter than the others. When she goes back and checks it, she sees that she did that assay at 8:00 on a Monday morning—someone was probably in using the machine on the weekend, and she forgot to reset it. Before re-running that sample, she checks to see if any files have too much data:

```
$ wc -l *.txt | sort | tail -5
```

```
 300 NENE02040A.txt
 300 NENE02040B.txt
 300 NENE02040Z.txt
 300 NENE02043A.txt
 300 NENE02043B.txt
```

Those numbers look good—but what's that 'Z' doing there in the third-to-last line? All of her samples should be marked 'A' or 'B'; by convention, her lab uses 'Z' to indicate samples with missing information. To find others like it, she does this:

```
$ ls *Z.txt
```

```
NENE01971Z.txt    NENE02040Z.txt
```

Sure enough, when she checks the log on her laptop, there's no depth recorded for either of those samples. Since it's too late to get the information any other way, she must exclude those two files from her analysis. She could just delete them using `rm`, but there are actually some analyses she might do later where depth doesn't matter, so instead, she'll just be careful later on to select files using the wildcard expression `*[AB].txt`. As always, the '*' matches any number of characters; the expression `[AB]` matches either an 'A' or a 'B', so this matches all the valid data files she has.

Key Points

- `command > file` redirects a command's output to a file.
- `first | second` is a pipeline: the output of the first command is used as the input to the second.
- The best way to use the shell is to use pipes to combine simple single-purpose programs (filters).

Challenge

If we run `sort` on this file:

```
10
2
19
22
6
```

the output is:

```
10
19
2
22
6
```

If we run `sort -n` on the same input, we get this instead:

```
2
6
10
19
22
```

Explain why `-n` has this effect.

Challenge

What is the difference between:

```
$ wc -l < mydata.dat
```

and:

```
$ wc -l mydata.dat
```

Challenge

The command `uniq` removes adjacent duplicated lines from its input. For example, if a file `salmon.txt` contains:

```
coho
coho
steelhead
coho
steelhead
steelhead
```

then `uniq salmon.txt` produces:

```
coho
steelhead
coho
steelhead
```

Why do you think `uniq` only removes *adjacent* duplicated lines? (Hint: think about very large data sets.) What other command could you combine with it in a pipe to remove all duplicated lines?

Challenge

A file called `animals.txt` contains the following data:

```
2012-11-05,deer
2012-11-05,rabbit
2012-11-05,raccoon
2012-11-06,rabbit
2012-11-06,deer
2012-11-06,fox
2012-11-07,rabbit
2012-11-07,bear
```

What text passes through each of the pipes and the final redirect in the pipeline below?

```
$ cat animals.txt | head -5 | tail -3 | sort -r > final.txt
```

Challenge

The command:

```
$ cut -d , -f 2 animals.txt
```

produces the following output:

```
deer
rabbit
raccoon
rabbit
deer
fox
rabbit
bear
```

What other command(s) could be added to this in a pipeline to find out what animals the file contains (without any duplicates in their names)?

2.5 Loops

Objectives

- Write a loop that applies one or more commands separately to each file in a set of files.
- Trace the values taken on by a loop variable during execution of the loop.
- Explain the difference between a variable's name and its value.
- Explain why spaces and some punctuation characters shouldn't be used in files' names.
- Demonstrate how to see what commands have recently been executed.
- Re-run recently executed commands without retyping them.

Wildcards and tab completion are two ways to reduce typing (and typing mistakes). Another is to tell the shell to do something over and over again. Suppose we have several hundred genome data files named `basilisk.dat`, `unicorn.dat`, and so on. When new files arrive, we'd like to rename the existing ones to `original-basilisk.dat` and `original-unicorn.dat`. We can't use:

```
$ mv *.dat original-*.dat
```

because that would expand (in the two-file case) to:

```
$ mv basilisk.dat unicorn.dat
```

This wouldn't back up our files: it would replace the content of `unicorn.dat` with whatever's in `basilisk.dat`.

Instead, we can use a *loop* to do some operation once for each thing in a list. Here's a simple example that displays the first three lines of each file in turn:

```
$ for filename in basilisk.dat unicorn.dat
> do
>    head -3 $filename
> done
```

```
COMMON NAME: basilisk
CLASSIFICATION: basiliscus vulgaris
UPDATED: 1745-05-02
COMMON NAME: unicorn
CLASSIFICATION: equus monoceros
UPDATED: 1738-11-24
```

When the shell sees the keyword `for`, it knows it is supposed to repeat a command (or group of commands) once for each thing in a list. In this case, the list is the two filenames. Each time through the loop, the name of the thing currently being operated on is assigned to the *variable* called `filename`. Inside the loop, we get the variable's value by putting `$` in front of it: `$filename` is `basilisk.dat` the first time through the loop, `unicorn.dat` the second, and so on. Finally, the command that's actually being run is our old friend `head`, so this loop prints out the first three lines of each data file in turn.

> **Follow the Prompt**
>
> The shell prompt changes from `$` to `>` and back again as we were typing in our loop. The second prompt, `>`, is different to remind us that we haven't finished typing a complete command yet.

We have called the variable in this loop `filename` in order to make its purpose clearer to human readers. The shell itself doesn't care what the variable is called; if we wrote this loop as:

```
for x in basilisk.dat unicorn.dat
do
    head -3 $x
done
```

or:

```
for temperature in basilisk.dat unicorn.dat
do
    head -3 $temperature
done
```

it would work exactly the same way. *Don't do this.* Programs are only useful if people can understand them, so meaningless names (like `x`) or misleading names (like `temperature`) increase the odds that the program won't do what its readers think it does.

Here's a slightly more complicated loop:

```
for filename in *.dat
do
    echo $filename
    head -100 $filename | tail -20
done
```

The shell starts by expanding `*.dat` to create the list of files it will process. The *loop body* then executes two commands for each of those files. The first, `echo`, just prints its command-line parameters to standard output. For example:

```
$ echo hello there
```

prints:

```
hello there
```

In this case, since the shell expands `$filename` to be the name of a file, `echo $filename` just prints the name of the file. Note that we can't write this as:

```
for filename in *.dat
do
    $filename
    head -100 $filename | tail -20
done
```

because then the first time through the loop, when `$filename` expanded to `basilisk.dat`, the shell would try to run `basilisk.dat` as a program. Finally, the `head` and `tail` combination selects lines 81-100 from whatever file is being processed.

Spaces in Names

Filename expansion in loops is another reason you should not use spaces in filenames. Suppose our data files are named:

```
basilisk.dat
red dragon.dat
unicorn.dat
```

If we try to process them using:

```
for filename in *.dat
do
    head -100 $filename | tail -20
done
```

then the shell will expand `*.dat` to create:

```
basilisk.dat red dragon.dat unicorn.dat
```

With older versions of Bash, or most other shells, `filename` will then be assigned the following values in turn:

```
basilisk.dat
red
dragon.dat
unicorn.dat
```

That's a problem: `head` can't read files called `red` and `dragon.dat` because they don't exist, and won't be asked to read the file `red dragon.dat`.

We can make our script a little bit more robust by *quoting* our use of the variable:

```
for filename in *.dat
do
    head -100 "$filename" | tail -20
done
```

but it's simpler just to avoid using spaces (or other special characters) in filenames.

Going back to our original file renaming problem, we can solve it using this loop:

```
for filename in *.dat
do
    mv $filename original-$filename
done
```

This loop runs the `mv` command once for each filename. The first time, when `$filename` expands to `basilisk.dat`, the shell executes:

```
mv basilisk.dat original-basilisk.dat
```

The second time, the command is:

```
mv unicorn.dat original-unicorn.dat
```

Measure Twice, Run Once

A loop is a way to do many things at once—or to make many mistakes at once if it does the wrong thing. One way to check what a loop *would* do is to echo the commands it would run instead of actually running them. For example, we could write our file renaming loop like this:

```
for filename in *.dat
do
    echo mv $filename original-$filename
done
```

Instead of running mv, this loop runs echo, which prints out:

```
mv basilisk.dat original-basilisk.dat
mv unicorn.dat original-unicorn.dat
```

without actually running those commands. We can then use up-arrow to redisplay the loop, back-arrow to get to the word echo, delete it, and then press “enter” to run the loop with the actual mv commands. This isn’t foolproof, but it’s a handy way to see what’s going to happen when you’re still learning how loops work.

Nelle’s Pipeline: Processing Files

Nelle is now ready to process her data files. Since she’s still learning how to use the shell, she decides to build up the required commands in stages. Her first step is to make sure that she can select the right files—remember, these are ones whose names end in ‘A’ or ‘B’, rather than ‘Z’:

```
$ cd north-pacific-gyre/2012-07-03
$ for datafile in *[AB].txt
> do
>     echo $datafile
> done

NENE01729A.txt
NENE01729B.txt
NENE01736A.txt
...
NENE02043A.txt
NENE02043B.txt
```

Her next step is to decide what to call the files that the goostats analysis program will create. Prefixing each input file’s name with “stats” seems simple, so she modifies her loop to do that:

```
$ for datafile in *[AB].txt
> do
>     echo $datafile stats-$datafile
> done

NENE01729A.txt stats-NENE01729A.txt
NENE01729B.txt stats-NENE01729B.txt
NENE01736A.txt stats-NENE01736A.txt
...
NENE02043A.txt stats-NENE02043A.txt
NENE02043B.txt stats-NENE02043B.txt
```

She hasn’t actually run goostats yet, but now she’s sure she can select the right files and generate the right output filenames.

Typing in commands over and over again is becoming tedious, though, and Nelle is worried about making mistakes, so instead of re-entering her loop, she presses the up arrow. In response, the shell redisplays the whole loop on one line (using semi-colons to separate the pieces):

```
$ for datafile in *[AB].txt; do echo $datafile stats-$datafile; done
```

Using the left arrow key, Nelle backs up and changes the command echo to goostats:

```
$ for datafile in *[AB].txt; do bash goostats $datafile stats-$datafile; done
```

When she presses enter, the shell runs the modified command. However, nothing appears to happen—there is no output. After a moment, Nelle realizes that since her script doesn't print anything to the screen any longer, she has no idea whether it is running, much less how quickly. She kills the job by typing Control-C, uses up-arrow to repeat the command, and edits it to read:

```
$ for datafile in *[AB].txt; do echo $datafile; bash goostats $datafile stats-$datafile; done
```

Beginning and End

We can move to the beginning of a line in the shell by typing `^A` (which means Control-A) and to the end using `^E`.

When she runs her program now, it produces one line of output every five seconds or so:

```
NENE01729A.txt
NENE01729B.txt
NENE01736A.txt
...
```

1518 times 5 seconds, divided by 60, tells her that her script will take about two hours to run. As a final check, she opens another terminal window, goes into `north-pacific-gyre/2012-07-03`, and uses `cat stats-NENE01729B.txt` to examine one of the output files. It looks good, so she decides to get some coffee and catch up on her reading.

Those Who Know History Can Choose to Repeat It

Another way to repeat previous work is to use the `history` command to get a list of the last few hundred commands that have been executed, and then to use `!123` (where "123" is replaced by the command number) to repeat one of those commands. For example, if Nelle types this:

```
$ history | tail -5
```

```
  456  ls -l NENE0*.txt
  457  rm stats-NENE01729B.txt.txt
  458  bash goostats NENE01729B.txt stats-NENE01729B.txt
  459  ls -l NENE0*.txt
  460  history
```

then she can re-run `goostats` on `NENE01729B.txt` simply by typing `!458`.

Key Points

- A `for` loop repeats commands once for every thing in a list.
- Every `for` loop needs a variable to refer to the current "thing".
- Use `$name` to expand a variable (i.e., get its value).
- Do not use spaces, quotes, or wildcard characters such as '*' or '?' in filenames, as it complicates variable expansion.
- Give files consistent names that are easy to match with wildcard patterns to make it easy to select them for looping.
- Use the up-arrow key to scroll up through previous commands to edit and repeat them.
- Use `history` to display recent commands, and `!number` to repeat a command by number.

Challenge

Suppose that `ls` initially displays:

```
fructose.dat    glucose.dat   sucrose.dat
```

What is the output of:

```
for datafile in *.dat
do
    ls *.dat
done
```

Challenge

In the same directory, what is the effect of this loop?

```
for sugar in *.dat
do
    echo $sugar
    cat $sugar > xylose.dat
done
```

1. Prints `fructose.dat`, `glucose.dat`, and `sucrose.dat`, and copies `sucrose.dat` to create `xylose.dat`.
2. Prints `fructose.dat`, `glucose.dat`, and `sucrose.dat`, and concatenates all three files to create `xylose.dat`.
3. Prints `fructose.dat`, `glucose.dat`, `sucrose.dat`, and `xylose.dat`, and copies `sucrose.dat` to create `xylose.dat`.
4. None of the above.

Challenge

The `expr` program does simple arithmetic using command-line parameters:

```
$ expr 3 + 5
```

```
8
```

```
$ expr 30 / 5 - 2
```

```
4
```

Given this, what is the output of:

```
for left in 2 3
do
    for right in $left
    do
        expr $left + $right
    done
done
```

Challenge

Describe in words what the following loop does.

```
for how in frog11 prcb redig
do
    $how -limit 0.01 NENE01729B.txt
done
```

2.6 Shell Scripts

Objectives

- Write a shell script that runs a command or series of commands for a fixed set of files.
- Run a shell script from the command line.
- Write a shell script that operates on a set of files defined by the user on the command line.
- Create pipelines that include user-written shell scripts.

We are finally ready to see what makes the shell such a powerful programming environment. We are going to take the commands we repeat frequently and save them in files so that we can re-run all those operations again later by typing a single command. For historical reasons, a bunch of commands saved in a file is usually called a *shell script*, but make no mistake: these are actually small programs.

Let's start by putting the following line in the file `middle.sh`:

```
head -20 cholesterol.pdb | tail -5
```

This is a variation on the pipe we constructed earlier: it selects lines 16-20 of the file `cholesterol.pdb`. Remember, we are *not* running it as a command just yet: we are putting the commands in a file.

Once we have saved the file, we can ask the shell to execute the commands it contains. Our shell is called `bash`, so we run the following command:

```
$ bash middle.sh
```

```
ATOM     14  C           1      -1.463  -0.666   1.001  1.00  0.00
ATOM     15  C           1       0.762  -0.929   0.295  1.00  0.00
ATOM     16  C           1       0.771  -0.937   1.840  1.00  0.00
ATOM     17  C           1      -0.664  -0.610   2.293  1.00  0.00
ATOM     18  C           1      -4.705   2.108  -0.396  1.00  0.00
```

Sure enough, our script's output is exactly what we would get if we ran that pipeline directly.

Text vs. Whatever

We usually call programs like Microsoft Word or LibreOffice Writer "text editors", but we need to be a bit more careful when it comes to programming. By default, Microsoft Word uses `.docx` files to store not only text, but also formatting information about fonts, headings, and so on. This extra information isn't stored as characters, and doesn't mean anything to tools like `head`: they expect input files to contain nothing but the letters, digits, and punctuation on a standard computer keyboard. When editing programs, therefore, you must either use a plain text editor, or be careful to save files as plain text.

What if we want to select lines from an arbitrary file? We could edit `middle.sh` each time to change the filename, but that would probably take longer than just retyping the command. Instead, let's edit `middle.sh` and replace `cholesterol.pdb` with a special variable called `$1`:

```
$ cat middle.sh
```

```
head -20 $1 | tail -5
```

Inside a shell script, `$1` means "the first filename (or other parameter) on the command line". We can now run our script like this:

```
$ bash middle.sh cholesterol.pdb
```

```
ATOM     14  C           1      -1.463  -0.666   1.001  1.00  0.00
ATOM     15  C           1       0.762  -0.929   0.295  1.00  0.00
ATOM     16  C           1       0.771  -0.937   1.840  1.00  0.00
ATOM     17  C           1      -0.664  -0.610   2.293  1.00  0.00
ATOM     18  C           1      -4.705   2.108  -0.396  1.00  0.00
```

or on a different file like this:

```
$ bash middle.sh vitamin-a.pdb
```

```
ATOM     14  C           1       1.788  -0.987  -0.861
ATOM     15  C           1       2.994  -0.265  -0.829
ATOM     16  C           1       4.237  -0.901  -1.024
ATOM     17  C           1       5.406  -0.117  -1.087
ATOM     18  C           1      -0.696  -2.628  -0.641
```

We still need to edit `middle.sh` each time we want to adjust the range of lines, though. Let's fix that by using the special variables `$2` and `$3`:

```
$ cat middle.sh
```

```
head $2 $1 | tail $3
```

```
$ bash middle.sh vitamin-a.pdb -20 -5
```

```
ATOM     14  C           1       1.788  -0.987  -0.861
ATOM     15  C           1       2.994  -0.265  -0.829
ATOM     16  C           1       4.237  -0.901  -1.024
ATOM     17  C           1       5.406  -0.117  -1.087
ATOM     18  C           1      -0.696  -2.628  -0.641
```

This works, but it may take the next person who reads `middle.sh` a moment to figure out what it does. We can improve our script by adding some *comments* at the top:

```
$ cat middle.sh
```

```
# Select lines from the middle of a file.
# Usage: middle.sh filename -end_line -num_lines
head $2 $1 | tail $3
```

A comment starts with a `#` character and runs to the end of the line. The computer ignores comments, but they're invaluable for helping people understand and use scripts.

What if we want to process many files in a single pipeline? For example, if we want to sort our `.pdb` files by length, we would type:

```
$ wc -l *.pdb | sort -n
```

because wc -l lists the number of lines in the files and sort -n sorts things numerically. We could put this in a file, but then it would only ever sort a list of .pdb files in the current directory. If we want to be able to get a sorted list of other kinds of files, we need a way to get all those names into the script. We can't use $1, $2, and so on because we don't know how many files there are. Instead, we use the special variable $*, which means, "All of the command-line parameters to the shell script." Here's an example:

```
$ cat sorted.sh

wc -l $* | sort -n

$ bash sorted.sh *.dat backup/*.dat

      29 chloratin.dat
      89 backup/chloratin.dat
      91 sphagnoi.dat
     156 sphag2.dat
     172 backup/sphag-merged.dat
     182 girmanis.dat
```

Why Isn't It Doing Anything?

What happens if a script is supposed to process a bunch of files, but we don't give it any filenames? For example, what if we type:

```
$ bash sorted.sh
```

but don't say *.dat (or anything else)? In this case, $* expands to nothing at all, so the pipeline inside the script is effectively:

```
wc -l | sort -n
```

Since it doesn't have any filenames, wc assumes it is supposed to process standard input, so it just sits there and waits for us to give it some data interactively. From the outside, though, all we see is it sitting there: the script doesn't appear to do anything.

We have two more things to do before we're finished with our simple shell scripts. If you look at a script like:

```
wc -l $* | sort -n
```

you can probably puzzle out what it does. On the other hand, if you look at this script:

```
# List files sorted by number of lines.
wc -l $* | sort -n
```

you don't have to puzzle it out—the comment at the top tells you what it does. A line or two of documentation like this make it much easier for other people (including your future self) to re-use your work. The only caveat is that each time you modify the script, you should check that the comment is still accurate: an explanation that sends the reader in the wrong direction is worse than none at all.

Second, suppose we have just run a series of commands that did something useful—for example, that created a graph we'd like to use in a paper. We'd like to be able to re-create the graph later if we need to, so we want to save the commands in a file. Instead of typing them in again (and potentially getting them wrong) we can do this:

```
$ history | tail -4 > redo-figure-3.sh
```

The file `redo-figure-3.sh` now contains:

```
297 goostats -r NENE01729B.txt stats-NENE01729B.txt
298 goodiff stats-NENE01729B.txt /data/validated/01729.txt > 01729-differences.txt
299 cut -d ',' -f 2-3 01729-differences.txt > 01729-time-series.txt
300 ygraph --format scatter --color bw --borders none 01729-time-series.txt figure-3.png
```

After a moment's work in an editor to remove the serial numbers on the commands, we have a completely accurate record of how we created that figure.

Unnumbering

Nelle could also use `colrm` (short for "column removal") to remove the serial numbers on her previous commands. Its parameters are the range of characters to strip from its input:

```
$ history | tail -5

  173  cd /tmp
  174  ls
  175  mkdir bakup
  176  mv bakup backup
  177  history | tail -5

$ history | tail -5 | colrm 1 7

cd /tmp
ls
mkdir bakup
mv bakup backup
history | tail -5
history | tail -5 | colrm 1 7
```

In practice, most people develop shell scripts by running commands at the shell prompt a few times to make sure they're doing the right thing, then saving them in a file for re-use. This style of work allows people to recycle what they discover about their data and their workflow with one call to `history` and a bit of editing to clean up the output and save it as a shell script.

Nelle's Pipeline: Creating a Script

An off-hand comment from her supervisor has made Nelle realize that she should have provided a couple of extra parameters to `goostats` when she processed her files. This might have been a disaster if she had done all the analysis by hand, but thanks to for loops, it will only take a couple of hours to re-do.

But experience has taught her that if something needs to be done twice, it will probably need to be done a third or fourth time as well. She runs the editor and writes the following:

```
# Calculate reduced stats for data files at J = 100 c/bp.
for datafile in $*
do
    echo $datafile
    goostats -J 100 -r $datafile stats-$datafile
done
```

(The parameters `-J 100` and `-r` are the ones her supervisor said she should have used.) She saves this in a file called `do-stats.sh` so that she can now re-do the first stage of her analysis by typing:

```
$ bash do-stats.sh *[AB].txt
```

She can also do this:

```
$ bash do-stats.sh *[AB].txt | wc -l
```

so that the output is just the number of files processed rather than the names of the files that were processed.

One thing to note about Nelle's script is that it lets the person running it decide what files to process. She could have written it as:

```
# Calculate reduced stats for  A and Site B data files at J = 100 c/bp.
for datafile in *[AB].txt
do
    echo $datafile
    goostats -J 100 -r $datafile stats-$datafile
done
```

The advantage is that this always selects the right files: she doesn't have to remember to exclude the 'Z' files. The disadvantage is that it *always* selects just those files—she can't run it on all files (including the 'Z' files), or on the 'G' or 'H' files her colleagues in Antarctica are producing, without editing the script. If she wanted to be more adventurous, she could modify her script to check for command-line parameters, and use `*[AB].txt` if none were provided. Of course, this introduces another tradeoff between flexibility and complexity.

Key Points

- Save commands in files (usually called shell scripts) for re-use.
- `bash filename` runs the commands saved in a file.
- `$*` refers to all of a shell script's command-line parameters.
- `$1`, `$2`, etc., refer to specified command-line parameters.
- Letting users decide what files to process is more flexible and more consistent with built-in Unix commands.

Challenge

Leah has several hundred data files, each of which is formatted like this:

```
2013-11-05,deer,5
2013-11-05,rabbit,22
2013-11-05,raccoon,7
2013-11-06,rabbit,19
2013-11-06,deer,2
2013-11-06,fox,1
2013-11-07,rabbit,18
2013-11-07,bear,1
```

Write a shell script called `species.sh` that takes any number of filenames as command-line parameters, and uses `cut`, `sort`, and `uniq` to print a list of the unique species appearing in each of those files separately.

Challenge

Write a shell script called `longest.sh` that takes the name of a directory and a filename extension as its parameters, and prints out the name of the file with the most lines in that directory with that extension. For example:

```
$ bash longest.sh /tmp/data pdb
```

would print the name of the `.pdb` file in `/tmp/data` that has the most lines.

Challenge

If you run the command:

```
$ history | tail -5 > recent.sh
```

the last command in the file is the `history` command itself, i.e., the shell has added `history` to the command log before actually running it. In fact, the shell *always* adds commands to the log before running them. Why do you think it does this?

Challenge

Joel's `data` directory contains three files: `fructose.dat`, `glucose.dat`, and `sucrose.dat`. Explain what a script called `example.sh` would do when run as `bash example.sh *.dat` if it contained the following lines:

```
# Script 1
echo *.*

# Script 2
for filename in $1 $2 $3
do
    cat $filename
done

# Script 3
echo $*.dat
```

2.7 Finding Things

Objectives

- Use `grep` to select lines from text files that match simple patterns.
- Use `find` to find files whose names match simple patterns.
- Use the output of one command as the command-line parameters to another command.
- Explain what is meant by "text" and "binary" files, and why many common tools don't handle the latter well.

You can guess someone's age by how they talk about search: young people use "Google" as a verb, while crusty old Unix programmers use "grep". The word is a contraction of "global/regular expression/print", a common sequence of operations in early Unix text editors. It is also the name of a very useful command-line program.

`grep` finds and prints lines in files that match a pattern. For our examples, we will use a file that contains three haikus taken from a 1998 competition in *Salon* magazine:

```
$ cat haiku.txt

The Tao that is seen
Is not the true Tao, until
You bring fresh toner.

With searching comes loss
and the presence of absence:
"My Thesis" not found.

Yesterday it worked
Today it is not working
Software is like that.
```

Forever, or Five Years

We haven't linked to the original haikus because they don't appear to be on *Salon*'s site any longer. As Jeff Rothenberg said[8], "Digital information lasts forever—or five years, whichever comes first."

Let's find lines that contain the word "not":

```
$ grep not haiku.txt

Is not the true Tao, until
"My Thesis" not found
Today it is not working
```

Here, `not` is the pattern we're searching for. It's pretty simple: every alphanumeric character matches against itself. After the pattern comes the name or names of the files we're searching in. The output is the three lines in the file that contain the letters "not".

Let's try a different pattern: "day".

```
$ grep day haiku.txt

Yesterday it worked
Today it is not working
```

This time, the output is lines containing the words "Yesterday" and "Today", which both have the letters "day". If we give `grep` the `-w` flag, it restricts matches to word boundaries, so that only lines with the word "day" will be printed:

```
$ grep -w day haiku.txt
```

In this case, there aren't any, so `grep`'s output is empty.

Another useful option is `-n`, which numbers the lines that match:

```
$ grep -n it haiku.txt

5:With searching comes loss
9:Yesterday it worked
10:Today it is not working
```

[8] http://www.clir.org/pubs/archives/ensuring.pdf

Here, we can see that lines 5, 9, and 10 contain the letters “it”.

We can combine flags as we do with other Unix commands. For example, since -i makes matching case-insensitive and -v inverts the match, using them both only prints lines that *don’t* match the pattern in any mix of upper and lower case:

```
$ grep -i -v the haiku.txt
```

```
You bring fresh toner.

With searching comes loss

Yesterday it worked
Today it is not working
Software is like that.
```

grep has lots of other options. To find out what they are, we can type man grep. man is the Unix “manual” command: it prints a description of a command and its options, and (if you’re lucky) provides a few examples of how to use it:

```
$ man grep
```

```
GREP(1)                                                                                               GREP(1)

NAME
grep, egrep, fgrep - print lines matching a pattern

SYNOPSIS
grep [OPTIONS] PATTERN [FILE...]
grep [OPTIONS] [-e PATTERN | -f FILE] [FILE...]

DESCRIPTION
grep  searches the named input FILEs (or standard input if no files are named, or if a single hyphen-
minus (-) is given as file name) for lines containing a match to the given PATTERN.  By default, grep
prints the matching lines.
...        ...        ...

OPTIONS
Generic Program Information
--help Print  a  usage  message  briefly summarizing these command-line options and the bug-reporting
address, then exit.

-V, --version
Print the version number of grep to the standard output stream.  This version number should be
included in all bug reports (see below).

Matcher Selection
-E, --extended-regexp
Interpret  PATTERN  as  an  extended regular expression (ERE, see below).  (-E is specified by
POSIX.)

-F, --fixed-strings
Interpret PATTERN as a list of fixed strings, separated by newlines, any of  which  is  to  be
matched.  (-F is specified by POSIX.)
...        ...        ...
```

Wildcards

grep’s real power doesn’t come from its options, though; it comes from the fact that patterns can include wildcards. (The technical name for these is *regular expressions*,

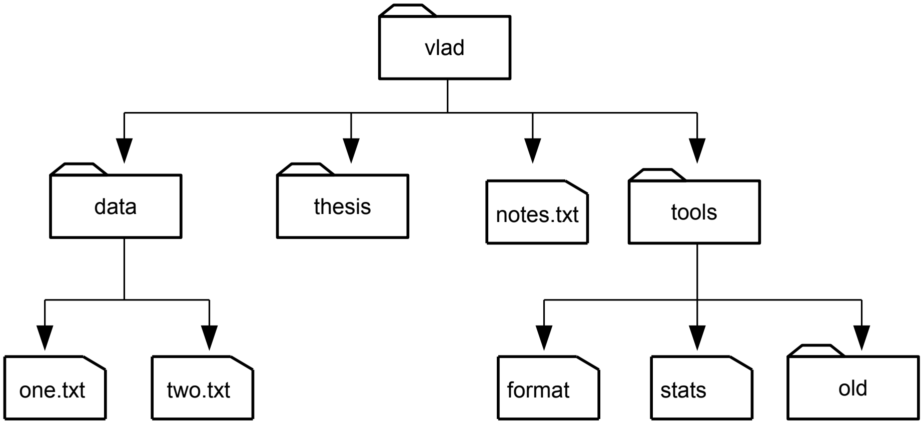


Figure 2.6: Directory Tree for `find` Examples

which is what the “re” in “grep” stands for.) Regular expressions are both complex and powerful; if you want to do complex searches, please look at the lesson on our website[9]. As a taster, we can find lines that have an ‘o’ in the second position like this:

```
$ grep -E '^.o' haiku.txt

You bring fresh toner.
Today it is not working
Software is like that.
```

We use the `-E` flag and put the pattern in quotes to prevent the shell from trying to interpret it. (If the pattern contained a ‘*’, for example, the shell would try to expand it before running `grep`.) The ‘\^’ in the pattern anchors the match to the start of the line. The ‘.’ matches a single character (just like ‘?’ in the shell), while the ‘o’ matches an actual ‘o’.

While `grep` finds lines in files, the `find` command finds files themselves. Again, it has a lot of options; to show how the simplest ones work, we’ll use the directory tree shown in Figure 2.6.

Vlad’s home directory contains one file called `notes.txt` and four subdirectories: `thesis` (which is sadly empty), `data` (which contains two files `one.txt` and `two.txt`), a `tools` directory that contains the programs `format` and `stats`, and an empty subdirectory called `old`.

For our first command, let’s run `find . -type d -print`. As always, the `.` on its own means the current working directory, which is where we want our search to start; `-type d` means “things that are directories”, and (unsurprisingly) `-print` means “print what’s found”. Sure enough, `find`’s output is the names of the five directories in our little tree (including `.`):

```
$ find . -type d -print

./
./data
./thesis
./tools
./tools/old
```

[9] http://software-carpentry.org

If we change `-type d` to `-type f`, we get a listing of all the files instead:

```
$ find . -type f -print
```

```
./data/one.txt
./data/two.txt
./notes.txt
./tools/format
./tools/stats
```

`find` automatically goes into subdirectories, their subdirectories, and so on to find everything that matches the pattern we've given it. If we don't want it to, we can use `-maxdepth` to restrict the depth of search:

```
$ find . -maxdepth 1 -type f -print
```

```
./notes.txt
```

The opposite of `-maxdepth` is `-mindepth`, which tells `find` to only report things that are at or below a certain depth. `-mindepth 2` therefore finds all the files that are two or more levels below us:

```
$ find . -mindepth 2 -type f -print
```

```
./data/one.txt
./data/two.txt
./tools/format
./tools/stats
```

Another option is `-empty`, which restricts matches to empty files and directories:

```
$ find . -empty -print
```

```
./thesis
./tools/old
```

Now let's try matching by name:

```
$ find . -name *.txt -print
```

```
./notes.txt
```

We expected it to find all the text files, but it only prints out `./notes.txt`. The problem is that the shell expands wildcard characters like `*` *before* commands run. Since `*.txt` in the current directory expands to `notes.txt`, the command we actually ran was:

```
$ find . -name notes.txt -print
```

`find` did what we asked; we just asked for the wrong thing.

To get what we want, let's do what we did with `grep`: put `*.txt` in single quotes to prevent the shell from expanding the `*` wildcard. This way, `find` actually gets the pattern `*.txt`, not the expanded filename `notes.txt`:

```
$ find . -name '*.txt' -print
```

```
./data/one.txt
./data/two.txt
./notes.txt
```

Listing vs. Finding

`ls` and `find` can be made to do similar things given the right options, but under normal circumstances, `ls` lists everything it can, while `find` searches for things with certain properties and shows them.

As we said earlier, the command line's power lies in combining tools. We've seen how to do that with pipes; let's look at another technique. As we just saw, `find . -name ’*.txt’ -print` gives us a list of all text files in or below the current directory. How can we combine that with `wc -l` to count the lines in all those files?

The simplest way is to put the `find` command inside `$()`:

```
$ wc -l $(find . -name '*.txt' -print)
```

```
70  ./data/one.txt
420  ./data/two.txt
30  ./notes.txt
520  total
```

When the shell executes this command, the first thing it does is run whatever is inside the `$()`. It then replaces the `$()` expression with that command's output. Since the output of `find` is the three filenames `./data/one.txt`, `./data/two.txt`, and `./notes.txt`, the shell constructs the command:

```
$ wc -l ./data/one.txt ./data/two.txt ./notes.txt
```

which is what we wanted. This expansion is exactly what the shell does when it expands wildcards like `*` and `?`, but lets us use any command we want as our own "wildcard".

It's very common to use `find` and `grep` together. The first finds files that match a pattern; the second looks for lines inside those files that match another pattern. Here, for example, we can find PDB files that contain iron atoms by looking for the string "FE" in all the `.pdb` files below the current directory:

```
$ grep FE $(find . -name '*.pdb' -print)
```

```
./human/heme.pdb:ATOM  25  FE  1  -0.924  0.535  -0.518
```

Binary Files

We have focused exclusively on finding things in text files. What if your data is stored as images, in databases, or in some other format? One option would be to extend tools like `grep` to handle those formats. This hasn't happened, and probably won't, because there are too many formats to support.

The second option is to convert the data to text, or extract the text-ish bits from the data. This is probably the most common approach, since it only requires people to build one tool per data format (to extract information). On the one hand, it makes simple things easy to do. On the negative side, complex things are usually impossible. For example, it's easy enough to write a program that will extract X and Y dimensions from image files for `grep` to play with, but how would you write something to find values in a spreadsheet whose cells contained formulas?

The third choice is to recognize that the shell and text processing have their limits, and to use a programming language such as Python instead. When the time comes to do this, don't be too hard on the shell: many modern programming languages, Python included, have borrowed a lot of ideas from it, and imitation is also the sincerest form of praise.

2.8 Conclusion

The Unix shell is older than most of the people who use it. It has survived so long because it is one of the most productive programming environments ever created—maybe even *the* most productive. Its syntax may be cryptic, but people who have mastered it can experiment with different commands interactively, then use what they have learned to automate their work. Graphical user interfaces may be better at the first, but the shell is still unbeaten at the second. And as Alfred North Whitehead wrote in 1911, "Civilization advances by extending the number of important operations which we can perform without thinking about them."

Key Points

- Use `find` to find files and directories, and `grep` to find text patterns in files.
- `$(command)` inserts a command's output in place.
- `man command` displays the manual page for a given command.

Challenge

Write a short explanatory comment for the following shell script:

```
find . -name '*.dat' -print | wc -l | sort -n
```

Challenge

The `-v` flag to `grep` inverts pattern matching, so that only lines which do *not* match the pattern are printed. Given that, which of the following commands will find all files in `/data` whose names end in `ose.dat` (e.g., `sucrose.dat` or `maltose.dat`), but do *not* contain the word `temp`?

1. `find /data -name '*.dat' -print | grep ose | grep -v temp`
2. `find /data -name ose.dat -print | grep -v temp`
3. `grep -v temp $(find /data -name '*ose.dat' -print)`
4. None of the above.

Chapter 3

Version Control with Git

Version control is the lab notebook of the digital world: it's what professionals use to keep track of what they've done and to collaborate with other people. Every large software development project relies on it, and most programmers use it for their small jobs as well. And it isn't just for software: books (like this one), papers, small data sets, and anything that changes over time or needs to be shared can and should be stored in a version control system.

3.1 Introducing Version Control

Wolfman and Dracula have been hired by Universal Missions (a space services spinoff from Euphoric State University) to figure out where the company should send its next planetary lander. They want to be able to work on the plans at the same time, but they have run into problems doing this in the past. If they take turns, each one will spend a lot of time waiting for the other to finish, but if they work on their own copies and email changes back and forth things will be lost, overwritten, or duplicated.

The right solution is to use *version control* to manage their work. Version control is better than mailing files back and forth because:

- Nothing that is committed to version control is ever lost. This means it can be used like the "undo" feature in an editor, and since all old versions of files are saved it's always possible to go back in time to see exactly who wrote what on a particular day, or what version of a program was used to generate a particular set of results.
- It keeps a record of who made what changes when, so that if people have questions later on, they know who to ask.
- It's hard (but not impossible) to accidentally overlook or overwrite someone's changes: the version control system automatically notifies users whenever there's a conflict between one person's work and another's.

This lesson shows how to use a popular open source version control system called Git. It is more complex than some alternatives, but it is widely used, both because it's easy to set up and because of a hosting site called GitHub[1]. No matter which version control system you use, the most important thing to learn is not the details of their more obscure commands, but the workflow that they encourage.

[1] http://github.com

3.2 A Better Kind of Backup

Objectives

- Explain which initialization and configuration steps are required once per machine, and which are required once per repository.
- Go through the modify-add-commit cycle for single and multiple files and explain where information is stored at each stage.
- Identify and Use Git revision numbers.
- Compare files with old versions of themselves.
- Restore old versions of files.
- Configure Git to ignore specific files, and explain why it is sometimes useful to do so.

We'll start by exploring how version control can be used to keep track of what one person did and when. Even if you aren't collaborating with other people, version control is much better for this than the common alternative (Figure 3.1).

"Piled Higher and Deeper" by Jorge Cham, http://www.phdcomics.com

Setting Up

The first time we use Git on a new machine, we need to configure a few things. Here's how Dracula sets up his new laptop:

```
$ git config --global user.name "Vlad Dracula"
$ git config --global user.email "vlad@tran.sylvan.ia"
$ git config --global color.ui "auto"
$ git config --global core.editor "nano"
```

(Please use your own name and email address instead of Dracula's, and please make sure you choose an editor that's actually on your system, such as `notepad` on Windows.)

Git commands are written `git verb`, where `verb` is what we actually want it to do. In this case, we're telling Git:

- our name and email address,
- to colorize output,
- what our favorite text editor is, and
- that we want to use these settings globally (i.e., for every project),

The four commands above only need to be run once: the flag `--global` tells Git to use the settings for every project on this machine.

Creating a Repository

Once Git is configured, we can start using it. Let's create a directory for our work:

```
$ mkdir planets
$ cd planets
```

and tell Git to make it a *repository*—a place where Git can store old versions of our files:

```
$ git init
```

If we use `ls` to show the directory's contents, it appears that nothing has changed:

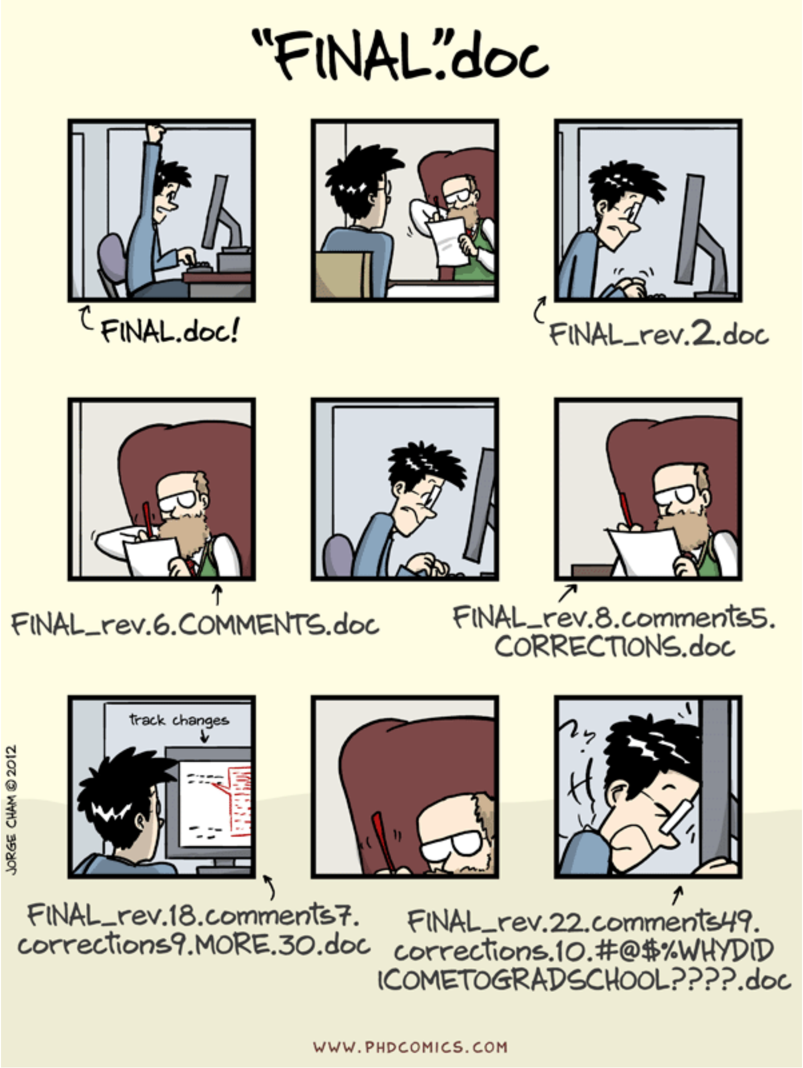


Figure 3.1: How Most Scientists Manage Files

```
$ ls
```

But if we add the `-a` flag to show everything, we can see that Git has created a hidden directory called `.git`:

```
$ ls -a
```

```
.  ..  .git
```

Git stores information about the project in this special sub-directory. If we ever delete it, we will lose the project's history.

We can check that everything is set up correctly by asking Git to tell us the status of our project:

```
$ git status
```

```
# On branch master
#
# Initial commit
#
nothing to commit (create/copy files and use "git add" to track)
```

Tracking Changes to Files

Let's create a file called `mars.txt` that contains some notes about the Red Planet's suitability as a base. (We'll use nano to edit the file; you can use whatever editor you like. In particular, this does not have to be the core.editor you set globally earlier.)

```
$ nano mars.txt
```

Type the text below into the `mars.txt` file:

```
Cold and dry, but everything is my favorite color
```

`mars.txt` now contains a single line:

```
$ ls
```

```
mars.txt
```

```
$ cat mars.txt
```

```
Cold and dry, but everything is my favorite color
```

If we check the status of our project again, Git tells us that it's noticed the new file:

```
$ git status
```

```
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   mars.txt
nothing added to commit but untracked files present (use "git add" to track)
```

The "untracked files" message means that there's a file in the directory that Git isn't keeping track of. We can tell Git that it should do so using `git add`:

```
$ git add mars.txt
```

and then check that the right thing happened:

```
$ git status
```

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#   new file:   mars.txt
#
```

Git now knows that it's supposed to keep track of `mars.txt`, but it hasn't yet recorded any changes for posterity as a commit. To get it to do that, we need to run one more command:

```
$ git commit -m "Starting to think about Mars"
```

```
[master (root-commit) f22b25e] Starting to think about Mars
 1 file changed, 1 insertion(+)
 create mode 100644 mars.txt
```

When we run `git commit`, Git takes everything we have told it to save by using `git add` and stores a copy permanently inside the special `.git` directory. This permanent copy is called a *revision* and its short identifier is `f22b25e`. (Your revision may have another identifier.)

We use the `-m` flag (for "message") to record a comment that will help us remember later on what we did and why. If we just run `git commit` without the `-m` option, Git will launch `nano` (or whatever other editor we configured at the start) so that we can write a longer message.

If we run `git status` now:

```
$ git status
```

```
# On branch master
nothing to commit, working directory clean
```

it tells us everything is up to date. If we want to know what we've done recently, we can ask Git to show us the project's history using `git log`:

```
$ git log
```

```
commit f22b25e3233b4645dabd0d81e651fe074bd8e73b
Author: Vlad Dracula <vlad@tran.sylvan.ia>
Date:   Thu Aug 22 09:51:46 2013 -0400

    Starting to think about Mars
```

`git log` lists all revisions made to a repository in reverse chronological order. The listing for each revision includes the revision's full identifier (which starts with the same characters as the short identifier printed by the `git commit` command earlier), the revision's author, when it was created, and the log message Git was given when the revision was created.

Where Are My Changes?

If we run `ls` at this point, we will still see just one file called `mars.txt`. That's because Git saves information about files' history in the special `.git` directory mentioned earlier so that our filesystem doesn't become cluttered (and so that we can't accidentally edit or delete an old version).

Changing a File

Now suppose Dracula adds more information to the file. (Again, we'll edit with nano and then `cat` the file to show its contents; you may use a different editor, and don't need to `cat`.)

```
$ nano mars.txt
$ cat mars.txt
```

```
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
```

When we run `git status` now, it tells us that a file it already knows about has been modified:

```
$ git status
```

```
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#   modified:   mars.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

The last line is the key phrase: "no changes added to commit". We have changed this file, but we haven't told Git we will want to save those changes (which we do with `git add`) much less actually saved them. Let's double-check our work using `git diff`, which shows us the differences between the current state of the file and the most recently saved version:

```
$ git diff
```

```
diff --git a/mars.txt b/mars.txt
index df0654a..315bf3a 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1 +1,2 @@
 Cold and dry, but everything is my favorite color
+The two moons may be a problem for Wolfman
```

The output is cryptic because it is actually a series of commands for tools like editors and `patch` telling them how to reconstruct one file given the other. If we can break it down into pieces:

1. The first line tells us that Git is using the Unix `diff` command to compare the old and new versions of the file.
2. The second line tells exactly which *revisions* of the file Git is comparing; `df0654a` and `315bf3a` are unique computer-generated labels for those revisions.
3. The remaining lines show us the actual differences and the lines on which they occur. In particular, the + markers in the first column show where we are adding lines.

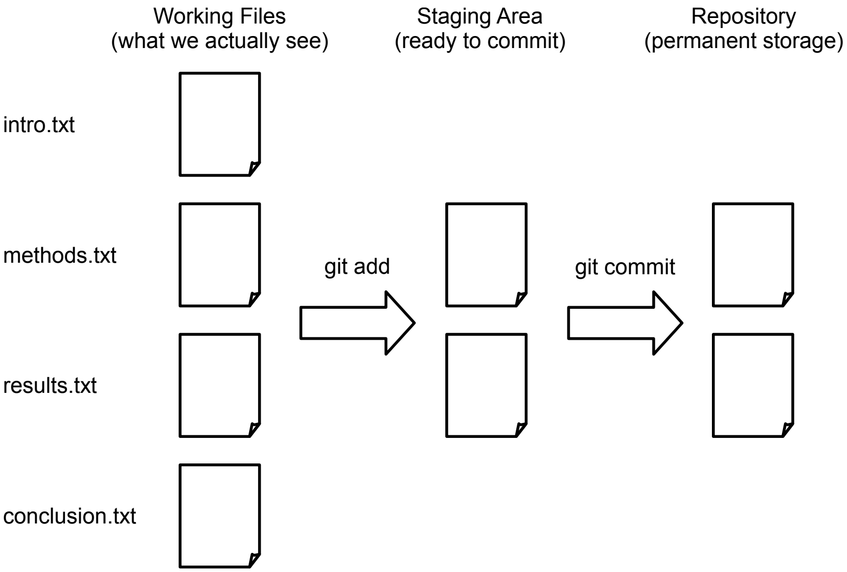


Figure 3.2: Git's Staging Area

Let's commit our change:

```
$ git commit -m "Concerns about Mars's moons on my furry friend"

# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#   modified:   mars.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Whoops: Git won't commit because we didn't use `git add` first. Let's fix that:

```
$ git add mars.txt
$ git commit -m "Concerns about Mars's moons on my furry friend"

[master 34961b1] Concerns about Mars's moons on my furry friend
 1 file changed, 1 insertion(+)
```

Git insists that we add files to the set we want to commit before actually committing anything because we may not want to commit everything at once. For example, suppose we're adding a few citations to our supervisor's work to our thesis. We might want to commit those additions, and the corresponding addition to the bibliography, but *not* commit the work we're doing on the conclusion (which we haven't finished yet).

To allow for this, Git has a special staging area where it keeps track of things that have been added to the current *change set* but not yet committed (Figure 3.2). `git add` puts things in this area, and `git commit` then copies them to long-term storage (as a commit).

Let's watch as our changes to a file move from our editor to the staging area and into long-term storage. First, we'll add another line to the file:

```
$ nano mars.txt
$ cat mars.txt

Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity

$ git diff

diff --git a/mars.txt b/mars.txt
index 315bf3a..b36abfd 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1,2 +1,3 @@
 Cold and dry, but everything is my favorite color
 The two moons may be a problem for Wolfman
+But the Mummy will appreciate the lack of humidity
```

So far, so good: we've added one line to the end of the file (shown with a `+` in the first column). Now let's put that change in the staging area and see what `git diff` reports:

```
$ git add mars.txt
$ git diff
```

There is no output: as far as Git can tell, there's no difference between what it's been asked to save permanently and what's currently in the directory. However, if we do this:

```
$ git diff --staged

diff --git a/mars.txt b/mars.txt
index 315bf3a..b36abfd 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1,2 +1,3 @@
 Cold and dry, but everything is my favorite color
 The two moons may be a problem for Wolfman
+But the Mummy will appreciate the lack of humidity
```

it shows us the difference between the last committed change and what's in the staging area. Let's save our changes:

```
$ git commit -m "Thoughts about the climate"

[master 005937f] Thoughts about the climate
 1 file changed, 1 insertion(+)
```

check our status:

```
$ git status

# On branch master
nothing to commit, working directory clean
```

and look at the history of what we've done so far:

```
$ git log
```

```
commit 005937fbe2a98fb83f0ade869025dc2636b4dad5
Author: Vlad Dracula <vlad@tran.sylvan.ia>
Date:   Thu Aug 22 10:14:07 2013 -0400

    Thoughts about the climate

commit 34961b159c27df3b475cfe4415d94a6d1fcd064d
Author: Vlad Dracula <vlad@tran.sylvan.ia>
Date:   Thu Aug 22 10:07:21 2013 -0400

    Concerns about Mars's moons on my furry friend

commit f22b25e3233b4645dabd0d81e651fe074bd8e73b
Author: Vlad Dracula <vlad@tran.sylvan.ia>
Date:   Thu Aug 22 09:51:46 2013 -0400

    Starting to think about Mars
```

Exploring History

If we want to see what we changed when, we use `git diff` again, but refer to old versions using the notation `HEAD~1`, `HEAD~2`, and so on:

```
$ git diff HEAD~1 mars.txt

diff --git a/mars.txt b/mars.txt
index 315bf3a..b36abfd 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1,2 +1,3 @@
 Cold and dry, but everything is my favorite color
 The two moons may be a problem for Wolfman
+But the Mummy will appreciate the lack of humidity

$ git diff HEAD~2 mars.txt

diff --git a/mars.txt b/mars.txt
index df0654a..b36abfd 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1 +1,3 @@
 Cold and dry, but everything is my favorite color
+The two moons may be a problem for Wolfman
+But the Mummy will appreciate the lack of humidity
```

In this way, we build up a chain of revisions. The most recent end of the chain is referred to as `HEAD`; we can refer to previous revisions using the ~ notation, so `HEAD~1` (pronounced "head minus one") means "the previous revision", while `HEAD~123` goes back 123 revisions from where we are now.

We can also refer to revisions using those long strings of digits and letters that `git log` displays. These are unique IDs for the changes, and "unique" really does mean unique: every change to any set of files on any machine has a unique 40-character identifier. Our first commit was given the ID f22b25e3233b4645dabd0d81e651fe074bd8e73b, so let's try this:

```
$ git diff f22b25e3233b4645dabd0d81e651fe074bd8e73b mars.txt
```

```
diff --git a/mars.txt b/mars.txt
index df0654a..b36abfd 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1 +1,3 @@
 Cold and dry, but everything is my favorite color
+The two moons may be a problem for Wolfman
+But the Mummy will appreciate the lack of humidity
```

That's the right answer, but typing random 40-character strings is annoying, so Git lets us use just the first few:

```
$ git diff f22b25e mars.txt
```

```
diff --git a/mars.txt b/mars.txt
index df0654a..b36abfd 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1 +1,3 @@
 Cold and dry, but everything is my favorite color
+The two moons may be a problem for Wolfman
+But the Mummy will appreciate the lack of humidity
```

Recovering Old Versions

All right: we can save changes to files and see what we've changed—how can we restore older versions of things? Let's suppose we accidentally overwrite our file:

```
$ nano mars.txt
$ cat mars.txt
```

```
We will need to manufacture our own oxygen
```

`git status` now tells us that the file has been changed, but those changes haven't been staged:

```
$ git status
```

```
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#   modified:   mars.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

We can put things back the way they were by using `git checkout`:

```
$ git checkout HEAD mars.txt
$ cat mars.txt
```

```
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
```

As you might guess from its name, `git checkout` checks out (i.e., restores) an old version of a file. In this case, we're telling Git that we want to recover the version of the file recorded in `HEAD`, which is the last saved revision. If we want to go back even further, we can use a revision identifier instead:

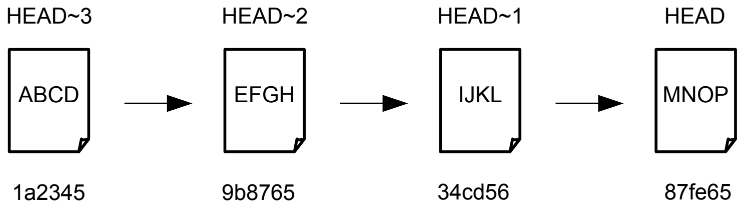


Figure 3.3: When Revisions Are Updated

```
$ git checkout f22b25e mars.txt
```

It's important to remember that we must use the revision number that identifies the state of the repository *before* the change we're trying to undo. A common mistake is to use the revision number of the commit in which we made the change we're trying to get rid of (Figure 3.3).

> **Simplifying the Common Case**
>
> If you read the output of `git status` carefully, you'll see that it includes this hint:
>
> ```
> (use "git checkout -- <file>..." to discard changes in working directory)
> ```
>
> As it says, `git checkout` without a version identifier restores files to the state saved in `HEAD`. The double dash `--` is needed to separate the names of the files being recovered from the command itself: without it, Git would try to use the name of the file as the revision identifier.

The fact that files can be reverted one by one tends to change the way people organize their work. If everything is in one large document, it's hard (but not impossible) to undo changes to the introduction without also undoing changes made later to the conclusion. If the introduction and conclusion are stored in separate files, on the other hand, moving backward and forward in time becomes much easier.

Ignoring Things

What if we have files that we do not want Git to track for us, like backup files created by our editor or intermediate files created during data analysis. Let's create a few dummy files:

```
$ mkdir results
$ touch a.dat b.dat c.dat results/a.out results/b.out
```

and see what Git says:

```
$ git status

# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   a.dat
#   b.dat
#   c.dat
#   results/
nothing added to commit but untracked files present (use "git add" to track)
```

Putting these files under version control would be a waste of disk space. What's worse, having them all listed could distract us from changes that actually matter, so let's tell Git to ignore them.

We do this by creating a file in the root directory of our project called `.gitignore`.

```
$ nano .gitignore
$ cat .gitignore
```

```
*.dat
results/
```

These patterns tell Git to ignore any file whose name ends in `.dat` and everything in the `results` directory. (If any of these files were already being tracked, Git would continue to track them.)

Once we have created this file, the output of `git status` is much cleaner:

```
$ git status
```

```
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   .gitignore
nothing added to commit but untracked files present (use "git add" to track)
```

The only thing Git notices now is the newly-created `.gitignore` file. You might think we wouldn't want to track it, but everyone we're sharing our repository with will probably want to ignore the same things that we're ignoring. Let's add and commit `.gitignore`:

```
$ git add .gitignore
$ git commit -m "Add the ignore file"
$ git status
```

```
# On branch master
nothing to commit, working directory clean
```

As a bonus, using `.gitignore` helps us avoid accidentally adding files to the repository that we don't want.

```
$ git add a.dat
```

```
The following paths are ignored by one of your .gitignore files:
a.dat
Use -f if you really want to add them.
fatal: no files added
```

If we really want to override our ignore settings, we can use `git add -f` to force Git to add something. We can also always see the status of ignored files if we want:

```
$ git status --ignored
```

```
# On branch master
# Ignored files:
#  (use "git add -f <file>..." to include in what will be committed)
#
#        a.dat
#        b.dat
#        c.dat
#        results/

nothing to commit, working directory clean
```

Key Points

- Use `git config` to configure a user name, email address, editor, and other preferences once per machine.
- `git init` initializes a repository.
- `git status` shows the status of a repository.
- Files can be stored in a project's working directory (which users see), the staging area (where the next commit is being built up) and the local repository (where snapshots are permanently recorded).
- `git add` puts files in the staging area.
- `git commit` creates a snapshot of the staging area in the local repository.
- Always write a log message when committing changes.
- `git diff` displays differences between revisions.
- `git checkout` recovers old versions of files.
- The `.gitignore` file tells Git what files to ignore.

Challenge

Create a new Git repository on your computer called `bio`. Write a three-line biography for yourself in a file called `me.txt`, commit your changes, then modify one line and add a fourth and display the differences between its updated state and its original state.

Challenge

The following sequence of commands creates one Git repository inside another:

```
$ cd           # return to home directory
$ mkdir alpha  # make a new directory alpha
$ cd alpha     # go into alpha
$ git init     # make the alpha directory a Git repository
$ mkdir beta   # make a sub-directory alpha/beta
$ cd beta      # go into alpha/beta
$ git init     # make the beta sub-directory a Git repository
```

Why is it a bad idea to do this?

3.3 Collaborating

Objectives

- Explain what remote repositories are and why they are useful.
- Explain what happens when a remote repository is cloned.
- Explain what happens when changes are pushed to or pulled from a remote repository.

Version control really comes into its own when we begin to collaborate with other people. We already have most of the machinery we need to do this; the only thing missing is to copy changes from one repository to another.

Systems like Git allow us to move work between any two repositories. In practice, though, it's easiest to use one copy as a central hub, and to keep it on the web rather than on someone's laptop. Most programmers use hosting services like GitHub[2] or BitBucket[3] to hold those master copies;

[2] http://github.com

[3] http://bitbucket.org

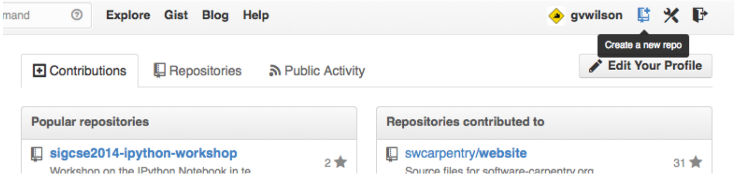
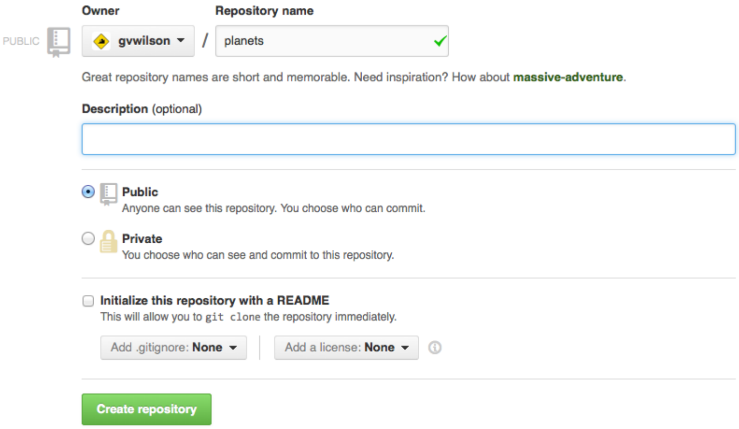


Figure 3.4: First Step in Creating a Repository



Owner: gvwilson / Repository name: planets

Great repository names are short and memorable. Need inspiration? How about **massive-adventure**.

Description (optional)

● **Public**
Anyone can see this repository. You choose who can commit.

○ **Private**
You choose who can see and commit to this repository.

□ **Initialize this repository with a README**
This will allow you to `git clone` the repository immediately.

Add .gitignore: **None**

Add a license: **None**

Figure 3.5: Second Step in Creating a Repository

we'll explore the pros and cons of this in the final section of this lesson.

Let's start by sharing the changes we've made to our current project with the world. Log in to GitHub, then click on the icon in the top right corner to create a new repository called `planets` (Figure 3.4).

Name your repository "planets" and then click "Create Repository" (Figure 3.5). As soon as the repository is created, GitHub displays a page with a URL and some information on how to configure your local repository (Figure 3.6).

This effectively does the following on GitHub's servers:

```
$ mkdir planets
$ cd planets
$ git init
```

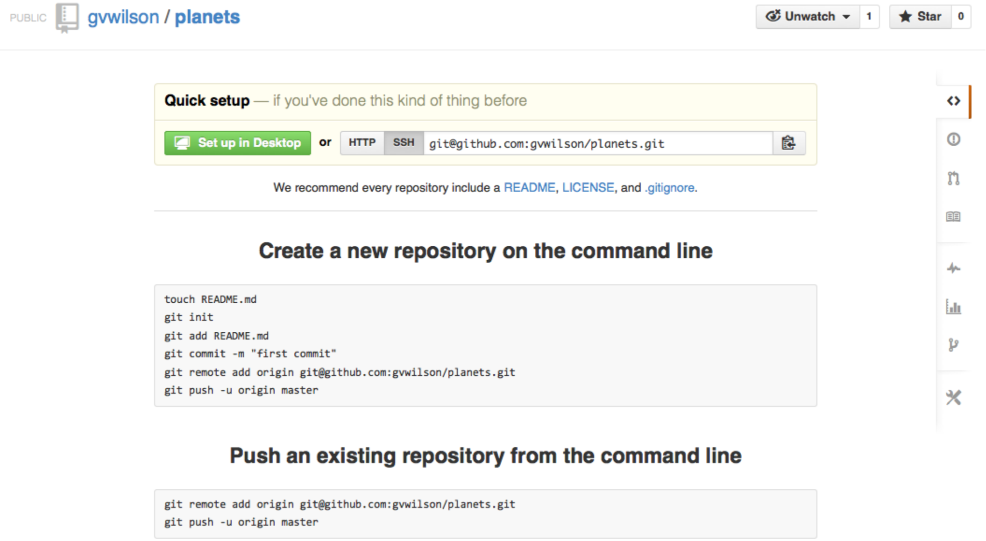
Our local repository still contains our earlier work on `mars.txt`, but the remote repository on GitHub doesn't contain any files yet (Figure 3.7).

The next step is to connect the two repositories. We do this by making the GitHub repository a *remote* for the local repository. The home page of the repository on GitHub includes the string we need to identify it (Figure 3.8).

Click on the 'HTTPS' link to change the *protocol* from SSH to HTTPS. It's slightly less convenient for day-to-day use, but much less work for beginners to set up (Figure 3.9).

Copy that URL from the browser, go into the local `planets` repository, and run this command:

```
$ git remote add origin https://github.com/vlad/planets
```



Quick setup — if you've done this kind of thing before

git@github.com:gvwilson/planets.git

We recommend every repository include a README, LICENSE, and .gitignore.

Create a new repository on the command line

```
touch README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin git@github.com:gvwilson/planets.git
git push -u origin master
```

Push an existing repository from the command line

```
git remote add origin git@github.com:gvwilson/planets.git
git push -u origin master
```

Figure 3.6: Third Step in Creating a Repository

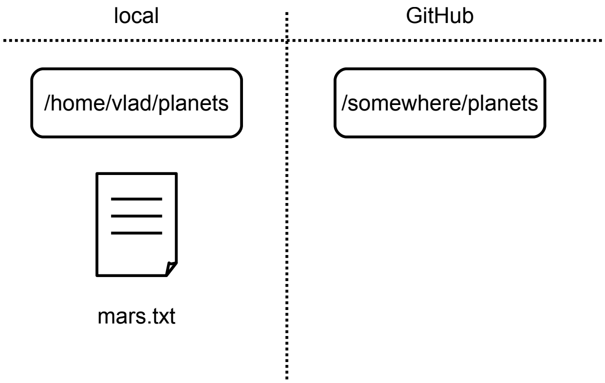
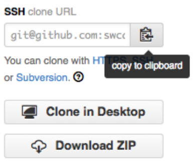
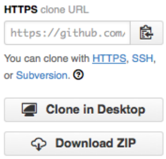


Figure 3.7: Fourth Step in Creating a Repository



You can clone with HT
or Subversion.

Figure 3.8: Fifth Step in Creating a Repository



HTTPS clone URL

https://github.com/

You can clone with HTTPS, SSH, or Subversion.

Figure 3.9: Sixth Step in Creating a Repository

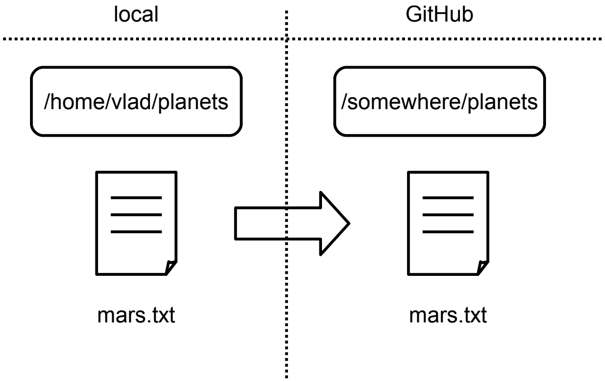


Figure 3.10: Repositories After First Push

Make sure to use the URL for your repository rather than Vlad's: the only difference should be your username instead of `vlad`.

We can check that the command has worked by running `git remote -v`:

```
$ git remote -v
```

```
origin   https://github.com/vlad/planets.git (push)
origin   https://github.com/vlad/planets.git (fetch)
```

The name `origin` is a local nickname for your remote repository: we could use something else if we wanted to, but `origin` is by far the most common choice.

Once the nickname `origin` is set up, this command will push the changes from our local repository to the repository on GitHub:

```
$ git push origin master
```

```
Counting objects: 9, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (9/9), 821 bytes, done.
Total 9 (delta 2), reused 0 (delta 0)
To https://github.com/vlad/planets
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

Our local and remote repositories are now in the state show in Figure 3.10.

The '-u' Flag

You may see a `-u` option used with `git push` in some documentation. It is related to concepts we cover in our intermediate lesson, and can safely be ignored for now.

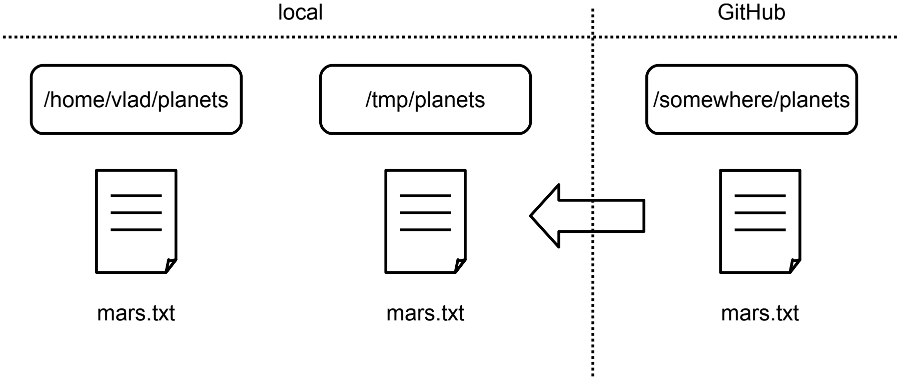


Figure 3.11: After Cloning

We can pull changes from the remote repository to the local one as well:

```
$ git pull origin master
```

```
From https://github.com/vlad/planets
 * branch            master     -> FETCH_HEAD
Already up-to-date.
```

Pulling has no effect in this case because the two repositories are already synchronized. If someone else had pushed some changes to the repository on GitHub, though, this command would download them to our local repository.

We can simulate working with a collaborator using another copy of the repository on our local machine. To do this, `cd` to the directory `/tmp`. (Note the absolute path: don't make `tmp` a subdirectory of the existing repository). Instead of creating a new repository here with `git init`, we will *clone* the existing repository from GitHub:

```
$ cd /tmp
$ git clone https://github.com/vlad/planets.git
```

`git clone` creates a fresh local copy of a remote repository. (We did it in `/tmp` or some other directory so that we don't overwrite our existing `planets` directory.) Our computer now has two copies of the repository (Figure 3.11).

Let's make a change in the copy in `/tmp/planets`:

```
$ cd /tmp/planets
$ nano pluto.txt
$ cat pluto.txt
```

```
It is so a planet!
```

```
$ git add pluto.txt
$ git commit -m "Some notes about Pluto"
```

```
 1 file changed, 1 insertion(+)
 create mode 100644 pluto.txt
```

then push the change to GitHub:

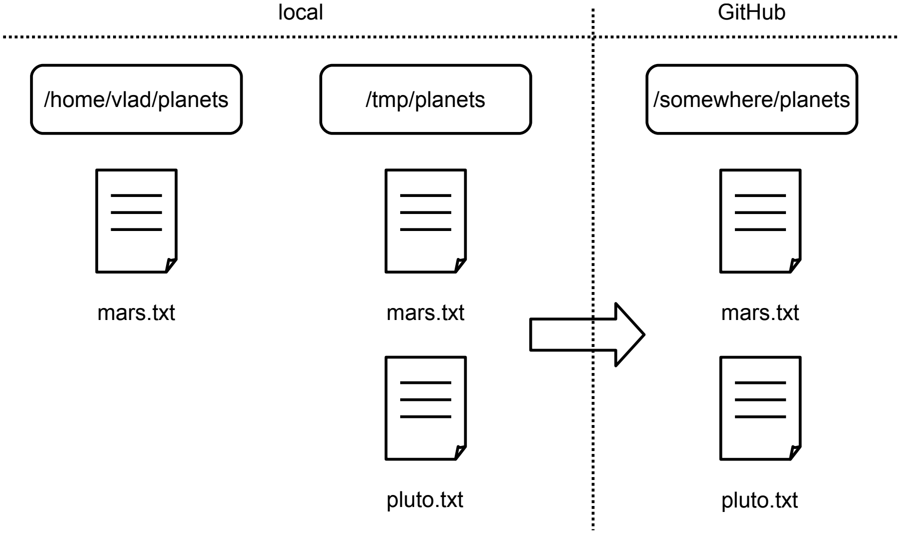


Figure 3.12: Repositories After Change to Duplicate

```
$ git push origin master

Counting objects: 4, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 306 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/vlad/planets.git
   9272da5..29aba7c  master -> master
```

Note that we didn't have to create a remote called `origin`: Git does this automatically, using that name, when we clone a repository. (This is why `origin` was a sensible choice earlier when we were setting up remotes by hand.)

Our three repositories are now as shown in Figure 3.12.

We can now download changes into the original repository on our machine:

```
$ cd ~/planets
$ git pull origin master

remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://github.com/vlad/planets
 * branch            master     -> FETCH_HEAD
Updating 9272da5..29aba7c
Fast-forward
 pluto.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 pluto.txt
```

which puts us in the state shown in Figure 3.13.

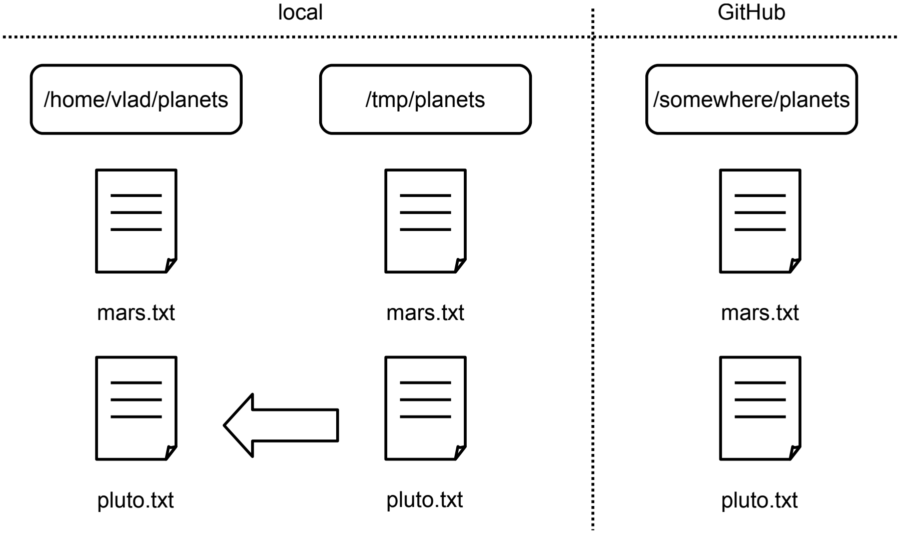


Figure 3.13: Repositories After Pulling

In practice, we would probably never have two copies of the same remote repository on our laptop at once. Instead, one of those copies would be on our laptop, and the other on a lab machine, or on someone else's computer. Pushing and pulling changes gives us a reliable way to share work between different people and machines.

Key Points

- A local Git repository can be connected to one or more remote repositories.
- Use the HTTPS protocol to connect to remote repositories until you have learned how to set up SSH.
- `git push` copies changes from a local repository to a remote repository.
- `git pull` copies changes from a remote repository to a local repository.
- `git clone` copies a remote repository to create a local repository with a remote called `origin` automatically set up.

Challenge

Create a repository on GitHub, clone it, add a file, push those changes to GitHub, and then look at the *timestamp* of the change on GitHub. How does GitHub record times, and why?

3.4 Conflicts

Objectives

- Explain what conflicts are and when they can occur.
- Resolve conflicts resulting from a merge.

As soon as people can work in parallel, someone's going to step on someone else's toes. This will even happen with a single person: if we are working on a piece of software on both our laptop and a server in the lab, we could make different changes to each copy. Version control helps us manage these *conflicts* by giving us tools to *resolve* overlapping changes.

To see how we can resolve conflicts, we must first create one. The file `mars.txt` currently looks like this in both local copies of our `planets` repository (the one in our home directory and the one in `/tmp`):

```
$ cat mars.txt
```

```
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
```

Let's add a line to the copy under our home directory:

```
$ nano mars.txt
$ cat mars.txt
```

```
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
This line added to our home copy
```

and then push the change to GitHub:

```
$ git add mars.txt
$ git commit -m "Adding a line in our home copy"
```

```
[master 5ae9631] Adding a line in our home copy
 1 file changed, 1 insertion(+)
```

```
$ git push origin master
```

```
Counting objects: 5, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 352 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
To https://github.com/vlad/planets
   29aba7c..dabb4c8  master -> master
```

Our repositories are now as shown in Figure 3.14.

Now let's switch to the copy under `/tmp` and make a different change there *without* updating from GitHub:

```
$ cd /tmp/planets
$ nano mars.txt
$ cat mars.txt
```

```
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
We added a different line in the temporary copy
```

We can commit the change locally:

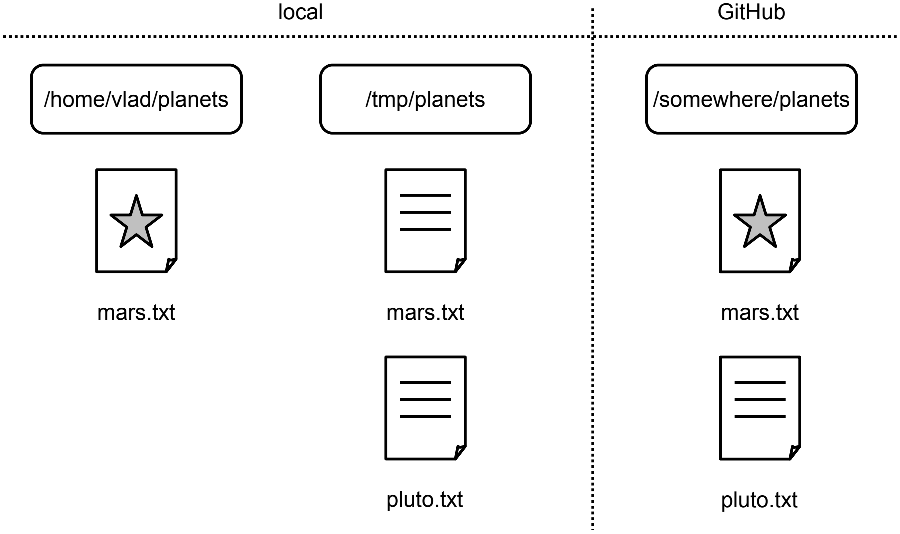


Figure 3.14: Repositories After First Conflict

```
$ git add mars.txt
$ git commit -m "Adding a line in the temporary copy"

[master 07ebc69] Adding a line in the temporary copy
 1 file changed, 1 insertion(+)
```

but Git won't let us push it to GitHub:

```
$ git push origin master

To https://github.com/vlad/planets.git
 ! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to 'https://github.com/vlad/planets.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Merge the remote changes (e.g. 'git pull')
hint: before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Git detects that the changes made in one copy overlap with those made in the other and stops us from trampling on our previous work. What we have to do is pull the changes from GitHub, *merge* them into the copy we're currently working in, and then push that. Let's start by pulling:

```
$ git pull origin master

remote: Counting objects: 5, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 3 (delta 1)
Unpacking objects: 100% (3/3), done.
From https://github.com/vlad/planets
 * branch            master     -> FETCH_HEAD
Auto-merging mars.txt
CONFLICT (content): Merge conflict in mars.txt
Automatic merge failed; fix conflicts and then commit the result.
```

`git pull` tells us there's a conflict, and marks that conflict in the affected file:

```
$ cat mars.txt
```

```
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
<<<<<<< HEAD
We added a different line in the temporary copy
=======
This line added to our home copy
>>>>>>> dabb4c8c450e8475aee9b14b4383acc99f42af1d
```

Our change—the one in `HEAD`—is preceded by <<<<<<<. Git has then inserted ======= as a separator between the conflicting changes and marked the end of the content downloaded from GitHub with >>>>>>>. (The string of letters and digits after that marker identifies the revision we've just downloaded.)

It is now up to us to edit this file to remove these markers and reconcile the changes. We can do anything we want: keep the change in this branch, keep the change made in the other, write something new to replace both, or get rid of the change entirely. Let's replace both so that the file looks like this:

```
$ cat mars.txt
```

```
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
We removed the conflict on this line
```

To finish merging, we add `mars.txt` to the changes being made by the merge and then commit:

```
$ git add mars.txt
$ git status
```

```
# On branch master
# All conflicts fixed but you are still merging.
#   (use "git commit" to conclude merge)
#
# Changes to be committed:
#
#   modified:   mars.txt
#
```

```
$ git commit -m "Merging changes from GitHub"
```

```
[master 2abf2b1] Merging changes from GitHub
```

Our repositories are now as shown in Figure 3.15.

so we push our changes to GitHub:

```
$ git push origin master
```

```
Counting objects: 10, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 697 bytes, done.
Total 6 (delta 2), reused 0 (delta 0)
To https://github.com/vlad/planets.git
   dabb4c8..2abf2b1  master -> master
```

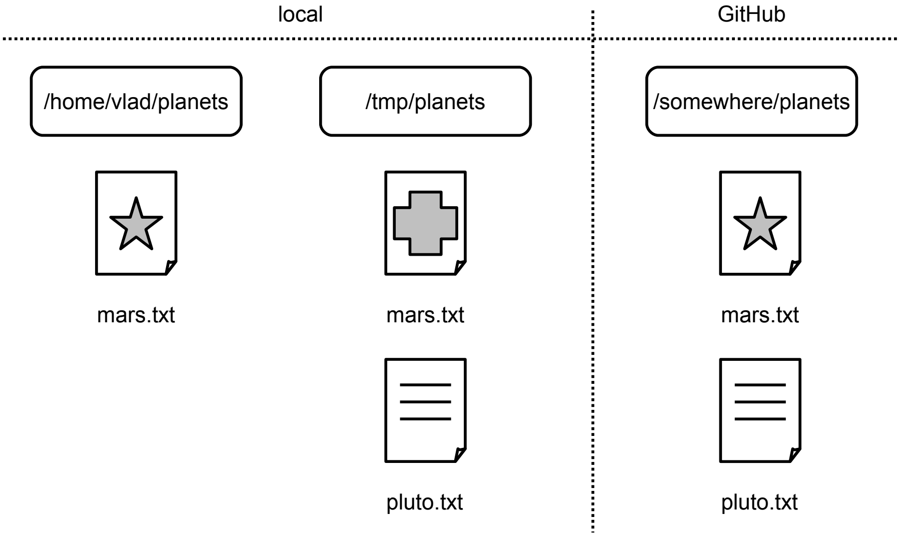


Figure 3.15: Repositories After Second Conflict

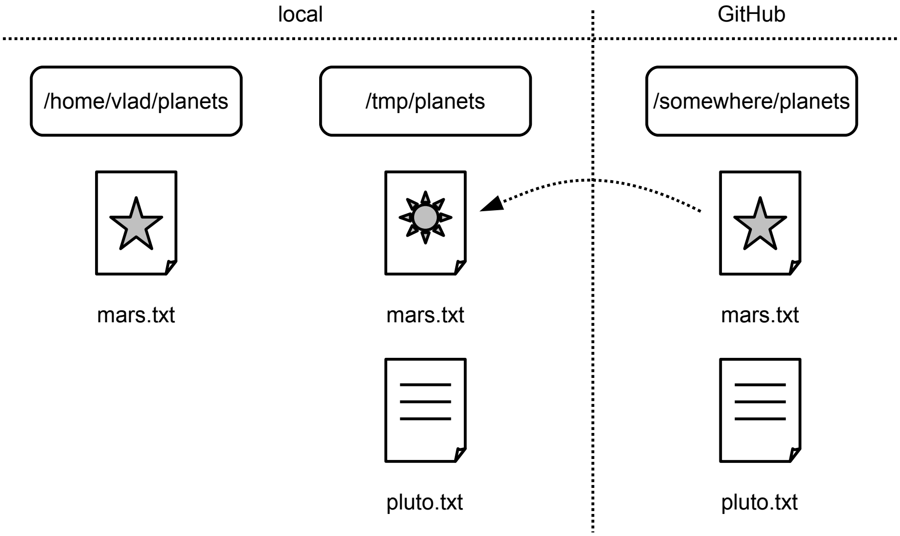


Figure 3.16: Repositories After Merging

to get the state shown in Figure 3.16.

Git keeps track of what we've merged with what, so we don't have to fix things by hand again if we switch back to the repository in our home directory and pull from GitHub:

```
$ cd ~/planets
$ git pull origin master
```

```
remote: Counting objects: 10, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 2), reused 6 (delta 2)
Unpacking objects: 100% (6/6), done.
From https://github.com/vlad/planets
 * branch            master     -> FETCH_HEAD
Updating dabb4c8..2abf2b1
Fast-forward
 mars.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

we get the merged file:

```
$ cat mars.txt
```

```
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
We removed the conflict on this line
```

We don't need to merge again because GitHub knows someone has already done that.

Version control's ability to merge conflicting changes is another reason users tend to divide their programs and papers into multiple files instead of storing everything in one large file. There's another benefit too: whenever there are repeated conflicts in a particular file, the version control system is essentially trying to tell its users that they ought to clarify who's responsible for what, or find a way to divide the work up differently.

Key Points

- Conflicts occur when two or more people change the same file(s) at the same time.
- The version control system does not allow people to blindly overwrite each other's changes. Instead, it highlights conflicts so that they can be resolved.

Challenge

Clone the repository created by your instructor. Add a new file to it, and modify an existing file (your instructor will tell you which one). When asked by your instructor, pull her changes from the repository to create a conflict, then resolve it.

Challenge

What does Git do when there is a conflict in an image or some other non-textual file that is stored in version control?

3.5 Open Science

Objectives

- Explain how the GNU Public License (GPL) differs from most other open licenses.
- Explain the four kinds of restrictions that can be combined in a Creative Commons license.
- Correctly add licensing and citation information to a project repository.
- Outline options for hosting code and data and the pros and cons of each.

> The opposite of "open" isn't "closed". The opposite of "open" is "broken".
> — John Wilbanks

Free sharing of information might be the ideal in science, but the reality is often more complicated. Normal practice today looks something like this:

- A scientist collects some data and stores it on a machine that is occasionally backed up by her department.
- She then writes or modifies a few small programs (which also reside on her machine) to analyze that data.
- Once she has some results, she writes them up and submits her paper. She might include her data—a growing number of journals require this—but she probably doesn't include her code.
- Time passes.
- The journal sends her reviews written anonymously by a handful of other people in her field. She revises her paper to satisfy them, during which time she might also modify the scripts she wrote earlier, and resubmits.
- More time passes.
- The paper is eventually published. It might include a link to an online copy of her data, but the paper itself will be behind a paywall: only people who have personal or institutional access will be able to read it.

For a growing number of scientists, though, the process looks like this:

- The data that the scientist collects is stored in an open access repository like figshare[4] or Dryad[5] as soon as it's collected, and given its own DOI.
- The scientist creates a new repository on GitHub to hold her work.
- As she does her analysis, she pushes changes to her scripts (and possibly some output files) to that repository. She also uses the repository for her paper; that repository is then the hub for collaboration with her colleagues.
- When she's happy with the state of her paper, she posts a version to arXiv[6] or some other preprint server to invite feedback from peers.
- Based on that feedback, she may post several revisions before finally submitting her paper to a journal.
- The published paper includes links to her preprint and to her code and data repositories, which makes it much easier for other scientists to use her work as starting point for their own research.

This open model accelerates discovery: the more open work is, the more widely it is cited and re-used. However, people who want to work this way need to make some decisions about what exactly "open" means in practice.

[4] http://figshare.com/

[5] http://datadryad.org/

[6] http://arxiv.org/

Licensing

The first question is licensing. Broadly speaking, there are two kinds of open license for software, and half a dozen for data and publications. For software, people can choose between the GNU Public License[7] (GPL) on the one hand, and licenses like the MIT[8] and BSD[9] licenses on the other. All of these licenses allow unrestricted sharing and modification of programs, but the GPL is *infective*: anyone who distributes a modified version of the code (or anything that includes GPL'd code) must make *their* code freely available as well.

Proponents of the GPL argue that this requirement is needed to ensure that people who are benefiting from freely-available code are also contributing back to the community. Opponents counter that many open source projects have had long and successful lives without this condition, and that the GPL makes it more difficult to combine code from different sources. At the end of the day, what matters most is that:

1. every project have a file in its home directory called something like `LICENSE` or `LICENSE.txt` that clearly states what the license is, and
2. people use existing licenses rather than writing new ones.

The second point is as important as the first: most scientists are not lawyers, so wording that may seem sensible to a layperson may have unintended gaps or consequences. The Open Source Initiative[10] maintains a list of open source licenses, and tl;drLegal[11] explains many of them in plain English.

When it comes to data, publications, and the like, scientists have many more options to choose from. The good news is that an organization called Creative Commons[12] has prepared a set of licenses using combinations of four basic restrictions:

- Attribution: derived works must give the original author credit for their work.
- No Derivatives: people may copy the work, but must pass it along unchanged.
- Share Alike: derivative works must license their work under the same terms as the original.
- Noncommercial: free use is allowed, but commercial use is not.

These four restrictions are abbreviated "BY", "ND", "SA", and "NC" respectively, so "CC-BY-ND" means, "People can re-use the work both for free and commercially, but cannot make changes and must cite the original." These short descriptions[13] summarize the six CC licenses in plain language, and include links to their full legal formulations.

There is one other important license that doesn't fit into this categorization. Scientists (and other people) can choose to put material in the public domain, which is often abbreviated "PD". In this case, anyone can do anything they want with it, without needing to cite the original or restrict further re-use. The table below shows how the six Creative Commons licenses and PD relate to one another:

[7] http://opensource.org/licenses/GPL-3.0

[8] http://opensource.org/licenses/MIT

[9] http://opensource.org/licenses/BSD-2-Clause

[10] http://opensource.org/

[11] http://www.tldrlegal.com/

[12] http://creativecommons.org/

[13] http://creativecommons.org/licenses/

	Licenses that can be used for derivative work or adaptation						
Original work	by	by-nc	by-nc-nd	by-nc-sa	by-nd	by-sa	pd
by	X	X	X	X	X	X	
by-nc		X	X	X			
by-nc-nd							
by-nc-sa				X			
by-nd							
by-sa						X	
pd	X	X	X	X	X	X	X

Software Carpentry[14] uses CC-BY for its lessons and the MIT License for its code in order to encourage the widest possible re-use. Again, the most important thing is for the `LICENSE` file in the root directory of your project to state clearly what your license is. You may also want to include a file called `CITATION` or `CITATION.txt` that describes how to reference your project; the one for Software Carpentry states:

```
To reference Software Carpentry in publications, please cite both of the following:

Greg Wilson: "Software Carpentry: Lessons Learned". arXiv:1307.5448, July 2013.

@online{wilson-software-carpentry-2013,
  author      = {Greg Wilson},
  title       = {Software Carpentry: Lessons Learned},
  version     = {1},
  date        = {2013-07-20},
  eprinttype  = {arxiv},
  eprint      = {1307.5448}
}
```

Hosting

The second big question for groups that want to open up their work is where to host their code and data. One option is for the lab, the department, or the university to provide a server, manage accounts and backups, and so on. The main benefit of this is that it clarifies who owns what, which is particularly important if any of the material is sensitive (i.e., relates to experiments involving human subjects or may be used in a patent application). The main drawbacks are the cost of providing the service and its longevity: a scientist who has spent ten years collecting data would like to be sure that data will still be available ten years from now, but that's well beyond the lifespan of most of the grants that fund academic infrastructure.

Another option is to purchase a domain and pay an *Internet service provider* (ISP) to host it. This gives the individual or group more control, and sidesteps problems that can arise when moving from one institution to another, but requires more time and effort to set up than either the option above or the option below.

The third option is to use a public hosting service like GitHub[15], BitBucket[16], Google Code[17], or SourceForge[18]. All of these allow people to create repositories through a web interface, and also provide mailing lists, ways to keep track of who's doing what, and so on. They all benefit from

[14] http://software-carpentry.org/license.html

[15] http://github.com

[16] http://bitbucket.org

[17] http://code.google.com

[18] http://sourceforge.net

economies of scale and network effects: it's easier to run one large service well than to run many smaller services to the same standard, and it's also easier for people to collaborate if they're using the same service, not least because it gives them fewer passwords to remember.

However, all of these services place some constraints on people's work. In particular, most give users a choice: if they're willing to share their work with others, it will be hosted for free, but if they want privacy, they may have to pay. Sharing might seem like the only valid choice for science, but many institutions may not allow researchers to do this, either because they want to protect future patent applications or simply because what's new is often also frightening.

Key Points

- Open scientific work is more useful and more highly cited than closed.
- People who incorporate GPL'd software into theirs must make theirs open; most other open licenses do not require this.
- The Creative Commons family of licenses allow people to mix and match requirements and restrictions on attribution, creation of derivative works, further sharing, and commercialization.
- People who are not lawyers should not try to write licenses from scratch.
- Projects can be hosted on university servers, on personal domains, or on public forges.
- Rules regarding intellectual property and storage of sensitive information apply no matter where code and data are hosted.

Challenge

Find out whether you are allowed to apply an open license to your software. Can you do this unilaterally, or do you need permission from someone in your institution? If so, who?

Challenge

Find out whether you are allowed to host your work openly on a public forge. Can you do this unilaterally, or do you need permission from someone in your institution? If so, who?

Chapter 4

Programming with Python

The best way to learn how to program is to do something useful, so this introduction to Python is built around a common scientific task: data analysis.

Our real goal isn't to teach you Python, but to teach you the basic concepts that all programming depends on. We use Python in our lessons because:

1. we have to use *something* for examples;
2. it's free, well-documented, and runs almost everywhere;
3. it has a large (and growing) user base among scientists; and
4. experience shows that it's easier for novices to pick up than most other languages.

But the two most important things are to use whatever language your colleagues are using, so that you can share you work with them easily, and to use that language *well*.

4.1 Analyzing Patient Data

We are studying inflammation in patients who have been given a new treatment for arthritis, and need to analyze the first dozen data sets. The data sets are stored in *comma-separated values* (CSV) format: each row holds information for a single patient, and the columns represent successive days. The first few rows of our first file look like this:

```
0,0,1,3,1,2,4,7,8,3,3,3,10,5,7,4,7,7,12,18,6,13,11,11,7,7,4,6,8,8,4,4,5,7,3,4,2,3,0,0
0,1,2,1,2,1,3,2,2,6,10,11,5,9,4,4,7,16,8,6,18,4,12,5,12,7,11,5,11,3,3,5,4,4,5,5,1,1,0,1
0,1,1,3,3,2,6,2,5,9,5,7,4,5,4,15,5,11,9,10,19,14,12,17,7,12,11,7,4,2,10,5,4,2,2,3,2,2,1,1
0,0,2,0,4,2,2,1,6,7,10,7,9,13,8,8,15,10,10,7,17,4,4,7,6,15,6,4,9,11,3,5,6,3,3,4,2,3,2,1
0,1,1,3,3,1,3,5,2,4,4,7,6,5,3,10,8,10,6,17,9,14,9,7,13,9,12,6,7,7,9,6,3,2,2,4,2,0,1,1
```

We want to:

- load that data into memory,
- calculate the average inflammation per day across all patients, and
- plot the result.

To do all that, we'll have to learn a little bit about programming.

Objectives

- Explain what a library is, and what libraries are used for.
- Load a Python library and use the things it contains.
- Read tabular data from a file into a program.
- Assign values to variables.
- Select individual values and subsections from data.
- Perform operations on arrays of data.
- Display simple graphs.

Loading Data

Words are useful, but what's more useful are the sentences and stories we use them to build. Similarly, while a lot of powerful tools are built into languages like Python, even more lives in the *libraries* they are used to build.

In order to load our inflammation data, we need to *import* a library called NumPy that knows how to operate on matrices:

```
import numpy
```

Importing a library is like getting a piece of lab equipment out of a storage locker and setting it up on the bench. Once it's done, we can ask the library to read our data file for us:

```
numpy.loadtxt(fname='inflammation-01.csv', delimiter=',')
```

```
array([[ 0.,  0.,  1., ...,  3.,  0.,  0.],
       [ 0.,  1.,  2., ...,  1.,  0.,  1.],
       [ 0.,  1.,  1., ...,  2.,  1.,  1.],
       ...,
       [ 0.,  1.,  1., ...,  1.,  1.,  1.],
       [ 0.,  0.,  0., ...,  0.,  2.,  0.],
       [ 0.,  0.,  1., ...,  1.,  1.,  0.]])
```

The expression `numpy.loadtxt(...)` is a *function call* that asks Python to run the function `loadtxt` that belongs to the `numpy` library. This *dotted notation* is used everywhere in Python to refer to the parts of things as `whole.part`.

`numpy.loadtxt` has two *parameters*: the name of the file we want to read, and the *delimiter* that separates values on a line. These both need to be character strings (or *strings* for short), so we put them in quotes.

When we are finished typing and press Shift+Enter, the notebook runs our command. Since we haven't told it to do anything else with the function's output, the notebook displays it. In this case, that output is the data we just loaded. By default, only a few rows and columns are shown (with `...` to omit elements when displaying big arrays). To save space, Python displays numbers as `1.` instead of `1.0` when there's nothing interesting after the decimal point.

Our call to `numpy.loadtxt` read our file, but didn't save the data in memory. To do that, we need to *assign* the array to a *variable*. A variable is just a name for a value, such as `x`, `current_temperature`, or `subject_id`. We can create a new variable simply by assigning a value to it using `=`:

```
weight_kg = 55
```

Once a variable has a value, we can print it:

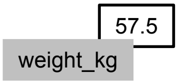


Figure 4.1: Assignment

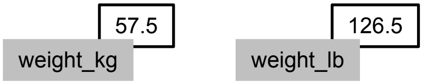


Figure 4.2: Before Changing Weight

```
print weight_kg
```

```
55
```

and do arithmetic with it:

```
print 'weight in pounds:', 2.2 * weight_kg
```

```
weight in pounds: 121.0
```

We can also change a variable's value by assigning it a new one:

```
weight_kg = 57.5
print 'weight in kilograms is now:', weight_kg
```

```
weight in kilograms is now: 57.5
```

As the example above shows, we can print several things at once by separating them with commas.

If we imagine the variable as a sticky note with a name written on it, assignment is like putting the sticky note on a particular value (Figure 4.1).

This means that assigning a value to one variable does *not* change the values of other variables. For example, let's store the subject's weight in pounds in a variable (Figure 4.2):

```
weight_lb = 2.2 * weight_kg
print 'weight in kilograms:', weight_kg, 'and in pounds:', weight_lb
```

```
weight in kilograms: 57.5 and in pounds: 126.5
```

and then change `weight_kg` (Figure 4.3):

```
weight_kg = 100.0
print 'weight in kilograms is now:', weight_kg, 'and weight in pounds is still:', weight_lb
```

```
weight in kilograms is now: 100.0 and weight in pounds is still: 126.5
```

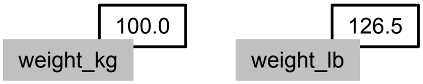


Figure 4.3: After Changing Weight

Since `weight_lb` doesn't "remember" where its value came from, it isn't automatically updated when `weight_kg` changes. This is different from the way spreadsheets work.

Now that we know how to assign things to variables, let's re-run `numpy.loadtxt` and save its result:

```
data = numpy.loadtxt(fname='inflammation-01.csv', delimiter=',')
```

This statement doesn't produce any output because assignment doesn't display anything. If we want to check that our data has been loaded, we can print the variable's value:

```
print data
```

```
[[ 0.  0.  1. ...,  3.  0.  0.]
 [ 0.  1.  2. ...,  1.  0.  1.]
 [ 0.  1.  1. ...,  2.  1.  1.]
 ...,
 [ 0.  1.  1. ...,  1.  1.  1.]
 [ 0.  0.  0. ...,  0.  2.  0.]
 [ 0.  0.  1. ...,  1.  1.  0.]]
```

Challenge

Draw diagrams showing what variables refer to what values after each statement in the following program:

```
mass = 47.5
age = 122
mass = mass * 2.0
age = age - 20
```

Challenge

What does the following program print out?

```
first, second = `Grace', `Hopper'
third, fourth = second, first
print third, fourth
```

Manipulating Data

Now that our data is in memory, we can start doing things with it. First, let's ask what *type* of thing `data` refers to:

```
print type(data)
```

```
<type 'numpy.ndarray'>
```

The output tells us that `data` currently refers to an N-dimensional array created by the NumPy library. We can see what its *shape* is like this:

```
print data.shape
```

```
(60, 40)
```

This tells us that `data` has 60 rows and 40 columns. `data.shape` is a *member* of `data`, i.e., a value that is stored as part of a larger value. We use the same dotted notation for the members of values that we use for the functions in libraries because they have the same part-and-whole relationship.

If we want to get a single value from the matrix, we must provide an *index* in square brackets, just as we do in math:

```
print 'first value in data:', data[0, 0]
```

```
first value in data: 0.0
```

```
print 'middle value in data:', data[30, 20]
```

```
middle value in data: 13.0
```

The expression `data[30, 20]` may not surprise you, but `data[0, 0]` might. Programming languages like Fortran and MATLAB start counting at 1, because that's what human beings have done for thousands of years. Languages in the C family (including C++, Java, Perl, and Python) count from 0 because that's simpler for computers to do. As a result, if we have an MÃŮN array in Python, its indices go from 0 to M-1 on the first axis and 0 to N-1 on the second. It takes a bit of getting used to, but one way to remember the rule is that the index is how many steps we have to take from the start to get the item we want.

> **In the Corner**
>
> What may also surprise you is that when Python displays an array, it shows the element with index `[0, 0]` in the upper left corner rather than the lower left. This is consistent with the way mathematicians draw matrices, but different from the Cartesian coordinates. The indices are (row, column) instead of (column, row) for the same reason.

An index like `[30, 20]` selects a single element of an array, but we can select whole sections as well. For example, we can select the first ten days (columns) of values for the first four (rows) patients like this:

```
print data[0:4, 0:10]
```

```
[[ 0.  0.  1.  3.  1.  2.  4.  7.  8.  3.]
 [ 0.  1.  2.  1.  2.  1.  3.  2.  2.  6.]
 [ 0.  1.  1.  3.  3.  2.  6.  2.  5.  9.]
 [ 0.  0.  2.  0.  4.  2.  2.  1.  6.  7.]]
```

The *slice* `0:4` means, "Start at index 0 and go up to, but not including, index 4." Again, the up-to-but-not-including takes a bit of getting used to, but the rule is that the difference between the upper and lower bounds is the number of values in the slice.

We don't have to start slices at 0:

```
print data[5:10, 0:10]
```

```
[[ 0.  0.  1.  2.  2.  4.  2.  1.  6.  4.]
 [ 0.  0.  2.  2.  4.  2.  2.  5.  5.  8.]
 [ 0.  0.  1.  2.  3.  1.  2.  3.  5.  3.]
 [ 0.  0.  0.  3.  1.  5.  6.  5.  5.  8.]
 [ 0.  1.  1.  2.  1.  3.  5.  3.  5.  8.]]
```

and we don't have to take all the values in the slice—if we provide a *stride*, Python takes values spaced that far apart:

```
print data[0:10:3, 0:10:2]
```

```
[[ 0.  1.  1.  4.  8.]
 [ 0.  2.  4.  2.  6.]
 [ 0.  2.  4.  2.  5.]
 [ 0.  1.  1.  5.  5.]]
```

Here, we have taken rows 0, 3, 6, and 9, and columns 0, 2, 4, 6, and 8. (Again, we always include the lower bound, but stop when we reach or cross the upper bound.)

We also don't have to include the upper and lower bound on the slice. If we don't include the lower bound, Python uses 0 by default; if we don't include the upper, the slice runs to the end of the axis, and if we don't include either (i.e., if we just use ':' on its own), the slice includes everything:

```
small = data[:3, 36:]
print 'small is:'
print small
```

```
small is:
[[ 2.  3.  0.  0.]
 [ 1.  1.  0.  1.]
 [ 2.  2.  1.  1.]]
```

Arrays also know how to perform common mathematical operations on their values. If we want to find the average inflammation for all patients on all days, for example, we can just ask the array for its mean value

```
print data.mean()
```

```
6.14875
```

`mean` is a *method* of the array, i.e., a function that belongs to it in the same way that the member `shape` does. If variables are nouns, methods are verbs: they are what the thing in question knows how to do. This is why `data.shape` doesn't need to be called (it's just a thing) but `data.mean()` does (it's an action). It is also why we need empty parentheses for `data.mean()`: even when we're not passing in any parameters, parentheses are how we tell Python to go and do something for us.

NumPy arrays have lots of useful methods:

```
print 'maximum inflammation:', data.max()
print 'minimum inflammation:', data.min()
print 'standard deviation:', data.std()
```

```
maximum inflammation: 20.0
minimum inflammation: 0.0
standard deviation: 4.61383319712
```

When analyzing data, though, we often want to look at partial statistics, such as the maximum value per patient or the average value per day. One way to do this is to select the data we want to create a new temporary array, then ask it to do the calculation:

```
patient_0 = data[0, :] # 0 on the first axis, everything on the second
print 'maximum inflammation for patient 0:', patient_0.max()
```

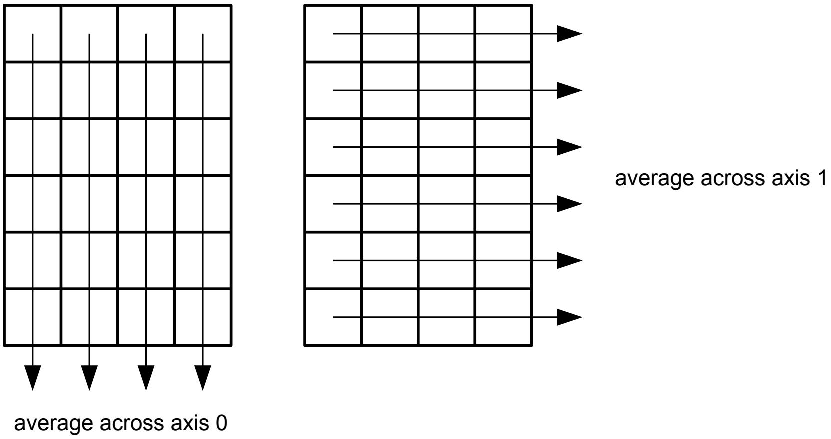


Figure 4.4: Operating Across an Axis

```
maximum inflammation for patient 0: 18.0
```

We don't actually need to store the row in a variable of its own. Instead, we can combine the selection and the method call:

```
print 'maximum inflammation for patient 2:', data[2, :].max()
```

```
maximum inflammation for patient 2: 19.0
```

What if we need the maximum inflammation for *all* patients, or the average for each day? As Figure 4.4 shows, we want to perform the operation across an axis. To support this, most array methods allow us to specify the axis we want to work on. If we ask for the average across axis 0, we get:

```
print data.mean(axis=0)
```

```
[  0.           0.45         1.11666667   1.75         2.43333333   3.15
   3.8          3.88333333   5.23333333   5.51666667   5.95         5.9
   8.35         7.73333333   8.36666667   9.5          9.58333333
  10.63333333  11.56666667  12.35        13.25        11.96666667
  11.03333333  10.16666667  10.           8.66666667   9.15         7.25
   7.33333333   6.58333333   6.06666667   5.95         5.11666667   3.6
   3.3          3.56666667   2.48333333   1.5          1.13333333
   0.56666667]
```

As a quick check, we can ask this array what its shape is:

```
print data.mean(axis=0).shape
```

```
(40,)
```

The expression `(40,)` tells us we have an NÃŮ1 vector, so this is the average inflammation per day for all patients. If we average across axis 1, we get:

```
print data.mean(axis=1)
```

```
[ 5.45   5.425  6.1    5.9    5.55   6.225  5.975  6.65   6.625  6.525
  6.775  5.8    6.225  5.75   5.225  6.3    6.55   5.7    5.85   6.55
  5.775  5.825  6.175  6.1    5.8    6.425  6.05   6.025  6.175  6.55
  6.175  6.35   6.725  6.125  7.075  5.725  5.925  6.15   6.075  5.75
  5.975  5.725  6.3    5.9    6.75   5.925  7.225  6.15   5.95   6.275  5.7
  6.1    6.825  5.975  6.725  5.7    6.25   6.4    7.05   5.9  ]
```

which is the average inflammation per patient across all days.

> **Slicing** A subsection of an array is called a *slice*. We can take slices of character strings as well:
>
> ```
> element = 'oxygen'
> print 'first three characters:', element[0:3]
> print 'last three characters:', element[3:6]
> ```
>
> ```
> first three characters: oxy
> last three characters: gen
> ```

Challenge

What is the value of `element[:4]`? What about `element[4:]`? Or `element[:]`?

Challenge

What is `element[-1]`? What is `element[-2]`? Given those answers, explain what `element[1:-1]` does.

Challenge

The expression `element[3:3]` produces an *empty string*, i.e., a string that contains no characters. If `data` holds our array of patient data, what does `data[3:3, 4:4]` produce? What about `data[3:3, :]`?

Plotting

The mathematician Richard Hamming once said, “The purpose of computing is insight, not numbers,” and the best way to develop insight is often to visualize data. Visualization deserves an entire lecture (or course) of its own, but we can explore a few features of Python’s `matplotlib` here. First, let’s tell the IPython Notebook that we want our plots displayed inline, rather than in a separate viewing window:

```
%matplotlib inline
```

The `%` at the start of the line signals that this is a command for the notebook, rather than a statement in Python. Next, we will import the `pyplot` module from `matplotlib` and use two of its functions to create and display a heat map of our data (Figure 4.5):

```
from matplotlib import pyplot
pyplot.imshow(data)
pyplot.show()
```

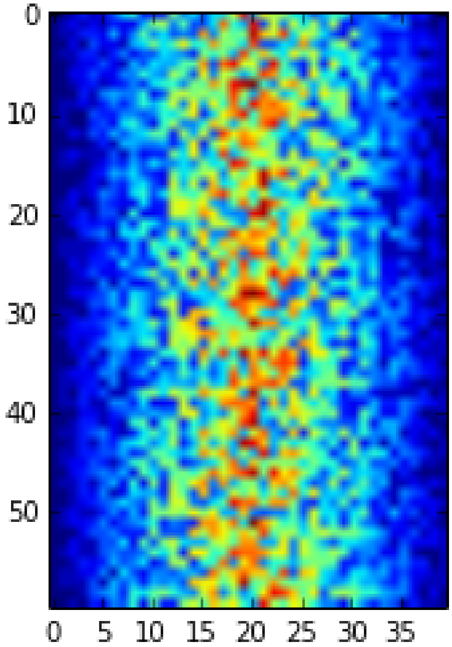


Figure 4.5: Heat Map of Inflammation Data

Blue regions in this heat map are low values, while red shows high values. As we can see, inflammation rises and falls over a 40-day period. Let's take a look at the average inflammation over time (Figure 4.6):

```
ave_inflammation = data.mean(axis=0)
pyplot.plot(ave_inflammation)
pyplot.show()
```

Here, we have put the average per day across all patients in the variable `ave_inflammation`, then asked `pyplot` to create and display a line graph of those values. The result is roughly a linear rise and fall, which is suspicious: based on other studies, we expect a sharper rise and slower fall. Let's have a look at two other statistics, the maximum (Figure 4.7) and minimum (Figure 4.8) per day:

```
pyplot.plot(data.max(axis=0))
pyplot.show()

pyplot.plot(data.min(axis=0))
pyplot.show()
```

The maximum value rises and falls perfectly smoothly, while the minimum seems to be a step function. Neither result seems particularly likely, so either there's a mistake in our calculations or something is wrong with our data.

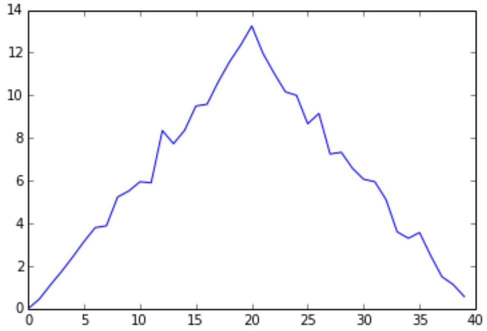


Figure 4.6: Average Inflammation over Time

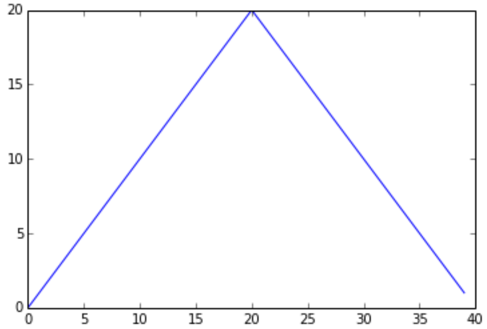


Figure 4.7: Maximum Inflammation per Day

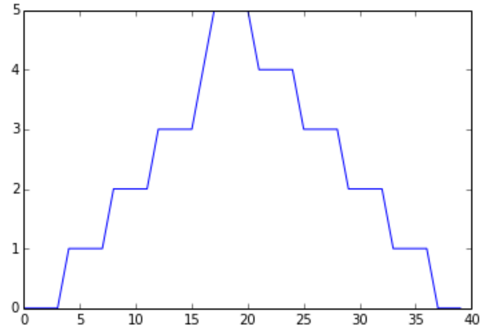


Figure 4.8: Minimum Inflammation per Day

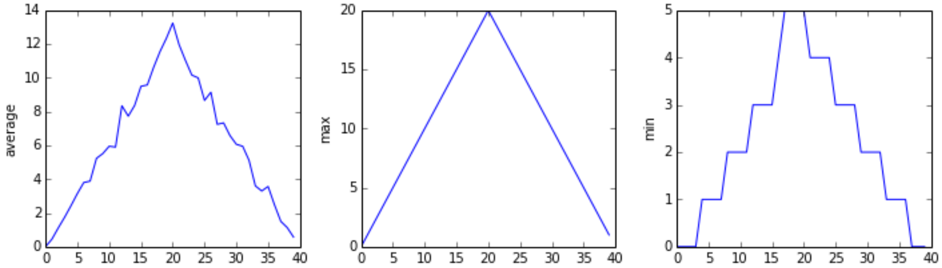


Figure 4.9: Inflammation Statistics vs. Time

Challenge

Why do all of our plots stop just short of the upper end of our graph? Why are the vertical lines in our plot of the minimum inflammation per day not vertical?

Challenge

Create a plot showing the standard deviation of the inflammation data for each day across all patients.

Wrapping Up

It's very common to create an *alias* for a library when importing it in order to reduce the amount of typing we have to do. Figure 4.9 shows our three plots side by side using aliases for numpy and `pyplot`:

```
import numpy as np
from matplotlib import pyplot as plt

data = np.loadtxt(fname='inflammation-01.csv', delimiter=',')

plt.figure(figsize=(10.0, 3.0))

plt.subplot(1, 3, 1)
plt.ylabel('average')
plt.plot(data.mean(0))

plt.subplot(1, 3, 2)
plt.ylabel('max')
plt.plot(data.max(0))

plt.subplot(1, 3, 3)
plt.ylabel('min')
plt.plot(data.min(0))

plt.tight_layout()
plt.show()
```

The first two lines re-load our libraries as `np` and `plt`, which are the aliases most Python programmers use. The call to `loadtxt` reads our data, and the rest of the program tells the plotting library how large we want the figure to be, that we're creating three sub-plots, what to draw for each one, and that we want a tight layout. (Perversely, if we leave out that call to `plt.tight_layout()`, the graphs will actually be squeezed together more closely.)

Challenge

Modify the program to display the three plots on top of one another instead of side by side.

Key Points

- Import a library into a program using `import libraryname`.
- Use the `numpy` library to work with arrays in Python.
- Use `variable = value` to assign a value to a variable in order to record it in memory.
- Variables are created on demand whenever a value is assigned to them.
- Use `print something` to display the value of `something`.
- The expression `array.shape` gives the shape of an array.
- Use `array[x, y]` to select a single element from an array.
- Array indices start at 0, not 1.
- Use `low:high` to specify a slice that includes the indices from `low` to `high-1`.
- All the indexing and slicing that works on arrays also works on strings.
- Use `# some kind of explanation` to add comments to programs.
- Use `array.mean()`, `array.max()`, and `array.min()` to calculate simple statistics.
- Use `array.mean(axis=0)` or `array.mean(axis=1)` to calculate statistics across the specified axis.
- Use the `pyplot` library from `matplotlib` for creating simple visualizations.

4.2 Creating Functions

If we only had one data set to analyze, it would probably be faster to load the file into a spreadsheet and use that to plot some simple statistics. But we have twelve files to check, and may have more in future. In this lesson, we'll learn how to write a function so that we can repeat several operations with a single command.

Objectives

- Define a function that takes parameters.
- Return a value from a function.
- Test and debug a function.
- Explain what a call stack is, and trace changes to the call stack as functions are called.
- Set default values for function parameters.
- Explain why we should divide programs into small, single-purpose functions.

Defining a Function

Let's start by defining a function `fahr_to_kelvin` that converts temperatures from Fahrenheit to Kelvin:

```
def fahr_to_kelvin(temp):
    return ((temp - 32) * (5/9)) + 273.15
```

The definition opens with the word `def`, which is followed by the name of the function and a parenthesized list of parameter names. The *body* of the function—the statements that are executed when it runs—is indented below the definition line, typically by four spaces.

When we call the function, the values we pass to it are assigned to those variables so that we can use them inside the function. Inside the function, we use a *return statement* to send a result back to whoever asked for it.

Let's try running our function. Calling our own function is no different from calling any other function:

```
print 'freezing point of water:', fahr_to_kelvin(32)
print 'boiling point of water:', fahr_to_kelvin(212)
```

```
freezing point of water: 273.15
boiling point of water: 273.15
```

We've successfully called the function that we defined, and we have access to the value that we returned. Unfortunately, the value returned doesn't look right. What went wrong?

Debugging a Function

Debugging is when we fix a piece of code that we know is working incorrectly. In this case, we know that `fahr_to_kelvin` is giving us the wrong answer, so let's find out why.

For big pieces of code, there are tools called *debuggers* that aid in this process.

We just have a short function, so we'll debug by choosing some parameter value, breaking our function into small parts, and printing out the value of each part.

```
# We'll use temp = 212, the boiling point of water, which was incorrect
print "212 - 32:", 212 - 32
```

```
212 - 32: 180
```

```
print "(212 - 32) * (5/9):", (212 - 32) * (5/9)
```

```
(212 - 32) * (5/9): 0
```

Aha! The problem comes when we multiply by `5/9`. This is because `5/9` is actually 0.

```
5/9
```

```
0
```

Computers store numbers in one of two ways: as *integers* or as *floating-point numbers* (or floats). The first are the numbers we usually count with; the second have fractional parts. Addition, subtraction and multiplication work on both as we'd expect, but division works differently. If we divide one integer by another, we get the quotient without the remainder:

```
print '10/3 is:', 10/3
```

```
10/3 is: 3
```

If either part of the division is a float, on the other hand, the computer creates a floating-point answer:

```
print '10.0/3 is:', 10.0/3
```

```
10.0/3 is: 3.33333333333
```

The computer does this for historical reasons: integer operations were much faster on early machines, and this behavior is actually useful in a lot of situations. It's still confusing, though, so Python 3 produces a floating-point answer when dividing integers if it needs to. We're still using Python 2.7 in this class, though, so if we want `5/9` to give us the right answer, we have to write it as `5.0/9`, `5/9.0`, or some other variation.

Let's fix our `fahr_to_kelvin` function with this new knowledge.

```
def fahr_to_kelvin(temp):
    return ((temp - 32) * (5.0/9.0)) + 273.15

print 'freezing point of water:', fahr_to_kelvin(32)
print 'boiling point of water:', fahr_to_kelvin(212)
```

```
freezing point of water: 273.15
boiling point of water: 373.15
```

It works!

Composing Functions

Now that we've seen how to turn Fahrenheit into Kelvin, it's easy to turn Kelvin into Celsius:

```
def kelvin_to_celsius(temp):
    return temp - 273.15

print 'absolute zero in Celsius:', kelvin_to_celsius(0.0)
```

```
absolute zero in Celsius: -273.15
```

What about converting Fahrenheit to Celsius? We could write out the formula, but we don't need to. Instead, we can *compose* the two functions we have already created:

```
def fahr_to_celsius(temp):
    temp_k = fahr_to_kelvin(temp)
    result = kelvin_to_celsius(temp_k)
    return result

print 'freezing point of water in Celsius:', fahr_to_celsius(32.0)
```

```
freezing point of water in Celsius: 0.0
```

This is our first taste of how larger programs are built: we define basic operations, then combine them in ever-large chunks to get the effect we want. Real-life functions will usually be larger than the ones shown here—typically half a dozen to a few dozen lines—but they shouldn't ever be much longer than that, or the next person who reads it won't be able to understand what's going on.

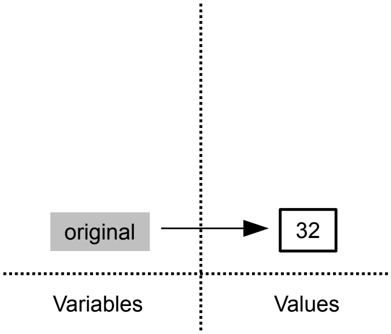


Figure 4.10: Initial Call Stack

Challenge

“Adding” two strings produces their concatention: `’a’ + ’b’` is `’ab’`. Write a function called `fence` that takes two parameters called `original` and `wrapper` and returns a new string that has the wrapper character at the beginning and end of the original:

```
print fence('name', '*')
```

```
*name*
```

Challenge

If the variable `s` refers to a string, then `s[0]` is the string’s first character and `s[-1]` is its last. Write a function called `outer` that returns a string made up of just the first and last characters of its input:

```
print outer('helium')
```

```
hm
```

The Call Stack

Let’s take a closer look at what happens when we call `fahr_to_celsius(32.0)`. To make things clearer, we’ll start by putting the initial value 32.0 in a variable and store the final result in one as well:

```
original = 32.0
final = fahr_to_celsius(original)
```

Figure 4.10 shows what memory looks like after the first line has been executed.

When we call `fahr_to_celsius`, Python *doesn’t* create the variable `temp` right away. Instead, it creates something called a *stack frame* to keep track of the variables defined by `fahr_to_kelvin`. Initially, this stack frame only holds the value of `temp` (Figure 4.11).

When we call `fahr_to_kelvin` inside `fahr_to_celsius`, Python creates another stack frame to hold `fahr_to_kelvin`’s variables (Figure 4.12).

It does this because there are now two variables in play called `temp`: the parameter to `fahr_to_celsius`, and the parameter to `fahr_to_kelvin`. Having two variables with the same name in the same part of the program would be ambiguous, so Python (and every other modern programming language)

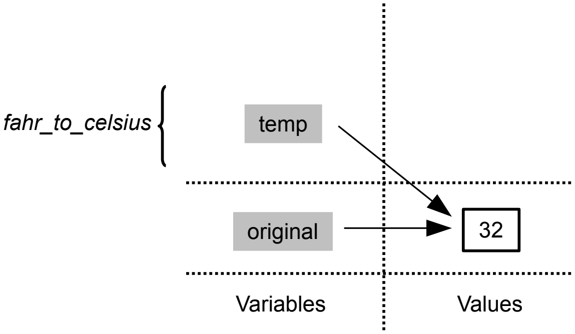


Figure 4.11: Call Stack Immediately Upon Call

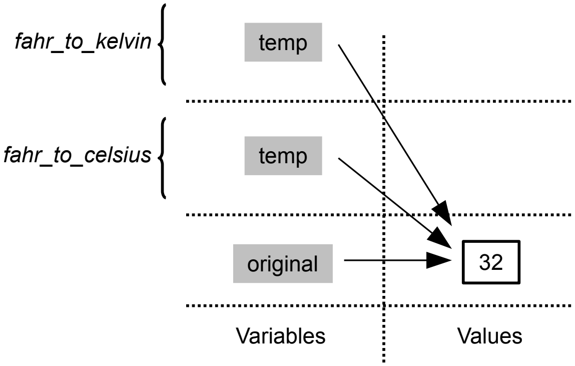


Figure 4.12: Call Stack During Nested Function Call

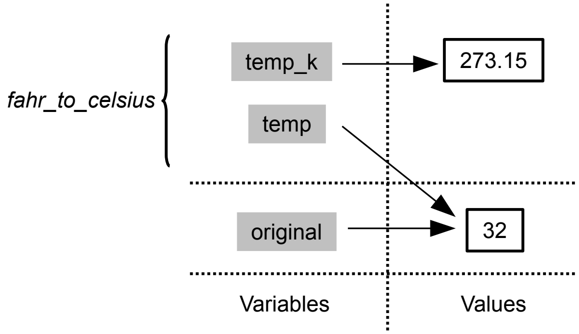


Figure 4.13: Call Stack Between Calls

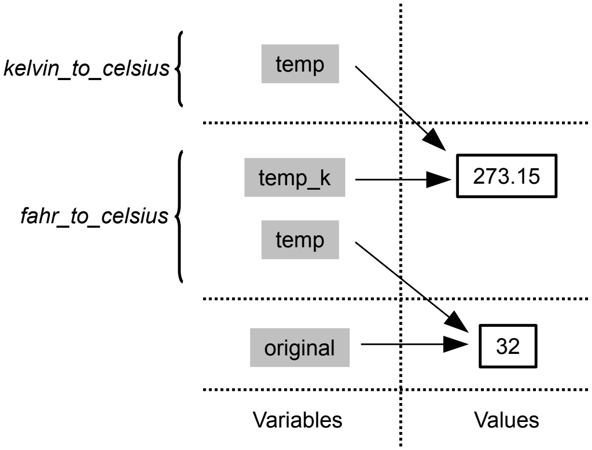


Figure 4.14: Call Stack During Second Call

creates a new stack frame for each function call to keep that function's variables separate from those defined by other functions.

When the call to `fahr_to_kelvin` returns a value, Python throws away `fahr_to_kelvin`'s stack frame and creates a new variable in the stack frame for `fahr_to_celsius` to hold the temperature in Kelvin (Figure 4.13).

It then calls `kelvin_to_celsius`, which means it creates a stack frame to hold that function's variables (Figure 4.14). Once again, Python throws away that stack frame when `kelvin_to_celsius` is done and creates the variable `result` in the stack frame for `fahr_to_celsius` (Figure 4.15).

Finally, when `fahr_to_celsius` is done, Python throws away *its* stack frame and puts its result in a new variable called `final` that lives in the stack frame we started with (Figure 4.16).

This final stack frame is always there; it holds the variables we defined outside the functions in our code. What it *doesn't* hold is the variables that were in the various stack frames. If we try to get the value of `temp` after our functions have finished running, Python tells us that there's no such thing:

```
print 'final value of temp after all function calls:', temp
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
```

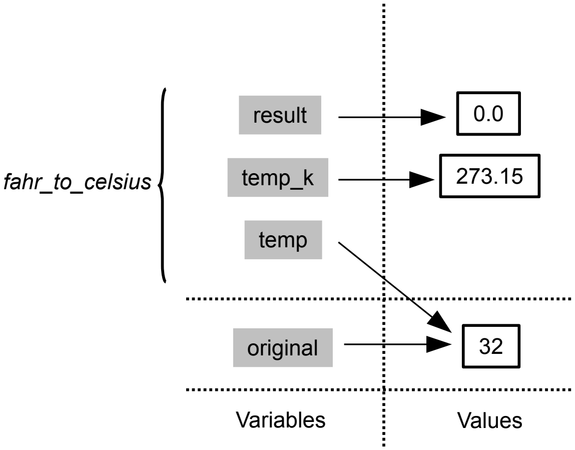


Figure 4.15: Call Stack Immediately Before Final Return

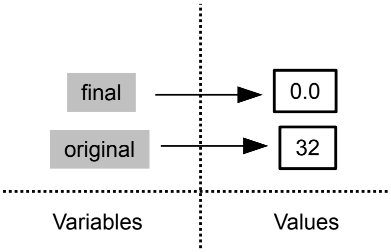


Figure 4.16: Call Stack After Calls Are Finished

```
<ipython-input-12-ffd9b4dbd5f1> in <module>()
----> 1 print 'final value of temp after all function calls:', temp

NameError: name 'temp' is not definedfinal value of temp after all function calls:
```

Why go to all this trouble? Well, here's a function called span that calculates the difference between the mininum and maximum values in an array:

```
import numpy

def span(a):
    diff = a.max() - a.min()
    return diff

data = numpy.loadtxt(fname='inflammation-01.csv', delimiter=',')
print 'span of data', span(data)

 span of data 20.0
```

Notice that span assigns a value to a variable called `diff`. We might very well use a variable with the same name to hold data:

```
diff = numpy.loadtxt(fname='inflammation-01.csv', delimiter=',')
print 'span of data:', span(diff)

span of data: 20.0
```

We don't expect `diff` to have the value 20.0 after this function call, so the name `diff` cannot refer to the same thing inside span as it does in the main body of our program. And yes, we could probably choose a different name than `diff` in our main program in this case, but we don't want to have to read every line of NumPy to see what variable names its functions use before calling any of those functions, just in case they change the values of our variables.

The big idea here is *encapsulation*, and it's the key to writing correct, comprehensible programs. A function's job is to turn several operations into one so that we can think about a single function call instead of a dozen or a hundred statements each time we want to do something. That only works if functions don't interfere with each other; if they do, we have to pay attention to the details once again, which quickly overloads our short-term memory.

Challenge

We previously wrote functions called `fence` and `outer`. Draw a diagram showing how the call stack changes when we run the following:

```
print outer(fence('carbon', `+'))
```

Testing and Documenting

Once we start putting things in functions so that we can re-use them, we need to start testing that those functions are working correctly. To see how to do this, let's write a function to center a dataset around a particular value:

```
def center(data, desired):
    return (data - data.mean()) + desired
```

We could test this on our actual data, but since we don't know what the values ought to be, it will be hard to tell if the result was correct. Instead, let's use NumPy to create a matrix of 0's and then center that around 3:

```
z = numpy.zeros((2,2))
print center(z, 3)
```

```
[[ 3.  3.]
 [ 3.  3.]]
```

That looks right, so let's try `center` on our real data:

```
data = numpy.loadtxt(fname='inflammation-01.csv', delimiter=',')
print center(data, 0)
```

```
[[-6.14875 -6.14875 -5.14875 ..., -3.14875 -6.14875 -6.14875]
 [-6.14875 -5.14875 -4.14875 ..., -5.14875 -6.14875 -5.14875]
 [-6.14875 -5.14875 -5.14875 ..., -4.14875 -5.14875 -5.14875]
 ...,
 [-6.14875 -5.14875 -5.14875 ..., -5.14875 -5.14875 -5.14875]
 [-6.14875 -6.14875 -6.14875 ..., -6.14875 -4.14875 -6.14875]
 [-6.14875 -6.14875 -5.14875 ..., -5.14875 -5.14875 -6.14875]]
```

It's hard to tell from the default output whether the result is correct, but there are a few simple tests that will reassure us:

```
print 'original min, mean, and max are:', data.min(), data.mean(), data.max()
centered = center(data, 0)
print 'min, mean, and and max of centered data are:', centered.min(), centered.mean(), centered.max()
```

```
original min, mean, and max are: 0.0 6.14875 20.0
min, mean, and and max of centered data are: -6.14875 -3.49054118942e-15 13.85125
```

That seems almost right: the original mean was about 6.1, so the lower bound from zero is how about -6.1. The mean of the centered data isn't quite zero—we'll explore why not in the challenges—but it's pretty close. We can even go further and check that the standard deviation hasn't changed:

```
print 'std dev before and after:', data.std(), centered.std()
```

```
std dev before and after: 4.61383319712 4.61383319712
```

Those values look the same, but we probably wouldn't notice if they were different in the sixth decimal place. Let's do this instead:

```
print 'difference in standard deviations before and after:', data.std() - centered.std()
```

```
difference in standard deviations before and after: -3.5527136788e-15
```

Again, the difference is very small. It's still possible that our function is wrong, but it seems unlikely enough that we should probably get back to doing our analysis. We have one more task first, though: we should write some *documentation* for our function to remind ourselves later what it's for and how to use it.

The usual way to put documentation in software is to add *comments* like this:

```
# center(data, desired): return a new array containing the original data centered around the desired value.
def center(data, desired):
    return (data - data.mean()) + desired
```

There's a better way, though. If the first thing in a function is a string that isn't assigned to a variable, that string is attached to the function as its documentation:

```
def center(data, desired):
    '''Return a new array containing the original data centered around the desired value.'''
    return (data - data.mean()) + desired
```

This is better because we can now ask Python's built-in help system to show us the documentation for the function:

```
help(center)
```

```
Help on function center in module __main__:

center(data, desired)
    Return a new array containing the original data centered around the desired value.
```

A string like this is called a *docstring*. We don't need to use triple quotes when we write one, but if we do, we can break the string across multiple lines:

```
def center(data, desired):
    '''Return a new array containing the original data centered around the desired value.
    Example: center([1, 2, 3], 0) => [-1, 0, 1]'''
    return (data - data.mean()) + desired
```

```
help(center)
```

```
Help on function center in module __main__:

center(data, desired)
    Return a new array containing the original data centered around the desired value.
    Example: center([1, 2, 3], 0) => [-1, 0, 1]
```

Challenge

Write a function called `analyze` that takes a filename as a parameter and displays the three graphs produced in the previous lesson[1], i.e., `analyze('inflammation-01.csv')` should produce the graphs already shown, while `analyze('inflammation-02.csv')` should produce corresponding graphs for the second data set. Be sure to give your function a docstring.

Challenge

Write a function `rescale` that takes an array as input and returns a corresponding array of values scaled to lie in the range 0.0 to 1.0. (If \(L\) and \(H\) are the lowest and highest values in the original array, then the replacement for a value \(v\) should be \((v-L) / (H-L)\).) Be sure to give the function a docstring.

[1] 01-numpy.ipynb

Challenge

Run the commands `help(numpy.arange)` and `help(numpy.linspace)` to see how to use these functions to generate regularly-spaced values, then use those values to test your `rescale` function.

Defining Defaults

We have passed parameters to functions in two ways: directly, as in `span(data)`, and by name, as in `numpy.loadtxt(fname=’something.csv’, delimiter=’,’)`. In fact, we can pass the filename to `loadtxt` without the `fname=`:

```
numpy.loadtxt('inflammation-01.csv', delimiter=',')
```

```
array([[ 0.,  0.,  1., ...,  3.,  0.,  0.],
       [ 0.,  1.,  2., ...,  1.,  0.,  1.],
       [ 0.,  1.,  1., ...,  2.,  1.,  1.],
       ...,
       [ 0.,  1.,  1., ...,  1.,  1.,  1.],
       [ 0.,  0.,  0., ...,  0.,  2.,  0.],
       [ 0.,  0.,  1., ...,  1.,  1.,  0.]])
```

but we still need to say `delimiter=`:

```
numpy.loadtxt('inflammation-01.csv', ',')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-26-e3bc6cf4fd6a> in <module>()
----> 1 numpy.loadtxt('inflammation-01.csv', ',')

/Users/gwilson/anaconda/lib/python2.7/site-packages/numpy/lib/npyio.pyc in loadtxt(fname, dtype, comments, delimiter,
    775     try:
    776         # Make sure we're dealing with a proper dtype
--> 777         dtype = np.dtype(dtype)
    778         defconv = _getconv(dtype)
    779

TypeError: data type ”,” not understood
```

To understand what’s going on, and make our own functions easier to use, let’s re-define our `center` function like this:

```
def center(data, desired=0.0):
    '''Return a new array containing the original data centered around the desired value (0 by default).
    Example: center([1, 2, 3], 0) => [-1, 0, 1]'''
    return (data - data.mean()) + desired
```

The key change is that the second parameter is now written `desired=0.0` instead of just `desired`. If we call the function with two arguments, it works as it did before:

```
test_data = numpy.zeros((2, 2))
print center(test_data, 3)
```

```
[[ 3.  3.]
 [ 3.  3.]]
```

But we can also now call it with just one parameter, in which case `desired` is automatically assigned the *default value* of 0.0:

```
more_data = 5 + numpy.zeros((2, 2))
print 'data before centering:', more_data
print 'centered data:', center(more_data)
```

```
data before centering: [[ 5.  5.]
 [ 5.  5.]]
centered data: [[ 0.  0.]
 [ 0.  0.]]
```

This is handy: if we usually want a function to work one way, but occasionally need it to do something else, we can allow people to pass a parameter when they need to but provide a default to make the normal case easier. The example below shows how Python matches values to parameters:

```
def display(a=1, b=2, c=3):
    print 'a:', a, 'b:', b, 'c:', c

print 'no parameters:'
display()
print 'one parameter:'
display(55)
print 'two parameters:'
display(55, 66)
```

```
no parameters:
a: 1 b: 2 c: 3
one parameter:
a: 55 b: 2 c: 3
two parameters:
a: 55 b: 66 c: 3
```

As this example shows, parameters are matched up from left to right, and any that haven't been given a value explicitly get their default value. We can override this behavior by naming the value as we pass it in:

```
print 'only setting the value of c'
display(c=77)
```

```
only setting the value of c
a: 1 b: 2 c: 77
```

With that in hand, let's look at the help for `numpy.loadtxt`:

```
help(numpy.loadtxt)
```

```
Help on function loadtxt in module numpy.lib.npyio:

loadtxt(fname, dtype=<type 'float'>, comments='#', delimiter=None, converters=None, skiprows=0, usecols=None, unpac
    Load data from a text file.

    Each row in the text file must have the same number of values.

    Parameters
    ----------
    fname : file or str
    File, filename, or generator to read.  If the filename extension is
    ``.gz`` or ``.bz2``, the file is first decompressed. Note that
    generators should return byte strings for Python 3k.
dtype : data-type, optional
    Data-type of the resulting array; default: float.  If this is a
    record data-type, the resulting array will be 1-dimensional, and
    each row will be interpreted as an element of the array.  In this
    case, the number of columns used must match the number of fields in
    the data-type.
comments : str, optional
    The character used to indicate the start of a comment;
    default: '#'.
delimiter : str, optional
    The string used to separate values.  By default, this is any
    whitespace.
converters : dict, optional
    A dictionary mapping column number to a function that will convert
    that column to a float.  E.g., if column 0 is a date string:
    ``converters = {0: datestr2num}``.  Converters can also be used to
    provide a default value for missing data (but see also `genfromtxt`):
    ``converters = {3: lambda s: float(s.strip() or 0)}``.  Default: None.
skiprows : int, optional
    Skip the first `skiprows` lines; default: 0.
usecols : sequence, optional
    Which columns to read, with 0 being the first.  For example,
    ``usecols = (1,4,5)`` will extract the 2nd, 5th and 6th columns.
    The default, None, results in all columns being read.
unpack : bool, optional
    If True, the returned array is transposed, so that arguments may be
    unpacked using ``x, y, z = loadtxt(...)``.  When used with a record
    data-type, arrays are returned for each field.  Default is False.
ndmin : int, optional
    The returned array will have at least `ndmin` dimensions.
    Otherwise mono-dimensional axes will be squeezed.
    Legal values: 0 (default), 1 or 2.
    .. versionadded:: 1.6.0

Returns
-------
out : ndarray
    Data read from the text file.

See Also
--------
load, fromstring, fromregex
genfromtxt : Load data with missing values handled as specified.
scipy.io.loadmat : reads MATLAB data files

Notes
-----
This function aims to be a fast reader for simply formatted files.  The
`genfromtxt` function provides more sophisticated handling of, e.g.,
lines with missing values.

Examples
--------
>>> from StringIO import StringIO   # StringIO behaves like a file object
>>> c = StringIO(”0 1\n2 3”)
>>> np.loadtxt(c)
```

```
array([[ 0.,  1.],
       [ 2.,  3.]])

>>> d = StringIO(”M 21 72\nF 35 58”)
>>> np.loadtxt(d, dtype={'names': ('gender', 'age', 'weight'),
...                      'formats': ('S1', 'i4', 'f4')})
array([('M', 21, 72.0), ('F', 35, 58.0)],
      dtype=[('gender', '|S1'), ('age', '<i4'), ('weight', '<f4')])

>>> c = StringIO(”1,0,2\n3,0,4”)
>>> x, y = np.loadtxt(c, delimiter=',', usecols=(0, 2), unpack=True)
>>> x
array([ 1.,  3.])
>>> y
array([ 2.,  4.])
```

There’s a lot of information here, but the most important part is the first couple of lines:

```
loadtxt(fname, dtype=<type 'float'>, comments='#', delimiter=None, converters=None, skiprows=0, usecols=None,
        unpack=False, ndmin=0)
```

This tells us that `loadtxt` has one parameter called `fname` that doesn’t have a default value, and eight others that do. If we call the function like this:

```
numpy.loadtxt('inflammation-01.csv', ',')
```

then the filename is assigned to `fname` (which is what we want), but the delimiter string `','` is assigned to `dtype` rather than `delimiter`, because `dtype` is the second parameter in the list. That’s why we don’t have to provide `fname=` for the filename, but *do* have to provide `delimiter=` for the second parameter.

Challenge

Rewrite the `rescale` function so that it scales data to lie between 0.0 and 1.0 by default, but will allow the caller to specify lower and upper bounds if they want. Compare your implementation to your neighbor’s: do the two functions always behave the same way?

Key Points

- Define a function using `def name(...params...)`.
- The body of a function must be indented.
- Call a function using `name(...values...)`.
- Numbers are stored as integers or floating-point numbers.
- Integer division produces the whole part of the answer (not the fractional part).
- Each time a function is called, a new stack frame is created on the *call stack* to hold its parameters and local variables.
- Python looks for variables in the current stack frame before looking for them at the top level.
- Use `help(thing)` to view help for something.
- Put docstrings in functions to provide help for that function.
- Specify default values for parameters when defining a function using `name=value` in the parameter list.
- Parameters can be passed by matching based on name, by position, or by omitting them (in which case the default value is used).

4.3 Analyzing Multiple Data Sets

We have created a function called `analyze` that creates graphs of the minimum, average, and maximum daily inflammation rates for a single data set:

```
import numpy as np
from matplotlib import pyplot as plt

def analyze(filename):
    data = np.loadtxt(fname=filename, delimiter=',')

    plt.figure(figsize=(10.0, 3.0))

    plt.subplot(1, 3, 1)
    plt.ylabel('average')
    plt.plot(data.mean(0))

    plt.subplot(1, 3, 2)
    plt.ylabel('max')
    plt.plot(data.max(0))

    plt.subplot(1, 3, 3)
    plt.ylabel('min')
    plt.plot(data.min(0))

    plt.tight_layout()
    plt.show()
```

We can use it to analyze other data sets one by one, but we have a dozen data sets right now and more on the way. We want to create plots for all our data sets with a single statement. To do that, we'll have to teach the computer how to repeat things.

Objectives

- Explain what a for loop does.
- Correctly write for loops to repeat simple calculations.
- Trace changes to a loop variable as the loop runs.
- Trace changes to other variables as they are updated by a for loop.
- Explain what a list is.
- Create and index lists of simple values.
- Use a library function to get a list of filenames that match a simple wildcard pattern.
- Use a for loop to process multiple files.

For Loops

Suppose we want to print each character in the word "lead" on a line of its own. One way is to use four `print` statements:

```
def print_characters(element):
    print element[0]
    print element[1]
    print element[2]
    print element[3]

print_characters('lead')
```

```
l
e
a
d
```

but that's a bad approach for two reasons:

1. It doesn't scale: if we want to print the characters in a string that's hundreds of letters long, we'd be better off just typing them in.
2. It's fragile: if we give it a longer string, it only prints part of the data, and if we give it a shorter one, it produces an error because we're asking for characters that don't exist.

```
print_characters('tin')
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-13-5bc7311e0bf3> in <module>()
----> 1 print_characters('tin')

<ipython-input-12-11460561ea56> in print_characters(element)
      3     print element[1]
      4     print element[2]
----> 5     print element[3]
      6 
      7 print_characters('lead')

IndexError: string index out of ranget
i
n
```

Here's a better approach:

```
def print_characters(element):
    for char in element:
        print char

print_characters('lead')
```

This is shorter—certainly shorter than something that prints every character in a hundred-letter string—and more robust as well:

```
print_characters('oxygen')
```

The improved version of `print_characters` uses a *for loop* to repeat an operation—in this case, printing—once for each thing in a collection. The general form of a loop is:

```
for variable in collection:
    do things with variable
```

We can call the *loop variable* anything we like, but there must be a colon at the end of the line starting the loop, and we must indent the body of the loop.

Here's another loop that repeatedly updates a variable:

```
length = 0
for vowel in 'aeiou':
    length = length + 1
print 'There are', length, 'vowels'
```

It’s worth tracing the execution of this little program step by step. Since there are five characters in `'aeiou'`, the statement on line 3 will be executed five times. The first time around, `length` is zero (the value assigned to it on line 1) and `vowel` is `'a'`. The statement adds 1 to the old value of `length`, producing 1, and updates `length` to refer to that new value. The next time around, `vowel` is `'e'` and `length` is 1, so `length` is updated to be 2. After three more updates, `length` is 5; since there is nothing left in `'aeiou'` for Python to process, the loop finishes and the `print` statement on line 4 tells us our final answer.

Note that a loop variable is just a variable that’s being used to record progress in a loop. It still exists after the loop is over, and we can re-use variables previously defined as loop variables as well:

```
letter = 'z'
for letter in 'abc':
    print letter
print 'after the loop, letter is', letter
```

Note also that finding the length of a string is such a common operation that Python actually has a built-in function to do it called `len`:

```
print len('aeiou')
```

`len` is much faster than any function we could write ourselves, and much easier to read than a two-line loop; it will also give us the length of many other things that we haven’t met yet, so we should always use it when we can.

Challenge

Python has a built-in function called `range` that creates a list of numbers: `range(3)` produces `[0, 1, 2]`, `range(2, 5)` produces `[2, 3, 4]`, and `range(2, 10, 3)` produces `[2, 5, 8]`. Using `range`, write a function that prints the \(N\) natural numbers:

```
print_N(3)
```

```
1 2 3
```

Challenge

Exponentiation is built into Python:

```
print 2**4
```

```
16
```

It also has a function called `pow` that calculates the same value. Write a function called `expo` that uses a loop to calculate the same result.

Challenge

Python’s strings have methods, just like NumPy’s arrays. One of these is called `reverse`:

```
  print `Newton'.reverse()
```

```
notweN
```

Write a function called `rev` that does the same thing:

```
print rev('Newton')
```

```
notweN
```

As always, be sure to include a docstring.

Lists

Just as a `for` loop is a way to do operations many times, a list is a way to store many values. Unlike NumPy arrays, there are built into the language. We create a list by putting values inside square brackets:

```
odds = [1, 3, 5, 7]
print 'odds are:', odds
```

We select individual elements from lists by indexing them:

```
print 'first and last:', odds[0], odds[-1]
```

and if we loop over a list, the loop variable is assigned elements one at a time:

```
for number in odds:
    print number
```

There is one important difference between lists and strings: we can change the values in a list, but we cannot change the characters in a string. For example:

```
names = ['Newton', 'Darwing', 'Turing'] # typo in Darwin's name
print 'names is originally:', names
names[1] = 'Darwin' # correct the name
print 'final value of names:', names
```

works, but:

```
name = 'Bell'
name[0] = 'b'
```

does not.

Ch-Ch-Ch-Changes

Data that can be changed is called *mutable*, while data that cannot be is called *immutable*. Like strings, numbers are immutable: there's no way to make the number 0 have the value 1 or vice versa (at least, not in Python—there actually *are* languages that will let people do this, with predictably confusing results). Lists and arrays, on the other hand, are mutable: both can be modified after they have been created.

Programs that modify data in place can be harder to understand than ones that don't because readers may have to mentally sum up many lines of code in order to figure out what the value of something actually is. On the other hand, programs that modify data in place instead of creating copies that are almost identical to the original every time they want to make a small change are much more efficient.

There are many ways to change the contents of in lists besides assigning to elements:

```
odds.append(11)
print 'odds after adding a value:', odds

del odds[0]
print 'odds after removing the first element:', odds

odds.reverse()
print 'odds after reversing:', odds
```

Challenge

Write a function called `total` that calculates the sum of the values in a list. (Python has a built-in function called `sum` that does this for you. Please don't use it for this exercise.)

Processing Multiple Files

We now have almost everything we need to process all our data files. The only thing that's missing is a library with a rather unpleasant name:

```
import glob
```

The `glob` library contains a single function, also called `glob`, that finds files whose names match a pattern. We provide those patterns as strings: the character `*` matches zero or more characters, while `?` matches any one character. We can use this to get the names of all the IPython Notebooks we have created so far:

```
print glob.glob('*.ipynb')
```

```
['01-numpy.ipynb', '02-func.ipynb', '03-loop.ipynb', '04-cond.ipynb', '05-defensive.ipynb', '06-cmdline.ipynb', 'spati
```

or to get the names of all our CSV data files:

```
print glob.glob('*.csv')
```

```
['inflammation-01.csv', 'inflammation-02.csv', 'inflammation-03.csv', 'inflammation-04.csv', 'inflammation-05.csv', 'i
```

As these examples show, `glob.glob`'s result is a list of strings, which means we can loop over it to do something with each filename in turn. In our case, the "something" we want is our `analyze` function. Let's test it by analyzing the first three files in the list (Figure 4.17):

```
filenames = glob.glob('*.csv')
filenames = filenames[0:3]
for f in filenames:
    print f
    analyze(f)
```

Sure enough, the maxima of these data sets show exactly the same ramp as the first, and their minima show the same staircase structure.

Challenge

Write a function called `analyze_all` that takes a filename pattern as its sole argument and runs `analyze` for each file whose name matches the pattern.

Key Points

- Use `for variable in collection` to process the elements of a collection one at a time.
- The body of a for loop must be indented.
- Use `len(thing)` to determine the length of something that contains other values.
- `[value1, value2, value3, ...]` creates a list.
- Lists are indexed and sliced in the same way as strings and arrays.
- Lists are mutable (i.e., their values can be changed in place).
- Strings are immutable (i.e., the characters in them cannot be changed).
- Use `glob.glob(pattern)` to create a list of files whose names match a pattern.
- Use `*` in a pattern to match zero or more characters, and `?` to match any single character.

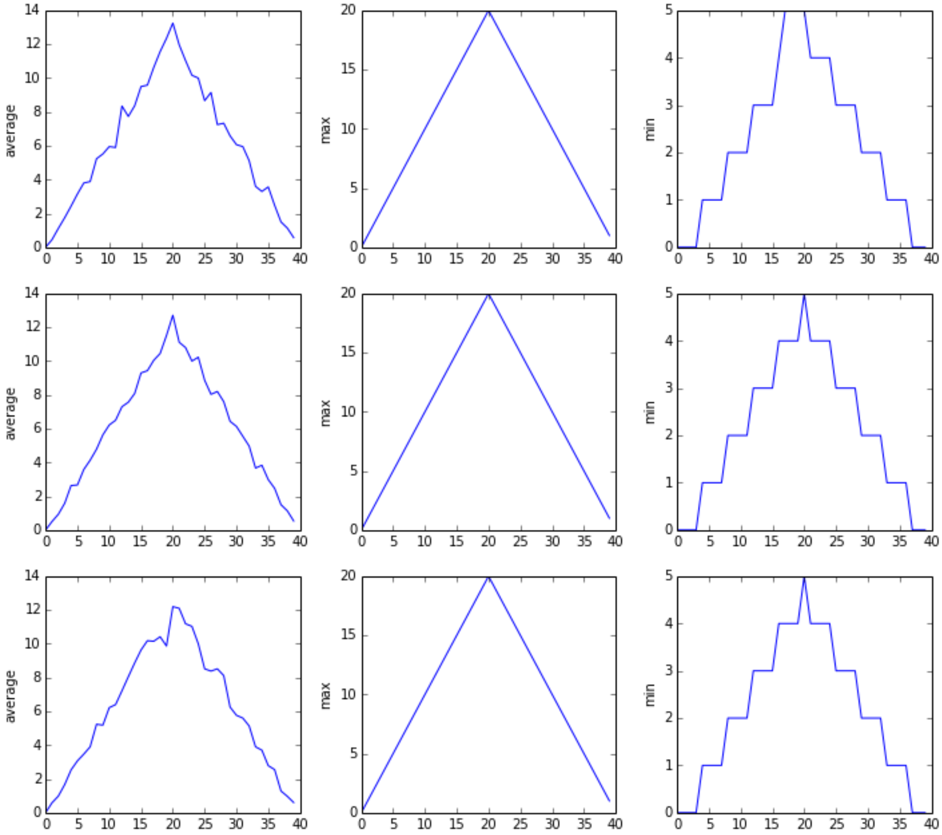


Figure 4.17: Inflammation Statistics for First Three Data Sets

4.4 Making Choices

Our previous lessons have shown us how to manipulate data, define our own functions, and repeat things. However, the programs we have written so far always do the same things, regardless of what data they're given. We want programs to make choices based on the values they are manipulating. To help us see what decisions they're making, we'll start by looking at how computers manipulate images.

Objectives

- Create a simple "image" made out of colored blocks.
- Explain how the RGB model represents colors.
- Explain the similarities and differences between tuples and lists.
- Write conditional statements including `if`, `elif`, and `else` branches.
- Correctly evaluate expressions containing `and` and `or`.
- Correctly write and interpret code containing nested loops and conditionals.
- Explain the advantages of putting frequently-modified code in a function.

Image Grids

Let's start by creating some simple heat maps of our own using a library called `ipythonblocks`. The first step is to create our own "image":

```
from ipythonblocks import ImageGrid
```

Unlike the `import` statements we have seen earlier, this one doesn't load the entire `ipythonblocks` library. Instead, it just loads `ImageGrid` from that library, since that's the only thing we need (for now).

Once we have `ImageGrid` loaded, we can use it to create a very simple grid of colored cells:

```
grid = ImageGrid(5, 3)
grid.show()
```

```
table.blockgrid {border: none;} .blockgrid tr {border: none;} .blockgrid td {padding: 0px;} #bl
```

Just like a NumPy array, an `ImageGrid` has some properties that hold information about it:

```
print 'grid width:', grid.width
print 'grid height:', grid.height
print 'grid lines on:', grid.lines_on
```

```
grid width: 5
grid height: 3
grid lines on: True
```

The obvious thing to do with a grid like this is color in its cells, but in order to do that, we need to know how computers represent color. The most common schemes are *RGB*, which is short for "red, green, blue". RGB is an *additive color model*: every shade is some combination of red, green, and blue intensities. We can think of these three values as being the axes in a cube (Figure 4.18).

An RGB color is an example of a multi-part value: like a Cartesian coordinate, it is one thing with several parts. We can represent such a value in Python using a *tuple*, which we write using parentheses instead of the square brackets used for a list:

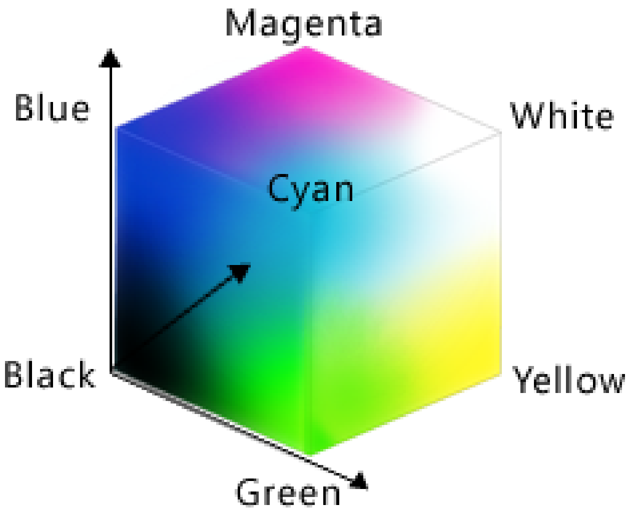


Figure 4.18: The RGB Color Cube

```
position = (12.3, 45.6)
print 'position is:', position
color = (10, 20, 30)
print 'color is:', color

position is: (12.3, 45.6)
color is: (10, 20, 30)
```

We can select elements from tuples using indexing, just as we do with lists and arrays:

```
print 'first element of color is:', color[0]

first element of color is: 10
```

Unlike lists and arrays, though, tuples cannot be changed after they are created—in technical terms, they are *immutable*:

```
color[0] = 40
print 'first element of color after change:', color[0]

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-11-9c3dd30a4e52> in <module>()
----> 1 color[0] = 40
      2 print 'first element of color after change:', color[0]

TypeError: 'tuple' object does not support item assignment
```

If a tuple represents an RGB color, its red, green, and blue components can take on values between 0 and 255. The upper bound may seem odd, but it's the largest number that can be represented in an

8-bit byte (i.e., 2^8-1). This makes it easy for computers to manipulate colors, while providing fine enough gradations to fool most human eyes, most of the time.

Let’s see what a few RGB colors actually look like:

```
row = ImageGrid(8, 1)
row[0, 0] = (0, 0, 0)   # no color => black
row[1, 0] = (255, 255, 255) # all colors => white
row[2, 0] = (255, 0, 0) # all red
row[3, 0] = (0, 255, 0) # all green
row[4, 0] = (0, 0, 255) # all blue
row[5, 0] = (255, 255, 0) # red and green
row[6, 0] = (255, 0, 255) # red and blue
row[7, 0] = (0, 255, 255) # green and blue
row.show()
```

```
table.blockgrid {border: none;} .blockgrid tr {border: none;} .blockgrid td {padding: 0px;} #bl
```

Simple color values like `(0,255,0)` are easy enough to decipher with a bit of practice, but what color is `(214,90,127)`? To help us, `ipythonblocks` provides a function called `show_color`:

```
from ipythonblocks import show_color
show_color(214, 90, 127)
```

It also provides a table of standard colors:

```
from ipythonblocks import colors
c = ImageGrid(3, 2)
c[0, 0] = colors['Fuchsia']
c[0, 1] = colors['Salmon']
c[1, 0] = colors['Orchid']
c[1, 1] = colors['Lavender']
c[2, 0] = colors['LimeGreen']
c[2, 1] = colors['HotPink']
c.show()
```

```
table.blockgrid {border: none;} .blockgrid tr {border: none;} .blockgrid td {padding: 0px;} #bl
```

Challenge

Fill in the `____` in the code below to create a bar that changes color from dark blue to black.

```
bar = ImageGrid(10, 1)
for x in range(10):
    bar[x, 0] = (0, 0, ____)
bar.show()
```

Challenge

Why do computers use red, green, and blue as their primary colors?

Conditionals

The other thing we need in order to create a heat map of our own is a way to pick a color based on a data value. The tool Python gives us for doing this is called a *conditional statement*, and looks like this:

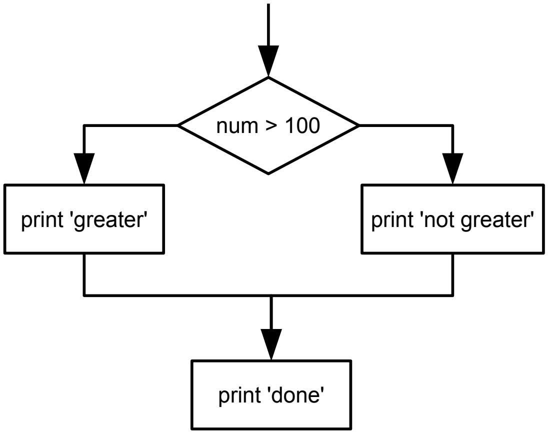


Figure 4.19: Flow of Control in a Conditional

```
num = 37
if num > 100:
    print 'greater'
else:
    print 'not greater'
print 'done'
```

```
not greater
done
```

The second line of this code uses the keyword `if` to tell Python that we want to make a choice. If the test that follows it is true, the body of the `if` (i.e., the lines indented underneath it) are executed. If the test is false, the body of the `else` is executed instead, so only one or the other is ever executed (Figure 4.19).

Conditional statements don't have to include an `else`. If there isn't one, Python simply does nothing if the test is false:

```
num = 53
print 'before conditional...'
if num > 100:
    print '53 is greater than 100'
print '...after conditional'
```

```
before conditional...
...after conditional
```

We can also chain several tests together using `elif`, which is short for "else if". This makes it simple to write a function that returns the sign of a number:

```
def sign(num):
    if num > 0:
        return 1
    elif num == 0:
        return 0
    else:
        return -1

print 'sign of -3:', sign(-3)
```

```
sign of -3: -1
```

One important thing to notice the code above is that we use a double equals sign `==` to test for equality rather than a single equals sign because the latter is used to mean assignment. This convention was inherited from C, and while many other programming languages work the same way, it does take a bit of getting used to…

We can also combine tests using `and` and `or`. `and` is only true if both parts are true:

```
if (1 > 0) and (-1 > 0):
    print 'both parts are true'
else:
    print 'one part is not true'
```

```
one part is not true
```

while `or` is true if either part is true:

```
if (1 < 0) or ('left' < 'right'):
    print 'at least one test is true'
```

```
at least one test is true
```

In this case, "either" means "either or both", not "either one or the other but not both".

Challenge

`True` and `False` aren't the only values in Python that are true and false. In fact, *any* value can be used in an `if` or `elif`. After reading and running the code below, explain what the rule is for which values are considered true and which are considered false. (Note that if the body of a conditional is a single statement, we can write it on the same line as the `if`.)

```
if '': print 'empty string is true'
if 'word': print 'word is true'
if []: print 'empty list is true'
if [1, 2, 3]: print 'non-empty list is true'
if 0: print 'zero is true'
if 1: print 'one is true'
```

Challenge

Write a function called `near` that returns `True` if its first parameter is within 10% of its second and `False` otherwise. Compare your implementation with your partner's: do you return the same answer for all possible pairs of numbers?

Nesting

Another thing to realize is that `if` statements can be combined with loops just as easily as they can be combined with functions. For example, if we want to sum the positive numbers in a list, we can write this:

```
numbers = [-5, 3, 2, -1, 9, 6]
total = 0
for n in numbers:
    if n >= 0:
        total = total + n
print 'sum of positive values:', total
```

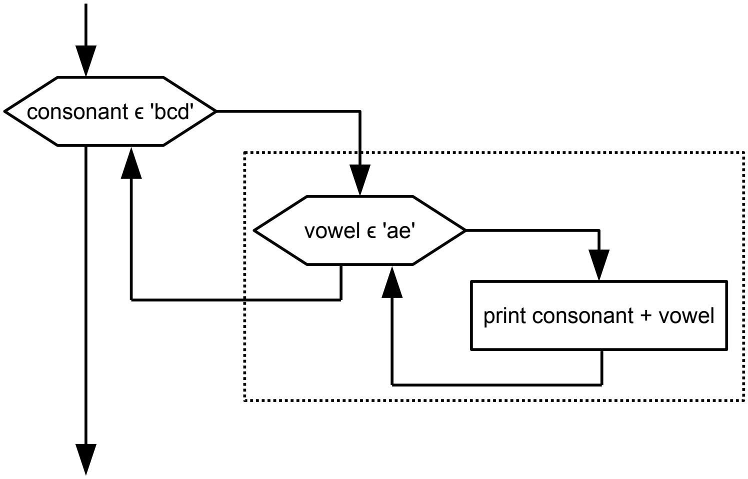


Figure 4.20: Execution of Nested Loops

```
sum of positive values: 20
```

We could equally well calculate the positive and negative sums in a single loop:

```
pos_total = 0
neg_total = 0
for n in numbers:
    if n >= 0:
        pos_total = pos_total + n
    else:
        neg_total = neg_total + n
print 'negative and positive sums are:', neg_total, pos_total

negative and positive sums are: -6 20
```

We can even put one loop inside another:

```
for consonant in 'bcd':
    for vowel in 'ae':
        print consonant + vowel

ba
be
ca
ce
da
de
```

As Figure 4.20 shows, the *inner loop* runs from start to finish each time the *outer loop* runs once. We can combine nesting and conditionals to create patterns in an image:

```
square = ImageGrid(5, 5)
for x in range(square.width):
    for y in range(square.height):
        if x < y:
            square[x, y] = colors['Fuchsia']
        elif x == y:
            square[x, y] = colors['Olive']
        else:
            square[x, y] = colors['SlateGray']
square.show()
```

```
table.blockgrid {border: none;} .blockgrid tr {border: none;} .blockgrid td {padding: 0px;} #bl
```

This is our first hand-made data visualization: the colors show where x is less than, equal to, or greater than y.

Challenge

Will changing the nesting of the loops in the code above—i.e., wrapping the Y-axis loop around the X-axis loop—change the final image? Why or why not?

Challenge

Python (and most other languages in the C family) provides *in-place operators* that work like this:

```
x = 1  # original value
x += 1 # add one to x, assigning result back to x
x *= 3 # multiply x by 3
print x
6
```

Rewrite the code that sums the positive and negative numbers in a list using in-place operators. Do you think the result is more or less readable than the original?

Creating a Heat Map

The last step is to turn our data into something we can see. As in previous lessons, the first step is to get the data into memory:

```
import numpy as np
data = np.loadtxt(fname='inflammation-01.csv', delimiter=',')
print 'data shape:', data.shape

data shape: (60, 40)
```

The second is to create an image grid that is the same size as the data:

```
width, height = data.shape
heatmap = ImageGrid(width, height)
```

(The first line of the code above takes advantage of a neat trick: we can unpack the values in a tuple by assigning it to as many variables as it has entries.)

The third step is to decide *how* we are going to color the cells in the heat map. To keep things simple, we will use red, green, and blue as our colors, and compare data values to the data set's mean. Here's the code:

```
for x in range(width):
    for y in range(height):
        if data[x, y] < data.mean():
            heatmap[x, y] = colors['Red']
        elif data[x, y] == data.mean():
            heatmap[x, y] = colors['Green']
        else:
            heatmap[x, y] = colors['Blue']
heatmap.show()
```

```
table.blockgrid {border: none;} .blockgrid tr {border: none;} .blockgrid td {padding: 0px;}
```

This may be what we asked for, but both the image and the code are hideous:

1. It’s too large for us to view the whole thing at once on a small laptop screen.
2. Our first heatmap had time along the X axis; this seems to have time along the Y axis.
3. Red against blue is pretty hard on the eyes.
4. The heatmap only shows two colors because none of the (integer) measurements has exactly the same value as the (fractional) mean.
5. We are calculating the mean of `data` either once or twice each time we go through the loop. That means that on a 40ÃŮ60 data set, we are performing the same calculation 2400 times.

Here’s how we can improve it:

1. We can give `ImageGrid` an optional parameter `block_size` to set the size of each block.
2. We can transpose our data before creating the grid.
3. We can pick better colors (I’m personally fond of orchid, fuchsia, and hot pink).
4. Instead of checking if values are exactly equal to the mean, we can see if they are close to it.
5. We can calculate the mean once, before we start our loops, and use that value over and over.

Our modified code looks like this:

```
flipped = data.transpose()
width, height = flipped.shape
heatmap = ImageGrid(width, height, block_size=5)
center = flipped.mean()
for x in range(width):
    for y in range(height):
        if flipped[x, y] < (0.8 * center):
            heatmap[x, y] = colors['Orchid']
        elif flipped[x, y] > (1.2 * center):
            heatmap[x, y] = colors['HotPink']
        else:
            heatmap[x, y] = colors['Fuchsia']
heatmap.show()
```

```
table.blockgrid {border: none;} .blockgrid tr {border: none;} .blockgrid td {padding: 0px;}
```

That’s a bit better—but now the contrast between the colors isn’t great enough. And there still aren’t very many fuchsia cells: we may want to widen the band around the mean that gets that color.

We could rewrite our loop a third time, but the right thing to do is to put our code in a function so that we can experiment with bands and colors more easily.

```
def make_heatmap(values, low_color, mid_color, high_color, low_band, high_band, block_size):
    '''Make a 3-colored heatmap from a 2D array of data.'''
    width, height = values.shape
    result = ImageGrid(width, height, block_size=block_size)
    center = values.mean()
    for x in range(width):
        for y in range(height):
            if values[x, y] < low_band * center:
                result[x, y] = low_color
            elif values[x, y] > high_band * center:
                result[x, y] = high_color
            else:
                result[x, y] = mid_color
    return result
```

To test this function, we'll run it with the settings we just used:

```
h = make_heatmap(flipped, colors['Orchid'], colors['Fuchsia'], colors['HotPink'], 0.8, 1.2, 5)
h.show()
```

```
table.blockgrid {border: none;} .blockgrid tr {border: none;} .blockgrid td {padding: 0px;} #bl
```

That seems right, so let's widen the band and use more dramatic colors:

```
h = make_heatmap(flipped, colors['Gray'], colors['YellowGreen'], colors['SpringGreen'], 0.5, 1.5, 5)
h.show()
```

```
table.blockgrid {border: none;} .blockgrid tr {border: none;} .blockgrid td {padding: 0px;} #bl
```

We'll probably want to experiment a bit more before publishing, but writing a function has made experimenting easy. We can make it even easier by re-defining our function one more time to give the parameters default values. While we're at it, let's put the low and high bands at the front, since they're more likely to change than our color choices:

```
def make_heatmap(values,
                 low_band=0.5, high_band=1.5,
                 low_color=colors['Gray'], mid_color=colors['YellowGreen'], high_color=colors['SpringGreen'],
                 block_size=5):
    '''Make a 3-colored heatmap from a 2D array of data.
    Default color scheme is gray to green.'''
    width, height = values.shape
    result = ImageGrid(width, height, block_size=block_size)
    center = values.mean()
    for x in range(width):
        for y in range(height):
            if values[x, y] < low_band * center:
                result[x, y] = low_color
            elif values[x, y] > high_band * center:
                result[x, y] = high_color
            else:
                result[x, y] = mid_color
    return result
```

Once default values are added, the function's first line is too long to fit comfortably on our screen. Rather than breaking it wherever it hits the right edge of the screen, we have divided the parameters into logical groups to make it more readable.

Again, our first test is to re-run it with the same values as before (which we give it in a different order, since we've changed the order of parameters):

```
h = make_heatmap(flipped, 0.5, 1.5, colors['Gray'], colors['YellowGreen'], colors['SpringGreen'], 5)
h.show()
```

```
table.blockgrid {border: none;} .blockgrid tr {border: none;} .blockgrid td {padding: 0px;}
```

We can now leave out everything except the data being visualized, or provide the data and the bands and re-use the default colors and block size:

```
h = make_heatmap(flipped, 0.4, 1.6)
h.show()
```

```
table.blockgrid {border: none;} .blockgrid tr {border: none;} .blockgrid td {padding: 0px;}
```

We can now explore our data with just a few keystrokes, which means we can concentrate on our science and not on our programming.

Challenge

Why did we transpose our data outside our heat map function? Why not have the function perform the transpose?

Challenge

Why does the heat map function return the grid rather than displaying it immediately? Do you think this is a good or bad design choice?

Challenge

Explain what the overall effect of this code is:

```
temp = left
left = right
right = temp
```

Compare it to:

```
left, right = right, left
```

Do they always do the same thing? Which do you find easier to read?

Key Points

- Use the `ImageGrid` class from the `ipythonblocks` library to create simple “images” made of colored blocks.
- Specify colors use (red, green, blue) triples, each component of which is an integer in the range 0..255.
- Use `if condition` to start a conditional statement, `elif condition` to provide additional tests, and `else` to provide a default.
- The bodies of the branches of conditional statements must be indented.
- Use == to test for equality.
- `X and Y` is only true if both X and Y are true.
- `X or Y` is true if either X or Y, or both, are true.

- Zero, the empty string, and the empty list are considered false; all other numbers, strings, and lists are considered true.
- Nest loops to operate on multi-dimensional data.
- Put code whose parameters change frequently in a function, then call it with different parameter values to customize its behavior.

4.5 Defensive Programming

Our previous lessons have introduced the basic tools of programming: variables and lists, file I/O, loops, conditionals, and functions. What they *haven't* done is show us how to tell whether a program is getting the right answer, and how to tell if it's *still* getting the right answer as we make changes to it.

To achieve that, we need to:

- write programs that check their own operation,
- write and run tests for widely-used functions, and
- make sure we know what "correct" actually means.

The good news is, doing these things will speed up our programming, not slow it down. As in real carpentry—the kind done with lumber—the time saved by measuring carefully before cutting a piece of wood is much greater than the time that measuring takes.

Objectives

- Explain what an assertion is.
- Add assertions to programs that correctly check the program's state.
- Correctly add precondition and postcondition assertions to functions.
- Explain what test-driven development is, and use it when creating new functions.
- Explain why variables should be initialized using actual data values rather than arbitrary constants.
- Debug code containing an error systematically.

Assertions

The first step toward getting the right answers from our programs is to assume that mistakes *will* happen and to guard against them. This is called *defensive programming*, and the most common way to do it is to add *assertions* to our code so that it checks itself as it runs. An assertion is simply a statement that something must be true at a certain point in a program. When Python sees one, it checks that the assertion's condition. If it's true, Python does nothing, but if it's false, Python halts the program immediately and prints the error message provided. For example, this piece of code halts as soon as the loop encounters a value that isn't positive:

```
numbers = [1.5, 2.3, 0.7, -0.001, 4.4]
total = 0.0
for n in numbers:
    assert n >= 0.0, 'Data should only contain positive values'
    total += n
print 'total is:', total
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-3-33d87ea29ae4> in <module>()
      2 total = 0.0
      3 for n in numbers:
----> 4     assert n >= 0.0, 'Data should only contain positive values'
      5     total += n
      6 print 'total is:', total

AssertionError: Data should only contain positive values
```

Programs like the Firefox browser are full of assertions: 10-20% of the code they contain are there to check that the other 80-90% are working correctly. Broadly speaking, assertions fall into three categories:

- A *precondition* is something that must be true at the start of a function in order for it to work correctly.
- A *postcondition* is something that the function guarantees is true when it finishes.
- An *invariant* is something that is always true at a particular point inside a piece of code.

For example, suppose we are representing rectangles using a tuple of four coordinates `(x0, y0, x1, y1)`. In order to do some calculations, we need to normalize the rectangle so that it is at the origin and 1.0 units long on its longest axis. This function does that, but checks that its input is correctly formatted and that its result makes sense:

```
def normalize_rectangle(rect):
    '''Normalizes a rectangle so that it is at the origin and 1.0 units long on its longest axis.'''
    assert len(rect) == 4, 'Rectangles must contain 4 coordinates'
    x0, y0, x1, y1 = rect
    assert x0 < x1, 'Invalid X coordinates'
    assert y0 < y1, 'Invalid Y coordinates'

    dx = x1 - x0
    dy = y1 - y0
    if dx > dy:
        scaled = float(dx) / dy
        upper_x, upper_y = 1.0, scaled
    else:
        scaled = float(dx) / dy
        upper_x, upper_y = scaled, 1.0

    assert 0 < upper_x <= 1.0, 'Calculated upper X coordinate invalid'
    assert 0 < upper_y <= 1.0, 'Calculated upper Y coordinate invalid'

    return (0, 0, upper_x, upper_y)
```

The preconditions on lines 2, 4, and 5 catch invalid inputs:

```
print normalize_rectangle( (0.0, 1.0, 2.0) ) # missing the fourth coordinate
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-5-3a97b1dcab70> in <module>()
----> 1 print normalize_rectangle( (0.0, 1.0, 2.0) ) # missing the fourth coordinate

<ipython-input-4-9f8adbfdcfc9> in normalize_rectangle(rect)
      1 def normalize_rectangle(rect):
```

```
      2     '''Normalizes a rectangle so that it is at the origin and 1.0 units long on its longest axis.'''
----> 3     assert len(rect) == 4, 'Rectangles must contain 4 coordinates'
      4     x0, y0, x1, y1 = rect
      5     assert x0 < x1, 'Invalid X coordinates'

AssertionError: Rectangles must contain 4 coordinates
```

```
print normalize_rectangle( (4.0, 2.0, 1.0, 5.0) ) # X axis inverted
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-6-f05ae7878a45> in <module>()
----> 1 print normalize_rectangle( (4.0, 2.0, 1.0, 5.0) ) # X axis inverted

<ipython-input-4-9f8adbfdcfc9> in normalize_rectangle(rect)
      3     assert len(rect) == 4, 'Rectangles must contain 4 coordinates'
      4     x0, y0, x1, y1 = rect
----> 5     assert x0 < x1, 'Invalid X coordinates'
      6     assert y0 < y1, 'Invalid Y coordinates'
      7

AssertionError: Invalid X coordinates
```

The post-conditions help us catch bugs by telling us when our calculations cannot have been correct. For example, if we normalize a rectangle that is taller than it is wide everything seems OK:

```
print normalize_rectangle( (0.0, 0.0, 1.0, 5.0) )
```

```
(0, 0, 0.2, 1.0)
```

but if we normalize one that's wider than it is tall, the assertion is triggered:

```
print normalize_rectangle( (0.0, 0.0, 5.0, 1.0) )
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-8-5f0ef7954aeb> in <module>()
----> 1 print normalize_rectangle( (0.0, 0.0, 5.0, 1.0) )

<ipython-input-4-9f8adbfdcfc9> in normalize_rectangle(rect)
     16
     17     assert 0 < upper_x <= 1.0, 'Calculated upper X coordinate invalid'
---> 18     assert 0 < upper_y <= 1.0, 'Calculated upper Y coordinate invalid'
     19
     20     return (0, 0, upper_x, upper_y)

AssertionError: Calculated upper Y coordinate invalid
```

Re-reading our function, we realize that line 10 should divide dy by dx rather than dx by dy. (You can display line numbers by typing Ctrl-M, then L.) If we had left out the assertion at the end of the function, we would have created and returned something that had the right shape as a valid answer, but wasn't. Detecting and debugging that would almost certainly have taken more time in the long run than writing the assertion.

But assertions aren't just about catching errors: they also help people understand programs. Each assertion gives the person reading the program a chance to check (consciously or otherwise) that their understanding matches what the code is doing.

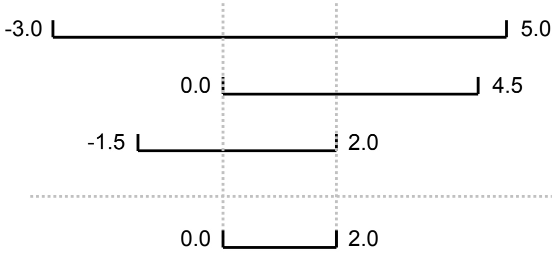


Figure 4.21: Overlapping Ranges

Most good programmers follow two rules when adding assertions to their code. The first is, "fail early, fail often[2]". The greater the distance between when and where an error occurs and when it's noticed, the harder the error will be to debug, so good code catches mistakes as early as possible.

The second rule is, "turn bugs into assertions or tests[3]". If you made a mistake in a piece of code, the odds are good that you have made other mistakes nearby, or will make the same mistake (or a related one) the next time you change it. Writing assertions to check that you haven't *regressed* (i.e., haven't re-introduced an old problem) can save a lot of time in the long run, and helps to warn people who are reading the code (including your future self) that this bit is tricky.

Challenge

Suppose you are writing a function called `average` that calculates the average of the numbers in a list. What pre-conditions and post-conditions would you write for it? Compare your answer to your neighbor's: can you think of a function that will past your tests but not hers or vice versa?

Challenge

Explain in words what the assertions in this code check, and for each one, give an example of input that will make that assertion fail.

```
def running(values):
    assert len(values) > 0
    result = [values[0]]
    for v in values[1:]:
        assert result[-1] >= 0
        result.append(result[-1] + v)
    assert result[-1] >= result[0]
    return result
```

Test-Driven Development

An assertion checks that something is true at a particular point in the program. The next step is to check the overall behavior of a piece of code, i.e., to make sure that it produces the right output when it's given a particular input. For example, suppose we need to find where two or more time series overlap. The range of each time series is represented as a pair of numbers, which are the time the interval started and ended. The output is the largest range that they all include (Figure 4.21).

[2] ../../rules.html#fail-early-fail-often

[3] ../../rules.html#turn-bugs-into-assertions-or-tests

Most novice programmers would solve this problem like this:

1. Write a function `range_overlap`.
2. Call it interactively on two or three different inputs.
3. If it produces the wrong answer, fix the function and re-run that test.

This clearly works—after all, thousands of scientists are doing it right now—but there's a better way:

1. Write a short function for each test.
2. Write a `range_overlap` function that should pass those tests.
3. If `range_overlap` produces any wrong answers, fix it and re-run the test functions.

Writing the tests *before* writing the function they exercise is called *test-driven development* (TDD). Its advocates believe it produces better code faster because:

1. If people write tests after writing the thing to be tested, they are subject to confirmation bias, i.e., they subconsciously write tests to show that their code is correct, rather than to find errors.
2. Writing tests helps programmers figure out what the function is actually supposed to do.

Here are three test functions for `range_overlap`:

```
assert range_overlap([ (0.0, 1.0) ]) == (0.0, 1.0)
assert range_overlap([ (0.0, 1.0), (0.0, 2.0) ]) == (0.0, 1.0)
assert range_overlap([ (0.0, 1.0), (0.0, 2.0), (-1.0, 1.0) ]) == (0.0, 1.0)
```

The error is actually reassuring: we haven't written `range_overlap` yet, so if the tests passed, it would be a sign that someone else had and that we were accidentally using their function.

And as a bonus of writing these tests, we've implicitly defined what our input and output look like: we expect a list of pairs as input, and produce a single pair as output.

Something important is missing, though. We don't have any tests for the case where the ranges don't overlap at all:

```
assert range_overlap([ (0.0, 1.0), (5.0, 6.0) ]) == ???
```

What should `range_overlap` do in this case: fail with an error message, produce a special value like `(0.0, 0.0)` to signal that there's no overlap, or something else? Any actual implementation of the function will do one of these things; writing the tests first helps us figure out which is best *before* we're emotionally invested in whatever we happened to write before we realized there was an issue.

And what about this case?

```
assert range_overlap([ (0.0, 1.0), (1.0, 2.0) ]) == ???
```

Do two segments that touch at their endpoints overlap or not? Mathematicians usually say "yes", but engineers usually say "no". The best answer is "whatever is most useful in the rest of our program", but again, any actual implementation of `range_overlap` is going to do *something*, and whatever it is ought to be consistent with what it does when there's no overlap at all.

Since we're planning to use the range this function returns as the X axis in a time series chart, we decide that:

1. every overlap has to have non-zero width, and
2. we will return the special value `None` when there's no overlap.

`None` is built into Python, and means "nothing here". (Other languages often call the equivalent value `null` or `nil`). With that decision made, we can finish writing our last two tests:

```
assert range_overlap([ (0.0, 1.0), (5.0, 6.0) ]) == None
assert range_overlap([ (0.0, 1.0), (1.0, 2.0) ]) == None
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-10-d877ef460ba2> in <module>()
----> 1 assert range_overlap([ (0.0, 1.0), (5.0, 6.0) ]) == None
      2 assert range_overlap([ (0.0, 1.0), (1.0, 2.0) ]) == None

AssertionError:
```

Again, we get an error because we haven't written our function, but we're now ready to do so:

```
def range_overlap(ranges):
    '''Return common overlap among a set of [low, high] ranges.'''
    lowest = 0.0
    highest = 1.0
    for (low, high) in ranges:
        lowest = max(lowest, low)
        highest = min(highest, high)
    return (lowest, highest)
```

(Take a moment to think about why we use `max` to raise `lowest` and `min` to lower `highest`.) We'd now like to re-run our tests, but they're scattered across three different cells. To make running them easier, let's put them all in a function:

```
def test_range_overlap():
    assert range_overlap([ (0.0, 1.0) ]) == (0.0, 1.0)
    assert range_overlap([ (0.0, 1.0), (0.0, 2.0) ]) == (0.0, 1.0)
    assert range_overlap([ (0.0, 1.0), (0.0, 2.0), (-1.0, 1.0) ]) == (0.0, 1.0)
    assert range_overlap([ (0.0, 1.0), (5.0, 6.0) ]) == None
    assert range_overlap([ (0.0, 1.0), (1.0, 2.0) ]) == None
```

We can now test `range_overlap` with a single function call:

```
test_range_overlap()
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-13-cf9215c96457> in <module>()
----> 1 test_range_overlap()

<ipython-input-12-34c3659163fc> in test_range_overlap()
      3     assert range_overlap([ (0.0, 1.0), (0.0, 2.0) ]) == (0.0, 1.0)
      4     assert range_overlap([ (0.0, 1.0), (0.0, 2.0), (-1.0, 1.0) ]) == (0.0, 1.0)
----> 5     assert range_overlap([ (0.0, 1.0), (5.0, 6.0) ]) == None
      6     assert range_overlap([ (0.0, 1.0), (1.0, 2.0) ]) == None

AssertionError:
```

The first of the tests that was supposed to produce `None` fails, so we know there's something wrong with our function. What we *don't* know, though, is whether the last of our five tests passed or failed, because Python halted the program as soon as it spotted the first error. Still, some information is better than none, and if we trace the behavior of the function with that input, we realize that we're initializing `lowest` and `highest` to 0.0 and 1.0 respectively, regardless of the input values. This violates another important rule of programming: "always initialize from data[4]". We'll leave it as an exercise to fix `range_overlap`.

[4] ../../rules.html#always-initialize-from-data

Challenge

Fix `range_overlap`. Re-run `test_range_overlap` after each change you make.

Debugging

Once testing has uncovered problems, the next step is to fix them. Many novices do this by making more-or-less random changes to their code until it seems to produce the right answer, but that's very inefficient (and the result is usually only correct for the one case they're testing). The more experienced a programmer is, the more systematically they debug, and most follow some variation on the rules explained below.

Know What It's Supposed to Do

The first step in debugging something is to know what it's supposed to do[5]. "My program doesn't work" isn't good enough: in order to diagnose and fix problems, we need to be able to tell correct output from incorrect. If we can write a test case for the failing case—i.e., if we can assert that with *these* inputs, the function should produce *that* result— then we're ready to start debugging. If we can't, then we need to figure out how we're going to know when we've fixed things.

But writing test cases for scientific software is frequently harder than writing test cases for commercial applications, because if we knew what the output of the scientific code was supposed to be, we wouldn't be running the software: we'd be writing up our results and moving on to the next program. In practice, scientists tend to do the following:

1. *Test with simplified data.* Before doing statistics on a real data set, we should try calculating statistics for a single record, for two identical records, for two records whose values are one step apart, or for some other case where we can calculate the right answer by hand.
2. *Test a simplified case.* If our program is supposed to simulate magnetic eddies in rapidly-rotating blobs of supercooled helium, our first test should be a blob of helium that isn't rotating, and isn't being subjected to any external electromagnetic fields. Similarly, if we're looking at the effects of climate change on speciation, our first test should hold temperature, precipitation, and other factors constant.
3. *Compare to an oracle.* A *test oracle* is something—experimental data, an older program whose results are trusted, or even a human expert—against which we can compare the results of our new program. If we have a test oracle, we should store its output for particular cases so that we can compare it with our new results as often as we like without re-running that program.
4. *Check conservation laws.* Mass, energy, and other quantitites are conserved in physical systems, so they should be in programs as well. Similarly, if we are analyzing patient data, the number of records should either stay the same or decrease as we move from one analysis to the next (since we might throw away outliers or records with missing values). If "new" patients start appearing out of nowhere as we move through our pipeline, it's probably a sign that something is wrong.
5. *Visualize.* Data analysts frequently use simple visualizations to check both the science they're doing and the correctness of their code (just as we did in the opening lesson[6] of this tutorial).

[5] ../../rules.html#know-what-its-supposed-to-do

[6] 01-numpy.html

This should only be used for debugging as a last resort, though, since it's very hard to compare two visualizations automatically.

Make It Fail Every Time

We can only debug something when it fails, so the second step is always to find a test case that makes it fail every time[7]. The "every time" part is important because few things are more frustrating than debugging an intermittent problem: if we have to call a function a dozen times to get a single failure, the odds are good that we'll scroll past the failure when it actually occurs.

As part of this, it's always important to check that our code is "plugged in", i.e., that we're actually exercising the problem that we think we are. Every programmer has spent hours chasing a bug, only to realize that they were actually calling their code on the wrong data set or with the wrong configuration parameters, or are using the wrong version of the software entirely. Mistakes like these are particularly likely to happen when we're tired, frustrated, and up against a deadline, which is one of the reasons late-night (or overnight) coding sessions are almost never worthwhile.

Make It Fail Fast

If it takes 20 minutes for the bug to surface, we can only do three experiments an hour. That doesn't must mean we'll get less data in more time: we're also more likely to be distracted by other things as we wait for our program to fail, which means the time we *are* spending on the problem is less focused. It's therefore critical to make it fail fast[8].

As well as making the program fail fast in time, we want to make it fail fast in space, i.e., we want to localize the failure to the smallest possible region of code:

1. The smaller the gap between cause and effect, the easier the connection is to find. Many programmers therefore use a divide and conquer strategy to find bugs, i.e., if the output of a function is wrong, they check whether things are OK in the middle, then concentrate on either the first or second half, and so on.
2. N things can interact in $N^{2/2}$ different ways, so every line of code that *isn't* run as part of a test means more than one thing we don't need to worry about.

Change One Thing at a Time, For a Reason

Replacing random chunks of code is unlikely to do much good. (After all, if you got it wrong the first time, you'll probably get it wrong the second and third as well.) Good programmers therefore change one thing at a time, for a reason[9] They are either trying to gather more information ("is the bug still there if we change the order of the loops?") or test a fix ("can we make the bug go away by sorting our data before processing it?").

Every time we make a change, however small, we should re-run our tests immediately, because the more things we change at once, the harder it is to know what's responsible for what (those N^2 interactions again). And we should re-run *all* of our tests: more than half of fixes made to code introduce (or re-introduce) bugs, so re-running all of our tests tells us whether we have *regressed*.

[7] ../../rules.html#make-it-fail-every-time

[8] ../../rules.html#make-it-fail-fast

[9] ../../rules.html#change-one-thing-at-a-time

Keep Track of What You've Done

Good scientists keep track of what they've done so that they can reproduce their work, and so that they don't waste time repeating the same experiments or running ones whose results won't be interesting. Similarly, debugging works best when we keep track of what we've done[10] and how well it worked. If we find ourselves asking, "Did left followed by right with an odd number of lines cause the crash? Or was it right followed by left? Or was I using an even number of lines?" then it's time to step away from the computer, take a deep breath, and start working more systematically.

Records are particularly useful when the time comes to ask for help. People are more likely to listen to us when we can explain clearly what we did, and we're better able to give them the information they need to be useful.

> **Version Control Revisited**
>
> Version control is often used to reset software to a known state during debugging, and to explore recent changes to code that might be responsible for bugs. In particular, most version control systems have a `blame` command that will show who last changed particular lines of code…

Be Humble

And speaking of help: if we can't find a bug in 10 minutes, we should be humble[11] and ask for help. Just explaining the problem aloud is often useful, since hearing what we're thinking helps us spot inconsistencies and hidden assumptions.

Asking for help also helps alleviate confirmation bias. If we have just spent an hour writing a complicated program, we want it to work, so we're likely to keep telling ourselves why it should, rather than searching for the reason it doesn't. People who aren't emotionally invested in the code can be more objective, which is why they're often able to spot the simple mistakes we have overlooked.

Part of being humble is learning from our mistakes. Programmers tend to get the same things wrong over and over: either they don't understand the language and libraries they're working with, or their model of how things work is wrong. In either case, taking note of why the error occurred and checking for it next time quickly turns into not making the mistake at all.

And that is what makes us most productive in the long run. As the saying goes, "A week of hard work can sometimes save you an hour of thought[12]." If we train ourselves to avoid making some kinds of mistakes, to break our code into modular, testable chunks, and to turn every assumption (or mistake) into an assertion, it will actually take us *less* time to produce working programs, not more.

Key Points

- Program defensively, i.e., assume that errors are going to arise, and write code to detect them when they do.
- Put assertions in programs to check their state as they run, and to help readers understand how those programs are supposed to work.
- Use preconditions to check that the inputs to a function are safe to use.
- Use postconditions to check that the output from a function is safe to use.

[10] ../../rules.html#keep-track-of-what-youve-done

[11] ../../rules.html#be-humble

[12] ../../rules.html#week-hard-work-hour-thought

- Write tests before writing code in order to help determine exactly what that code is supposed to do.
- Know what code is supposed to do *before* trying to debug it.
- Make it fail every time.
- Make it fail fast.
- Change one thing at a time, and for a reason.
- Keep track of what you've done.
- Be humble.

4.6 Command-Line Programs

The IPython Notebook and other interactive tools are great for prototyping code and exploring data, but sooner or later we will want to use our program in a pipeline or run it in a shell script to process thousands of data files. In order to do that, we need to make our programs work like other Unix command-line tools. For example, we may want a program that reads a data set and prints the average inflammation per patient:

```
$ python readings.py --mean inflammation-01.csv
```

```
5.45
5.425
6.1
...
6.4
7.05
5.9
```

but we might also want to look at the minimum of the first four lines

```
$ head -4 inflammation-01.csv | python readings.py --min
```

or the maximum inflammations in several files one after another:

```
$ python readings.py --max inflammation-*.csv
```

Our overall requirements are:

1. If no filename is given on the command line, read data from *standard input*.
2. If one or more filenames are given, read data from them and report statistics for each file separately.
3. Use the `--min`, `--mean`, or `--max` flag to determine what statistic to print.

To make this work, we need to know how to handle command-line arguments in a program, and how to get at standard input. We'll tackle these questions in turn below.

Objectives

- Use the values of command-line arguments in a program.
- Handle flags and files separately in a command-line program.
- Read data from standard input in a program so that it can be used in a pipeline.

Command-Line Arguments

Using the text editor of your choice, save the following in a text file:

```
!cat sys-version.py
```

```
import sys
print 'version is', sys.version
```

The first line imports a library called `sys`, which is short for “system”. It defines values such as `sys.version`, which describes which version of Python we are running. We can run this script from within the IPython Notebook like this:

```
%run sys-version.py
```

```
version is 2.7.5 |Anaconda 1.8.0 (x86_64)| (default, Oct 24 2013, 07:02:20)
[GCC 4.0.1 (Apple Inc. build 5493)]
```

or like this:

```
!ipython sys-version.py
```

```
version is 2.7.5 |Anaconda 1.8.0 (x86_64)| (default, Oct 24 2013, 07:02:20)
[GCC 4.0.1 (Apple Inc. build 5493)]
```

The first method, `%run`, uses a special command in the IPython Notebook to run a program in a `.py` file. The second method is more general: the exclamation mark `!` tells the Notebook to run a shell command, and it just so happens that the command we run is `ipython` with the name of the script.

Here’s another script that does something more interesting:

```
!cat argv-list.py
```

```
import sys
print 'sys.argv is', sys.argv
```

The strange name `argv` stands for “argument values”. Whenever Python runs a program, it takes all of the values given on the command line and puts them in the list `sys.argv` so that the program can determine what they were. If we run this program with no arguments:

```
!ipython argv-list.py
```

```
sys.argv is ['/Users/gwilson/s/bc/python/novice/argv-list.py']
```

the only thing in the list is the full path to our script, which is always `sys.argv[0]`. If we run it with a few arguments, however:

```
!ipython argv-list.py first second third
```

```
sys.argv is ['/Users/gwilson/s/bc/python/novice/argv-list.py', 'first', 'second', 'third']
```

then Python adds each of those arguments to that magic list.

With this in hand, let’s build a version of `readings.py` that always prints the per-patient mean of a single data file. The first step is to write a function that outlines our implementation, and a placeholder for the function that does the actual work. By convention this function is usually called `main`, though we can call it whatever we want:

```
# readings-01.py
import sys
import numpy as np

def main():
    script = sys.argv[0]
    filename = sys.argv[1]
    data = np.loadtxt(filename, delimiter=',')
    for m in data.mean(axis=1):
        print m
```

This function gets the name of the script from `sys.argv[0]`, because that's where it's always put, and the name of the file to process from `sys.argv[1]`. Here's a simple test:

```
$ python readings-01.py inflammation-01.csv
```

There is no output because we have defined a function, but haven't actually called it. Let's add a call to `main`:

```
# readings-02.py
import sys
import numpy as np

def main():
    script = sys.argv[0]
    filename = sys.argv[1]
    data = np.loadtxt(filename, delimiter=',')
    for m in data.mean(axis=1):
        print m

main()
```

and run that:

```
$ python readings-02.py inflammation-01.csv

5.45
5.425
6.1
5.9
5.55
6.225
5.975
6.65
6.625
6.525
6.775
5.8
6.225
5.75
5.225
6.3
6.55
5.7
5.85
6.55
5.775
5.825
```

```
6.175
6.1
5.8
6.425
6.05
6.025
6.175
6.55
6.175
6.35
6.725
6.125
7.075
5.725
5.925
6.15
6.075
5.75
5.975
5.725
6.3
5.9
6.75
5.925
7.225
6.15
5.95
6.275
5.7
6.1
6.825
5.975
6.725
5.7
6.25
6.4
7.05
5.9
```

The Right Way to Do It

If our programs can take complex parameters or multiple filenames, we shouldn't handle `sys.argv` directly. Instead, we should use Python's `argparse` library, which handles common cases in a systematic way, and also makes it easy for us to provide sensible error messages for our users.

Challenge

Write a command-line program that does addition and subtraction:

```
python arith.py 1 + 2
```

```
3
```

```
python arith.py 3 - 4
```

```
-1
```

What goes wrong if you try to add multiplication using '*' to the program?

Challenge

Using the glob module introduced 03-loop.ipynb[13], write a simple version of ls that shows files in the current directory with a particular suffix:

```
$ python my_ls.py py
```

```
left.py right.py zero.py
```

Handling Multiple Files

The next step is to teach our program how to handle multiple files. Since 60 lines of output per file is a lot to page through, we'll start by creating three smaller files, each of which has three days of data for two patients:

```
$ ls small-*.csv
```

```
small-01.csv small-02.csv small-03.csv
```

```
$ cat small-01.csv
```

```
0,0,1
0,1,2
```

```
$ python readings-02.py small-01.csv
```

```
0.333333333333
1.0
```

Using small data files as input also allows us to check our results more easily: here, for example, we can see that our program is calculating the mean correctly for each line, whereas we were really taking it on faith before. This is yet another rule of programming: "test the simple things first[14]."

We want our program to process each file separately, so we need a looop that executes once for each filename. If we specify the files on the command line, the filenames will be in sys.argv, but we need to be careful: sys.argv[0] will always be the name of our script, rather than the name of a file. We also need to handle an unknown number of filenames, since our program could be run for any number of files.

The solution to both problems is to loop over the contents of sys.argv[1:]. The '1' tells Python to start the slice at location 1, so the program's name isn't included; since we've left off the upper bound, the slice runs to the end of the list, and includes all the filenames. Here's our changed program:

```
# readings-03.py
import sys
import numpy as np

def main():
    script = sys.argv[0]
    for filename in sys.argv[1:]:
        data = np.loadtxt(filename, delimiter=',')
        for m in data.mean(axis=1):
            print m

main()
```

[13] earlier

[14] ../../rules.html#test-simple-first

and here it is in action:

```
$ python readings-03.py small-01.csv small-02.csv

0.333333333333
1.0
13.6666666667
11.0
```

Note: at this point, we have created three versions of our script called `readings-01.py`, `readings-02.py`, and `readings-03.py`. We wouldn't do this in real life: instead, we would have one file called `readings.py` that we committed to version control every time we got an enhancement working. For teaching, though, we need all the successive versions side by side.

Challenge

Write a program called `check.py` that takes the names of one or more inflammation data files as arguments and checks that all the files have the same number of rows and columns. What is the best way to test your program?

Handling Command-Line Flags

The next step is to teach our program to pay attention to the `--min`, `--mean`, and `--max` flags. These always appear before the names of the files, so we could just do this:

```
# readings-04.py
import sys
import numpy as np

def main():
    script = sys.argv[0]
    action = sys.argv[1]
    filenames = sys.argv[2:]

    for f in filenames:
        data = np.loadtxt(f, delimiter=',')

        if action == '--min':
            values = data.min(axis=1)
        elif action == '--mean':
            values = data.mean(axis=1)
        elif action == '--max':
            values = data.max(axis=1)

        for m in values:
            print m

main()
```

This works:

```
$ run readings-04.py --max small-01.csv

1.0
2.0
```

but there are seveal things wrong with it:

1. `main` is too large to read comfortably.
2. If `action` isn't one of the three recognized flags, the program loads each file but does nothing with it (because none of the branches in the conditional match). *Silent failures* like this are always hard to debug.

This version pulls the processing of each file out of the loop into a function of its own. It also checks that `action` is one of the allowed flags before doing any processing, so that the program fails fast:

```
# readings-05.py
import sys
import numpy as np

def main():
    script = sys.argv[0]
    action = sys.argv[1]
    filenames = sys.argv[2:]
    assert action in ['--min', '--mean', '--max'], \
           'Action is not one of --min, --mean, or --max: ' + action
    for f in filenames:
        process(f, action)

def process(filename, action):
    data = np.loadtxt(filename, delimiter=',')

    if action == '--min':
        values = data.min(axis=1)
    elif action == '--mean':
        values = data.mean(axis=1)
    elif action == '--max':
        values = data.max(axis=1)

    for m in values:
        print m

main()
```

This is four lines longer than its predecessor, but broken into more digestible chunks of 8 and 12 lines.

Python has a module named argparse[15] that helps handle complex command-line flags. We will not cover this module in this lesson but you can go to Tshepang Lekhonkhobe's Argparse tutorial[16] that is part of Python's Official Documentation.

[15] http://docs.python.org/dev/library/argparse.html

[16] http://docs.python.org/dev/howto/argparse.html

Challenge

Rewrite this program so that it uses `-n`, `-m`, and `-x` instead of `--min`, `--mean`, and `--max` respectively. Is the code easier to read? Is the program easier to understand?

Challenge

Separately, modify the program so that if no parameters are given (i.e., no action is specified and no filenames are given), it prints a message explaining how it should be used.

Challenge

Separately, modify the program so that if no action is given it displays the means of the data.

Handling Standard Input

The next thing our program has to do is read data from standard input if no filenames are given so that we can put it in a pipeline, redirect input to it, and so on. Let's experiment in another script:

```
# count-stdin.py
import sys

count = 0
for line in sys.stdin:
    count += 1

print count, 'lines in standard input'
```

This little program reads lines from a special "file" called `sys.stdin`, which is automatically connected to the program's standard input. We don't have to open it—Python and the operating system take care of that when the program starts up— but we can do almost anything with it that we could do to a regular file. Let's try running it from within the IPython notebook as if it were a regular command-line program:

```
!ipython count-stdin.py < small-01.csv
```

```
2 lines in standard input
```

What if we run it using `%run` (again, from within the notebook)?

```
%run count-stdin.py < fractal_1.txt
```

```
0 lines in standard input
```

As you can see, `%run` doesn't understand file redirection: that's a shell thing.

A common mistake is to try to run something that reads from standard input like this:

```
$ python count_stdin.py fractal_1.txt
```

i.e., to forget the < character that redirect the file to standard input. In this case, there's nothing in standard input, so the program waits at the start of the loop for someone to type something on the keyboard. Since there's no way for us to do this, our program is stuck, and we have to halt it using the `Interrupt` option from the `Kernel` menu in the Notebook.

We now need to rewrite the program so that it loads data from `sys.stdin` if no filenames are provided. Luckily, `numpy.loadtxt` can handle either a filename or an open file as its first parameter, so we don't actually need to change `process`. That leaves `main`:

```
# readings-06.py
def main():
    script = sys.argv[0]
    action = sys.argv[1]
    filenames = sys.argv[2:]
    assert action in ['--min', '--mean', '--max'], \
           'Action is not one of --min, --mean, or --max: ' + action
    if len(filenames) == 0:
        process(sys.stdin, action)
    else:
        for f in filenames:
            process(f, action)
```

Let's try it out from inside the notebook (we'll see in a moment why we send the output through `head`):

```
!ipython readings-06.py --mean < small-01.csv | head -10
```

```
[TerminalIPythonApp] CRITICAL | Bad config encountered during initialization:
[TerminalIPythonApp] CRITICAL | Unrecognized flag: '--mean'
=========
 IPython
=========

Tools for Interactive Computing in Python
=========================================

    A Python shell with automatic history (input and output), dynamic object
    introspection, easier configuration, command completion, access to the
    system shell and more.  IPython can also be embedded in running programs.
```

Whoops: why are we getting IPython's help rather than the line-by-line average of our data? The answer is that IPython has a hard time telling which command-line arguments are meant for it, and which are meant for the program it's running. To make our meaning clear, we have to use `--` (a double dash) to separate the two:

```
!ipython readings-06.py -- --mean < small-01.csv
```

```
0.333333333333
1.0
```

That's better. In fact, that's done: the program now does everything we set out to do.

Challenge

Write a program called `line-count.py` that works like the Unix `wc` command:

- If no filenames are given, it reports the number of lines in standard input.
- If one or more filenames are given, it reports the number of lines in each, followed by the total number of lines.

Key Points

- The `sys` library connects a Python program to the system it is running on.
- The list `sys.argv` contains the command-line arguments that a program was run with.
- Avoid silent failures.
- The “file” `sys.stdin` connects to a program’s standard input.
- The “file” `sys.stdout` connects to a program’s standard output.

Chapter 5

Using Databases and SQL

Almost everyone has used spreadsheets, and almost everyone has eventually run up against their limitations. The more complicated a data set is, the harder it is to filter data, express relationships between different rows and columns, or handle missing values.

Databases pick up where spreadsheets leave off. While they are not as simple to use if all we want is the sum of a dozen numbers, they can do a lot of things that spreadsheets can't, on much larger data sets, faster. And even if we never need to create a database ourselves, knowing how they work will help us understand why so many of the systems we use behave the way we do, and why they insist on structuring data in certain ways.

5.1 Selecting Data

In the late 1920s and early 1930s, William Dyer, Frank Pabodie, and Valentina Roerich led expeditions to the Pole of Inaccessibility[1] in the South Pacific, and then onward to Antarctica. Two years ago, their expeditions were found in a storage locker at Miskatonic University. We have scanned and OCR'd the data they contain, and we now want to store that information in a way that will make search and analysis easy.

We basically have three options: text files, a spreadsheet, or a database. Text files are easiest to create, and work well with version control, but then we would then have to build search and analysis tools ourselves. Spreadsheets are good for doing simple analysis, they don't handle large or complex data sets very well. We would therefore like to put this data in a database, and these lessons will show how to do that.

Objectives

- Explain the difference between a table, a record, and a field.
- Explain the difference between a database and a database manager.
- Write a query to select all values for specific fields from a single table.

[1] http://en.wikipedia.org/wiki/Pole_of_inaccessibility

A Few Definitions

A *relational database* is a way to store and manipulate information that is arranged as *tables*. Each table has columns (also known as *fields*) which describe the data, and rows (also known as *records*) which contain the data.

When we are using a spreadsheet, we put formulas into cells to calculate new values based on old ones. When we are using a database, we send commands (usually called *queries*) to a *database manager*: a program that manipulates the database for us. The database manager does whatever lookups and calculations the query specifies, returning the results in a tabular form that we can then use as a starting point for further queries.

> **Compatibility**
>
> Every database manager—Oracle, IBM DB2, PostgreSQL, MySQL, Microsoft Access, and SQLite—stores data in a different way, so a database created with one cannot be used directly by another. However, every database manager can import and export data in a variety of formats, so it *is* possible to move information from one to another.

Queries are written in a language called *SQL*, which stands for “Structured Query Language”. SQL provides hundreds of different ways to analyze and recombine data; we will only look at a handful, but that handful accounts for most of what scientists do.

The tables below show the database we will use in our examples:

Person: people who took readings.

ident	personal	family
dyer	William	Dyer
pb	Frank	Pabodie
lake	Anderson	Lake
roe	Valentina	Roerich
danforth	Frank	Danforth

Site: locations where readings were taken.

name	lat	long
DR-1	-49.85	-128.57
DR-3	-47.15	-126.72
MSK-4	-48.87	-123.4

Visited: when readings were taken at specific sites.

ident	site	dated
619	DR-1	1927-02-08
622	DR-1	1927-02-10
734	DR-3	1939-01-07
735	DR-3	1930-01-12
751	DR-3	1930-02-26
752	DR-3	
837	MSK-4	1932-01-14
844	DR-1	1932-03-22

Survey: the actual readings.

taken	person	quant	reading
619	dyer	rad	9.82
619	dyer	sal	0.13
622	dyer	rad	7.8
622	dyer	sal	0.09
734	pb	rad	8.41
734	lake	sal	0.05
734	pb	temp	-21.5
735	pb	rad	7.22
735		sal	0.06
735		temp	-26.0
751	pb	rad	4.35
751	pb	temp	-18.5
751	lake	sal	0.1
752	lake	rad	2.19
752	lake	sal	0.09
752	lake	temp	-16.0
752	roe	sal	41.6
837	lake	rad	1.46
837	lake	sal	0.21
837	roe	sal	22.5
844	roe	rad	11.25

Notice that three entries—one in the `Visited` table, and two in the `Survey` table—are shown in red because they don't contain any actual data: we'll return to these missing values *later*. For now, let's write an SQL query that displays scientists' names. We do this using the SQL command `select`, giving it the names of the columns we want and the table we want them from. Our query and its output look like this:

```
select family, personal from Person;
```

```
Dyer      William
Pabodie   Frank
Lake      Anderson
Roerich   Valentina
Danforth  Frank
```

The semi-colon at the end of the query tells the database manager that the query is complete and ready to run. We have written our commands and column names in lower case, and the table name in Title Case, but we don't have to: as the example below shows, SQL is *case insensitive*.

```
SeLeCt FaMiLy, PeRsOnAl FrOm PeRsOn;
```

```
Dyer      William
Pabodie   Frank
Lake      Anderson
Roerich   Valentina
Danforth  Frank
```

Whatever casing convention you choose, please be consistent: complex queries are hard enough to read without the extra cognitive load of random capitalization.

Going back to our query, it's important to understand that the rows and columns in a database table aren't actually stored in any particular order. They will always be *displayed* in some order, but we can control that in various ways. For example, we could swap the columns in the output by writing our query as:

```
select personal, family from Person;
```

```
William    Dyer
Frank      Pabodie
Anderson   Lake
Valentina  Roerich
Frank      Danforth
```

or even repeat columns:

```
select ident, ident, ident from Person;
```

```
dyer      dyer      dyer
pb        pb        pb
lake      lake      lake
roe       roe       roe
danforth  danforth  danforth
```

As a shortcut, we can select all of the columns in a table using `*`:

```
select * from Person;
```

```
dyer      William    Dyer
pb        Frank      Pabodie
lake      Anderson   Lake
roe       Valentina  Roerich
danforth  Frank      Danforth
```

Challenge

Write a query that selects only site names from the `Site` table.

Challenge

Many people format queries as:

```
SELECT personal, family FROM person;
```

or as:

```
select Personal, Family from PERSON;
```

What style do you find easiest to read, and why?

Key Points

- A relational database stores information in tables, each of which has a fixed set of columns and a variable number of records.
- A database manager is a program that manipulates information stored in a database.
- We write queries in a specialized language called SQL to extract information from databases.
- SQL is case-insensitive.

5.2 Sorting and Removing Duplicates

Objectives

- Write queries that display results in a particular order.
- Write queries that eliminate duplicate values from data.

Data is often redundant, so queries often return redundant information. For example, if we select the quantitites that have been measured from the `survey` table, we get this:

```
select quant from Survey;
```

```
rad
sal
rad
sal
rad
sal
temp
rad
sal
temp
rad
temp
sal
rad
sal
temp
sal
rad
sal
sal
rad
```

We can eliminate the redundant output to make the result more readable by adding the `distinct` keyword to our query:

```
select distinct quant from Survey;
```

```
rad
sal
temp
```

If we select more than one column—for example, both the survey site ID and the quantity measured—then the distinct pairs of values are returned:

```
select distinct taken, quant from Survey;
```

```
619  rad
619  sal
622  rad
622  sal
734  rad
734  sal
734  temp
735  rad
735  sal
735  temp
751  rad
751  temp
751  sal
752  rad
752  sal
752  temp
837  rad
837  sal
844  rad
```

Notice in both cases that duplicates are removed even if they didn't appear to be adjacent in the database. Again, it's important to remember that rows aren't actually ordered: they're just displayed that way.

Challenge

Write a query that selects distinct dates from the `Site` table.

As we mentioned earlier, database records are not stored in any particular order. This means that query results aren't necessarily sorted, and even if they are, we often want to sort them in a different way, e.g., by the name of the project instead of by the name of the scientist. We can do this in SQL by adding an `order by` clause to our query:

```
select * from Person order by ident;
```

```
danforth  Frank      Danforth
dyer      William    Dyer
lake      Anderson   Lake
pb        Frank      Pabodie
roe       Valentina  Roerich
```

By default, results are sorted in ascending order (i.e., from least to greatest). We can sort in the opposite order using `desc` (for "descending"):

```
select * from person order by ident desc;
```

```
roe       Valentina  Roerich
pb        Frank      Pabodie
lake      Anderson   Lake
dyer      William    Dyer
danforth  Frank      Danforth
```

(And if we want to make it clear that we're sorting in ascending order, we can use `asc` instead of `desc`.)

We can also sort on several fields at once. For example, this query sorts results first in ascending order by `taken`, and then in descending order by `person` within each group of equal `taken` values:

```
select taken, person from Survey order by taken asc, person desc;
```

```
619   dyer
619   dyer
622   dyer
622   dyer
734   pb
734   pb
734   lake
735   pb
735   None
735   None
751   pb
751   pb
751   lake
752   roe
752   lake
752   lake
752   lake
837   roe
837   lake
837   lake
844   roe
```

This is easier to understand if we also remove duplicates:

```
select distinct taken, person from Survey order by taken asc, person desc;
```

```
619   dyer
622   dyer
734   pb
734   lake
735   pb
735   None
751   pb
751   lake
752   roe
752   lake
837   roe
837   lake
844   roe
```

Challenge

Write a query that returns the distinct dates in the `Visited` table.

Challenge

Write a query that displays the full names of the scientists in the `Person` table, ordered by family name.

Key Points

- The records in a database table are not intrinsically ordered: if we want to display them in some order, we must specify that explicitly.
- The values in a database are not guaranteed to be unique: if we want to eliminate duplicates, we must specify that explicitly as well.

5.3 Filtering

Objectives

- Write queries that select records that satisfy user-specified conditions.
- Explain the order in which the clauses in a query are executed.

One of the most powerful features of a database is the ability to *filter* data, i.e., to select only those records that match certain criteria. For example, suppose we want to see when a particular site was visited. We can select these records from the `Visited` table by using a `where` clause in our query:

```
select * from Visited where site='DR-1';
```

```
619   DR-1   1927-02-08
622   DR-1   1927-02-10
844   DR-1   1932-03-22
```

The database manager executes this query in two stages. First, it checks at each row in the `Visited` table to see which ones satisfy the `where`. It then uses the column names following the `select` keyword to determine what columns to display.

This processing order means that we can filter records using `where` based on values in columns that aren't then displayed (Figure 5.1):

```
select ident from Visited where site='DR-1';
```

```
619
622
844
```

We can use many other Boolean operators to filter our data. For example, we can ask for all information from the DR-1 site collected since 1930:

```
select * from Visited where (site='DR-1') and (dated>='1930-00-00');
```

```
844   DR-1   1932-03-22
```

(The parentheses around the individual tests aren't strictly required, but they help make the query easier to read.)

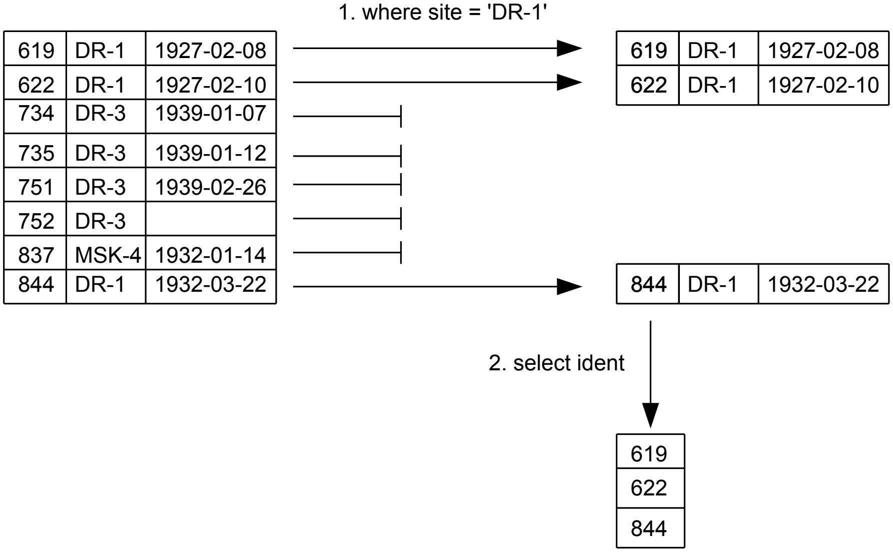


Figure 5.1: Filtering in SQL

Handling Dates

Most database managers have a special data type for dates. In fact, many have two: one for dates, such as “May 31, 1971”, and one for durations, such as “31 days”. SQLite doesn’t: instead, it stores dates as either text (in the ISO-8601 standard format “YYYY-MM-DD HH:MM:SS.SSSS”), real numbers (the number of days since November 24, 4714 BCE), or integers (the number of seconds since midnight, January 1, 1970). If this sounds complicated, it is, but not nearly as complicated as figuring out historical dates in Sweden[2].

If we want to find out what measurements were taken by either Lake or Roerich, we can combine the tests on their names using `or`:

```
select * from Survey where person='lake' or person='roe';
```

```
734   lake   sal    0.05
751   lake   sal    0.1
752   lake   rad    2.19
752   lake   sal    0.09
752   lake   temp   -16.0
752   roe    sal    41.6
837   lake   rad    1.46
837   lake   sal    0.21
837   roe    sal    22.5
844   roe    rad    11.25
```

Alternatively, we can use `in` to see if a value is in a specific set:

```
select * from Survey where person in ('lake', 'roe');
```

[2] http://en.wikipedia.org/wiki/Swedish_calendar

```
734  lake  sal   0.05
751  lake  sal   0.1
752  lake  rad   2.19
752  lake  sal   0.09
752  lake  temp  -16.0
752  roe   sal   41.6
837  lake  rad   1.46
837  lake  sal   0.21
837  roe   sal   22.5
844  roe   rad   11.25
```

We can combine `and` with `or`, but we need to be careful about which operator is executed first. If we *don't* use parentheses, we get this:

```
select * from Survey where quant='sal' and person='lake' or person='roe';
```

```
734  lake  sal  0.05
751  lake  sal  0.1
752  lake  sal  0.09
752  roe   sal  41.6
837  lake  sal  0.21
837  roe   sal  22.5
844  roe   rad  11.25
```

which is salinity measurements by Lake, and *any* measurement by Roerich. We probably want this instead:

```
select * from Survey where quant='sal' and (person='lake' or person='roe');
```

```
734  lake  sal  0.05
751  lake  sal  0.1
752  lake  sal  0.09
752  roe   sal  41.6
837  lake  sal  0.21
837  roe   sal  22.5
```

Finally, we can use `distinct` with `where` to give a second level of filtering:

```
select distinct person, quant from Survey where person='lake' or person='roe';
```

```
lake  sal
lake  rad
lake  temp
roe   sal
roe   rad
```

But remember: `distinct` is applied to the values displayed in the chosen columns, not to the entire rows as they are being processed.

Growing Queries

What we have just done is how most people "grow" their SQL queries. We started with something simple that did part of what we wanted, then added more clauses one by one, testing their effects as we went. This is a good strategy—in fact, for complex queries it’s often the *only* strategy—but it depends on quick turnaround, and on us recognizing the right answer when we get it.

The best way to achieve quick turnaround is often to put a subset of data in a temporary database and run our queries against that, or to fill a small database with synthesized records. For example, instead of trying our queries against an actual database of 20 million Australians, we could run it against a sample of ten thousand, or write a small program to generate ten thousand random (but plausible) records and use that.

Challenge

Suppose we want to select all sites that lie more than 30Âř from the poles. Our first query is:

```
select * from Site where (lat > -60) or (lat < 60);
```

Explain why this is wrong, and rewrite the query so that it is correct.

Challenge

Normalized salinity readings are supposed to be between 0.0 and 1.0. Write a query that selects all records from `Survey` with salinity values outside this range.

Challenge

The SQL test `*column-name* like *pattern*` is true if the value in the named column matches the pattern given; the character ‘%’ can be used any number of times in the pattern to mean “match zero or more characters”.

Expression	Value
`’a’ like ’a’`	True
`’a’ like ’%a’`	True
`’b’ like ’%a’`	False
`’alpha’ like ’a%’`	True
`’alpha’ like ’a%p%’`	True

The expression `*column-name* not like *pattern*` inverts the test. Using `like`, write a query that finds all the records in `Visited` that *aren’t* from sites labelled ‘DR-something’.

Key Points

- Use `where` to filter records according to Boolean conditions.
- Filtering is done on whole records, so conditions can use fields that are not actually displayed.

5.4 Calculating New Values

Objectives

- Write queries that calculate new values for each selected record.

After carefully re-reading the expedition logs, we realize that the radiation measurements they report may need to be corrected upward by 5%. Rather than modifying the stored data, we can do this calculation on the fly as part of our query:

```
select 1.05 * reading from Survey where quant='rad';
```

```
10.311
8.19
8.8305
7.581
4.5675
2.2995
1.533
11.8125
```

When we run the query, the expression `1.05 * reading` is evaluated for each row. Expressions can use any of the fields, all of usual arithmetic operators, and a variety of common functions. (Exactly which ones depends on which database manager is being used.) For example, we can convert temperature readings from Fahrenheit to Celsius and round to two decimal places:

```
select taken, round(5*(reading-32)/9, 2) from Survey where quant='temp';
```

```
734   -29.72
735   -32.22
751   -28.06
752   -26.67
```

We can also combine values from different fields, for example by using the string concatenation operator `||`:

```
select personal || ' ' || family from Person;
```

```
William Dyer
Frank Pabodie
Anderson Lake
Valentina Roerich
Frank Danforth
```

Challenge

After further reading, we realize that Valentina Roerich was reporting salinity as percentages. Write a query that returns all of her salinity measurements from the `Survey` table with the values divided by 100.

Challenge

The `union` operator combines the results of two queries:

```
select * from Person where ident='dyer' union select * from Person where ident='roe';
```

```
dyer   William     Dyer
roe    Valentina   Roerich
```

Use `union` to create a consolidated list of salinity measurements in which Roerich's, and only Roerich's, have been corrected as described in the previous challenge. The output should be something like:

619	0.13
622	0.09
734	0.05
751	0.1
752	0.09
752	0.416
837	0.21
837	0.225

Challenge

The site identifiers in the `Visited` table have two parts separated by a '-':

```
select distinct site from Visited;
```

DR-1
DR-3
MSK-4

Some major site identifiers are two letters long and some are three. The "in string" function `instr(X, Y)` returns the 1-based index of the first occurrence of string Y in string X, or 0 if Y does not exist in X. The substring function `substr(X, I)` returns the substring of X starting at index I. Use these two functions to produce a list of unique major site identifiers. (For this data, the list should contain only "DR" and "MSK").

Key Points

- SQL can perform calculations using the values in a record as part of a query.

5.5 Missing Data

Objectives

- Explain how databases represent missing information.
- Explain the three-valued logic databases use when manipulating missing information.
- Write queries that handle missing information correctly.

Real-world data is never complete—there are always holes. Databases represent these holes using special value called `null`. `null` is not zero, `False`, or the empty string; it is a one-of-a-kind value that means "nothing here". Dealing with `null` requires a few special tricks and some careful thinking.

To start, let's have a look at the `Visited` table. There are eight records, but #752 doesn't have a date—or rather, its date is null:

```
select * from Visited;
```

```
619   DR-1    1927-02-08
622   DR-1    1927-02-10
734   DR-3    1939-01-07
735   DR-3    1930-01-12
751   DR-3    1930-02-26
752   DR-3
837   MSK-4   1932-01-14
844   DR-1    1932-03-22
```

Null doesn't behave like other values. If we select the records that come before 1930:

```
select * from Visited where dated<'1930-00-00';
```

```
619   DR-1   1927-02-08
622   DR-1   1927-02-10
```

we get two results, and if we select the ones that come during or after 1930:

```
select * from Visited where dated>='1930-00-00';
```

```
734   DR-3    1939-01-07
735   DR-3    1930-01-12
751   DR-3    1930-02-26
837   MSK-4   1932-01-14
844   DR-1    1932-03-22
```

we get five, but record #752 isn't in either set of results. The reason is that `null<'1930-00-00'` is neither true nor false: null means, "We don't know," and if we don't know the value on the left side of a comparison, we don't know whether the comparison is true or false. Since databases represent "don't know" as null, the value of `null<'1930-00-00'` is actually `null`. `null>='1930-00-00'` is also null because we can't answer to that question either. And since the only records kept by a `where` are those for which the test is true, record #752 isn't included in either set of results.

Comparisons aren't the only operations that behave this way with nulls. `1+null` is `null`, `5*null` is `null`, `log(null)` is `null`, and so on. In particular, comparing things to null with = and != produces null:

```
select * from Visited where dated=NULL;
```

```
select * from Visited where dated!=NULL;
```

To check whether a value is `null` or not, we must use a special test `is null`:

```
select * from Visited where dated is NULL;
```

```
752   DR-3
```

or its inverse `is not null`:

```
select * from Visited where dated is not NULL;
```

```
619   DR-1    1927-02-08
622   DR-1    1927-02-10
734   DR-3    1939-01-07
735   DR-3    1930-01-12
751   DR-3    1930-02-26
837   MSK-4   1932-01-14
844   DR-1    1932-03-22
```

Null values cause headaches wherever they appear. For example, suppose we want to find all the salinity measurements that weren't taken by Dyer. It's natural to write the query like this:

```
select * from Survey where quant='sal' and person!='lake';
```

```
619  dyer  sal  0.13
622  dyer  sal  0.09
752  roe   sal  41.6
837  roe   sal  22.5
```

but this query filters omits the records where we don't know who took the measurement. Once again, the reason is that when `person` is `null`, the `!=` comparison produces `null`, so the record isn't kept in our results. If we want to keep these records we need to add an explicit check:

```
select * from Survey where quant='sal' and (person!='lake' or person is null);
```

```
619  dyer  sal  0.13
622  dyer  sal  0.09
735  None  sal  0.06
752  roe   sal  41.6
837  roe   sal  22.5
```

We still have to decide whether this is the right thing to do or not. If we want to be absolutely sure that we aren't including any measurements by Lake in our results, we need to exclude all the records for which we don't know who did the work.

Challenge

Write a query that sorts the records in `Visited` by date, omitting entries for which the date is not known (i.e., is null).

Challenge

What do you expect the query:

```
select * from Visited where dated in ('1927-02-08', null);
```

to produce? What does it actually produce?

Challenge

Some database designers prefer to use a *sentinel value* to mark missing data rather than `null`. For example, they will use the date "0000-00-00" to mark a missing date, or -1.0 to mark a missing salinity or radiation reading (since actual readings cannot be negative). What does this simplify? What burdens or risks does it introduce?

Key Points

- Databases use `null` to represent missing information.
- Any arithmetic or Boolean operation involving `null` produces `null` as a result.
- The only operators that can safely be used with `null` are `is null` and `is not null`.

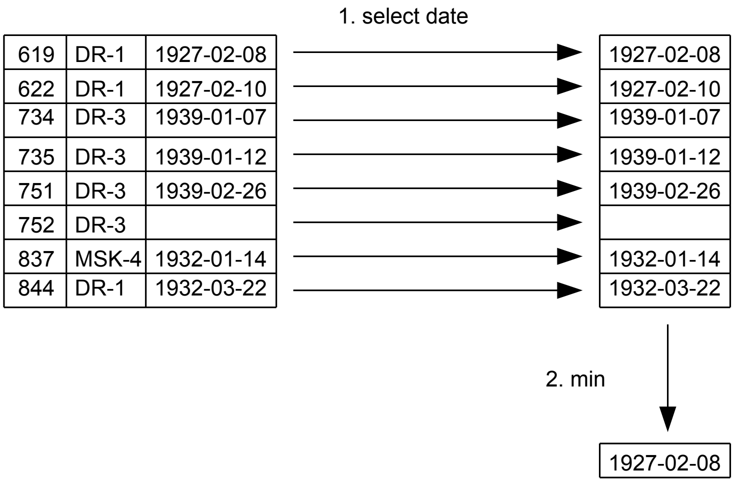


Figure 5.2: Aggregation in SQL

5.6 Aggregation

Objectives

- Define "aggregation" and give examples of its use.
- Write queries that compute aggregated values.
- Trace the execution of a query that performs aggregation.
- Explain how missing data is handled during aggregation.

We now want to calculate ranges and averages for our data. We know how to select all of the dates from the `Visited` table:

```
select dated from Visited;
```

```
1927-02-08
1927-02-10
1939-01-07
1930-01-12
1930-02-26

1932-01-14
1932-03-22
```

but to combine them, wee must use an *aggregation function* such as `min` or `max`. Each of these functions takes a set of records as input, and produces a single record as output (Figure 5.2):

```
select min(dated) from Visited;
```

```
1927-02-08
```

```
select max(dated) from Visited;
```

```
1939-01-07
```

`min` and `max` are just two of the aggregation functions built into SQL. Three others are `avg`, `count`, and `sum`:

```
select avg(reading) from Survey where quant='sal';
```

```
7.20333333333
```

```
select count(reading) from Survey where quant='sal';
```

```
9
```

```
select sum(reading) from Survey where quant='sal';
```

```
64.83
```

We used `count(reading)` here, but we could just as easily have counted `quant` or any other field in the table, or even used `count(*)`, since the function doesn't care about the values themselves, just how many values there are.

SQL lets us do several aggregations at once. We can, for example, find the range of sensible salinity measurements:

```
select min(reading), max(reading) from Survey where quant='sal' and reading<=1.0;
```

```
0.05    0.21
```

We can also combine aggregated results with raw results, although the output might surprise you:

```
select person, count(*) from Survey where quant='sal' and reading<=1.0;
```

```
lake    7
```

Why does Lake's name appear rather than Roerich's or Dyer's? The answer is that when it has to aggregate a field, but isn't told how to, the database manager chooses an actual value from the input set. It might use the first one processed, the last one, or something else entirely.

Another important fact is that when there are no values to aggregate, aggregation's result is "don't know" rather than zero or some other arbitrary value:

```
select person, max(reading), sum(reading) from Survey where quant='missing';
```

One final important feature of aggregation functions is that they are inconsistent with the rest of SQL in a very useful way. If we add two values, and one of them is null, the result is null. By extension, if we use `sum` to add all the values in a set, and any of those values are null, the result should also be null. It's much more useful, though, for aggregation functions to ignore null values and only combine those that are non-null. This behavior lets us write our queries as:

```
select min(dated) from Visited;
```

```
1927-02-08
```

instead of always having to filter explicitly:

```
select min(dated) from Visited where dated is not null;
```

```
1927-02-08
```

Aggregating all records at once doesn't always make sense. For example, suppose Gina suspects that there is a systematic bias in her data, and that some scientists' radiation readings are higher than others. We know that this doesn't work:

```
select person, count(reading), round(avg(reading), 2)
from  Survey
where quant='rad';
```

```
roe    8    6.56
```

because the database manager selects a single arbitrary scientist's name rather than aggregating separately for each scientist. Since there are only five scientists, she could write five queries of the form:

```
select person, count(reading), round(avg(reading), 2)
from  Survey
where quant='rad'
and   person='dyer';
```

```
dyer    2    8.81
```

but this would be tedious, and if she ever had a data set with fifty or five hundred scientists, the chances of her getting all of those queries right is small.

What we need to do is tell the database manager to aggregate the hours for each scientist separately using a `group by` clause:

```
select   person, count(reading), round(avg(reading), 2)
from     Survey
where    quant='rad'
group by person;
```

```
dyer    2    8.81
lake    2    1.82
pb      3    6.66
roe     1    11.25
```

`group by` does exactly what its name implies: groups all the records with the same value for the specified field together so that aggregation can process each batch separately. Since all the records in each batch have the same value for `person`, it no longer matters that the database manager is picking an arbitrary one to display alongside the aggregated `reading` values.

Just as we can sort by multiple criteria at once, we can also group by multiple criteria. To get the average reading by scientist and quantity measured, for example, we just add another field to the `group by` clause:

```
select   person, quant, count(reading), round(avg(reading), 2)
from     Survey
group by person, quant;
```

```
        sal     1    0.06
        temp    1    -26.0
dyer    rad     2    8.81
dyer    sal     2    0.11
lake    rad     2    1.82
lake    sal     4    0.11
lake    temp    1    -16.0
pb      rad     3    6.66
pb      temp    2    -20.0
roe     rad     1    11.25
roe     sal     2    32.05
```

Note that we have added `person` to the list of fields displayed, since the results wouldn't make much sense otherwise.

Let's go one step further and remove all the entries where we don't know who took the measurement:

```
select   person, quant, count(reading), round(avg(reading), 2)
from     Survey
where    person is not null
group by person, quant
order by person, quant;
```

dyer	rad	2	8.81
dyer	sal	2	0.11
lake	rad	2	1.82
lake	sal	4	0.11
lake	temp	1	-16.0
pb	rad	3	6.66
pb	temp	2	-20.0
roe	rad	1	11.25
roe	sal	2	32.05

Looking more closely, this query:

1. selected records from the `Survey` table where the `person` field was not null;
2. grouped those records into subsets so that the `person` and `quant` values in each subset were the same;
3. ordered those subsets first by `person`, and then within each sub-group by `quant`; and
4. counted the number of records in each subset, calculated the average `reading` in each, and chose a `person` and `quant` value from each (it doesn't matter which ones, since they're all equal).

Challenge

How many temperature readings did Frank Pabodie record, and what was their average value?

Challenge

The average of a set of values is the sum of the values divided by the number of values. Does this mean that the `avg` function returns 2.0 or 3.0 when given the values 1.0, `null`, and 5.0?

Challenge

We want to calculate the difference between each individual radiation reading and the average of all the radiation readings. We write the query:

```
select reading - avg(reading) from Survey where quant='rad';
```

What does this actually produce, and why?

Challenge

The function `group_concat(field, separator)` concatenates all the values in a field using the specified separator character (or ',' if the separator isn't specified). Use this to produce a one-line list of scientists' names, such as:

```
William Dyer, Frank Pabodie, Anderson Lake, Valentina Roerich, Frank Danforth
```

Can you find a way to order the list by surname?

Key Points

- An aggregation function combines many values to produce a single new value.
- Aggregation functions ignore `null` values.
- Aggregation happens after filtering.

5.7 Combining Data

Objectives

- Explain the operation of a query that joins two tables.
- Explain how to restrict the output of a query containing a join to only include meaningful combinations of values.
- Write queries that join tables on equal keys.
- Explain what primary and foreign keys are, and why they are useful.
- Explain what atomic values are, and why database fields should only contain atomic values.

In order to submit her data to a web site that aggregates historical meteorological data, Gina needs to format it as latitude, longitude, date, quantity, and reading. However, her latitudes and longitudes are in the `Site` table, while the dates of measurements are in the `Visited` table and the readings themselves are in the `Survey` table. She needs to combine these tables somehow.

The SQL command to do this is `join`. To see how it works, let's start by joining the `Site` and `Visited` tables:

```
select * from Site join Visited;
```

```
DR-1    -49.85  -128.57  619  DR-1   1927-02-08
DR-1    -49.85  -128.57  622  DR-1   1927-02-10
DR-1    -49.85  -128.57  734  DR-3   1939-01-07
DR-1    -49.85  -128.57  735  DR-3   1930-01-12
DR-1    -49.85  -128.57  751  DR-3   1930-02-26
DR-1    -49.85  -128.57  752  DR-3
DR-1    -49.85  -128.57  837  MSK-4  1932-01-14
DR-1    -49.85  -128.57  844  DR-1   1932-03-22
DR-3    -47.15  -126.72  619  DR-1   1927-02-08
DR-3    -47.15  -126.72  622  DR-1   1927-02-10
DR-3    -47.15  -126.72  734  DR-3   1939-01-07
DR-3    -47.15  -126.72  735  DR-3   1930-01-12
DR-3    -47.15  -126.72  751  DR-3   1930-02-26
DR-3    -47.15  -126.72  752  DR-3
DR-3    -47.15  -126.72  837  MSK-4  1932-01-14
DR-3    -47.15  -126.72  844  DR-1   1932-03-22
MSK-4   -48.87  -123.4   619  DR-1   1927-02-08
MSK-4   -48.87  -123.4   622  DR-1   1927-02-10
MSK-4   -48.87  -123.4   734  DR-3   1939-01-07
MSK-4   -48.87  -123.4   735  DR-3   1930-01-12
MSK-4   -48.87  -123.4   751  DR-3   1930-02-26
MSK-4   -48.87  -123.4   752  DR-3
MSK-4   -48.87  -123.4   837  MSK-4  1932-01-14
MSK-4   -48.87  -123.4   844  DR-1   1932-03-22
```

`join` creates the *cross product* of two tables, i.e., it joins each record of one with each record of the other to give all possible combinations. Since there are three records in `Site` and eight in `Visited`, the join's output has 24 records. And since each table has three fields, the output has six fields.

What the join *hasn't* done is figure out if the records being joined have anything to do with each other. It has no way of knowing whether they do or not until we tell it how. To do that, we add a clause specifying that we're only interested in combinations that have the same site name:

```
select * from Site join Visited on Site.name=Visited.site;
```

```
DR-1    -49.85  -128.57  619  DR-1   1927-02-08
DR-1    -49.85  -128.57  622  DR-1   1927-02-10
DR-1    -49.85  -128.57  844  DR-1   1932-03-22
DR-3    -47.15  -126.72  734  DR-3   1939-01-07
DR-3    -47.15  -126.72  735  DR-3   1930-01-12
DR-3    -47.15  -126.72  751  DR-3   1930-02-26
DR-3    -47.15  -126.72  752  DR-3
MSK-4   -48.87  -123.4   837  MSK-4  1932-01-14
```

`on` does the same job as `where`: it only keeps records that pass some test. (The difference between the two is that `on` filters records as they're being created, while `where` waits until the join is done and then does the filtering.) Once we add this to our query, the database manager throws away records that combined information about two different sites, leaving us with just the ones we want.

Notice that we used `table.field` to specify field names in the output of the join. We do this because tables can have fields with the same name, and we need to be specific which ones we're talking about. For example, if we joined the person and visited tables, the result would inherit a field called ident from each of the original tables.

We can now use the same dotted notation to select the three columns we actually want out of our join:

```
select Site.lat, Site.long, Visited.dated
from   Site join Visited
on     Site.name=Visited.site;
```

```
-49.85   -128.57   1927-02-08
-49.85   -128.57   1927-02-10
-49.85   -128.57   1932-03-22
-47.15   -126.72
-47.15   -126.72   1930-01-12
-47.15   -126.72   1930-02-26
-47.15   -126.72   1939-01-07
-48.87   -123.4    1932-01-14
```

If joining two tables is good, joining many tables must be better. In fact, we can join any number of tables simply by adding more join clauses to our query, and more on tests to filter out combinations of records that don't make sense:

```
select Site.lat, Site.long, Visited.dated, Survey.quant, Survey.reading
from   Site join Visited join Survey
on     Site.name=Visited.site
and    Visited.ident=Survey.taken
and    Visited.dated is not null;
```

```
-49.85   -128.57   1927-02-08   rad    9.82
-49.85   -128.57   1927-02-08   sal    0.13
-49.85   -128.57   1927-02-10   rad    7.8
-49.85   -128.57   1927-02-10   sal    0.09
-47.15   -126.72   1939-01-07   rad    8.41
-47.15   -126.72   1939-01-07   sal    0.05
-47.15   -126.72   1939-01-07   temp   -21.5
-47.15   -126.72   1930-01-12   rad    7.22
-47.15   -126.72   1930-01-12   sal    0.06
-47.15   -126.72   1930-01-12   temp   -26.0
-47.15   -126.72   1930-02-26   rad    4.35
-47.15   -126.72   1930-02-26   sal    0.1
-47.15   -126.72   1930-02-26   temp   -18.5
-48.87   -123.4    1932-01-14   rad    1.46
-48.87   -123.4    1932-01-14   sal    0.21
-48.87   -123.4    1932-01-14   sal    22.5
-49.85   -128.57   1932-03-22   rad    11.25
```

We can tell which records from Site, Visited, and Survey correspond with each other because those tables contain *primary keys* and *foreign keys*. A primary key is a value, or combination of values, that uniquely identifies each record in a table. A foreign key is a value (or combination of values) from one table that identifies a unique record in another table. Another way of saying this is that a foreign key is the primary key of one table that appears in some other table. In our database, Person.ident is the primary key in the Person table, while Survey.person is a foreign key relating the Survey table's entries to entries in Person.

Most database designers believe that every table should have a well-defined primary key. They also believe that this key should be separate from the data itself, so that if we ever need to change the data, we only need to make one change in one place. One easy way to do this is to create an arbitrary, unique ID for each record as we add it to the database. This is actually very common: those IDs have names like “student numbers” and “patient numbers”, and they almost always turn out to have originally been a unique record identifier in some database system or other. As the query below demonstrates, SQLite automatically numbers records as they’re added to tables, and we can use those record numbers in queries:

```
select rowid, * from Person;
```

1	dyer	William	Dyer
2	pb	Frank	Pabodie
3	lake	Anderson	Lake
4	roe	Valentina	Roerich
5	danforth	Frank	Danforth

Data Hygiene

Now that we have seen how joins work, we can see why the relational model is so useful and how best to use it. The first rule is that every value should be *atomic*, i.e., not contain parts that we might want to work with separately. We store personal and family names in separate columns instead of putting the entire name in one column so that we don’t have to use substring operations to get the name’s components. More importantly, we store the two parts of the name separately because splitting on spaces is unreliable: just think of a name like “Eloise St. Cyr” or “Jan Mikkel Steubart”.

The second rule is that every record should have a unique primary key. This can be a serial number that has no intrinsic meaning, one of the values in the record (like the `ident` field in the `Person` table), or even a combination of values: the triple `(taken, person, quant)` from the `Survey` table uniquely identifies every measurement.

The third rule is that there should be no redundant information. For example, we could get rid of the `Site` table and rewrite the `Visited` table like this:

619	-49.85	-128.57	1927-02-08
622	-49.85	-128.57	1927-02-10
734	-47.15	-126.72	1939-01-07
735	-47.15	-126.72	1930-01-12
751	-47.15	-126.72	1930-02-26
752	-47.15	-126.72	
837	-48.87	-123.40	1932-01-14
844	-49.85	-128.57	1932-03-22

In fact, we could use a single table that recorded all the information about each reading in each row, just as a spreadsheet would. The problem is that it’s very hard to keep data organized this way consistent: if we realize that the date of a particular visit to a particular site is wrong, we have to change multiple records in the database. What’s worse, we may have to guess which records to change, since other sites may also have been visited on that date.

The fourth rule is that the units for every value should be stored explicitly. Our database doesn’t do this, and that’s a problem: Roerich’s salinity measurements are several orders of magnitude larger than anyone else’s, but we don’t know if that means she was using parts per million instead of parts per thousand, or whether there actually was a saline anomaly at that site in 1932.

Stepping back, data and the tools used to store it have a symbiotic relationship: we use tables and joins because it's efficient, provided our data is organized a certain way, but organize our data that way because we have tools to manipulate it efficiently if it's in a certain form. As anthropologists say, the tool shapes the hand that shapes the tool.

Challenge

Write a query that lists all radiation readings from the DR-1 site.

Challenge

Write a query that lists all sites visited by people named "Frank".

Challenge

Describe in your own words what the following query produces:

```
select Site.name from Site join Visited
on Site.lat<-49.0 and Site.name=Visited.site and Visited.dated>='1932-00-00';
```

Key Points

- Every fact should be represented in a database exactly once.
- A join produces all combinations of records from one table with records from another.
- A primary key is a field (or set of fields) whose values uniquely identify the records in a table.
- A foreign key is a field (or set of fields) in one table whose values are a primary key in another table.
- We can eliminate meaningless combinations of records by matching primary keys and foreign keys between tables.
- Keys should be atomic values to make joins simpler and more efficient.

5.8 Creating and Modifying Data

Objectives

- Write queries that creates tables.
- Write queries to insert, modify, and delete records.

So far we have only looked at how to get information out of a database, both because that is more frequent than adding information, and because most other operations only make sense once queries are understood. If we want to create and modify data, we need to know two other pairs of commands.

The first pair are `create table` and `drop table`. While they are written as two words, they are actually single commands. The first one creates a new table; its arguments are the names and types of the table's columns. For example, the following statements create the four tables in our survey database:

```
create table Person(ident text, personal text, family text);
create table Site(name text, lat real, long real);
create table Visited(ident integer, site text, dated text);
create table Survey(taken integer, person text, quant real, reading real);
```

We can get rid of one of our tables using:

```
drop table Survey;
```

Be very careful when doing this: most databases have some support for undoing changes, but it's better not to have to rely on it.

Different database systems support different data types for table columns, but most provide the following:

integer	a signed integer
real	a floating point number
text	a character string
blob	a "binary large object", such as an image

Most databases also support Booleans and date/time values; SQLite uses the integers 0 and 1 for the former, and represents the latter as discussed *earlier*. An increasing number of databases also support geographic data types, such as latitude and longitude. Keeping track of what particular systems do or do not offer, and what names they give different data types, is an unending portability headache.

When we create a table, we can specify several kinds of constraints on its columns. For example, a better definition for the `Survey` table would be:

```
create table Survey(
    taken   integer not null, -- where reading taken
    person  text,             -- may not know who took it
    quant   real not null,    -- the quantity measured
    reading real not null,    -- the actual reading
    primary key(taken, quant),
    foreign key(taken) references Visited(ident),
    foreign key(person) references Person(ident)
);
```

Once again, exactly what constraints are avialable and what they're called depends on which database manager we are using.

Once tables have been created, we can add and remove records using our other pair of commands, `insert` and `delete`. The simplest form of `insert` statement lists values in order:

```
insert into Site values('DR-1', -49.85, -128.57);
insert into Site values('DR-3', -47.15, -126.72);
insert into Site values('MSK-4', -48.87, -123.40);
```

We can also insert values into one table directly from another:

```
create table JustLatLong(lat text, long text);
insert into JustLatLong select lat, long from site;
```

Deleting records can be a bit trickier, because we have to ensure that the database remains internally consistent. If all we care about is a single table, we can use the `delete` command with a `where` clause that matches the records we want to discard. For example, once we realize that Frank Danforth didn't take any measurements, we can remove him from the `Person` table like this:

```
delete from Person where ident = "danforth";
```

But what if we removed Anderson Lake instead? Our `Survey` table would still contain seven records of measurements he'd taken, but that's never supposed to happen: `Survey.person` is a foreign key into the `Person` table, and all our queries assume there will be a row in the latter matching every value in the former.

This problem is called *referential integrity*: we need to ensure that all references between tables can always be resolved correctly. One way to do this is to delete all the records that use `'lake'` as a foreign key before deleting the record that uses it as a primary key. If our database manager supports it, we can automate this using *cascading delete*. However, this technique is outside the scope of this chapter.

> **Mix and Match**
>
> Many applications use a hybrid storage model instead of putting everything into a database: the actual data (such as astronomical images) is stored in files, while the database stores the files' names, their modification dates, the region of the sky they cover, their spectral characteristics, and so on. This is also how most music player software is built: the database inside the application keeps track of the MP3 files, but the files themselves live on disk.

Challenge

Write an SQL statement to replace all uses of `null` in `Survey.person` with the string `'unknown'`.

Challenge

One of our colleagues has sent us a *CSV* file containing temperature readings by Robert Olmstead, which is formatted like this:

```
Taken,Temp
619,-21.5
622,-15.5
```

Write a small Python program that reads this file in and prints out the SQL `insert` statements needed to add these records to the survey database. Note: you will need to add an entry for Olmstead to the `Person` table. If you are testing your program repeatedly, you may want to investigate SQL's `insert or replace` command.

Challenge

SQLite has several administrative commands that aren't part of the SQL standard. One of them is `.dump`, which prints the SQL commands needed to re-create the database. Another is `.load`, which reads a file created by `.dump` and restores the database. A colleague of yours thinks that storing dump files (which are text) in version control is a good way to track and manage changes to the database. What are the pros and cons of this approach? (Hint: records aren't stored in any particular order.)

Key Points

- Database tables are created using queries that specify their names and the names and properties of their fields.
- Records can be inserted, updated, or deleted using queries.
- It is simpler and safer to modify data when every record has a unique primary key.

5.9 Programming with Databases

Objectives

- Write short programs that execute SQL queries.
- Trace the execution of a program that contains an SQL query.
- Explain why most database applications are written in a general-purpose language rather than in SQL.

To close, let's have a look at how to access a database from a general-purpose programming language like Python. Other languages use almost exactly the same model: library and function names may differ, but the concepts are the same.

Here's a short Python program that selects latitudes and longitudes from an SQLite database stored in a file called `survey.db`:

```
import sqlite3
connection = sqlite3.connect("survey.db")
cursor = connection.cursor()
cursor.execute("select site.lat, site.long from site;")
results = cursor.fetchall()
for r in results:
    print r
cursor.close()
connection.close()
```

```
(-49.85, -128.57)
(-47.15, -126.72)
(-48.87, -123.4)
```

The program starts by importing the `sqlite3` library. If we were connecting to MySQL, DB2, or some other database, we would import a different library, but all of them provide the same functions, so that the rest of our program does not have to change (at least, not much) if we switch from one database to another.

Line 2 establishes a connection to the database. Since we're using SQLite, all we need to specify is the name of the database file. Other systems may require us to provide a username and password as well. Line 3 then uses this connection to create a *cursor*; just like the cursor in an editor, its role is to keep track of where we are in the database.

On line 4, we use that cursor to ask the database to execute a query for us. The query is written in SQL, and passed to `cursor.execute` as a string. It's our job to make sure that SQL is properly formatted; if it isn't, or if something goes wrong when it is being executed, the database will report an error.

The database returns the results of the query to us in response to the `cursor.fetchall` call on line 5. This result is a list with one entry for each record in the result set; if we loop over that list (line 6) and print those list entries (line 7), we can see that each one is a tuple with one element for each field we asked for.

Finally, lines 8 and 9 close our cursor and our connection, since the database can only keep a limited number of these open at one time. Since establishing a connection takes time, though, we shouldn't open a connection, do one operation, then close the connection, only to reopen it a few microseconds later to do another operation. Instead, it's normal to create one connection that stays open for the lifetime of the program.

Queries in real applications will often depend on values provided by users. For example, this function takes a user's ID as a parameter and returns their name:

```
def get_name(database_file, person_ident):
    query = "select personal || ' ' || family from Person where ident='" + person_ident + "';"

    connection = sqlite3.connect(database_file)
    cursor = connection.cursor()
    cursor.execute(query)
    results = cursor.fetchall()
    cursor.close()
    connection.close()

    return results[0][0]

print "full name for dyer:", get_name('survey.db', 'dyer')
```

```
full name for dyer: William Dyer
```

We use string concatenation on the first line of this function to construct a query containing the user ID we have been given. This seems simple enough, but what happens if someone gives us this string as input?

```
dyer'; drop table Survey; select '
```

It looks like there's garbage after the name of the project, but it is very carefully chosen garbage. If we insert this string into our query, the result is:

```
select personal || ' ' || family from Person where ident='dyer'; drop tale Survey; select '';
```

If we execute this, it will erase one of the tables in our database.

This is called an *SQL injection attack*, and it has been used to attack thousands of programs over the years. In particular, many web sites that take data from users insert values directly into queries without checking them carefully first.

Since a villain might try to smuggle commands into our queries in many different ways, the safest way to deal with this threat is to replace characters like quotes with their escaped equivalents, so that we can safely put whatever the user gives us inside a string. We can do this by using a *prepared statement* instead of formatting our statements as strings. Here's what our example program looks like if we do this:

```
def get_name(database_file, person_ident):
    query = "select personal || ' ' || family from Person where ident=?;"

    connection = sqlite3.connect(database_file)
    cursor = connection.cursor()
    cursor.execute(query, [person_ident])
    results = cursor.fetchall()
    cursor.close()
    connection.close()

    return results[0][0]

print "full name for dyer:", get_name('survey.db', 'dyer')
```

```
full name for dyer: William Dyer
```

The key changes are in the query string and the `execute` call. Instead of formatting the query ourselves, we put question marks in the query template where we want to insert values. When we call `execute`, we provide a list that contains as many values as there are question marks in the query. The library matches values to question marks in order, and translates any special characters in the values into their escaped equivalents so that they are safe to use.

Challenge

Write a Python program that creates a new database in a file called `original.db` containing a single table called `Pressure`, with a single field called `reading`, and inserts 100,000 random numbers between 10.0 and 25.0. How long does it take this program to run? How long does it take to run a program that simply writes those random numbers to a file?

Challenge

Write a Python program that creates a new database called `backup.db` with the same structure as `original.db` and copies all the values greater than 20.0 from `original.db` to `backup.db`. Which is faster: filtering values in the query, or reading everything into memory and filtering in Python?

Key Points

- We usually write database applications in a general-purpose language, and embed SQL queries in it.
- To connect to a database, a program must use a library specific to that database manager.
- A program may open one or more connections to a single database, and have one or more cursors active in each.
- Programs can read query results in batches or all at once.

Chapter 6

A Few Extras

A few things come up in our classes that don't fit naturally into the flow of our lessons. We have gathered several of them here.

6.1 Branching in Git

Here's where we are right now:

```
$ git log

commit 005937fbe2a98fb83f0ade869025dc2636b4dad5
Author: Vlad Dracula <vlad@tran.sylvan.ia>
Date:   Thu Aug 22 10:14:07 2013 -0400

    Thoughts about the climate

commit 34961b159c27df3b475cfe4415d94a6d1fcd064d
Author: Vlad Dracula <vlad@tran.sylvan.ia>
Date:   Thu Aug 22 10:07:21 2013 -0400

    Concerns about Mars's moons on my furry friend

commit f22b25e3233b4645dabd0d81e651fe074bd8e73b
Author: Vlad Dracula <vlad@tran.sylvan.ia>
Date:   Thu Aug 22 09:51:46 2013 -0400

    Starting to think about Mars

$ cat mars.txt
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
```

We can draw the history of the repository as shown in Figure 6.1 (we'll see in a second why there's a box called `master`):

Let's run this command:

```
$ git branch moons
```

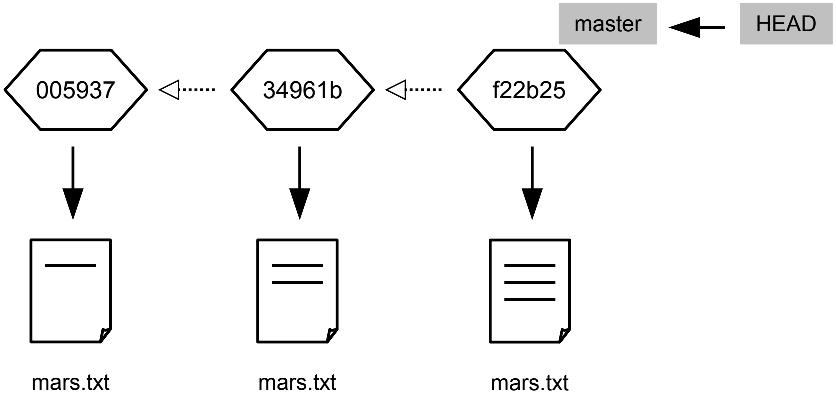


Figure 6.1: Initial History of Repository

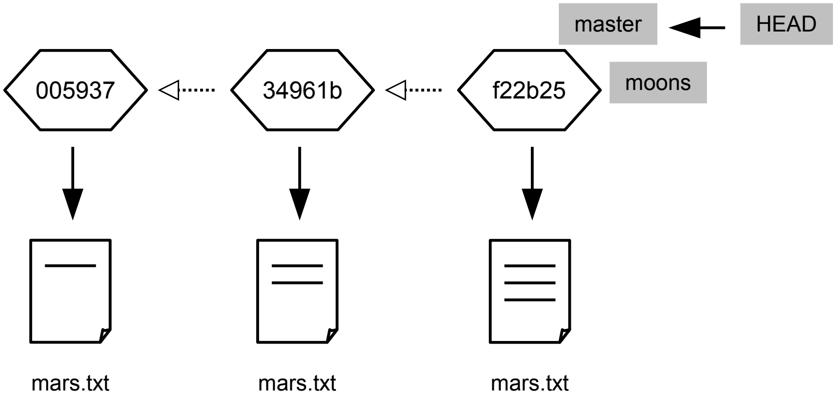


Figure 6.2: Repository Immediately After Creating Branch

It appears to do nothing, but behind the scenes it has created a new *branch* called moons (Figure 6.2):

```
$ git branch
```

```
* master
  moons
```

Git is now maintaining two named bookmarks in our history: master, which was created automatically when we set up the repository, and moons, which we just made. They both point to the same revision right now, but we can change that. Let's make moons the active branch (Figure 6.3):

```
$ git checkout moons
```

```
Switched to branch 'moons'
```

```
$ git branch
```

```
  master
* moons
```

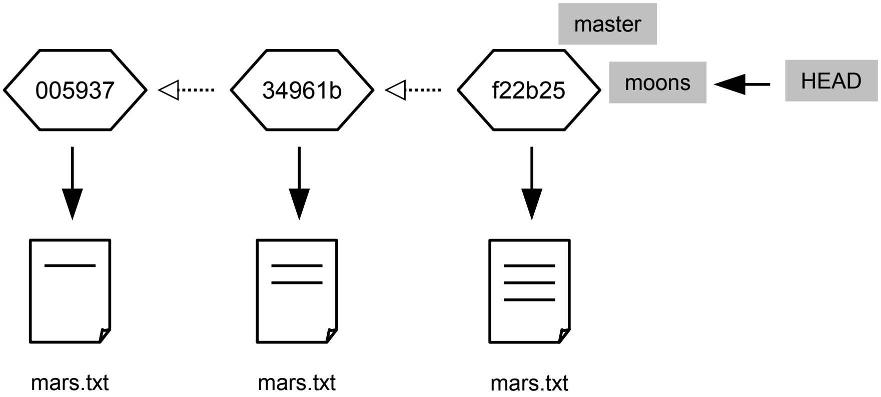


Figure 6.3: Repository After Switching Branches

Our file looks the same:

```
$ cat mars.txt
```

```
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
```

because it *is* the same: Let's add another line to it:

```
$ echo "Maybe we should put the base on one of the moons instead?" >> mars.txt
```

and add an entirely new file:

```
$ echo "Phobos is larger than Deimos" > moons.txt
$ ls
```

```
mars.txt    moons.txt
```

Git now tells us that we have one changed file and one new file:

```
$ git status
```

```
# On branch moons
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#    modified:   mars.txt
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#    moons.txt
no changes added to commit (use "git add" and/or "git commit -a")
```

Let's add and commit those changes (the `-A` flag to `git commit` means "add everything"):

```
$ git add -A
$ git status
```

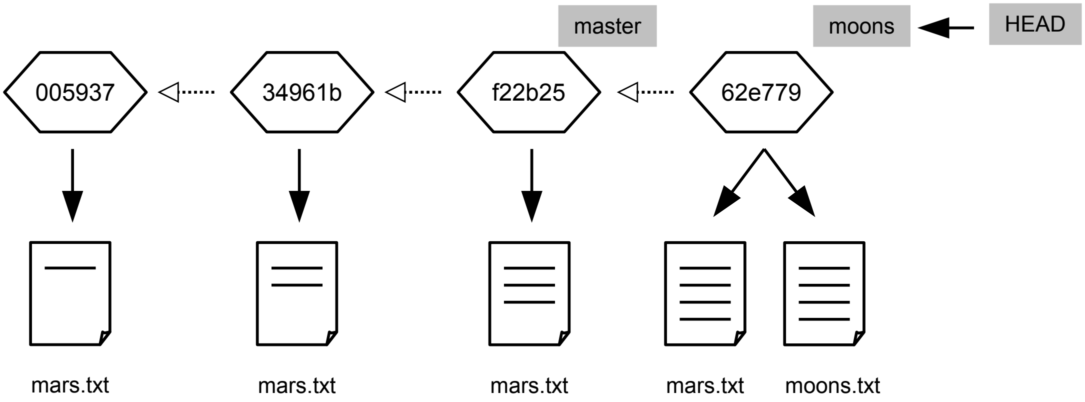


Figure 6.4: Repository After Committing to Branch

```
# On branch moons
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#    modified:   mars.txt
#    new file:   moons.txt
#

$ git commit -m "Thinking about the moons"

[moons 62e7791] Thinking about the moons
 2 files changed, 2 insertions(+)
 create mode 100644 moons.txt
```

Our repository is now in the state shown in Figure 6.4. The `moons` branch has advanced to record the changes we just made, but `master` is still where it was. If we switch back to `master`:

```
$ git checkout master
```

our changes seem to disappear:

```
$ ls

mars.txt

$ cat mars.txt

Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
```

They're still in the repository—they're just not in the revision that `master` is currently pointing to. In essence, we've created a parallel timeline that shares some history with the original one before diverging.

Let's make some changes in the `master` branch to further illustrate this point:

```
$ echo "Should we go with a classical name like Ares Base?" > names.txt
$ git status
```

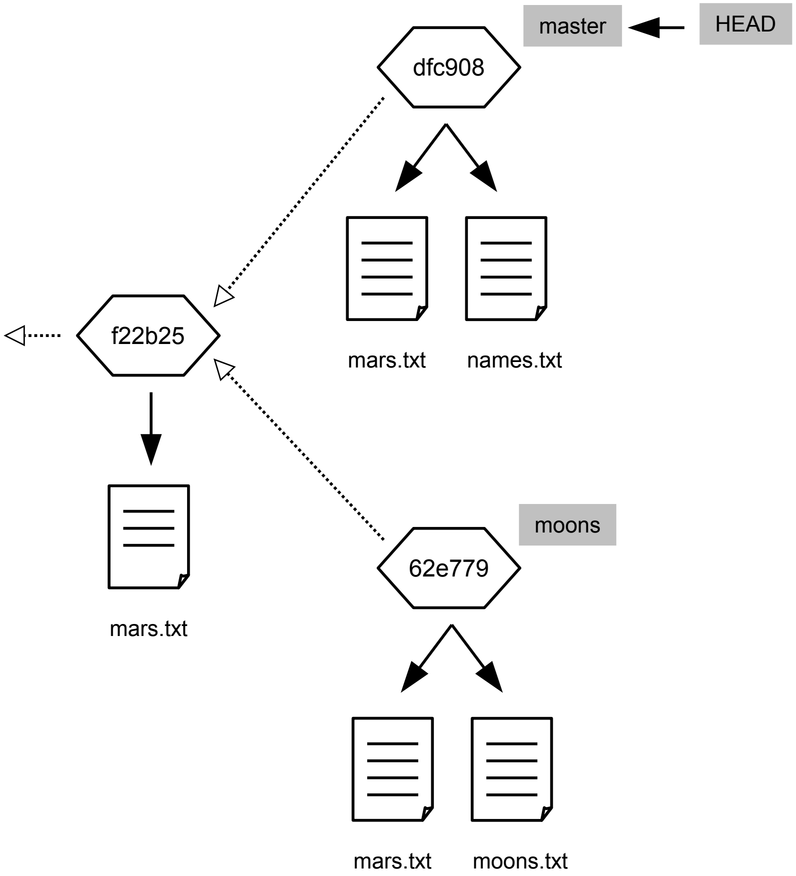


Figure 6.5: Repository After Branches Have Diverged

```
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#    names.txt
nothing added to commit but untracked files present (use "git add" to track)

$ git add names.txt
$ git commit -m "We will need a cool name for our secret base"

[master dfcf908] We will need a cool name for our secret base
 1 file changed, 1 insertion(+)
 create mode 100644 names.txt
```

Our repository is now as shown in Figure 6.5. `master` and `moons` have both moved on from their original common state, but in different ways. They could continue independent existence indefinitely, but at some point we'll probably want to *merge* our changes. Let's do that now:

```
$ git branch
```

```
* master
  moons

$ git merge moons
```

When we run the `git merge` command, Git opens an editor to let us write a log entry about what we're doing. The editor session initially contains this:

```
Merge branch 'moons'

# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
```

If we notice that something is wrong and decide not to complete the merge, we must delete everything in the file—Git interprets an empty log message to mean, "Don't proceed." Otherwise, everything that isn't marked as a comment with # will be saved to the log. In this case, we'll stick with the default log message. When we save the file and exit the editor, Git displays this:

```
Merge made by the 'recursive' strategy.
 mars.txt  | 1 +
 moons.txt | 1 +
 2 files changed, 2 insertions(+)
 create mode 100644 moons.txt
```

We now have all of our changes in one place:

```
$ ls

mars.txt    moons.txt    names.txt
```

and our repository looks like Figure 6.6:

We can ask Git to draw a diagram of the repository's history with this command:

```
$ git log --oneline --topo-order --graph

*   e0cf8ab Merge branch 'moons'
|\
| * 62e7791 Thinking about the moons
* | dfcf908 We will need a cool name for our secret base
|/
* 005937f Thoughts about the climate
* 34961b1 Concerns about Mars's moons on my furry friend
* f22b25e Starting to think about Mars
```

This ASCII art is fine for small sets of changes, but for anything significant, it's much better to use a GUI that can draw graphs using lines instead of characters.

Branching strikes most newcomers as unnecessary complexity, particularly for single-author projects. After all, if we need to make some changes to a project, what do we gain by creating parallel universes?

The answer is that branching makes it easy for us to concentrate on one thing at a time. Suppose we are part-way through rewriting a function that calculates spatial correlations when we realize that the task would be easier if our file I/O routines always stored things as complex numbers. Most

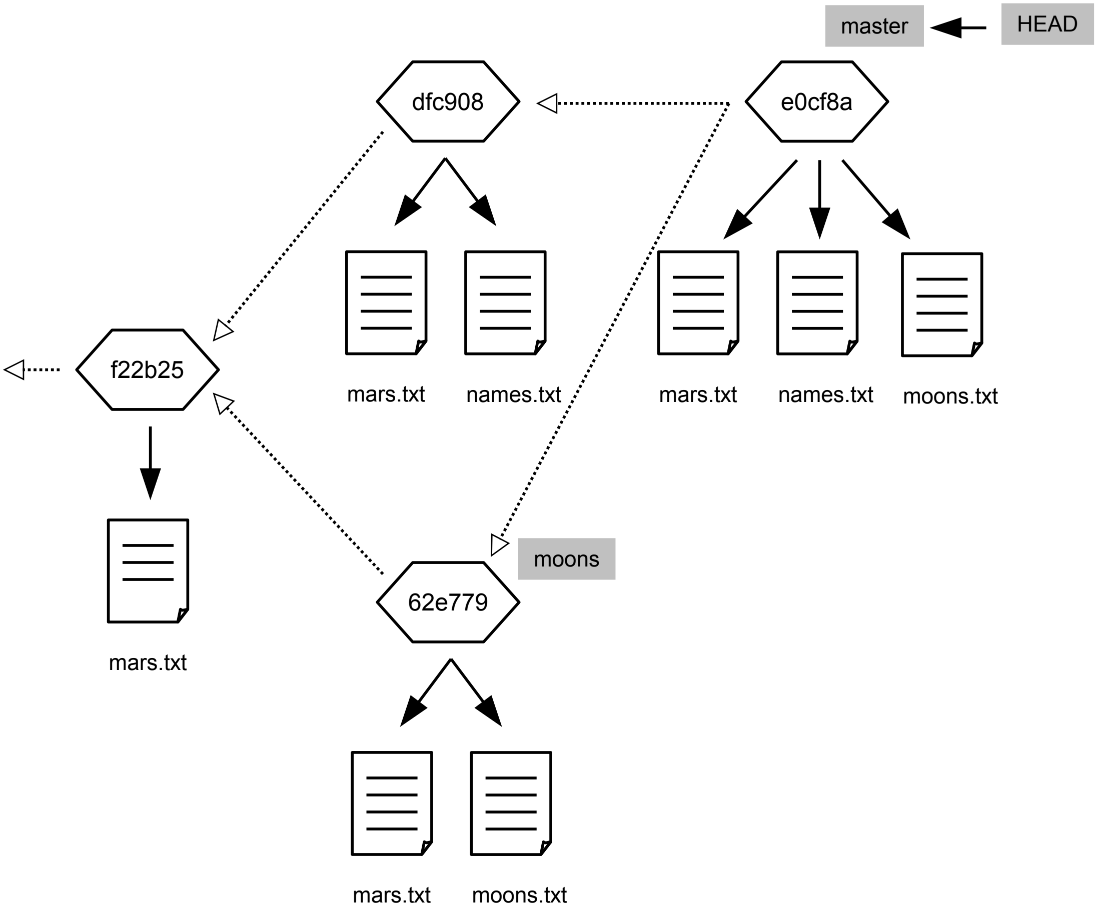


Figure 6.6: Repository After Merging

people would put the spatial correlation changes aside, change the file I/O, then (hopefully) come back to the original task.

The problem with this is that we have to remember what we were doing, even if we realize halfway through rewriting file I/O that we should also rewrite our error handling. It's quite common to wind up with half a dozen tasks stacked on top of one another, and quite hard to them all straight. Branching allows us to put what we're doing in a safe place, solve the new problem, then resume our earlier work.

In practice, most developers never make changes directly in the `master` branch. Instead, they create a new branch from it for every change they want to make, then merge those branches back to `master` when the work is complete. Returning to our hypothetical example, we would:

1. create a branch called something like `better-spatial-correlation` for those changes;
2. go back to master and create another branch called `file-input-produces-complex-values` for *those* changes;
3. merge `file-input-produces-complex-values` into `master`;
4. merge `master` into `better-spatial-correlation`; and
5. finish work on the spatial correlation function and merge it all back into `master`.

And if, partway through this process, our supervisor asked us to change the graph-plotting routines to conform to the university's new style guide, we would simply switch back to `master`, create a branch for that, make the changes, produce the desired graphs, and leave the changes parked in that branch until we were ready to merge them.

6.2 Code Review

The model shown in the main lesson[1] in which everyone pushes and pulls from a single repository, is perfectly usable, but there's one thing it *doesn't* let us do: *code review*. Suppose Dracula wants to be able to look at Wolfman's changes before merging them into the master copy on GitHub, just as he would review Wolfman's paper before publishing it (or perhaps even before submitting it for publication). We need to arrange things so that Wolfman can commit his changes and Dracula can compare them with the master copy; in fact, we want Wolfman to be able to commit many times, so that he can incorporate Dracula's feedback and get further review as often as necessary.

To allow code review, most programmers take a slightly more roundabout route to merging. When the project starts, Dracula creates a repository on GitHub in exactly the same way as we created the `planets` repository[2] and then *clones* it to his desktop:

```
$ git clone https://github.com/vlad/undersea.git
```

```
Cloning into 'undersea'...
warning: You appear to have cloned an empty repository.
```

`git clone` automatically adds the original repository on GitHub as a remote of the local repository called `origin`—this is why we chose `origin` as a remote name in our previous example:

```
$ cd undersea
$ git remote -v
```

```
origin https://github.com/vlad/undersea.git (fetch)
origin https://github.com/vlad/undersea.git (push)
```

[1] ../git/02-collab.html

[2] ../git/02-collab.html

Figure 6.7: Forking a Repository on Git

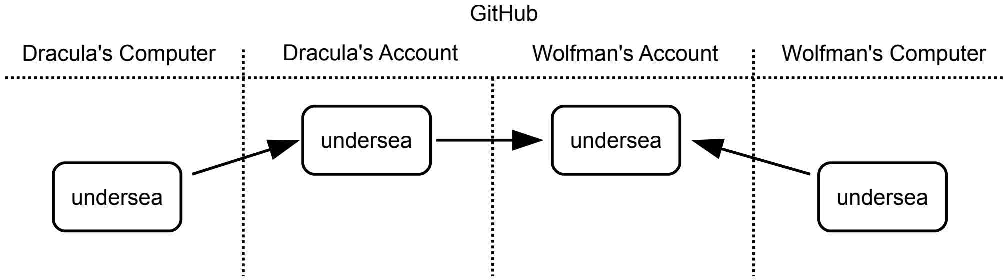


Figure 6.8: Repositories After Cloning the Fork

Dracula can now push and pull changes just as before.

Wolfman doesn't clone Dracula's GitHub repository directly. Instead, he *forks* it, i.e., clones it on GitHub. He does this using the GitHub web interface (Figure 6.7).

He then clones his own GitHub repository, not Dracula's, to give himself a desktop copy (Figure 6.8.

This may seem like unnecessary work, but it allows Wolfman and Dracula to collaborate much more effectively. Suppose Wolfman makes a change to the project. He commits it to his local repository, then uses `git push` to copy those changes to GitHub (Figure 6.9).

He then creates a *pull request*, which notifies Dracula that Wolfman wants to merge some changes into Dracula's repository (Figure 6.10).

A pull request is a merge waiting to happen. When Dracula views it online, he can see and comment on the changes Wolfman wants to make. Commenting is the crucial step here, and half the reason Wolfman went to the trouble of forking the repository on GitHub. Dracula, or anyone else involved in the project, can now give Wolfman feedback on what he is trying to do: this function is too long, that one contains a bug, there's a special case that isn't being handled anywhere, and so on. Wolfman can then update his code, commit locally, and push those changes to GitHub to update the

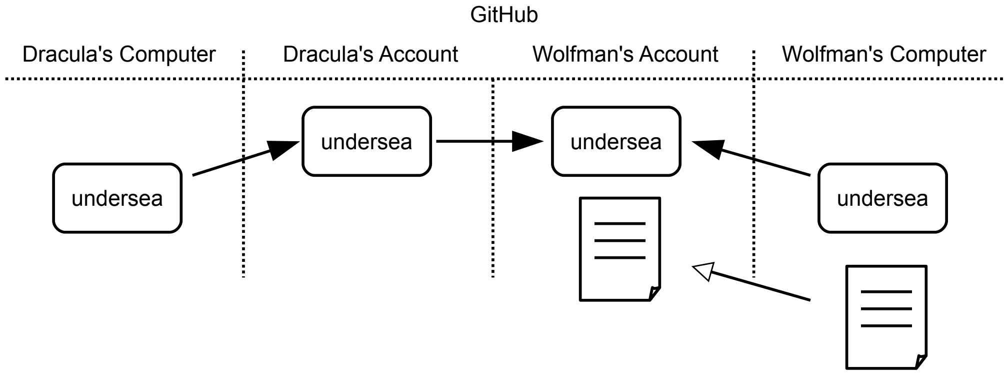


Figure 6.9: Repositories After Pushing Changes to Personal Copy

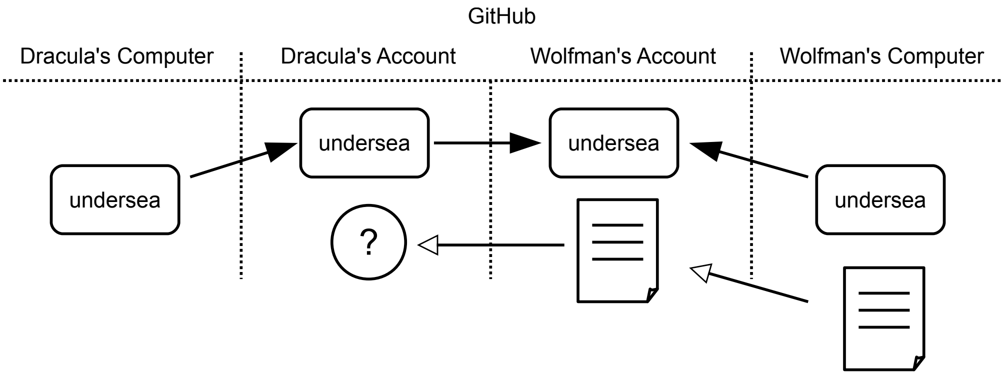


Figure 6.10: Creating a Pull Request

pull request.

This process is exactly like peer review of papers, though usually much faster. In large open source projects like Firefox, it's very common for a pull request to be updated several times before finally being accepted and merged. Working this way not only helps maintain the quality of the code, it is also a very effective way to transfer knowledge.

If Wolfman wants to do more work while he's waiting for Dracula to review his first modification, he creates a new branch in his local repository, pushes it to GitHub, and then issues a pull request from that. We can now see why Git, Mercurial, and other modern version control systems use branching so much: it helps people work together, but on their own time. It might take Dracula several days to get around to reviewing Wolfman's changes. Rather than being stalled until then, Wolfman can just switch to another branch and work on something else, then switch back when Dracula's review finally comes in. Once the changes in a particular branch have been accepted, Wolfman can delete it; provided it has been merged into `master` (or some other branch), the only thing that will be lost is the pointer with the branch name, not the changes themselves.

We said above that code review is half the reason every developer should have their own repository on GitHub (or whatever service is being used). The other reason is that working this way allows people to explore ideas without needing permission from any central authority. If you want to change this tutorial, you can fork the Software Carpentry repository on GitHub[3] and start rewriting things in your repository. You can send us a pull request if you want to share you changes with us, but you don't have to. And if other people like your version better than ours, they can start forking your repository and sending pull requests to you instead of to us.

If this sounds familiar, it's because it is the way science itself works. When someone publishes a new method or result, other scientists can immediately start building on top of it—essentially, they can create their own fork of the work and start committing changes to it. If the first scientist likes the second's work, she can incorporate those findings into her next paper, which is analogous to merging a pull request. If she doesn't, then it's up to other scientists to decide whose work to build on, or whether to try to combine both approaches.

[3] https://github.com/swcarpentry/bc

6.3 Manual Pages

We can get help for any Unix command with the `man` (short for manual) command. For example, here is the command to look up information on `cp`:

```
$ man cp
```

The output displayed is referred to as the "man page".

The man page will be displayed in the default file viewer for our shell, which usually a program called `more`. When `more` displays a colon ':', we can press the space bar to get the next page, the letter 'h' to get help, or the letter 'q' to quit.

`man`'s output is typically complete but concise, as it is designed to be used as a reference rather than a tutorial. Most man pages are divided into sections:

- NAME: gives the name of the command and a brief description
- SYNOPSIS: how to run the command, including optional and mandatory parameters. (We will explain the syntax later.)
- DESCRIPTION: a fuller description than the synopsis, including a description of all the options to the command. This section may also include example usage or details about how the command works.
- EXAMPLES: self-explanatory.
- SEE ALSO: list other commands that we might find useful or other sources of information that might help us.

Other sections we might see include AUTHOR, REPORTING BUGS, COPYRIGHT, HISTORY, (known) BUGS, and COMPATIBILITY.

How to Read the Synopsis

Here is the is synopsis for the `cp` command on Ubuntu Linux:

```
SYNOPSIS
   cp [OPTION]... [-T] SOURCE DEST
   cp [OPTION]... SOURCE... DIRECTORY
   cp [OPTION]... -t DIRECTORY SOURCE...
```

This tells the reader that there are three ways to use the command. Let's look at the first usage:

```
cp [OPTION]... [-T] SOURCE DEST
```

`[OPTION]` means the `cp` command can be followed by one or more optional *flags*. We can tell they're optional because of the square brackets, and we can tell that one or more are welcome because of the ellipsis (...). For example, the fact that `[-T]` is in square brackets, but after the ellipsis, means that it's optional, but if it's used, it must come after all the other options.

`SOURCE` refers to the source file or directory, and `DEST` to the destination file or directory. Their precise meanings are explained at the top of the `DESCRIPTION` section.

The other two usage examples can be read in similar ways. Note that to use the last one, the `-t` option is mandatory (because it isn't shown in square brackets).

The `DESCRIPTION` section starts with a few paragraphs explaining the command and its use, then expands on the possible options one by one:

```
     The following options are available:

     -a    Same as -pPR options. Preserves structure and attributes of
           files but not directory structure.

     -f    If the destination file cannot be opened, remove it and create
           a new file, without prompting for confirmation regardless of
           its permissions.  (The -f option overrides any previous -n
           option.)

           The target file is not unlinked before the copy.  Thus, any
           existing access rights will be retained.

      ...  ...
```

Finding Help on Specific Options

If we want to skip ahead to the option you're interested in, we can search for it using the slash key '/'. (This isn't part of the `man` command: it's a feature of `more`.) For example, to find out about `-t`, we can type `/-t` and press return. After that, we can use the 'n' key to navigate to the next match until we find the detailed information we need:

```
-t, --target-directory=DIRECTORY
     copy all SOURCE arguments into DIRECTORY
```

This means that this option has the short form `-t` and the long form `--target-directory` and that it takes an argument. Its meaning is to copy all the SOURCE arguments into DIRECTORY. Thus, we can give the destination explicitly instead of relying on having to place the directory at the end.

Limitations of Man Pages

Man pages can be useful for a quick confirmation of how to run a command, but they are not famous for being readable. If you can't find what you need in the man page—or you can't understand what you've found—try entering "unix command copy file" into your favorite search engine: it will often produce more helpful results.

You May Also Enjoy…

The explainshell.com[4] site does a great job of breaking complex Unix commands into parts and explaining what each does. Sadly, it doesn't work in reverse…

6.4 Permissions

Unix controls who can read, modify, and run files using *permissions*. We'll discuss how Windows handles permissions at the end of the section: the concepts are similar, but the rules are different.

Let's start with Nelle. She has a unique *user name*, `nnemo`, and a *user ID*, 1404.

[4] http://explainshell.com/

Why Integer IDs?

Why integers for IDs? Again, the answer goes back to the early 1970s. Character strings like `alan.turing` are of varying length, and comparing one to another takes many instructions. Integers, on the other hand, use a fairly small amount of storage (typically four characters), and can be compared with a single instruction. To make operations fast and simple, programmers often keep track of things internally using integers, then use a lookup table of some kind to translate those integers into user-friendly text for presentation. Of course, programmers being programmers, they will often skip the user-friendly string part and just use the integers, in the same way that someone working in a lab might talk about Experiment 28 instead of "the chronotypical alpha-response trials on anacondas".

Users can belong to any number of *groups*, each of which has a unique *group name* and numeric *group ID*. The list of who's in what group is usually stored in the file `/etc/group`. (If you're in front of a Unix machine right now, try running `cat /etc/group` to look at that file.)

Now let's look at files and directories. Every file and directory on a Unix computer belongs to one owner and one group. Along with each file's content, the operating system stores the numeric IDs of the user and group that own it.

The user-and-group model means that for each file every user on the system falls into one of three categories: the owner of the file, someone in the file's group, and everyone else.

For each of these three categories, the computer keeps track of whether people in that category can read the file, write to the file, or execute the file (i.e., run it if it is a program).

For example, if a file had the following set of permissions:

user

group

all

read

yes

yes

no

write

yes

no

no

execute

no

no

no

it would mean that:

- the file's owner can read and write it, but not run it;
- other people in the file's group can read it, but not modify it or run it; and
- everybody else can do nothing with it at all.

Let's look at this model in action. If we `cd` into the `labs` directory and run `ls -F`, it puts a `*` at the end of `setup`'s name. This is its way of telling us that `setup` is executable, i.e., that it's (probably) something the computer can run.

```
$ cd labs
$ ls -F

safety.txt    setup*     waiver.txt
```

Necessary But Not Sufficient

The fact that something is marked as executable doesn't actually mean it contains a program of some kind. We could easily mark this HTML file as executable using the commands that are introduced below. Depending on the operating system we're using, trying to "run" it will either fail (because it doesn't contain instructions the computer recognizes) or cause the operating system to open the file with whatever application usually handles it (such as a web browser).

Now let's run the command `ls -l`:

```
$ ls -l

-rw-rw-r-- 1 vlad bio  1158  2010-07-11 08:22 safety.txt
-rwxr-xr-x 1 vlad bio 31988  2010-07-23 20:04 setup
-rw-rw-r-- 1 vlad bio  2312  2010-07-11 08:23 waiver.txt
```

The `-l` flag tells `ls` to give us a long-form listing. It's a lot of information, so let's go through the columns in turn.

On the right side, we have the files' names. Next to them, moving left, are the times and dates they were last modified. Backup systems and other tools use this information in a variety of ways, but you can use it to tell when you (or anyone else with permission) last changed a file.

Next to the modification time is the file's size in bytes and the names of the user and group that owns it (in this case, `vlad` and `bio` respectively). We'll skip over the second column for now (the one showing `1` for each file) because it's the first column that we care about most. This shows the file's permissions, i.e., who can read, write, or execute it.

Let's have a closer look at one of those permission strings: `-rwxr-xr-x`. The first character tells us what type of thing this is: '-' means it's a regular file, while 'd' means it's a directory, and other characters mean more esoteric things.

The next three characters tell us what permissions the file's owner has. Here, the owner can read, write, and execute the file: `rwx`. The middle triplet shows us the group's permissions. If the permission is turned off, we see a dash, so `r-x` means "read and execute, but not write". The final triplet shows us what everyone who isn't the file's owner, or in the file's group, can do. In this case, it's 'r-x' again, so everyone on the system can look at the file's contents and run it.

To change permissions, we use the `chmod` command (whose name stands for "change mode"). Here's a long-form listing showing the permissions on the final grades in the course Vlad is teaching:

```
$ ls -l final.grd

-rwxrwxrwx 1 vlad bio  4215  2010-08-29 22:30 final.grd
```

Whoops: everyone in the world can read it—and what's worse, modify it! (They could also try to run the grades file as a program, which would almost certainly not work.)

The command to change the owner's permissions to `rw-` is:

```
$ chmod u=rw final.grd
```

The ‘u’ signals that we’re changing the privileges of the user (i.e., the file’s owner), and `rw` is the new set of permissions. A quick `ls -l` shows us that it worked, because the owner’s permissions are now set to read and write:

```
$ ls -l final.grd
```

```
-rw-rwxrwx 1 vlad bio  4215  2010-08-30 08:19 final.grd
```

Let’s run `chmod` again to give the group read-only permission:

```
$ chmod g=r final.grd
$ ls -l final.grd
```

```
-rw-r--rw- 1 vlad bio  4215  2010-08-30 08:19 final.grd
```

And finally, let’s give “all” (everyone on the system who isn’t the file’s owner or in its group) no permissions at all:

```
$ chmod a= final.grd
$ ls -l final.grd
```

```
-rw-r----- 1 vlad bio  4215  2010-08-30 08:20 final.grd
```

Here, the ‘a’ signals that we’re changing permissions for “all”, and since there’s nothing on the right of the “=”, “all”’s new permissions are empty.

We can search by permissions, too. Here, for example, we can use `-type f -perm -u=x` to find files that the user can execute:

```
$ find . -type f -perm -u=x
```

```
./tools/format
./tools/stats
```

Before we go any further, let’s run `ls -a -l` to get a long-form listing that includes directory entries that are normally hidden:

```
$ ls -a -l
```

```
drwxr-xr-x 1 vlad bio     0  2010-08-14 09:55 .
drwxr-xr-x 1 vlad bio  8192  2010-08-27 23:11 ..
-rw-rw-r-- 1 vlad bio  1158  2010-07-11 08:22 safety.txt
-rwxr-xr-x 1 vlad bio 31988  2010-07-23 20:04 setup
-rw-rw-r-- 1 vlad bio  2312  2010-07-11 08:23 waiver.txt
```

The permissions for `.` and `..` (this directory and its parent) start with a ‘d’. But look at the rest of their permissions: the ‘x’ means that “execute” is turned on. What does that mean? A directory isn’t a program—how can we “run” it?

In fact, ‘x’ means something different for directories. It gives someone the right to *traverse* the directory, but not to look at its contents. The distinction is subtle, so let’s have a look at an example. Vlad’s home directory has three subdirectories called `venus`, `mars`, and `pluto` (Figure 6.11).

Each of these has a subdirectory in turn called `notes`, and those sub-subdirectories contain various files. If a user’s permissions on `venus` are ‘r-x’, then if she tries to see the contents of `venus` and `venus/notes` using `ls`, the computer lets her see both. If her permissions on `mars` are just ‘r–’, then she is allowed to read the contents of both `mars` and `mars/notes`. But if her permissions on `pluto` are only ‘–x’, she cannot see what’s in the `pluto` directory: `ls pluto` will tell her she doesn’t have permission to view its contents. If she tries to look in `pluto/notes`, though, the computer will let her do that. She’s allowed to go through `pluto`, but not to look at what’s there. This trick gives people a way to make some of their directories visible to the world as a whole without opening up everything else.

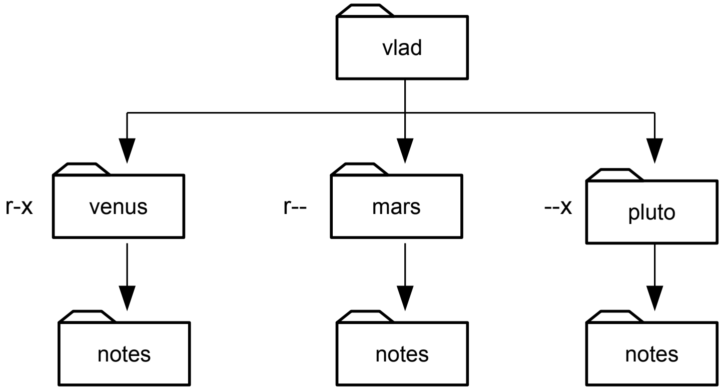


Figure 6.11: “Execute” Permission for Directories

What about Windows?

Those are the basics of permissions on Unix. As we said at the outset, though, things work differently on Windows. There, permissions are defined by *access control lists*, or ACLs. An ACL is a list of pairs, each of which combines a “who” with a “what”. For example, you could give the Mummy permission to append data to a file without giving him permission to read or delete it, and give Frankenstein permission to delete a file without being able to see what it contains.

This is more flexible that the Unix model, but it’s also more complex to administer and understand on small systems. (If you have a large computer system, *nothing* is easy to administer or understand.) Some modern variants of Unix support ACLs as well as the older read-write-execute permissions, but hardly anyone uses them.

6.5 Shell Variables

The shell is just a program, and like other programs, it has variables. Those variables control its execution, so by changing their values you can change how the shell and other programs behave.

Let’s start by running the command `set` and looking at some of the variables in a typical shell session:

```
$ set

COMPUTERNAME=TURING
HOME=/home/vlad
HOMEDRIVE=C:
HOSTNAME=TURING
HOSTTYPE=i686
NUMBER_OF_PROCESSORS=4
OS=Windows_NT
PATH=/Users/vlad/bin:/usr/local/git/bin:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin
PWD=/home/vlad
UID=1000
USERNAME=vlad
...
```

As you can see, there are quite a few—in fact, four or five times more than what's shown here. And yes, using `set` to *show* things might seem a little strange, even for Unix, but if you don't give it any arguments, it might as well show you things you *could* set.

Every variable has a name. By convention, variables that are always present are given uppercase names. All shell variables' values are strings, even those (like `UID`) that look like numbers. It's up to programs to convert these strings to other types when necessary. For example, if a program wanted to find out how many processors the computer had, it would convert the value of the `NUMBER_OF_PROCESSORS` variable from a string to an integer.

Similarly, some variables (like `PATH`) store lists of values. In this case, the convention is to use a colon ':' as a separator. If a program wants the individual elements of such a list, it's the program's responsibility to split the variable's string value into pieces.

The `PATH` Variable

Let's have a closer look at that `PATH` variable. Its value defines the shell's *search path*, i.e., the list of directories that the shell looks in for runnable programs when you type in a program name without specifying what directory it is in.

For example, when we type a command like `analyze`, the shell needs to decide whether to run `./analyze` or `/bin/analyze`. The rule it uses is simple: the shell checks each directory in the `PATH` variable in turn, looking for a program with the requested name in that directory. As soon as it finds a match, it stops searching and runs the program.

To show how this works, here are the components of `PATH` listed one per line:

```
/Users/vlad/bin
/usr/local/git/bin
/usr/bin
/bin
/usr/sbin
/sbin
/usr/local/bin
```

On our computer, there are actually three programs called `analyze` in three different directories: `/bin/analyze`, `/usr/local/bin/analyze`, and `/users/vlad/analyze`. Since the shell searches the directories in the order they're listed in `PATH`, it finds `/bin/analyze` first and runs that. Notice that it will *never* find the program `/users/vlad/analyze` unless we type in the full path to the program, since the directory `/users/vlad` isn't in `PATH`.

Showing the Value of a Variable

Let's show the value of the variable `HOME`:

```
$ echo HOME
```

```
HOME
```

That just prints "HOME", which isn't what we wanted (though it is what we actually asked for). Let's try this instead:

```
$ echo $HOME
```

```
/home/vlad
```

The dollar sign tells the shell that we want the *value* of the variable rather than its name. This works just like wildcards: the shell does the replacement *before* running the program we've asked for. Thanks to this expansion, what we actually run is `echo /home/vlad`, which displays the right thing.

Creating and Changing Variables

Creating a variable is easy—we just assign a value to a name using "=":

```
$ SECRET_IDENTITY=Dracula
$ echo $SECRET_IDENTITY

Dracula
```

To change the value, just assign a new one:

```
$ SECRET_IDENTITY=Camilla
$ echo $SECRET_IDENTITY

Camilla
```

If we want to set some variables automatically every time we run a shell, we can put commands to do this in a file called `.bashrc` in our home directory. (The '.' character at the front prevents `ls` from listing this file unless we specifically ask it to using `-a`: we normally don't want to worry about it. The "rc" at the end is an abbreviation for "run control", which meant something really important decades ago, and is now just a convention everyone follows without understanding why.)

For example, here are two lines in `/home/vlad/.bashrc`:

```
export SECRET_IDENTITY=Dracula
export TEMP_DIR=/tmp
export BACKUP_DIR=$TEMP_DIR/backup
```

These three lines create the variables `SECRET_IDENTITY`, `TEMP_DIR`, and `BACKUP_DIR`, and export them so that any programs the shell runs can see them as well. Notice that `BACKUP_DIR`'s definition relies on the value of `TEMP_DIR`, so that if we change where we put temporary files, our backups will be relocated automatically.

While we're here, it's also common to use the `alias` command to create shortcuts for things we frequently type. For example, we can define the alias `backup` to run `/bin/zback` with a specific set of arguments:

```
alias backup=/bin/zback -v --nostir -R 20000 $HOME $BACKUP_DIR
```

As you can see, aliases can save us a lot of typing, and hence a lot of typing mistakes. You can find interesting suggestions for other aliases and other bash tricks by searching for "sample bashrc" in your favorite search engine.

6.6 Working Remotely

Let's take a closer look at what happens when we use the shell on a desktop or laptop computer. The first step is to log in so that the operating system knows who we are and what we're allowed to do. We do this by typing our username and password; the operating system checks those values against its records, and if they match, runs a shell for us.

As we type commands, the 1's and 0's that represent the characters we're typing are sent from the keyboard to the shell. The shell displays those characters on the screen to represent what we type, and then, if what we typed was a command, the shell executes it and displays its output (if any).

What if we want to run some commands on another machine, such as the server in the basement that manages our database of experimental results? To do this, we have to first log in to that machine. We call this a *remote login*, and the other computer a remote computer. Once we do this, everything we type is passed to a shell running on the remote computer. That shell runs those commands on our behalf, just as a local shell would, then sends back output for our computer to display.

The tool we use to log in remotely is the [secure shell)(#g:secure-shell), or SSH. In particular, the command `ssh username@computer` runs SSH and connects to the remote computer we have specified. After we log in, we can use the remote shell to use the remote computer's files and directories. Typing `exit` or Control-D terminates the remote shell and returns us to our previous shell.

In the example below, the remote machine's command prompt is `moon>` instead of just `$`. To make it clearer which machine is doing what, we'll indent the commands sent to the remote machine and their output.

```
$ pwd

/users/vlad

$ ssh vlad@moon.euphoric.edu
Password: ********

    moon> hostname

    moon

    moon> pwd

    /home/vlad

    moon> ls -F

    bin/     cheese.txt   dark_side/   rocks.cfg

    moon> exit

$ pwd

/users/vlad
```

The secure shell is called "secure" to contrast it with an older program called `rsh`, which stood for "remote shell". Back in the day, when everyone trusted each other and knew every chip in their computer by its first name, people didn't encrypt anything except the most sensitive information when sending it over a network. However, that meant that villains could watch network traffic, steal usernames and passwords, and use them for all manner of nefarious purposes. SSH was invented to prevent this (or at least slow it down). It uses several sophisticated, and heavily tested, encryption protocols to ensure that outsiders can't see what's in the messages going back and forth between different computers.

`ssh` has a companion program called `scp`, which stands for "secure copy". It allows us to copy files to or from a remote computer using the same kind of connection as SSH. The command's name combines `cp`'s and `ssh`'s, and so does its operation. To copy a file, we specify the source and destination paths, either of which may include computer names. If we leave out a computer name, `scp` assumes we mean the machine we're running on. For example, this command copies our latest results to the backup server in the basement, printing out its progress as it does so:

```
$ scp results.dat vlad@backupserver:backups/results-2011-11-11.dat
Password: ********

results.dat             100%  9  1.0 MB/s 00:00
```

Copying a whole directory is similar: we just use the `-r` option to signal that we want copying to be recursive. For example, this command copies all of our results from the backup server to our laptop:

```
$ scp -r vlad@backupserver:backups ./backups
Password: ********

results-2011-09-18.dat             100%  7  1.0 MB/s 00:00
results-2011-10-04.dat             100%  9  1.0 MB/s 00:00
results-2011-10-28.dat             100%  8  1.0 MB/s 00:00
results-2011-11-11.dat             100%  9  1.0 MB/s 00:00
```

Here's one more thing SSH can do for us. Suppose we want to check whether we have already created the file `backups/results-2011-11-12.dat` on the backup server. Instead of logging in and then typing `ls`, we could do this:

```
$ ssh vlad@backupserver "ls results"
Password: ********

results-2011-09-18.dat  results-2011-10-28.dat
results-2011-10-04.dat  results-2011-11-11.dat
```

SSH takes the argument after our remote username and passes them to the shell on the remote computer. (We have to put quotes around it to make it look like a single argument.) Since those arguments are a legal command, the remote shell runs `ls results` for us and sends the output back to our local shell for display.

> **All Those Passwords**
>
> Typing our password over and over again is annoying, especially if the commands we want to run remotely are in a loop. To remove the need to do this, we can create an *authentication key* to tell the remote machine that it should always trust us. We discuss authentication keys in our intermediate lessons.

6.7 Exceptions

Assertions help us catch errors in our code, but things can go wrong for other reasons, like missing or badly-formatted files. Most modern programming languages allow programmers to use *exceptions* to separate what the program should do if everything goes right from what it should do if something goes wrong. Doing this makes both cases easier to read and understand.

For example, here's a small piece of code that tries to read parameters and a grid from two separate files, and reports an error if either goes wrong:

```
try:
    params = read_params(param_file)
    grid = read_grid(grid_file)
except:
    log.error('Failed to read input file(s)')
    sys.exit(ERROR)
```

We join the normal case and the error-handling code using the keywords `try` and `except`. These work together like `if` and `else`: the statements under the `try` are what should happen if everything works, while the statements under `except` are what the program should do if something goes wrong.

We have actually seen exceptions before without knowing it, since by default, when an exception occurs, Python prints it out and halts our program. For example, trying to open a nonexistent file triggers a type of exception called an `IOError`, while trying to access a list element that doesn't exist causes an `IndexError`:

```
open('nonexistent-file.txt', 'r')
```

```
---------------------------------------------------------------------------
IOError                                   Traceback (most recent call last)

<ipython-input-13-58cbde3dd63c> in <module>()
----> 1 open('nonexistent-file.txt', 'r')

IOError: [Errno 2] No such file or directory: 'nonexistent-file.txt'
```

```
values = [0, 1, 2]
print values[999]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)

<ipython-input-14-7fed13afc650> in <module>()
1 values = [0, 1, 2]
----> 2 print values[999]

IndexError: list index out of range
```

We can use `try` and `except` to deal with these errors ourselves if we don't want the program simply to fall over:

```
try:
    reader = open('nonexistent-file.txt', 'r')
except IOError:
    print 'Whoops!'
```

```
Whoops!
```

When Python executes this code, it runs the statement inside the `try`. If that works, it skips over the `except` block without running it. If an exception occurs inside the `try` block, though, Python compares the type of the exception to the type specified by the `except`. If they match, it executes the code in the `except` block.

`IOError` is the particular kind of exception Python raises when there is a problem related to input and output, such as files not existing or the program not having the permissions it needs to read them. We can put as many lines of code in a `try` block as we want, just as we can put many statements under an `if`. We can also handle several different kinds of errors afterward. For example, here's some code to calculate the entropy at each point in a grid:

```
try:
    params = read_params(param_file)
    grid = read_grid(grid_file)
    entropy = lee_entropy(params, grid)
    write_entropy(entropy_file, entropy)
```

```
except IOError:
    report_error_and_exit('IO error')
except ArithmeticError:
    report_error_and_exit('Arithmetic error')
```

Python tries to run the four functions inside the try as normal. If an error occurs in any of them, Python immediately jumps down and tries to find an except of the corresponding type: if the exception is an IOError, Python jumps into the first error handler, while if it's an ArithmeticError, Python jumps into the second handler instead. It will only execute one of these, just as it will only execute one branch of a series of if/elif/else statements.

This layout has made the code easier to read, but we've lost something important: the message printed out by the IOError branch doesn't tell us which file caused the problem. We can do better if we capture and hang on to the object that Python creates to record information about the error:

```
try:
    params = read_params(param_file)
    grid = read_grid(grid_file)
    entropy = lee_entropy(params, grid)
    write_entropy(entropy_file, entropy)
except IOError as err:
    report_error_and_exit('Cannot read/write' + err.filename)
except ArithmeticError as err:
    report_error_and_exit(err.message)
```

If something goes wrong in the try, Python creates an exception object, fills it with information, and assigns it to the variable err. (There's nothing special about this variable name—we can use anything we want.) Exactly what information is recorded depends on what kind of error occurred; Python's documentation describes the properties of each type of error in detail, but we can always just print the exception object. In the case of an I/O error, we print out the name of the file that caused the problem. And in the case of an arithmetic error, printing out the message embedded in the exception object is what Python would have done anyway.

So much for how exceptions work: how should they be used? Some programmers use try and except to give their programs default behaviors. For example, if this code can't read the grid file that the user has asked for, it creates a default grid instead:

```
try:
    grid = read_grid(grid_file)
except IOError:
    grid = default_grid()
```

Other programmers would explicitly test for the grid file, and use if and else for control flow:

```
if file_exists(grid_file):
    grid = read_grid(grid_file)
else:
    grid = default_grid()
```

It's mostly a matter of taste, but we prefer the second style. As a rule, exceptions should only be used to handle exceptional cases. If the program knows how to fall back to a default grid, that's not an unexpected event. Using if and else instead of try and except sends different signals to anyone reading our code, even if they do the same thing.

Novices often ask another question about exception handling style, but before we address it, there's something in our example that you might not have noticed. Exceptions can actually be thrown a long way: they don't have to be handled immediately. Take another look at this code:

```
try:
    params = read_params(param_file)
    grid = read_grid(grid_file)
    entropy = lee_entropy(params, grid)
    write_entropy(entropy_file, entropy)
except IOError as err:
    report_error_and_exit('Cannot read/write' + err.filename)
except ArithmeticError as err:
    report_error_and_exit(err.message)
```

The four lines in the try block are all function calls. They might catch and handle exceptions themselves, but if an exception occurs in one of them that *isn't* handled internally, Python looks in the calling code for a matching except. If it doesn't find one there, it looks in that function's caller, and so on. If we get all the way back to the main program without finding an exception handler, Python's default behavior is to print an error message like the ones we've been seeing all along.

This rule is the origin of the rule throw low, catch high[5]. There are many places in our program where an error might occur. There are only a few, though, where errors can sensibly be handled. For example, a linear algebra library doesn't know whether it's being called directly from the Python interpreter, or whether it's being used as a component in a larger program. In the latter case, the library doesn't know if the program that's calling it is being run from the command line or from a GUI. The library therefore shouldn't try to handle or report errors itself, because it has no way of knowing what the right way to do this is. It should instead just raise an exception, and let its caller figure out how best to handle it.

Finally, we can raise exceptions ourselves if we want to. In fact, we *should* do this, since it's the standard way in Python to signal that something has gone wrong. Here, for example, is a function that reads a grid and checks its consistency:

```
def read_grid(grid_file):
    data = read_raw_data(grid_file)
    if not grid_consistent(data):
        raise Exception('Inconsistent grid: ' + grid_file)
    result = normalize_grid(data)
    return result
```

The raise statement creates a new exception with a meaningful error message. Since read_grid itself doesn't contain a try/except block, this exception will always be thrown up and out of the function, to be caught and handled by whoever is calling read_grid. We can define new types of exceptions if we want to. And we should, so that errors in our code can be distinguished from errors in other people's code. However, this involves classes and objects, which is outside the scope of these lessons.

6.8 Numbers

Let's start by looking at how numbers are stored. If we only have the two digits 0 and 1, the natural way to store a positive integer is to use base 2, so 1001_2 is $(12^3) + (02^2) + (02^1) + (12^0) = 9_{10}$. It's equally natural to extend this scheme to negative numbers by reserving one bit for the sign. If, for example, we use 0 for positive numbers and 1 for those that are negative, $+9^{10}$ would be 01001^2 and -9^{10} would be 11001^{10}.

[5] ../rules.html#throw-low-catch-high

There are two problems with this. The first is that this scheme gives us two representations for zero (00000_2 and 10000_2). This isn't necessarily fatal, but any claims this scheme has to being "natural" disappear when we have to write code like:

```
if (length != +0) and (length != -0)
```

As for the other problem, it turns out that the circuits needed to do addition and other arithmetic on this *sign and magnitude representation* are more complicated than the hardware needed for another called *two's complement*. Instead of mirroring positive values, two's complement rolls over when going below zero, just like a car's odometer. If we're using four bits per number, so that 0_{10} is 0000_2, then -1_{10} is 1111_2. -2_{10} is 1110_2, -3_{10} is 1101_2, and so on until we reach the most negative number we can represent, 1000_2, which is -8. Our representation then wraps around again, so that 0111_2 is 7_{10}.

This scheme isn't intuitive, but it solves sign and magnitude's "double zero" problem, and the hardware to handle it is faster and cheaper. As a bonus, we can still tell whether a number is positive or negative by looking at the first bit: negative numbers have a 1, positives have a 0. The only odd thing is its asymmetry: because 0 counts as a positive number, numbers go from -8 to 7, or -16 to 15, and so on. As a result, even if `x` is a valid number, `-x` may not be.

Finding a good representation for real numbers (called *floating point numbers*, since the decimal point can move around) is a much harder problem. The root of the problem is that we cannot represent an infinite number of real values with a finite set of bit patterns. And unlike integers, no matter what values we *do* represent, there will be an infinite number of values between each of them that we can't.

Floating point numbers are usually represented using sign, magnitude, and an exponent. In a 32-bit word, the IEEE 754 standard calls for 1 bit of sign, 23 bits for the magnitude (or *mantissa*), and 8 bits for the exponent. To illustrate the problems with floating point, we'll use a much dumber representation: we'll only worry about positive values without fractional parts, and we'll only use 3 for the magnitude and 2 for the exponent.

		Exponent			
		00	01	10	11
	000	0	0	0	0
	001	1	2	4	8
	010	2	4	8	16
Mantissa	011	3	6	12	24
	100	4	8	16	32
	101	5	10	20	40
	110	6	12	24	48
	111	7	14	28	56

The table above shows the values that we can represent this way. Each one is the mantissa times two to the exponent. For example, the decimal values 48 is binary 110 times 2 to the binary 11 power, which is 6 times 2 to the third, or 6 times 8. (Note that real floating point representations like the IEEE 754 standard don't have the redundancy shown in this table, but that doesn't affect our argument.)

The first thing you should notice is that there are a lot of values we *can't* store. We can do 8 and 10, for example, but not 9. This is exactly like the problems hand calculators have with fractions like 1/3: in decimal, we have to round that to 0.3333 or 0.3334.

But if this scheme has no representation for 9, then 8+1 must be stored as either 8 or 10. This raises an interesting question: if 8+1 is 8, what is 8+1+1? If we add from the left, 8+1 is 8, plus another 1 is 8 again. If we add from the right, though, 1+1 is 2, and 2+8 is 10. Changing the order of operations can make the difference between right and wrong. There's no randomness involved—a particular order of operations will always produce the same result—but as the number of steps increases, so too does the difficulty of figuring out what the best order is.

This is the sort of problem that numerical analysts spend their time on. In this case, if we sort the values we're adding, then add from smallest to largest, it gives us a better chance of getting the best possible answer. In other situations, like inverting a matrix, the rules are much more complicated.

Here's another observation about our uneven number line: the spacing between the values we can represent is uneven, but the relative spacing between each set of values stays the same, i.e., the first group is separated by 1, then the separation becomes 2, then 4, then 8, so that the ratio of the spacing to the values stays roughly constant. This happens because we're multiplying the same fixed set of mantissas by ever-larger exponents, and it points us at a couple of useful definitions.

The *absolute error* in some approximation is simply the absolute value of the difference between the actual value and the approximation. The *relative error*, on the other hand, is the ratio of the absolute error to the value we're approximating. For example, if we're off by 1 in approximating 8+1 and 56+1, the absolute error is the same in both cases, but the relative error in the first case is 1/9 = 11%, while the relative error in the second case is only 1/57 = 1.7%. When we're thinking about floating point numbers, relative error is almost always more useful than absolute error. After all, it makes little sense to say that we're off by a hundredth when the value in question is a billionth.

To see why this matters, let's have a look at a little program:

```
nines = []
sums = []
current = 0.0
for i in range(1, 10):
    num = 9.0 / (10.0 ** i)
    nines.append(num)
    current += num
    sums.append(current)

for i in range(len(nines)):
    print '%.18f %.18f' % (nines[i], sums[i])
```

The loop runs over the integers from 1 to 9 inclusive. Using those values, we create the numbers 0.9, 0.09, 0.009, and so on, and put them in the list `vals`. We then calculate the sum of those numbers. Clearly, this should be 0.9, 0.99, 0.999, and so on. But is it?

```
1   0.900000000000000022   0.900000000000000022
2   0.089999999999999997   0.989999999999999991
3   0.008999999999999999   0.998999999999999999
4   0.000900000000000000   0.999900000000000011
5   0.000090000000000000   0.999990000000000046
6   0.000009000000000000   0.999999000000000082
7   0.000000900000000000   0.999999900000000053
8   0.000000090000000000   0.999999990000000061
9   0.000000009000000000   0.999999999000000028
```

Here are our answers. The first column is the loop index; the second, what we actually got when we tried to calculate 0.9, 0.09, and so on, and the third is the cumulative sum.

The first thing you should notice is that the very first value contributing to our sum is already slightly off. Even with 23 bits for a mantissa, we cannot exactly represent 0.9 in base 2, any more than we can exactly represent 1/3 in base 10. Doubling the size of the mantissa would reduce the error, but we can't ever eliminate it.

The second thing to notice is that our approximation to 0.0009 actually appears accurate, as do all of the approximations after that. This may be misleading, though: after all, we've only printed things out to 18 decimal places. As for the errors in the last few digits of the sums, there doesn't appear to be any regular pattern in the way they increase and decrease.

This phenomenon is one of the things that makes testing scientific programs hard. If a function uses floating point numbers, what do we compare its result to if we want to check that it's working correctly? If we compared the sum of the first few numbers in `vals` to what it's supposed to be, the answer could be `False`, even if we're initializing the list with the right values, and calculating the sum correctly. This is a genuinely hard problem, and no one has a good generic answer. The root of our problem is that we're using approximations, and each approximation has to be judged on its own merits.

There are things you can do, though. The first rule is, compare what you get to analytic solutions whenever you can. For example, if you're looking at the behavior of drops of liquid helium, start by checking your program's output on a stationary spherical drop in zero gravity. You should be able to calculate the right answer in that case, and if your program doesn't work for that, it probably won't work for anything else.

The second rule is to compare more complex versions of your code to simpler ones. If you're about to replace a simple algorithm for calculating heat transfer with one that's more complex, but hopefully faster, don't throw the old code away. Instead, use its output as a check on the correctness of the new code. And if you bump into someone at a conference who has a program that can calculate some of the same results as yours, swap data sets: it'll help you both.

The third rule is, never use == (or `!=`) on floating point numbers, because two numbers calculated in different ways will probably not have exactly the same bits. Instead, check to see whether two values are within some tolerance, and if they are, treat them as equal. Doing this forces you to make your tolerances explicit, which is useful in its own right (just as putting error bars on experimental results is useful).

6.9 Why I Teach

Several independent studies have confirmed that my daughter Madeline (Figure 6.12) is the cutest child on the planet (plus or minus 5%, 19 times out of 20). She means the world to me, but by the time she's grown, the world is going to be dealing with the consequences of our generation's short-sightedness. Climate change, resource shortages, drug-resistant diseases—the list is a long one, and the only things that will save us are more science and more courage.

That's why I teach. I teach because I need your help to make this world fit for her to live in. I teach because I need you to discover more and invent faster so that the world she inherits from you will be better than the one you're inheriting from my generation. Science is not just the greatest adventure of our time: it is, quite literally, a matter of life and death, and I hope that what we have taught you will help you do it better.

Thank you for your time: may you always see the world through the eyes of a child.

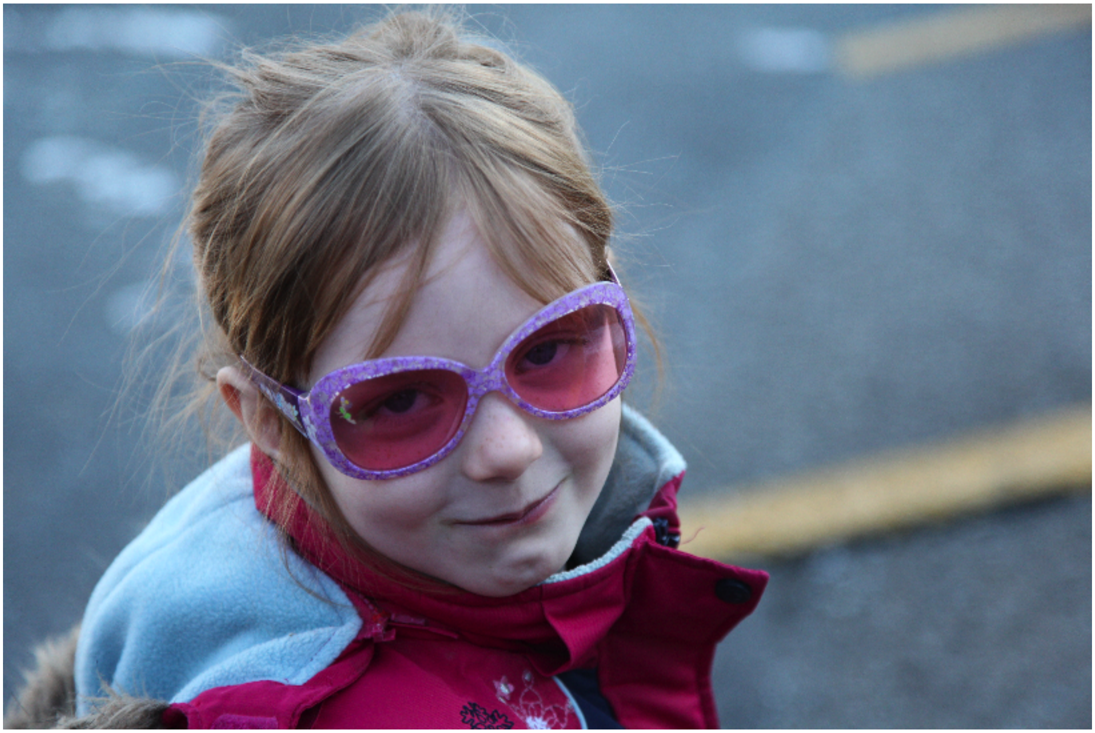


Figure 6.12: Madeleine

Chapter 7

Instructor's Guide

As a species, we know a lot about how brains learn, how effective various teaching practices are, and how society's needs and expectations shape how and how well we learn. As individuals, though, most people who teach at college and university either don't know this knowledge exists, or haven't incorporated it into their teaching. It's as if only doctors knew about the connection between smoking and cancer.

The best guide to evidence-based learning we have found is *How Learning Works: Seven Research-Based Principles for Smart Teaching* (see the bibliography for the full citation). Its advice is based in equal parts on theory, research, and experience. While some of the recommendations may seem banal, the full-length explanations in the book itself are anything but.

We try to incorporate these ideas into our teaching, and into our instructor training course[1]. If you are interested in taking part, please get in touch.

7.1 General Advice

This is a placeholder for general notes about teaching. For up-to-date information about software installation and configuration problems, and their solutions, see this wiki page[2].

Teaching Notes

- For bootcamps that extend over more than two days (e.g., four afternoons spread over two weeks), it's a good idea to email the learners at the end of each day with a summary of what was taught (with links to the relevant online notes). This allows absent learners to catch up before the next session, and is also a great opporunity to present the lessons of the day in the context of the entire bootcamp.
- Point learners at http://software-carpentry.org/v5/[3], which is the permanent home of the current learning materials, and at http://software-carpentry.org/v4/[4], which is where our previous materials live. The former corresponds to what they're being taught; the latter covers more

[1] http://teaching.software-carpentry.org

[2] https://github.com/swcarpentry/bc/wiki/Configuration-Problems-and-Solutions

[3] http://software-carpentry.org/v5/

[4] http://software-carpentry.org/v4/

ground in video as well as in slides and prose. They should also be direct to Software Carpentry's FAQ[5].

- Explain that the lesson materials can all be freely re-mixed and re-used under the Creative Commons - Attribution[6] (CC-BY) license, provided people cite us as the original source (e.g., provide a link back to our site). However, Software Carpentry's name and logo are trademarked, and cannot be used without our permission. We normally grant this to any class that (a) covers our core topics and (b) has at least one badged instructor on the teaching roster, but are happy to discuss specifics.
- Plan for the first 30-60 minutes of the bootcamp to be spent on installation and setup, because it's going to happen anyway. Running a pre-bootcamp "help desk" doesn't really affect this: the people who are most likely to have installation problems probably won't show up. (We fantasize occasionally about turning people away if they haven't installed software, or at least downloaded the installers, but in practice it's hard to do.)
- Emphasise that good software development skills contribute to productive, reproducible, reusable research.
- Have learners post a red sticky note on their laptop whenever they have a question or need help. Have them take down their sticky notes at the start of each practical exercise, and then post a green one when they're done (and a red one when they need help).
- At lunch and again at the end of the day, ask learners to write one good point (i.e., something they learned or enjoyed) on their green sticky note and one bad point (i.e., something that didn't work, that they didn't understand, or that they already knew) on their red one. It only takes a couple of minutes to sort through these, and it's a quick way to find out how things are actually going.
- At the very end of the bootcamp, ask learners to alternately give one good point or one bad one about the entire bootcamp. Write these down on the whiteboard as they come in, and do not allow repeats (i.e., every point has to be new). The first few negative points will usually be pretty innocuous; after those have been exhausted, you will start to get the real feedback.
- As a variation on the red/green sticky notes, make little name tents out of red and green paper, held together with name tag labels. The learners write their names on the name tags, and prop the tents either green side up or red side up depending on the feedback they want to give about the lesson being too fast or too slow.
- Back up the material with your own anecdotes, experiences and evidence—it makes you more credible, helps learners understand how to apply what you're teaching to their own problems, and prevents the lectures from becoming too dry.
- Keep a running list of the commands encountered so far in the lesson in the Etherpad or on a whiteboard adjacent to the projection screen. Encourage learners (particularly ones who already know the material and might otherwise get bored) to take notes in the Etherpad as well. This reduces the effort per learner, gives you a chance to see what they think you're saying, and provides a record after the bootcamp of what was actually taught.
- When the co-instructor isn't teaching, she can answer questions on the Etherpad and update it with the key points made by the instructor (along with commands and any related points the instructor may not have mentioned). It's less disruptive to the "live" instructor than interjecting with these points, but allows the attendees to get the shared expertise from both instructors.

[5] http://software-carpentry.org/faq.html

[6] ../../LICENSE.html

- For bootcamps that extend over more than two days (e.g. four afternoons spread over two weeks), it's a good idea to email the learners at the end of each day with a summary of what was taught (with links to the relevant online notes). Not only does this allow absent learners to catch up before the next session, it's also a great opporunity to present the lessons of the day in the context of the entire bootcamp.
- The long-form notes are intended as a script for instructors and as self-study material for learners. Do *not* show these notebooks to learners: instead, start with a blank notebook when teaching and add code as you go. This helps prevent you from racing ahead of learners and makes it easier for you to improvise in response to their questions.
- Point learners at the online versions of the long-form notes (either on your bootcamp's home page or at http://software-carpentry.org/v5/[7] *after* the lesson is done: if you do it before the lesson, they'll try to read the notes while you're trying to talk.
- If you're really keen, keep the SVG's of the diagrams handy in the directory where you're doing your teaching so that you can include them in your notebooks by adding an `<img src="novice/teaching/...">` element to a Markdown cell (or just display them in your browser). Most people don't ever actually do this though, either because they forget to or because they have a whiteboard or flipchart handy.
- There are (at least) three ways to get data files to learners at the start of a lesson:
    1. Create a zip file, add it to your bootcamp's repository, and put a link to it in your bootcamp's `index.html` page so that they can click, download, and unzip. This uses something everyone already understands, but does assume they know how to navigate from their download directory to their working (lesson) directory, which is often not the case.
    2. Create a throwaway Git repository on GitHub and tell them to run one command to clone it at the start of class. This (usually) works even if they've never used Git, and as a bonus, lets you identify people who (are going to) have Git problems early.
    3. Paste the data into an Etherpad for learners to copy. As a bonus, this lets you identify people who (are going to) have trouble using a text editor early.
- If you are using multiple windows (e.g., a command window and an editor window) make sure they are both large enough to be visible by all attendees. Remember to pause when switching from one window to the other so that learners don't become confused. If possible, use different background colors for different text windows to make it easier for learners to tell them apart (but keep in mind red-green and blue-yellow color blindness).
- As you type at the command line, read out what you're typing. Remember that most learners can only go half as fast as you, because they have to watch you type then type it all in again themselves.

Pitfalls

Instructing at a bootcamp isn't trivial. The most important thing is to remember that no lesson plan survives contact with the audience. Whether it's the network going down or the sudden realization that many of your learners *don't* know how to use SSH, you will frequently need to improvise. And even when there aren't hiccups like those, try your best to adjust your pace and examples based on learners' questions, puzzled looks, and sighs of impatience.

Allow enough time for setup.

[7] http://software-carpentry.org/v5/

In almost all cases, learners want to use their own laptops during bootcamps so that they leave with a working environment set up. Even if you ask attendees to prepare beforehand, and give them detailed instructions, some will not have time, or will have run into problems that they're not yet able to fix. You should therefore schedule at least 20 minutes for *checking the learners' machines* at the beginning of the first day. Some bootcamps start early on the first day to allow time to troubleshoot setup problems.

Don't ignore your learners.

You're not there to reproduce one of our online videos in person: you're there to interact with people so that they get a better learning experience. You shouldn't ever go more than two or three minutes without asking a question (and listening to the answer), and if it has been 15 minutes since any of your learners asked one, odds are you've either lost them or are boring them.

Don't bore your learners.

Your audience will never care more about what you're teaching than you appear to, so if they get the feeling you're not interested in it, they won't be either. This does *not* mean you have to shout, crack three jokes a minute, or harangue them about how this stuff is really, really important, but you do owe it to your audience to show up mentally as well as physically.

Don't be all talk, no action.

The more time folks spend with their hands on the keyboards doing exercises, the more time they're actually paying attention. The students have their computers in front of them: if you talk for more than five minutes without asking them to use their computers, they'll do so anyway—on Facebook.

Don't use magic.

Typing too fast, using shortcuts or commands learners haven't seen yet—basically, any time you say, "Don't worry about this just now," or they say, "Wait, how did you do that?" or, "Can you please slow down, I can't keep up," you're no longer actually helping them.

Don't ignore feedback.

The feedback you get from learners on sticky notes or through surveys is pointless if you don't pay attention to it (or worse, if you explain it away). There's no point collecting feedback during and after each bootcamp if you don't change what and how you teach to reflect it.

Tell learners "why".

Most of our learners are graduate students in science and engineering, so they know what evidence looks like, and why working practices should be evidence-based. That doesn't mean you have to have the whole of empirical software engineering at your fingertips, but please do read *Facts and Fallacies of Software Engineering*[8] and sprinkle a few of the findings it quotes into your lessons.

Don't show them the forest but not the trees.

The things we teach reinforce each other, so tie them together at every opportunity. Point out that connecting things with pipes in the shell is like chaining functions together, or that they can use a shell script to re-run a bunch of different tests before committing to version control, and so on. If possible, take 15 minutes or so each day to show them how you use these tools in your day-to-day work.

Don't underestimate setup requirements.

Do you have enough power outlets? (Are you sure?) Do you have enough bandwidth to handle fifty people hitting your version control repository at the same time? (How do you know?) Can everyone actually log in? Are the washrooms unlocked? Does campus security know you're using the room over the weekend?

[8] http://www.amazon.com/Facts-Fallacies-Software-Engineering-Robert/dp/0321117425/

Don't let your learners ignore each other.

Software Carpentry bootcamps are a great networking opportunity for our learners (and for us, too). Get to know your learners by name, have them work in pairs, and get them to mix up the pairs at least a couple of times. Encourage them to chat to one another at coffee breaks and lunch, and to get a pizza or some curry together for dinner on the first day.

Relax.

Something always fails to install for someone (or they fail to install anything at all), or a bunch of learners are accidentally locked out of the building after lunch, or whoever was supposed to drop off power bars didn't. Roll with it, and remember to laugh (even if it's a bit hysterically).

7.2 The Unix Shell

Many people have questioned whether we should still teach the shell. After all, anyone who wants to rename several thousand data files can easily do so interactively in the Python interpreter, and anyone who's doing serious data analysis is probably going to do most of their work inside the IPython Notebook or R Studio. So why teach the shell?

The first answer is, "Because so much else depends on it." Installing software, configuring your default editor, and controlling remote machines frequently assume a basic familiarity with the shell, and with related ideas like standard input and output. Many tools also use its terminology (for example, the `%ls` and `%cd` magic commands in IPython).

The second answer is, "Because it's an easy way to introduce some fundamental ideas about how to use computers." As we teach people how to use the Unix shell, we teach them that they should get the computer to repeat things (via tab completion, `!` followed by a command number, and `for` loops) rather than repeating things themselves. We also teach them to take things they've discovered they do frequently and save them for later re-use (via shell scripts), to give things sensible names, and to write a little bit of documentation (like comment at the top of shell scripts) to make their future selves' lives better.

Finally, and perhaps most importantly, teaching people the shell lets us teach them to think about programming in terms of function composition. In the case of the shell, this takes the form of pipelines rather than nested function calls, but the core idea of "small pieces, loosely joined" is the same.

All of this material can be covered in three hours as long as learners using Windows do not run into roadblocks such as:

- not being able to figure out where their home directory is (particularly if they're using Cygwin);
- not being able to run a plain text editor; and
- the shell refusing to run scripts that include DOS line endings.

Teaching Notes

- Have learners open a shell and then do `whoami`, `pwd`, and `ls`. Then have them create a directory called `bootcamp` and `cd` into it, so that everything else they do during the lesson is unlikely to harm whatever files they already have.
- Get them to run an editor and save a file in their `bootcamp` directory as early as possible. Doing this is usually the biggest stumbling block during the entire lesson: many will try to run the same editor as the instructor (which may leave them trapped in the awful nether hell that is Vim), or will not know how to navigate to the right directory to save their file, or will run a

word processor rather than a plain text editor. The quickest way past these problems is to have more knowledgeable learners help those who need it.

- Tab completion sounds like a small thing: it isn't. Re-running old commands using `!123` or `!wc` isn't a small thing either, and neither are wildcard expansion and `for` loops. Each one is an opportunity to repeat one of the big ideas of Software Carpentry: if the computer *can* repeat it, some programmer somewhere will almost certainly have built some way for the computer *to* repeat it.
- Building up a pipeline with four or five stages, then putting it in a shell script for re-use and calling that script inside a `for` loop, is a great opportunity to show how "seven plus or minus two" connects to programming. Once we have figured out how to do something moderately complicated, we make it re-usable and give it a name so that it only takes up one slot in working memory rather than several. It is also a good opportunity to talk about exploratory programming: rather than designing a program up front, we can do a few useful things and then retroactively decide which are worth encapsulating for future re-use.
- We have to leave out many important things because of time constraints, including file permissions, job control, and SSH. If learners already understand the basic material, this can be covered instead using the online lessons as guidelines. These limitations also have follow-on consequences:
- It's hard to discuss `#!` (shebang) wihtout first discussing permissions, which we don't do.
- Installing Bash and a reasonable set of Unix commands on Windows always involves some fiddling and frustration. Please see the latest set of installation guidelines for advice, and try it out yourself *before* teaching a class.
- On Windows, it appears that:

  ```
  $ cd
  $ cd Desktop
  ```

  will always put someone on their desktop. Have them create the example directory for the shell exercises there so that they can find it easily and watch it evolve.

Windows

Installing Bash and a reasonable set of Unix commands on Windows always involves some fiddling and frustration. Please see the latest set of installation guidelines for advice, and try it out yourself *before* teaching a class. Options we have explored include:

1. msysGit[9] (also called "Git Bash"),
2. Cygwin[10],
3. using a desktop virtual machine, and
4. having learners connect to a remote Unix machine (typically a VM in the cloud).

Cygwin was the preferred option until mid-2013, but once we started teaching Git, msysGit proved to work better. Desktop virtual machines and cloud-based VMs work well for technically sophisticated learners, and can reduce installation and configuration at the start of the bootcamp, but:

1. they don't work well on underpowered machines,
2. they're confusing for novices (because simple things like copy and paste work differently),

[9] http://msysgit.github.io/
[10] http://www.cygwin.com/

3. learners leave the workshop without a working environment on their operating system of choice, and
4. learners may show up without having downloaded the VM or the wireless will go down (or become congested) during the lesson.

Whatever you use, please *test it yourself* on a Windows machine *before* your bootcamp: things may always have changed behind your back since your last bootcamp. And please also make use of our Windows setup helper.

7.3 Version Control with Git

Version control might be the most important topic we teach, but Git is definitely the most complicated tool. However, GitHub presently dominates the open software forge landscape, so we have to help novices learn just enough Git to feel they can and should learn more on their own.

This is why we don't teach branching: while it is a power tool in the hands of a knowledgeable user, it is an extra cognitive burden for someone who is new to the idea of version control. This is also why we don't get into hashes and commit objects with novices, but try instead to convince them that version control is:

1. a better backup system;
2. a better Dropbox; and
3. a better way to collaborate than mailing files back and forth.

We close with material on licensing because:

1. questions about who owns what, or can use what, arise naturally once we start talking about using public services like GitHub to store files; and
2. the discussion gives learners a chance to catch their breath after what is often a frustrating couple of hours.

Teaching Notes

- Make sure the network is working *before* starting this lesson.
- Give learners a five-minute overview of what version control does for them before diving into the watch-and-do practicals. Most of them will have tried to co-author papers by emailing files back and forth, or will have biked into the office only to realize that the USB key with last night's work is still on the kitchen table. Instructors can also make jokes about directories with names like "final version", "final version revised", "final version with reviewer three's corrections", "really final version", and, "come on this really has to be the last version" to motivate version control as a better way to collaborate and as a better way to back work up.
- Version control is typically taught after the shell, so collect learners' names during that session and create a repository for them to share with their names as both their IDs and their passwords.
- If your learners are advanced enough to be comfortable SSH, tell them they can use keys to authenticate on GitHub instead of passwords, but don't try to set this up during class: it takes too long, and is a distraction from the core ideas of the lesson.
- Be very clear what files learners are to edit and what user IDs they are to use when giving instructions: it is common for them to edit the file the instructor is working on rather than their own, or to use the instructor's user ID and password when committing.

- Be equally clear *when* they are to edit things: it's also common for someone to edit the file the instructor is editing and commit changes while the instructor is explaining what's going on, so that a conflict occurs when the instructor comes to commit the file.
- If some learners are using Windows, there will inevitably be issues merging files with different line endings. (Even if everyone's on some flavor of Unix, different editors may or may not add a newline to the last line of a file.) Take a moment to explain these issues, since learners will almost certainly trip over them again.
- We don't use a Git GUI in these notes because we haven't found one that installs easily and runs reliably on the three major operating systems, and because we want learners to understand what commands are being run. That said, instructors should demo a GUI on their desktop at some point during this lesson and point learners at this page[11].
- Instructors should also show learners graphical diff/merge tools like DiffMerge[12].
- When appropriate, explain that we teach Git rather than CVS, Subversion, or Mercurial primarily because of GitHub's growing popularity: CVS and Subversion are now seen as legacy systems, and Mercurial isn't nearly as widely used in the sciences right now.

Text Editor

We suggest instructors and students use `nano` as the text editor for this lessons because:

- it runs in all three major operating systems,
- it runs inside the shell (switching windows can be confusing to students), and
- it has shortcut help at the bottom of the window.

Please point out to students during setup that they can and should use another text editor if they're already familiar with it. Below you will find some tips that could help solving problems when using other editors.

Gedit You should use

```
$ git config --global core.editor 'gedit --standalone'
```

to avoid this error occurring if the student already has a Gedit window open:

```
$ git commit
```

```
Aborting commit due to empty commit message.
```

7.4 Programming with Python

This lesson is written as an introduction to Python, but its real purpose is to introduce the single most important idea in programming: how to solve problems by building functions, each of which can fit in a programmer's working memory. In order to teach that, we must teach people a little about the mechanics of manipulating data with lists and file I/O so that their functions can do things they actually care about. Our teaching order tries to show practical uses of every idea as soon as it is introduced; instructors should resist the temptation to explain the "other 90%" of the language as well.

[11] http://git-scm.com/downloads/guis

[12] https://sourcegear.com/diffmerge/

The secondary goal of this lesson is to give them a usable mental model of how programs run (what computer science educators call a *notional machine* so that they can debug things when they go wrong. In particular, they must understand how function call stacks work.

The final example asks them to build a command-line tool that works with the Unix pipe-and-filter model. We do this because it is a useful skill and because it helps learners see that the software they use isn't magical. Tools like `grep` might be more sophisticated than the programs our learners can write at this point in their careers, but it's crucial they realize this is a difference of scale rather than kind.

Teaching Notes

- Explain that we use Python because:
  - It's free.
  - It has a lot of scientific libraries, and more are constantly being added.
  - It has a large scientific user community.
  - It's easier for novices to learn than most of the mature alternatives. (Software Carpentry originally used Perl; when we switched, we found that we could cover as much material in two days in Python as we'd covered in three days in Perl, and that retention was higher.)
- We do *not* include instructions on running the IPython Notebook in the tutorial because we want to focus on the language rather than the tools. Instructors should, however, walk learners through some basic operations:
  - Launch from the command line with `ipython notebook`.
  - Create a new notebook.
  - Enter code or data in a cell and execute it.
  - Explain the difference between `In[#]` and `Out[#]`.
- Watching the instructor grow programs step by step is as helpful to learners as anything to do with Python. Resist the urge to update a single cell repeatedly (which is what you'd probably do in real life). Instead, clone the previous cell and write the update in the new copy so that learners have a complete record of how the program grew. Once you've done this, you can say, "Now why don't we just breaks things into small functions right from the start?"
- The discussion of command-line scripts assumes that students understand standard I/O and building filters, which are covered in the lesson on the shell.
- Do *not* start the notebook with:

  ```
  ipython notebook --pylab [backend]
  ```

  The `--pylab` option has been deprecated for a long time, and is being removed soon.

7.5 Using Databases and SQL

Relational databases are not as widely used in science as in business, but they are still a common way to store large data sets with complex structure. Even when the data itself isn't in a database, the metadata could be: for example, meteorological data might be stored in files on disk, but data about when and where observations were made, data ranges, and so on could be in a database to make it easier for scientists to find what they want to.

Teaching Notes

- The first few sections (up to "Missing Data") usually go very quickly. The pace usually slows down a bit when null values are discussed mostly because learners have a lot of details to keep straight by this point. Things *really* slow down during the discussion of joins, but this is the key idea in the whole lesson: important ideas like primary keys and referential integrity only make sense once learners have seen how they're used in joins. It's worth going over things a couple of times if necessary (with lots of examples).
- The sections on creating and modifying data, and programming with databases, can be dropped if time is short. Of the two, people seem to care most about how to add data (which only takes a few minutes to demonstrate).
- Overall, this material takes three hours to present assuming that a short exercise is done with each topic.
- Simple calculations are actually easier to do in a spreadsheet; the advantages of using a database become clear as soon as filtering and joins are needed. Instructors may therefore want to show a spreadsheet with the information from the four database tables consolidated into a single sheet, and demonstrate what's needed in both systems to answer questions like, "What was the average radiation reading in 1931?"
- Some learners may have heard that NoSQL databases (i.e., ones that don't use the relational model) are the next big thing, and ask why we're not teaching those. The answers are:
    1. Relational databases are far more widely used than NoSQL databases.
    2. We have far more experience with relational databases than with any other kind, so we have a better idea of what to teach and how to teach it.
    3. NoSQL databases are as different from each other as they are from relational databases. Until a leader emerges, it isn't clear *which* NoSQL database we should teach.
- Run `sqlite3 survey.db < gen-survey-database.sql` to re-create survey database before loading notebooks.

Chapter 8

Reference

These short reference guides cover the basic tools and ideas introduced in our lessons.

8.1 Shell Reference

Basic Commands

- `cat` displays the contents of its inputs.
- `cd path` changes the current working directory.
- `cp old new` copies a file.
- `find` finds files with specific properties that match patterns.
- `grep` selects lines in files that match patterns.
- `head` displays the first few lines of its input.
- `ls path` prints a listing of a specific file or directory; `ls` on its own lists the current working directory.
- `man command` displays the manual page for a given command.
- `mkdir path` creates a new directory.
- `mv old new` moves (renames) a file or directory.
- `pwd` prints the user's current working directory.
- `rm path` removes (deletes) a file.
- `rmdir path` removes (deletes) an empty directory.
- `sort` sorts its inputs.
- `tail` displays the last few lines of its input.
- `touch path` creates an empty file if it doesn't already exist.
- `wc` counts lines, words, and characters in its inputs.
- `whoami` shows the user's current identity.

Paths

- `/path/from/root` is an absolute path.
- `/` on its own refers to the root of the filesystem.
- `path/without/leading/slash` is a relative path.

- `.` refers to the current directory, `..` to its parent.
- `*` matches zero or more characters in a filename, so `*.txt` matches all files ending in `.txt`.
- `?` matches any single character in a filename, so `?.txt` matches `a.txt` but not `any.txt`.

Combining Commands

- `command > file` redirects a command's output to a file.
- `first | second` connects the output of the first command to the input of the second.
- A `for` loop repeats commands once for every thing in a list:

  ```
  for variable in name_1 name_2 name_3
  do
      ...commands refering to $variable...
  done
  ```

- Use `$name` to expand a variable (i.e., get its value).
- `history` displays recent commands, and `!number` to repeat a command by number.
- `bash filename` runs commands saved in `filename`.
- `$*` refers to all of a shell script's command-line parameters.
- `$1`, `$2`, etc., refer to specified command-line parameters.
- `$(command)` inserts a command's output in place.

8.2 Git Reference

Set global configuration (only needs to be done once per machine):

```
git config --global user.name "Your Name"
git config --global user.email "you@some.domain"
git config --global color.ui "auto"
git config --global core.editor "your_editor"
```

Initialize the current working directory as a repository:

```
git init
```

Display the status of the repository:

```
git status
```

Add specific files to the staging area:

```
git add filename_1 filename_2
```

Add all modified files in the current directory and its sub-directories to the staging area:

```
git add -A .
```

Commit changes in the staging area to the repository's history: (Without `-m` and a message, this command runs a text editor.)

```
git commit -m ”Some message”
```

View the history of the repository:

```
git log
```

Display differences between the current state of the repository and the last saved state:

```
git diff
```

Display differences between the current state of a particular file and the last saved state:

```
git diff path/to/file
```

Display differences between the last saved state and what’s in the staging area:

```
git diff --staged
```

Compare files to the previously saved state:

```
git diff HEAD~1 path/to/file
```

Compare files to an earlier saved state:

```
git diff HEAD~27 path/to/file
```

Compare files to a specific earlier state:

```
git diff 123456 path/to/file
```

Erase changes since the last save:

```
git reset --hard HEAD
```

Restore file to its state in a previous revision:

```
git checkout 123456 path/to/file
```

Add a remote to a repository:

```
git remote add nickname remote_specification
```

Display a repository’s remotes:

```
git remote -v
```

Push changes from a local repository to a remote (assuming `master` already exists in the remote):

```
git push nickname master
```

Push changes from a local repository to a remote (if `master` doesn’t yet exist in the remote):

```
git push -u nickname master
```

Pull changes from a remote repoisitory:

```
git pull nickname master
```

Note: `master` may be replaced with another branch name by more advanced users.
Clone a remote repository:

```
git clone remote_specification
```

Markers used to show conflict in a file during a merge:

```
<<<<<<< HEAD
lines from local file
=======
lines from remote file
>>>>>>> 1234567890abcdef1234567890abcdef12345678
```

8.3 Python Reference

Basic Operations

- Use `variable = value` to assign a value to a variable.
- Use `print first, second, third` to display values.
- Python counts from 0, not from 1.
- # starts a comment.
- Statements in a block must be indented (usually by four spaces).
- `help(thing)` displays help.
- `len(thing)` produces the length of a collection.
- `[value1, value2, value3, ...]` creates a list.
- `list_name[i]` selects the i'th value from a list.

Control Flow

- Create a `for` loop to process elements in a collection one at a time:

  ```
  for variable in collection:
      ...body...
  ```

- Create a conditional using `if`, `elif`, and `else`:

  ```
  if condition_1:
      ...body...
  elif condition_2:
      ...body...
  else:
      ...body...
  ```

- Use == to test for equality.
- `X and Y` is only true if both X and Y are true.
- `X or Y` is true if either X or Y, or both, are true.
- Use `assert condition, message` to check that something is true when the program is running.

Functions

- `def name(...params...)` defines a new function.
- `def name(param=default)` specifies a default value for a parameter.
- Call a function using `name(...values...)`.

Libraries

- Import a library into a program using `import libraryname`.
- The `sys` library contains:
  - `sys.argv`: the command-line arguments a program was run with.
  - `sys.stdin`, `sys.stdout`: standard input and output.
- `glob.glob(pattern)` returns a list of files whose names match a pattern.

Arrays

- `import numpy` to load the NumPy library.
- `array.shape` gives the shape of an array.
- `array[x, y]` selects a single element from an array.
- `low:high` specifies a slice including elements from `low` to `high-1`.
- `array.mean()`, `array.max()`, and `array.min()` calculate simple statistics.
- `array.mean(axis=0)` calculates statistics across the specified axis.

8.4 SQL Reference

Basic Queries

Select one or more columns from a table:

```
SELECT column_name_1, column_name_2 FROM table_name;
```

Select all columns from a table:

```
SELECT * FROM table_name;
```

Get only unique results in a query:

```
SELECT DISTINCT column_name FROM table_name;
```

Perform calculations in a query:

```
SELECT column_name_1, ROUND(column_name_2 / 1000.0) FROM table_name;
```

Sort results in ascending order:

```
SELECT * FROM table_name ORDER BY column_name_1;
```

Sort results in ascending and descending order:

```
SELECT * FROM table_name ORDER BY column_name_1 ASC, column_name_2 DESC;
```

Filtering

Select only data meeting a condition:

```
SELECT * FROM table_name WHERE column_name > 0;
```

Select only data meeting a combination of conditions:

```
SELECT * FROM table_name WHERE (column_name_1 >= 1000) AND (column_name_2 = 'A' OR column_name_
```

Missing Data

Use NULL to represent missing data.

NULL is not 0, false, or any other specific value. Operations involving NULL produce NULL, so 1+NULL, 2>NULL, and 3=NULL are all NULL.

Test whether a value is null:

```
SELECT * FROM table_name WHERE column_name IS NULL;
```

Test whether a value is not null:

```
SELECT * FROM table_name WHERE column_name IS NOT NULL;
```

Grouping and Aggregation

Combine values using aggregation functions:

```
SELECT SUM(column_name_1) FROM table_name;
```

Combine data into groups and calculate combined values in groups:

```
SELECT column_name_1, SUM(column_name_2), COUNT(*) FROM table_name GROUP BY column_name_1;
```

Joins

Join data from two tables:

```
SELECT * FROM table_name_1 JOIN table_name_2 ON table_name_1.column_name = table_name_2.column_
```

Writing Queries

SQL commands must be combined in the following order: SELECT, FROM, JOIN, ON, WHERE, GROUP BY, ORDER BY.

Creating Tables

Create tables by specifying column names and types. Include primary and foreign key relationships and other constraints.

```
CREATE TABLE survey(
    taken   INTEGER NOT NULL,
    person  TEXT,
    quant   REAL NOT NULL,
    PRIMARY KEY(taken, quant),
    FOREIGN KEY(person) REFERENCES person(ident)
);
```

Programming

Execute queries in a general-purpose programming language by:

- loading the appropriate library
- creating a connection
- creating a cursor
- repeatedly:
  - execute a query
  - fetch some or all results
- disposing of the cursor
- closing the connection

Python example:

```
import sqlite3
connection = sqlite3.connect("database_name")
cursor = connection.cursor()
cursor.execute("...query...")
for r in cursor.fetchall():
    ...process result r...
cursor.close()
connection.close()
```

Chapter 9

Recommended Reading

9.1 Books

Susan A. Ambrose, Michael W. Bridges, Michele DiPietro, Marsha C. Lovett, and Marie K. Norman: *How L*
The best single-volume guide to evidence-based practices in education around.

Chris Fehily: *SQL: Visual QuickStart Guide* (3rd ed). Peachpit Press, 0321553578, 2002. Describes the 5% of SQL that covers 95% of real-world needs.

Karl Fogel: *Producing Open Source Software: How to Run a Successful Free Software Project*. O'Reilly Medi
An guide to how open source projects actually work, full of practical advice on how to earn commit privileges on a project, get it more attention, or fork it in case of irreconcilable differences.

Steve Haddock and Casey Dunn: *Practical Computing for Biologists*. Sinauer, 0878933913, 2010.
An excellent general introduction to "the other 90%" of scientific computing.

Andy Oram and Greg Wilson (eds): *Making Software: What Really Works, and Why We Believe It*. O'Reilly,
Leading software engineering researchers take a chapter each to describe key empirical results and the evidence behind them. Topics range from the impact of programming languages on programmers' productivity to whether we can predict software faults using statistical techniques.

Deborah S. Ray and Eric J. Ray: *Unix and Linux: Visual QuickStart Guide*. Peachpit Press, 0321636783, 200
A gentle introduction to Unix, with many examples.

Papers

Paul F. Dubois: "Maintaining Correctness in Scientific Programs". *Computing in Science & Engineering*, M
Shows how several good programming practices fit together to create defense in depth, so that errors missed by one will be caught by another.

Matthew Gentzkow and Jesse M. Shapiro. 2014: "Code and Data for the Social Sciences: A Practitioner's G
An excellent description of how to move from doing data analysis with SAS and Excel to using maintainable scripts on well-organized data files in a reproducible way.

Jo Erskine Hannay, Hans Petter Langtangen, Carolyn MacLeod, Dietmar Pfahl, Janice Singer, and Greg Wi
The largest study survey done of how scientists use computers in their research and how much time they spend doing so.

William Stafford Noble: “A Quick Guide to Organizing Computational Biology Projects”. *PLoS Computational*

How and why one scientist organizes his data and scripts.

Ethan P. White, Elita Baldridge, Zachary T. Brym, Kenneth J. Locey, Daniel J. McGlinn, and Sarah R. Supp: “

Delivers exactly what the title promises: a straightforward set of practices that will make it easier for other scientists to use your data.

Greg Wilson, D. A. Aruliah, C. Titus Brown, Neil P. Chue Hong, Matt Davis, Richard T. Guy, Steven H. D. Had

Describes a set of best practices for scientific software development that have solid foundations in research and experience, and that improve scientists’ productivity and the reliability of their software.

Greg Wilson: “Software Carpentry: Lessons Learned”. *F1000 Research*, 3(62), 2014, doi:10.12688/f1000researc

Describes what we’ve learned about how to teach programming to scientists over the last 15 years.

Chapter 10

Glossary

absolute error : The absolute value of the difference between a mathematical value and its finite approximation in a computer.

absolute path : A *path* that refers to a particular location in a file system. Absolute paths are usually written with respect to the file system's *root directory*, and begin with either "/" (on Unix) or "\" (on Microsoft Windows). See also: *relative path*.

access control list (ACL): A list of permissions attached to a file or directory that specifies who can do what with it.

additive color model : A way to represent colors as the sum of contributions from primary colors such as *red, green, and blue*.

aggregation function : A function such as `sum` or `max` that combines many values to produce a single result.

alias (a library): To give a *library* a nickname while importing it.

assertion : An expression which is supposed to be true at a particular point in a program. Programmers typically put assertions in their code to check for errors; if the assertion fails (i.e., if the expression evaluates as false), the program halts and produces an error message. See also: *invariant*, *precondition*, *postcondition*.

assignment : To give a value a name by associating a variable with it.

atomic value : A value that cannot be decomposed into smaller pieces. For example, the number 12 is usually considered atomic (unless we are teaching addition to school children, in which case we might decompose it into tens and ones).

branch : A "parallel universe" in a *version control repository*. Programmers typically use branches to isolate different sets of changes from one another during development so that they can concentrate on one problem at a time. See also: *merge*.

call stack : A data structure inside a running program that keeps track of active function calls. Each call's variables are stored in a *stack frame*; a new stack frame is put on top of the stack for each call, and discarded when the call is finished.

cascading delete : The practice of automatically deleting things in a database that depend on a record when that record is deleted. See also: *referential integrity*.

case insensitive : Treating text as if upper and lower case characters were the same. See also: *case sensitive*.

catch (an exception): To handle an *exception* that has been *raised* somewhere else in a program.

change set : A group of changes to one or more files that are *committed* to a *version control repository* in a single operation.

clone (a repository): To make a local copy of a *version control repository*. See also: *fork*.

code review : A systematic peer review of a piece of software, or of changes to a piece of software. Peer review is often conducted on *pull requests* before they are *merged* into a *repository*.

comma-separated values (CSV): A common textual representation for tables in which the values in each row are separated by commas.

command-line interface (CLI): An interface based on typing commands, usually at a *REPL*. See also: *graphical user interface*.

comment : A remark in a program that is intended to help human readers understand what is going on, but is ignored by the computer. Comments in Python, R, and the Unix shell start with a # character and run to the end of the line; comments in SQL start with `--`, and other languages have other conventions.

conditional statement : A statement in a program that might or might not be executed depending on whether a test is true or false.

conflict : A change made by one user of a *version control system* that is incompatible with changes made by other users. Helping users *resolve* conflicts is one of version control's major tasks.

cross product : A pairing of all elements of one set with all elements of another.

current working directory : The directory that *relative paths* are calculated from; equivalently, the place where files referenced by name only are searched for. Every *process* has a current working directory. The current working directory is usually referred to using the shorthand notation `.` (pronounced "dot").

cursor : A pointer into a database that keeps track of outstanding operations.

data type : A kind of data value, such as *integer* or *character string*.

database manager : A program that manages a *relational database*.

default parameter value : A value to use for a parameter if nothing is specified explicitly.

defensive programming : The practice of writing programs that check their own operation to catch errors as early as possible.

delimiter : A character or characters used to separate individual values, such as the commas between columns in a *CSV* file.

disk block : The smallest unit of storage that can be allocated on a computer disk. Disk blocks are typically 512 bytes in size.

docstring : Short for "documentation string", this refers to textual documentation embedded in Python programs. Unlike comments, docstrings are preserved in the running program and can be examined in interactive sessions.

documentation : Human-language text written to explain what software does, how it works, or how to use it.

dotted notation : A two-part notation used in many programming languages in which `thing.component` refers to the `component` belonging to `thing`.

empty string : A character string containing no characters, often thought of as the "zero" of text.

encapsulation : The practice of hiding something's implementation details so that the rest of a program can worry about *what* it does rather than *how* it does it.

exception : An event that disrupts the normal or expected execution of a program. Most modern languages record information about what went wrong in a piece of data (also called an exception). See also: *catch*, *raise*.

field (of a database): A set of data values of a particular type, one for each *record* in a *table*.

filename extension : The portion of a file's name that comes after the final "." character. By convention this identifies the file's type: `.txt` means "text file", `.png` means "Portable Network Graphics file", and so on. These conventions are not enforced by most operating systems: it is perfectly possible to name an MP3 sound file `homepage.html`. Since many applications use filename extensions to identify the *MIME type* of the file, misnaming files may cause those applications to fail.

filesystem : A set of files, directories, and I/O devices (such as keyboards and screens). A filesystem may be spread across many physical devices, or many filesystems may be stored on a single physical device; the *operating system* manages access.

filter : A program that transforms a stream of data. Many Unix command-line tools are written as filters: they read data from *standard input*, process it, and write the result to *standard output*.

flag : A terse way to specify an option or setting to a command-line program. By convention Unix applications use a dash followed by a single letter, such as `-v`, or two dashes followed by a word, such as `--verbose`, while DOS applications use a slash, such as `/V`. Depending on the application, a flag may be followed by a single argument, as in `-o /tmp/output.txt`.

floating point number : A number containing a fractional part and an exponent. See also: *integer*.

for loop : A loop that is executed once for each value in some kind of set, list, or range. See also: *while loop*.

foreign key : One or more values in a *database table* that identify a *records* in another table.

fork : To *clone* a *version control repository* on a server.

function call : A use of a function in another piece of software.

function body : The statements that are executed inside a function.

function composition : The immediate application of one function to the result of another, such as `f(g(x))`.

graphical user interface (GUI): A graphical user interface, usually controlled by using a mouse. See also: *command-line interface*.

home directory : The default directory associated with an account on a computer system. By convention, all of a user's files are stored in or below her home directory.

HTTP : The Hypertext Transfer *Protocol* used for sharing web pages and other data on the World Wide Web.

immutable : Unchangeable. The value of immutable data cannot be altered after it has been created. See also: *mutable*.

import : To load a *library* into a program.

in-place operator : An operator such as `+=` that provides a shorthand notation for the common case in which the variable being assigned to is also an operand on the right hand side of the assignment. For example, the statement `x += 3` means the same thing as `x = x + 3`.

index : A subscript that specifies the location of a single value in a collection, such as a single pixel in an image.

infective license : A license such as the GPL[1] that compels people who incorporate material into their own work to place similar sharing requirements on it.

inner loop : A loop that is inside another loop. See also: *outer loop*.

integer : A whole number, such as -12343. See also: *floating-point number*.

invariant : An expression whose value doesn't change during the execution of a program, typically

[1] http://opensource.org/licenses/GPL-3.0

used in an *assertion*. See also: *precondition*, *postcondition*.

library : A family of code units (functions, classes, variables) that implement a set of related tasks.

loop body : The set of statements or commands that are repeated inside a *for loop* or *while loop*.

loop variable : The variable that keeps track of the progress of the loop.

member : A variable contained within an *object*.

merge (a repository): To reconcile two sets of change to a *repository*.

method : A function which is tied to a particular *object*. Each of an object's methods typically implements one of the things it can do, or one of the questions it can answer.

mutable : Changeable. The value of mutable data can be updated in place. See also: *immutable*.

notional machine : An abstraction of a computer used to think about what it can and will do.

object : A particular "chunk" of data associated with specific operations called *methods*.

orthogonal : To have meanings or behaviors that are independent of each other. If a set of concepts or tools are orthogonal, they can be combined in any way.

outer loop : A loop that contains another loop. See also: *inner loop*.

parameter : A value passed into a function, or a variable named in the function's declaration that is used to hold such a value.

parent directory : The directory that "contains" the one in question. Every directory in a file system except the *root directory* has a parent. A directory's parent is usually referred to using the shorthand notation `..` (pronounced "dot dot").

pipe : A connection from the output of one program to the input of another. When two or more programs are connected in this way, they are called a "pipeline".

pipe and filter : A model of programming in which *filters* that process *streams* of data are connected end-to-end. The pipe and filter model is used extensively in the Unix *shell*.

postcondition : A condition that a function (or other block of code) guarantees is true once it has finished running. Postconditions are often represented using *assertions*.

precondition : A condition that must be true in order for a function (or other block of code) to run correctly.

prepared statement : A template for an *SQL* query in which some values can be filled in.

primary key : One or more *fields* in a *database table* whose values are guaranteed to be unique for each *record*, i.e., whose values uniquely identify the entry.

process : A running instance of a program, containing code, variable values, open files and network connections, and so on. Processes are the "actors" that the *operating system* manages; it typically runs each process for a few milliseconds at a time to give the impression that they are executing simultaneously.

prompt : A character or characters display by a *REPL* to show that it is waiting for its next command.

protocol : A set of rules that define how one computer communicates with another. Common protocols on the Internet include *HTTP* and *SSH*.

pull request : A set of changes created in one *version control repository* that is being offered to another for *merging*.

query : A database operation that reads values but does not modify anything. Queries are expressed in a special-purpose language called *SQL*.

quoting (in the shell): Using quotation marks of various kinds to prevent the shell from interpreting special characters. For example, to pass the string `*.txt` to a program, it is usually necessary to write it as `'*.txt'` (with single quotes) so that the shell will not try to expand the `*` wildcard.

raise (an exception): To explicitly signal that an *exception* has occured in a program. See also: *catch*.

read-eval-print loop (REPL): A *command-line interface* that reads a command from the user, executes it, prints the result, and waits for another command.

record (in a database): A set of related values making up a single entry in a *database table*, typically shown as a row. See also: *field*.

redirect : To send a command's output to a file rather than to the screen or another command, or equivalently to read a command's input from a file.

referential integrity : The internal consistency of values in a database. If an entry in one table contains a *foreign key*, but the corresponding *records* don't exist, referential integrity has been violated.

regression : To re-introduce a bug that was once fixed.

regular expressions (RE): A pattern that specifies a set of character strings. REs are most often used to find sequences of characters in strings.

relational database : A collection of data organized into *tables*.

relative error : The ratio of the *absolute error* in an approximation of a value to the value being approximated.

relative path : A *path* that specifies the location of a file or directory with respect to the *current working directory*. Any path that does not begin with a separator character ("/" or "\") is a relative path. See also: *absolute path*.

remote login : To connect to a computer over a network, e.g., to run a *shell* on it. See also: *SSH*.

remote repository : A version control *repository* other than the current one that the current one is somehow connected to or mirroring.

repository : A storage area where a *version control* system stores old *revisions* of files and information about who changed what, when.

resolve : To eliminate the *conflicts* between two or more incompatible changes to a file or set of files being managed by a *version control* system.

return statement : A statement that causes a function to stop executing and return a value to its caller immediately.

revision : A recorded state of a *version control repository*.

RGB : An *additive model* that represents colors as combinations of red, green, and blue. Each color's value is typically in the range 0..255 (i.e., a one-byte integer).

root directory : The top-most directory in a *filesystem*. Its name is "/" on Unix (including Linux and Mac OS X) and "\" on Microsoft Windows.

search path : The list of directories in which the *shell* searches for programs when they are run.

sentinel value : A value in a collection that has a special meaning, such as 999 to mean "age unknown".

sequence : An ordered collection of values whose elements can be specified with a single integer index, such as a vector.

shape (of an array): An array's dimensions, represented as a vector. For example, a 5ÃŮ3 array's shape is `(5,3)`.

shell : A *command-line interface* such as Bash (the Bourne-Again Shell) or the Microsoft Windows DOS shell that allows a user to interact with the *operating system*.

shell script : A set of *shell* commands stored in a file for re-use. A shell script is a program executed by the shell; the name "script" is used for historical reasons.

sign and magnitude : A scheme for representing numbers in which one bit indicates the sign (positive or negative) and the other bits store the number's absolute value. See also: *two's complement*.

silent failure : Failing without producing any warning messages. Silent failures are hard to detect and debug.

slice : A regular subsequence of a larger sequence, such as the first five elements or every second element.

SQL (Structured Query Language): A special-purpose language for describing operations on *relational databases*.

SQL injection attack : An attack on a program in which the user's input contains malicious SQL statements. If this text is copied directly into an SQL statement, it will be executed in the database.

SSH : The Secure Shell *protocol* used for secure communication between computers. SSH is often used for *remote login* between computers.

SSH key : A digital key that identifies one computer or user to another.

stack frame : A data structure that provides storage for a function's local variables. Each time a function is called, a new stack frame is created and put on the top of the *call stack*. When the function returns, the stack frame is discarded.

standard input (stdin): A process's default input stream. In interactive command-line applications, it is typically connected to the keyboard;; in a *pipe*, it receives data from the *standard output* of the preceding process.

standard output (stdout): A process's default output stream. In interactive command-line applications, data sent to standard output is displayed on the screen; in a *pipe*, it is passed to the *standard input* of the next process.

stride : The offset between successive elements of a *slice*.

string : Short for "character string", a *sequence* of zero or more characters.

sub-directory : A directory contained within another directory.

tab completion : A feature provided by many interactive systems in which pressing the Tab key triggers automatic completion of the current word or command.

table (in a database): A set of data in a *relational database* organized into a set of *records*, each having the same named *fields*.

test oracle : A program, device, data set, or human being against which the results of a test can be compared.

test-driven development (TDD): The practice of writing unit tests *before* writing the code they test.

timestamp : A record of when a particular event occurred.

tuple : An *immutable sequence* of values.

two's complement : A scheme for representing numbers which wraps around like an odometer so that 111...111 represents -1. See also: *sign and magnitude*.

user group : A set of users on a computer system.

user group ID : A numerical ID that specifies a *user group*.

user group name : A textual name for a *user group*.

user ID : A numerical ID that specifies an individual user on a computer system. See also: *user name*.

user name : A textual name for a user on a computer system. See also: *user ID*.

variable : A name in a program that is associated with a value or a collection of values.

version control : A tool for managing changes to a set of files. Each set of changes creates a new *revision* of the files; the version control system allows users to recover old revisions reliably, and helps manage conflicting changes made by different users.

while loop : A loop that keeps executing as long as some condition is true. See also: *for loop*.

wildcard : A character used in pattern matching. In the Unix shell, the wildcard "*" matches zero or more characters, so that `*.txt` matches all files whose names end in `.txt`.

Chapter 11

Licenses

11.1 Lesson Material

All Software Carpentry instructional material is made available under the Creative Commons Attribution license. You are free:

- to **Share**—to copy, distribute and transmit the work
- to **Remix**—to adapt the work

Under the following conditions:

- **Attribution**—You must attribute the work using "Copyright (c) Software Carpentry" (but not in any way that suggests that we endorse you or your use of the work). Where practical, you must also include a hyperlink to http://software-carpentry.org.

With the understanding that:

- **Waiver**—Any of the above conditions can be waived if you get permission from the copyright holder.
- **Other Rights**—In no way are any of the following rights affected by the license:
  - Your fair dealing or fair use rights;
  - The author's moral rights;
  - Rights other persons may have either in the work itself or in how the work is used, such as publicity or privacy rights. *
- **Notice**—For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to http://creativecommons.org/licenses/by/3.0/[1].

For the full legal text of this license, please see http://creativecommons.org/licenses/by/3.0/legalcode[2].

11.2 Software

Except where otherwise noted, the example programs and other software provided by Software Carpentry are made available under the OSI[3]-approved MIT license[4].

[1] http://creativecommons.org/licenses/by/3.0/

[2] http://creativecommons.org/licenses/by/3.0/legalcode

[3] http://opensource.org

[4] http://opensource.org/licenses/mit-license.html

11.3 Trademark

“Software Carpentry” and the Software Carpentry logo are registered trademarks of Software Carpentry, Ltd.