THE POWER OF GO TOOLS

"Superb and well written"
—Lee Gibson

JOHN ARUNDEL

The Power of Go: Tools

2024 edition

John Arundel

Bitfield Consulting

September 5, 2023

© 2023 John Arundel

The Power of Go: Tools

- Praise for ‘The Power of Go: Tools’
- Introduction
  - Who is this book for?
  - What should I know before reading it?
  - What version of Go does it cover?
  - Where to find the code examples
  - What you’ll learn
- 1. Packages
  - The universal library
    - Packages are a force multiplier
    - The universal Go library is huge
    - Sharing our code benefits us all
    - Writing packages, not programs
    - Command-line tools
    - Zen mountaineering
    - Guided by tests
  - Building a `hello` package
    - The structure of a test
    - Tests are bug detectors
    - So when should a test fail?
    - Where does `fmt.Println` print to?
    - Meet `bytes.Buffer`, an off-the-shelf `io.Writer`
    - Failure standards
  - Implementing `hello`, guided by tests
    - We start with a failing test
    - Creating a module
    - One folder, one package
    - A null implementation
    - The real implementation
    - The naming of tests
    - Refactoring to use our new package

- Going further
- 2. Paperwork
  - Making our package more flexible
    - Mandatory arguments are annoying
    - What if we allow users to pass `nil`?
    - Maybe some global variable instead?
    - A struct is the answer
    - A convenience wrapper with defaults
  - A simple line counter
    - Focus on behaviour, not implementation
    - One possible first version
    - What about configuration?
    - Config structs don't solve the problem
  - An elegant option API
    - Okay, so what's an "option"?
    - Options are functions
    - "Always valid" fields
    - Some internal paperwork
    - Handling internal errors
    - A line counter with options
    - Methodical options
  - Going further
- 3. Arguments
  - Designing the behaviour
    - Deciding the scope
    - Testing CLI arguments
    - A first attempt
  - Test data files
    - Creating the test data
    - Using the data in a test
    - The failing test
  - Implementing file-reading
    - Empty slice checking
    - Testing the "no args" behaviour
    - Closing files and other resources
  - Updating the user interface

- Some user testing
- Setting exit status
- Test scripts
  - Running the program as a binary
  - Introducing `testscript`
  - Invoking test scripts
  - Defining custom commands
  - The delegate `Main` function
  - The `stdout` assertion
  - Testing arguments
- Going further

4. Flags
- Commands
  - The Unix way
  - Multiple `main` packages
  - A `words` command
  - A test for word counting
  - Updating the test scripts
- Flags
  - Introducing flags
  - Adding a `-lines` flag
  - Implementing the behaviour
  - Help and usage information
- Going further

5. Files
- Writing to files
  - The art of judicious logging
  - Testing a `WriteToFile` function
  - Designing errors out of existence
  - Looking for inspiration
  - What are we really testing?
  - The `go-cmp` module
- Implementing a `WriteToFile` function
  - File permissions
  - When the directory doesn’t exist
  - A disastrous bug

- Using `t.TempDir`
- Finishing the job
  - It’s clobbering time
  - Ensuring permissions
  - What’s the worst that could happen?
  - A security leak
- Going further
- 6. Filesystems
  - Files and filesystems
    - What even is a file?
    - Organising files
  - A simple file finder
    - Handling folders recursively
    - Filesystems and `io/fs`
    - Matching files by name
    - Walking the tree
  - A file-finding tree-walker
    - Starting at the top
    - The `fs.FS` abstraction
    - Any path-value map is a “filesystem”
    - The `fstest.MapFS` filesystem
  - Adding a filesystem to our API
    - Timing potentially slow operations
    - Writing benchmark functions
    - Taking `fs.FS` makes APIs more flexible
  - Going further
- 7. Commands
  - The exec package
    - What even is a process?
    - Managing command output
    - When not to use exec
    - When to use exec
  - Migrating from shell scripts to Go
    - Why use Go to run commands?
  - A command wrapper in Go
    - The `pmset` command

- What can we test?
- Breaking down behaviour into chunks
- Parsing command output
- Testing the parsing function
- Parsing command output
  - Clarifying the problem
  - Writing a regular expression
  - Using the regexp
- Integration tests
  - Why isolate integration tests?
  - Build tags
  - Testing the command runner
- Running the command
  - Capturing output
- When to import a third-party package
- Going further

8. Shells
- A simple shell
  - Defining some behaviour
  - Identifying the first test
  - Comparing the incomparable
  - Parsing user input
- Prototyping
  - A pseudocode outline
  - Main-driven development
  - Feedback from user testing
- A stateful shell session
  - What would we like to write?
  - What object would make sense?
  - Designing the `Session` object
- Running a session
  - Testing the Run method
  - Dependencies on external commands
  - A dry-run mode
  - Implementing Run
- What’s still missing

- Globbing
- Redirection
- Piping
- Quoting
- Going further

9. Pipelines
- A realistic operations task
  - Matching and counting log lines
  - A quick shell spell
  - Solving the problem with Go
  - In what language would this be easy?
- Programs as pipelines
  - A fluent API
  - Errors in sequenced operations
  - An error-safe writer
- Putting the pieces together
  - How does data flow from one method to another?
  - Testing the end of the pipeline
  - Breaking ground
  - Getting the test passing
  - Adding error safety
  - Obviousness-oriented programming
- Trying it out
  - Hello, world
  - The world strikes back
  - Setting defaults with a constructor
  - Reading data from files
  - Another pipeline explosion
- Filtering data
  - Extracting columns
  - A `String` sink
  - Testing `Column`
  - Validating arguments
  - Implementing `Column`
- The `script` package
  - Was this a waste of time?

- Introducing `script`
- Some simple one-liners
- More sophisticated programs
- Concurrent pipeline stages
- Custom filter functions
- Solving problems
- The landscape of simple programs
- Going further
- 10. Data
  - Marshalling
    - Serialisation
    - The gob package
  - A file-based data store
    - Testing data persistence
    - Setters and getters
    - A sensible key-value API
    - Testing the key-value machinery
    - Implementing the store
    - Adding persistence
    - What are we really testing here?
    - An end-to-end persistence test
    - Saving and loading
    - Tightening up the tests
  - JSON: a text data format
    - The `json` package
    - JSON is a useful auxiliary language
    - A JSON output option
  - Adding JSON to the `battery` package
    - Producing JSON strings
    - What could go wrong?
    - Testing `ToJSON`
  - Implementing `ToJSON`
    - Pretty-printing with indentation
    - Querying JSON output
  - YAML: a less verbose JSON
    - Parsing YAML in Go

- Designing the API with a test
- Defining our types
- The `go-yaml` package
  - Decoding
  - When unmarshalling doesn’t work
  - The format of struct tags
  - Setting defaults
  - Eliminating config structs
- Going further
11. Clients
- A simple weather client
  - What do we need?
  - Kicking the tyres
  - Environmental credentials
  - Making a GET request
  - Initial user testing
  - A second pass
- Parsing JSON responses
  - Testing `ParseResponse`
  - To decode or to unmarshal?
  - A temporary struct type
  - The response struct
  - Implementing `ParseResponse`
  - What could go wrong?
  - Other kinds of invalid data
  - What are we really testing here?
  - Back to a running program
- Constructing the request URL
  - A `FormatURL` function
  - Getting the location as input
  - Refactoring the remaining code
  - A paperwork-reducing `GetWeather` function
- Testing against a local HTTP server
  - A simple `httptest` example
  - A trustful TLS client
- A `weather` client object

- A familiar pattern
- Refactoring the tests to use our client
- Writing the client constructor
- Refactoring `GetWeather` as a client method
  - Testing `GetWeather`
  - Implementing `GetWeather`
  - A convenience wrapper
  - Adding a `Main` function
- Adding temperature support
  - Parsing temperature data
  - More user testing
  - Handling quantities with units
  - Presenting temperatures in Celsius
  - Defining a `Temperature` type
- Tackling more complex APIs
  - Request data
  - “CRUD” methods
- Last words
- Going further
About this book
- Who wrote this?
- Feedback
- Mailing list
- For the Love of Go
- The Power of Go: Tests
- Know Go: Generics
- Further reading
- Video course
- Credits
Acknowledgements
A sneak preview
1. Programming with confidence
- Self-testing code
- The adventure begins
- Verifying the test
- Running tests with `go test`

Using `cmp.Diff` to compare results
New behaviour? New test.
Test cases
Adding cases one at a time
Quelling a panic
Refactoring
Well, that was easy
Sounds good, now what?

Praise for ‘The Power of Go: Tools’

Curse you for derailing my day with another fascinating book! The content is absolutely awesome.
—Peter Nunn

Superb and well written: all the examples worked and were very helpful.
—Lee Gibson

It’s fantastic! I really like the approach.
—Sal DiStefano

The book does a great job of teaching what to do *with Go. I really liked the emphasis on testing.*
—Pedro Sandoval

Exactly the book I was looking for next! I love it.
—Elliot Thomas

What I really love about this book is it takes the newbie to the next level via real-world, relatable examples. The narrative flow clicks with me.
—Rajaseelan Ganeswaran

It’s clear, concise, and practical, especially for folks like me who enjoy the art of making end-user tools with Go. It’s really made me think when I write code about how simple and easy I can make it for folks to use, without, as John likes to say, ‘a lot of paperwork’.
—Josh Feierman

Everywhere I go, I’m asked if I think the university stifles writers. My opinion is that they don’t stifle enough of them.
—Flannery O’Connor

Introduction

> *The guiding motto in the life of every natural philosopher should be, "Seek simplicity and distrust it."*
> —Alfred North Whitehead, "The Concept of Nature"

Hello, and welcome to the book! It's great to have you here.

Who is this book for?

There are lots of books that will teach you Go, but not many that will show you what to *do* with it. In other words, once you've learned how to write Go code, what code should you write?

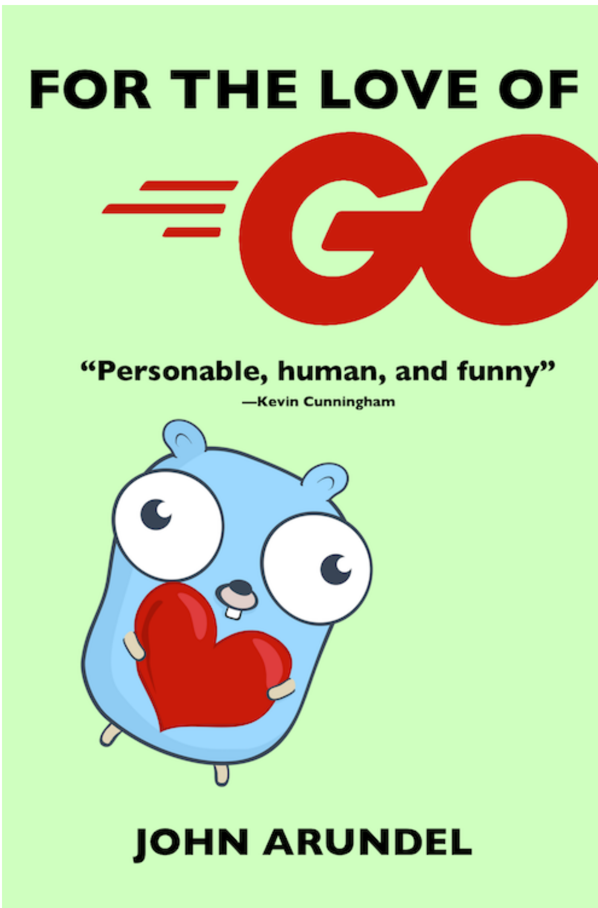
This book is aimed at those who have a little experience with Go (or even a lot), and would now like to learn how to build good software with it. What is "good" software anyway? What would it look like in Go? And how do we get there from here?

If software engineering is a craft, which it surely is, then how do we go about mastering it? It's all very well to say "just write programs", but how? How do we take some problem and start designing a program to solve it? How can we incorporate tests into the design? What are we even aiming to do here?

I hope you'll find at least some useful answers to these questions in this book, which focuses on developing *command-line tools*, but most of it applies to any kind of Go program.

What should I know before reading it?

While you don't need to be a confident or expert Go programmer, I'll assume in this book that you're familiar with at least the basics: compiling and running Go programs, how structs and slices work, what functions do, and so on. If you're entirely new to Go, or even to programming in general, I recommend you read my previous book, "For the Love of Go", first:



Once you’ve read it, you’ll be in the ideal place to start reading *this* book. Go ahead! I’ll wait right here until you come back.

What version of Go does it cover?

This book uses Go 1.21, released in August 2023, and all the code samples have been tested against at least that version. However, Go puts a strong emphasis on backward compatibility, so all the code in this book will still work perfectly well with later Go 1.x versions.

In general it should also work well with *earlier* versions, though I recommend you use the latest version of Go you can. If Go 1.21 isn’t yet available as a package in your operating system distribution, you can build it from source

or download a suitable binary package from the Go website directly:

- https://go.dev/learn/

Where to find the code examples

There are dozens of challenges for you to solve throughout the book, each designed to help you test your understanding of the concepts you've just learned.

Throughout the book, you'll see a number of code goals for you to achieve, marked with the word **GOAL**, like this:

GOAL: Get the test passing.

When you see this, stop reading at that point and see if you can figure out how to solve the problem. You can try to write the code and check it against the tests, or just think about what you would do.

If you reckon you have the answer, or alternatively if you've got a bit stuck and aren't sure what to do, you can then read on for a **HINT**, or read on even further for step-by-step instructions on how to construct the **SOLUTION**.

If you run into trouble, or just want to check your code, each challenge is accompanied by a complete sample solution, with tests.

All these solutions are also available in a public GitHub repo here:

- https://github.com/bitfield/tpg-tools2

Each listing in the book is accompanied by a name and number (for example, listing `hello/1`), and you'll find the

solution to that exercise in the corresponding folder of the repo.

What you'll learn

By reading through this book and completing the exercises, you'll learn:

- How to build reusable *packages* instead of one-off programs

- How to design user-friendly *APIs* and packages, without annoying paperwork

- How to write robust, testable tools that take command-line *flags* and *arguments*

- How to design Go packages that work with *files* and other kinds of streaming data

- How to write flexible tools to operate on *trees* of files, and more generally path-value databases such as URLs or zip archives

- How to use Go to drive external *commands* and provide elegant APIs to abstract their functionality

- How to write commands that interact extensively with users, such as *shells*

- How to sequence operations into simple *pipelines* that abstract away the details of handling streaming data and errors

- How to encode and decode data in binary format, and translate Go data to and from JSON and YAML formats

- How to create robust, reusable *client* packages for HTTP services and other APIs

1. Packages

> *When it was announced that the Library contained all books, the first reaction was unbounded joy. All… felt themselves the possessors of an intact and secret treasure. There was no personal problem, no world problem, whose eloquent solution did not exist—somewhere…*
> —Jorge Luis Borges, [“The Library of Babel”]

The universal library

What’s wrong with this program?

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println("Hello, world")
}
```

(Listing `hello/1`)

If you looked in vain, unable to spot the bug, I don’t blame you. The fact is, there’s nothing wrong with this program *as such*. It works, to the extent that it does what the author intended: it prints a message to the terminal, and exits.

But there are some limitations on what we can *do* with this program. The most serious of these is that it’s not an *importable package*. Let’s talk about why this is a big deal.

The earliest computers were single-purpose machines. They could only execute the specific computation they were designed for: the trajectory of an artillery shell, say. To compute something different meant physically re-wiring the machine.

It took a leap of insight to realise that a much more useful kind of computer would be one that could execute *any* computation, without being re-wired. That is to say, it would be *programmable*.

Packages are a force multiplier

A further significant advance was the idea that we don’t need to write every program from scratch every time. Instead, we could create reusable “routines”. For example, the code to calculate square roots is complicated and easy to get wrong, but it only has to be written once. We can then copy and use that routine in any program that needs to take a square root.

Nowadays, we would call such independent chunks of software *modules*, or *packages*, and they’re fundamental to all modern programming.

Without packages we would always have to instruct the computer about every detail of what we want to do. It would be hard to write useful programs in Go without the packages in the standard library, for example.

Packages, in other words, are an immensely powerful force multiplier. When we’re programming in a language like Go that has a rich ecosystem of importable packages, we never have to reinvent the wheel.

If we can figure out how to break down our unsolved problems into a bunch of mini-problems that have already been solved by existing packages, then we’re 90% done.

The universal Go library is huge

The Go *language* itself is pretty small, in the sense that there aren’t many keywords and there’s not a vast amount of syntax to learn. And that’s great news for those of us learning it.

But since this little language ships with a big standard library, full of all sorts of useful and well-designed packages, we can use it to construct some really powerful programs right away.

That’s nice, but are we restricted to importing only packages from the standard library? Certainly not. We can import any published package in the world, and there are plenty available.

We might call this wider ecosystem of importable Go packages the *universal* library. A quick search on GitHub reveals something close to *half a million* packages in this universal library (and there are more published in other places).

If you can imagine it, in other words, there's probably a Go package that provides it. So many, indeed, that simply *finding* the package you want can be a challenge. The pkg.go.dev site lets you search and browse the whole universal library:

- https://pkg.go.dev

If you're looking for a package to tackle some specific problem, a good place to check first is awesome-go, a carefully-curated list of a couple of thousand or so of the very best Go packages, by subject area:

- https://github.com/avelino/awesome-go

This is one of the many reasons that Go is such a popular choice for developing software nowadays. In many cases, all we need to do to create a particular program is to figure out the right way to connect up the various packages that we need.

We can then make *our* program a package that other people can use, and the process continues like a chain reaction. It's packages all the way down!

Sharing our code benefits us all

No program is an island, in other words. But that's what's wrong with the "hello, world" program in listing `hello/1`: it *is* isolated, breaking the chain of importability.

The syntax rules of Go mean that all code has to be in *some* package, and it happens that ours is in package `main`. But there's something special about the `main` package in Go: it can't be imported. That makes sense, because it could only *be* imported into some program that already has a `main` package, so we'd have a namespace conflict.

That means no one else can benefit from our wonderful code. What a shame! We're *taking* from the universal package ecosystem, as it were, but not giving anything back. That's just rude. If our program is worth writing, it's worth sharing with the millions of other Gophers who also benefit from the universal library.

This isn't just wide-eyed idealism, though there's nothing wrong with that: it also makes good commercial sense. For us to be able to write software at the scale we need today, it has to be an industrial process using standardised, modular components.

> *Today's software developer is like the colonial-era gunsmith, lovingly handcrafting every nut and screw. We haven't yet made an industrial revolution leap from "filing away at software like gunsmiths at iron bars" to "commercially robust repositories of trusted, stable components".*
> —Scott Rosenberg, "Dreaming in Code"

Good programmers, then, are always thinking in terms of writing importable packages, not mere dead-end programs. And it turns out that this is a great way to write correct, reliable software that's easy to understand and maintain, too. Modularity is just good design.

So what do we do differently when we're writing packages, not programs?

Writing packages, not programs

What Bill Kennedy has aptly called *package-oriented design* represents a fundamental shift of mindset:

> *All design decisions start and end with the package. The purpose of a package is to provide a solution to a*

> *specific problem domain. To be purposeful, packages must provide, not contain. The more focused each package's purpose is, the more clear it should be what the package provides.*
> —Bill Kennedy, [“Design Philosophy On Packaging”]

In other words, we start with some problem that we need to solve. Instead of jumping straight to a program that solves the problem, we first of all design a well-focused *package* that solves the problem, and then we can use it in a program.

The biggest shift in our thinking, then, is from solving our very specific and parochial problem, to solving a general *class* of problems that includes ours. For example, instead of writing code to calculate the square root of 2, we write a package that calculates *any* square root, and then we apply it to the number 2.

Both approaches produce the square root of 2, but the package approach is much more valuable because it also solves the square root problem for all developers, for all time. We can then contribute our package back to the universal library, so that everybody else can benefit from our work in the same way that we benefit every day from theirs.

Command-line tools

We can write as many packages as we want, of course, but nothing will actually *happen* until we run some executable binary. That means there must be a `main` package, because that's how Go builds executable binaries.

But we've already said that the substantive code in our program should be in some *importable* package, which means it can't be in `main`. So, while a `main` package has to

be present, it follows that it should do as little as possible. It should import our package and call some entrypoint function to start the real program, and that’s all.

We don’t really want to write any non-trivial *logic* in the `main` package, since it can’t be (directly) tested. And whether it’s correct or not, it can’t be imported and used in other programs, so it’s a dead end as far as the open-source community is concerned.

Let’s see what would be left if we extracted all the substantive code out of `main`, then, and replaced it with a call to some hypothetical package that does the real work.

We’ll start by creating a new folder for this project (call it `hello`, or whatever you like). Within that folder, we’ll create a file named `main.go`, containing the following code (you can copy and paste it from the example repo, if you like):

```
package main

import (
    "github.com/bitfield/hello"
)

func main() {
    hello.Print()
}
```

([Listing `hello/2`](#))

This won’t work yet, of course, because we haven’t written the `hello` package, but we can see how it *would* work.

We import the `hello` package from the module `github.com/bitfield/hello`. In Go, a *module* is a

collection of packages that share a common *version*.

Now we can use the behaviour exported by the `hello` package. To do that, we call the function `hello.Print`, which will presumably do the actual printing of “Hello, world”.

But how could we ever write such a `main` function until we actually have the `hello` package available? How can we call a function that doesn’t exist?

Well, design is always an exercise in imagination. If the thing you want already exists, there’s no need to design it! So it follows that when we design a package, we have to start by imagining what it would do.

Zen mountaineering

As we’re writing `main`, we can ask ourselves “What kind of function would we *like* to call here?” What name would make sense for it? Would it need to take any arguments? If so, what? Would it return any results? How many? What type? And so on.

We’d always have to make these decisions at *some* point, of course. The only question is “when?”

There’s a Zen saying that applies here:

> *If you want to climb a mountain, begin at the top.*

In other words, if we want to design a package, a great way to begin is by pretending it already exists, and writing code that uses it to solve our problem.

When this code looks clear, simple, and readable, we probably have at least the beginnings of a nice design. We

will probably end up tweaking it as the project goes on, but that's okay. The amount of tweaking required will most likely be less than it would if we had tried to design the whole thing up front, with no idea of how it would be used in practice.

By writing this `main` function, as minimal as it seems, we're starting this process. We've already done a little design work on `hello`, even though we haven't written a single line of code in that package.

For example, we know there will be a `Print` function that takes no parameters, returns nothing, and whose behaviour is to print a message to the terminal. At least, that's the API implied by the way we use it in `main`.

That API might have to change, and it probably will, but let's follow the process a little further to see how it works.

Guided by tests

We've decided one of the behaviours we need from the `hello` package: printing the message "Hello, world". Supposing we'd already written the code to do this, how would we know if it were *correct*?

Well, we could run the `main` package and see what happens, and that's always an option, but this kind of manual testing doesn't scale well as the software becomes more complicated. What we really want is an *automated* test, and Go has excellent support for writing such tests.

I'm going to suggest we do something that comes naturally to some programmers, but not others: namely, build our package *guided* by tests. What do I mean by that? Specifically, that we'll write the test for each behaviour *before* we write the code to implement it.

This might sound weird, but it actually makes perfect sense when you think about it. Unless you know in detail what the code should *do*, how can you write it? And to “know in detail what the code should do” is equivalent to writing an automated test for it.

In other words, we’re not thinking about tests primarily as a way of checking the correctness of code we’ve already written, though they can certainly do that. Instead, we’re treating the writing of a test as part of the *design* process.

Writing a test forces you to think clearly and precisely about the observable behaviour of the component under test, rather than the irrelevant details of its implementation. With that thinking work already done, it’s much easier to write the component itself. And when the test starts passing, we know we’ve got it right.

Building a `hello` package

Let’s try this idea with our `hello`-printing example. We’ll write a test and then see if we can come up with the right code to pass it.

Let’s add the test to our existing `hello` project folder. We’ll create a new file named `hello_test.go`, and start there.

Here’s a first attempt:

```
package hello_test

import (
    "testing"

    "github.com/bitfield/hello"
)
```

```go
func TestPrintPrintsHelloMessageToTerminal(t
*testing.T) {
    t.Parallel()
    hello.Print()
}
```

Let’s break this down, line by line. Every Go file must begin with a package clause defining what package its code belongs to. Since this code will be about testing the `hello` package, let’s put it in package `hello_test`.

We need to import the standard library `testing` package for tests, and we will also need our `hello` package.

The structure of a test

If you’re not already familiar with writing tests in Go, I recommend you read [For the Love of Go](), which will give you some helpful background on what follows.

As you probably know, every Go test function takes a single parameter, conventionally named `t`, which is a pointer to a `testing.T` struct. We can see that in the signature of our test:

```go
func TestPrintPrintsHelloMessageToTerminal(t
*testing.T)
```

This `t` value contains the state of the test during its execution, and we use its methods to control the outcome of the test (for example, calling `t.Error` to fail the test).

The call to `t.Parallel` signals that the test should be run concurrently with other tests, and is a standard prelude to any test. Now here comes the substantive part, where we actually call the function under test:

```
hello.Print()
```

It doesn't seem, from the way we've used it in our modified `main` package, that the `Print` function needs to take any parameters, and at the moment there don't seem to be any useful results it could return.

And that's the end of the test. But there's a problem: when would this test *fail*?

Tests are bug detectors

Go tests pass by default, so unless we take some specific action to make it fail (we'll see how to do that in a moment), this test will *always* pass. That might sound great, at first: we like passing tests!

But how useful is a test that can never fail? What is the point of a test, actually? Why do we write them? Partly to help us design the API of the system under test, as we've seen, but another important role of any test is as a *bug detector*.

To see how that works, let's think about potential bugs that could be in our `hello.Print` function. There are many, of course: it could print the wrong message, print the right message in the wrong language, print nothing but delete all your files, and so on.

These would all be important bugs, and we'd like to be able to detect and fix them, but let's strip it right back to basics. The most fundamental bug that could be in `Print` is that it doesn't do anything at all.

For example, suppose the implementation looked like this:

```go
package hello

func Print() {}
```

The question we need to ask ourselves here is “would our test detect this bug?” In other words, will the test fail when `Print` does nothing? Well, we already know the answer to that. The test can’t fail at all, so it can’t detect *any* bugs, including this one.

Actually, there’s one kind of bug it could detect: one that would cause the function to *panic*. It doesn’t matter whether the code explicitly calls the built-in `panic` function for some reason, or whether it does something that causes a *runtime* panic, such as accessing a non-existent slice element. In either case, the test will fail.

Being able to assure ourselves that the code doesn’t panic is better than nothing, but it’s still a pretty low bar. No code should *ever* panic, so while it’s good that tests can catch this, let’s be a bit more ambitious. Let’s extend the test to check, first, that the code does anything at all, and then, no less importantly, that it does the *right* thing.

So when should a test fail?

Most tests have one or more ways of failing under specific circumstances. In other words, we usually have some code in the test like this:

```go
if CONDITION {
    t.Errorf("test failed because %s", REASONS)
}
```

So what would `CONDITION` be in our case? It’s not easy to work out at first. Let’s remind ourselves what the desired behaviour of `Print` is. Its job is to print a hello message to

the terminal. Okay, so we can see if that’s worked by running the program and looking at the terminal, but how could we carry out that same check from a *test*?

We can’t. In general, programs have no way of accessing what they have or haven’t printed to the controlling terminal. So let’s do a little judo move. If we can’t check what we’ve printed to the *terminal*, could we check what we’ve printed somewhere else?

If so, perhaps there’s a way we can make it possible for the calling code to select where the output gets printed. When running the real program, that will be the terminal, just as before. But when running *tests*, the output could go somewhere else that the test could then inspect.

In the original program from listing `hello/1`, we used `fmt.Println` to do the printing, and we found that its output ends up going to the terminal.

So what determines this destination for output, and could we change it? Let’s take a look at the source code. In most editors, such as Visual Studio Code, you can place the cursor on some function call like this, right-click, and select *Go to definition* to see how it’s implemented (or you can just hold down the Cmd key and click on the function name).

This is helpful for navigating our own projects, but it works with imported packages too, even packages like `fmt` that are part of the standard library. This *code spelunking* is fun and interesting, and I recommend you spend some time exploring the standard library, or even packages from the universal library that interest you.

Where does `fmt.Println` print to?

Here’s what we find when we spelunk into the definition of `fmt.Println`:

```
func Println(a ...any) (n int, err error) {
    return Fprintln(os.Stdout, a...)
}
```

Notice that `Println` doesn’t seem to really do anything substantive itself. Instead, it just calls some other function, `Fprintln`, passing on the arguments `a...` it received from us, which are the things to be printed.

But it doesn’t just pass on its arguments: it adds a new one. The first argument to `Fprintln` is `os.Stdout`, which is a file handle representing the standard output. This is what actually determines where stuff ends up getting printed.

This looks promising. We can’t control where `Println` sends its output, because that’s *hard-wired*: it’s always `os.Stdout`. But we *can* control where `Fprintln`’s output goes, because the destination is the first argument to this function.

If we could construct some object that `Fprintln` will accept as a destination, then we’d have a way to solve our testing problem. So what would that look like? Here’s the signature of `Fprintln`:

```
func Fprintln(w io.Writer, a ...any) (n int, err
error)
```

The parameter we’re interested in is `w io.Writer`, so what’s `io.Writer`? It’s a standard library interface that means “thing you can write to”. Writing is the same as printing, for our purposes. Anything that conforms to the `io.Writer` interface is an acceptable destination as far as `Fprintln` is concerned.

Clearly that's true of `os.Stdout`, but what could we construct instead?

Meet `bytes.Buffer`, an off-the-shelf `io.Writer`

Like all good interfaces, `io.Writer` is easy to implement, because it's small. All we need is something with a `Write` method.

For example, we could create some struct type whose `Write` method stores the supplied text inside the struct, and we could add a `String` method so that we can *retrieve* that text and inspect it in the test.

That's not necessary, though, because such a type already exists in the standard library: `bytes.Buffer`. It's an all-purpose `io.Writer` that remembers what we write to it, so it's ideal for testing.

Let's rewrite our test using a buffer as the print destination, then. We'll start by giving it a new name:

```
func TestPrintTo_PrintsHelloMessageToGivenWriter(t
*testing.T)
```

Since we'll need a `bytes.Buffer` as our printing destination, let's create one and name it `buf`. What's next?

Now comes the call to the function under test. Since the behaviour of the function has changed, its name should change too. It now takes an `io.Writer` argument to tell it where to print its message *to*, so let's rename it `PrintTo`. And that reads quite naturally in the code:

```
hello.PrintTo(buf) // prints hello message to buf
```

Indeed, this is probably the easiest way to *invent* good names for things: write the code that refers to them, and see what makes sense in context. Then use that name.

Again, this usually works better than deciding in advance what the name should be, and then trying to jam it awkwardly into code where it doesn't fit. It's easy to tell when you're reading a program where this has happened, because the names don't seem to be very good descriptions of the things they're attached to.

Failure standards

We can now do what proved impossible with the first version of this test: we can write down the conditions under which it should fail.

First, we know what message we *want* to see printed to `buf`; it's "Hello, world". Let's name that variable `want`.

What we actually *got*, though, we'll find out when we call `buf.String`, which returns all the text written to `buf` since it was created. We'll name this value `got`.

And now we know the failure condition: if `got` is not equal to `want`, then we should fail. We have a useful test at last! And here it is:

```
func TestPrintTo_PrintsHelloMessageToGivenWriter(t
*testing.T) {
    t.Parallel()
    buf := new(bytes.Buffer)
    hello.PrintTo(buf)
    want := "Hello, world\n"
    got := buf.String()
    if want != got {
        t.Errorf("want %q, got %q", want, got)
```

```
    }
}
```

(Listing `hello/3`)

Implementing `hello`, guided by tests

So, how close are we to having this test pass? Well, that would obviously require `PrintTo` to exist, and it doesn't yet, even though we've done a lot of useful thinking about it. And we're still not quite ready to write it, because there's one thing we need first. We need to see our test *fail*.

We start with a failing test

If you think that sounds weird, I don't blame you. A failing test is usually bad news—and it would be, if we were under the impression that we'd successfully implemented the `PrintTo` function.

But we know we haven't, so if we can get the test to run at all, it *should* fail, shouldn't it? The alternative would be that it passes, even though the `PrintTo` function does nothing whatsoever. *That* would be weird.

We agree, I hope, that if `PrintTo` does nothing, then that's a bug. Would we detect that bug with *this* test? Yes, I think we would. Running this test against an empty `PrintTo` function should fail. Let's see why.

If `PrintTo` doesn't write anything to the buffer, then when we call `buf.String` we'll get the empty string. That won't be equal to `want`, which is the string "Hello, world".

That mismatch should fail the test, and it should also give us some confidence in our bug detector. No doubt there

could be other bugs in `PrintTo` that we *won't* detect at the moment. That's nearly always true.

But we feel at least we should detect *this* one, so let's find out.

Creating a module

We can't actually run the test just yet, though. At the moment, running `go test` gives us a compile error:

```
go: cannot find main module
```

This makes sense, since we haven't yet created a `go.mod` file to identify our module. A *module* in Go is a collection of (usually related) packages. While a module can contain any number of packages, it's a good idea to keep things simple and put each of our packages in its own module.

We'll do that with the `hello` package, so we'll name its containing module `hello` too. To do that, we need a little bit of paperwork.

Each Go module needs to have a special file named `go.mod` that identifies it by name. It's just a text file, so we could create it by hand, but an easier way is to use the go tool itself:

```
go mod init github.com/YOUR_GITHUB_ID/hello

go: creating new go.mod: module
github.com/YOUR_GITHUB_ID/hello
```

This name will be the import path for people who want to use our module, so that's why it needs to contain your GitHub username.

One folder, one package

But we still have a problem when we try to run the tests:

```
go test
```

```
found packages hello (hello_test.go) and main
(main.go)
```

Our old `main.go` file from listing hello/1 is still kicking around in this folder, and that's causing the compiler to get confused.

The rule in Go is that each distinct package must occupy a separate folder. That is to say, you can't have a file that declares package `hello` in the same folder as another file that declares package `main`, for example. And right now that's what we have.

Let's move `main.go` to a subfolder, then. We'll come back to it later:

```
mkdir -p cmd/hello
mv main.go cmd/hello
```

A null implementation

Let's try running the tests again:

```
go test
```

```
go build hello: no non-test Go files
```

That's absolutely correct. In order to run tests, there must be some package to build, and there isn't. So let's create a `hello.go` file that declares package `hello`. What else shall we put in it?

Well, we need the `PrintTo` function to be at least defined, or our test won't compile. On the other hand, we don't want

it to actually do anything yet, because we want to verify that our test can tell when it doesn't. So let's write a *null implementation*: just the same empty function we saw before.

```
package hello

func PrintTo(w io.Writer) {}
```

For it to compile, the function must take a parameter, and we know its type should be `io.Writer`. Nothing else is actually *required*, syntactically, since Go is quite happy for you to write empty functions that do nothing (your boss may take a different view).

But we think the test should fail, so let's see what happens:

```
--- FAIL:
TestPrintTo_PrintsHelloMessageToGivenWriter
(0.00s)
    hello_test.go:16: want "Hello, world", got ""
```

Nice! That's exactly what we hoped for, even though it *looks* like something bad happened. Don't think of it as a failing test: think of it instead as a *succeeding* bug detector. We know there *is* a bug, so if the test passed at this stage, we would have to conclude that our bug-detecting machinery were faulty.

The real implementation

The test is now doing one half of its job: detecting when the function doesn't work. The other half of its job is to detect when the function *does* work. So does it?

Let's write the real implementation of `PrintTo` and find out:

```
func PrintTo(w io.Writer) {
    fmt.Fprintln(w, "Hello, world")
}
```

Here's the result:

```
go test
```

```
PASS
```

Ideal. We now have an importable, testable package that makes our "print hello to some `io.Writer`" behaviour available to other users, provided that we remember to publish it with an appropriate software licence.

The naming of tests

Not only do we have a great test for our package, but the *name* of that test itself conveys some useful information. To help us think about what our test names are saying about the behaviour of the system, we can use the [`gotestdox`](https://github.com/bitfield/gotestdox) tool:

- https://github.com/bitfield/gotestdox

It simply rewrites the test names as space-separated words that we can read as a sentence. Here's how to install it:

```
go install
github.com/bitfield/gotestdox/cmd/gotestdox@latest
```

Let's run it in our project folder and see what it says:

```
gotestdox
```

```
github.com/bitfield/hello:
 ✔ PrintTo prints hello message to given writer
```

Nice! As our collection of tests grows, the documentation produced by `gotestdox` will be an increasingly valuable way to get an overview of what our package does, and how to use it.

Refactoring to use our new package

While we wait for the world to rush to our door to take advantage of our generous contribution to the Go universal library, let's use the new package ourselves to rebuild our `hello` command.

In fact, we only need to make minor changes to our `main` package from listing hello/2. Specifically, we now need to pass in the destination for `PrintTo` to print to:

```
package main

import (
    "os"

    "github.com/bitfield/hello"
)

func main() {
    hello.PrintTo(os.Stdout)
}
```

(Listing hello/3)

Let's run it and see if that works:

```
go run ./cmd/hello

Hello, world
```

This is great. We're back precisely where we started. From one point of view, we've added no value at all, since the program's behaviour is exactly the same. But from another point of view, we've added all the value in the world, since we created an importable package.

Someone else who wants this behaviour can now add it to their own program as a pluggable component. And because we've made it flexible about where it prints to, that program needn't even be a command-line tool. It could be, for example, a web server:

```
func main() {
    fmt.Println("Listening on
http://localhost:9001")
    http.ListenAndServe(":9001", http.HandlerFunc(
        func(w http.ResponseWriter, r
*http.Request) {
            hello.PrintTo(w)
        })
    )
}
```

(Listing hello/3)

Going further

If you're impatient to read more, by all means go on to the next chapter. But if you'd like to explore some of these ideas further, try this mini-project. It will give you a chance to practice building an importable package, guided by tests, in the way that we've talked about.

- Instead of a program that simply prints "Hello, world", try writing a program that asks the user for their name, then prints "Hello, [NAME]". Put all the substantive

behaviour of the program in an importable package that's used by `main`.

You'll find one possible solution in listing `greet/1`.

2. Paperwork

> *Yes, he did have a clean desk. But that was because he was throwing all the paperwork away.*
> —Terry Pratchett, “The Fifth Elephant”

We can feel very warm and fuzzy, after our efforts in the previous chapter, that we’ve added something useful to the wider Go ecosystem: that is, to the universal library. Okay, maybe it doesn’t seem *super* useful so far, but that’s not really for us to judge. If *we* want it, someone else might, too.

Designing packages instead of programs means we need to think about how *other* people might use our code. It’s no longer about just solving some specific problem that *we* have, but a wider class of problems. And that means we need to have a way for users of our package to specify *options*.

Making our package more flexible

Let’s put ourselves in the position of someone trying out our package, then. Maybe they’ve been struggling to write a Go

program that prints "hello, world", and they've just discovered to their delight that a new package `hello` has been published to do exactly that.

Mandatory arguments are annoying

Their first step will probably be to try it out, using a program something like the one we wrote in listing hello/3:

```
func main() {
    hello.PrintTo(os.Stdout)
}
```

From a developer experience point of view, this is *okay*, but not great. Why do we have to pass `os.Stdout` as an argument here? Isn't it obvious that's what we'll almost always want? Why make us do this apparently useless paperwork?

Well, *we* know why: that argument has to be there because in the tests, it won't be `os.Stdout`; it will be something like a `*bytes.Buffer`. Tests are great, of course, but do *users* actually want to pass this argument?

The answer is that they *might*: the HTTP handler in listing hello/3 needs to pass the ResponseWriter here so that the message will go to the user's browser instead of the operator's terminal. But surely this is a niche use case. Probably 95% of the time `os.Stdout` is what users will want.

It's a shame, then, that everybody has to pay the usability penalty of supplying this mandatory argument. Sure, it's not a big deal that you have to pass in this one little value, but it's *paperwork*. Small inconveniences add up, over a big enough API, to become a major annoyance, so we'd like to eliminate them wherever possible.

Is there any way, then, to allow users to omit this argument *unless* they want "advanced customisation" mode? What we'd *like* is for the program to print to `os.Stdout` by default, unless we choose to override that with something else (like in the test, for example).

What if we allow users to pass `nil`?

Here's one idea. Suppose, instead of requiring users to pass some valid `io.Writer` value as an argument, we let them pass `nil` instead. We can check for that in the function and set the writer to our default `os.Stdout` in this case:

```
func PrintTo(w io.Writer) {
    if w == nil {
        w = os.Stdout
    }
    ...
```

This is a *little* better, because the user doesn't have to remember to pass `os.Stdout`, specifically. Their code now looks like this:

```
hello.PrintTo(nil)
```

But this is objectively worse! We at least understood why `os.Stdout` was there, but this `nil` is completely baffling. What, you didn't know that passing `nil` means "print to standard output"? Too bad.

This can't be the right idea, can it? If there's one thing worse than mandatory paperwork, it's mandatory *meaningless* paperwork. You don't ever want your users to have to pass you a "mystery `nil`".

The standard library isn't entirely free of this problem. For example, you may well have seen code like this:

```
http.ListenAndServe("localhost:8080", nil)
```

What's that `nil` about? It's not immediately clear, is it? If that argument isn't always necessary, why does the API force us to supply it? Or if it *is* always necessary, then why is it okay for it to be `nil` here?

The answer, which again you just have to know, or go code-diving to find out, is that this is where the *handler* goes. If you don't need a custom handler, you can pass `nil`, and the result will be that the *default* handler is used. That's a little too much magic for our tastes.

Maybe some global variable instead?

If we don't want to make users pass us a mystery `nil`, we can't make the writer a parameter. If it's not a parameter, then it must be a variable. But where should that variable be set in such a way that users can override its default value if they need to?

We could make it a variable in the `hello` package, and set its default value at the same time:

```
var Output io.Writer = os.Stdout
```

Now we can drop the argument to `PrintTo` completely (and its name therefore becomes once again just `Print`):

```
hello.Print()
```

Note that we have to export the name `Output`, or we won't be able to set it from the test code like this:

```
buf := new(bytes.Buffer)
hello.Output = buf
hello.Print()
```

This gives us a tidier API, that is easier to use, and we don't mind doing a bit of paperwork in the tests for the unusual case where we need to override the default output.

However, this approach wouldn't be concurrency-safe. In other words, supposing two different goroutines (parallel tests, for example) were both trying to print something. With each one simultaneously trying to set `Output` to a different writer, chaos would ensue. At best, the program would crash; at worst, it would produce the wrong results.

While *we* don't at the moment call into the `hello` package concurrently, we certainly want *users* to be able to do that, and any mutable global state makes that impossible.

Calling `t.Parallel` in all our tests can help catch any such mistakes. If we had parallel tests that relied on setting `Output`, they couldn't possibly pass reliably if it were a global variable. If we'd accidentally made that mistake, then parallel tests would warn us about it, which is nice.

A struct is the answer

If a global variable is no good, then it follows elegantly that we need some *local* variable instead. But what?

Suppose that instead of a function, `Print` were a *method* on some struct—let's call it a `Printer`, since that's what it is.

```
type Printer struct {
    Output io.Writer
}
```

Now each `Printer` could have its own individual `Output` writer, distinct from all other Printers:

```go
func (p *Printer) Print() {
    fmt.Fprintln(p.Output, "Hello, world")
}
```

So what would user code that uses the `Printer` look like, then? Something like this, perhaps:

```go
p := &hello.Printer{
    Output: os.Stdout,
}
p.Print()
```

This seems like a step backwards, though: we've added several lines of paperwork, and we still need an explicit mention of `os.Stdout`. Couldn't we make that the default `Output`? It seems like a sensible choice.

But "default" is doing a lot of work in that sentence. Go doesn't allow us to specify default values for struct fields; they'll just get whatever the zero value is for their type. In this case, since `io.Writer` is an interface, the default `Output` will be `nil`.

That's no good, so we'll need to absorb this paperwork into a *constructor* instead. That way, we can set `Output` to the default value when we create the `Printer`, and users are welcome to set it to something else if they want to.

```go
func NewPrinter() *Printer {
    return &Printer{
        Output: os.Stdout,
    }
}
```

The result returned by `NewPrinter` is ready to use, but we *can* customise it when necessary. In the test, for example,

we'll want to set that `Output` to a buffer, so we can inspect what gets printed.

And here's what that would look like:

```
func TestPrintPrintsHelloMessageToOutput(t
*testing.T) {
    t.Parallel()
    buf := new(bytes.Buffer)
    p := hello.NewPrinter()
    p.Output = buf
    p.Print()
    want := "Hello, world\n"
    got := buf.String()
    if want != got {
        t.Errorf("want %q, got %q", want, got)
    }
}
```

(Listing hello/4)

Real users, though, don't need to set anything in order to print to the default output, which is nice:

```
p := hello.NewPrinter()
p.Print()
```

We could even eliminate the p variable altogether, and write simply:

```
hello.NewPrinter().Print()
```

Of course, now the user has to call two functions instead of one, which is annoying. Isn't there some way round that?

A convenience wrapper with defaults

Well, this is easily solved. We can just provide some trivial wrapper function that absorbs the unnecessary paperwork. Let's call it simply `Main`, since it's effectively acting as our main function:

```
func Main() {
    NewPrinter().Print()
}
```

So here's the complete program:

```
type Printer struct {
    Output io.Writer
}

func NewPrinter() *Printer {
    return &Printer{
        Output: os.Stdout,
    }
}

func (p *Printer) Print() {
    fmt.Fprintln(p.Output, "Hello, world")
}

func Main() {
    NewPrinter().Print()
}
```

([Listing hello/4](#))

Now the user really is living their best life. All they need to do is call:

```go
func main() {
    hello.Main()
}
```

This is, in fact, exactly the kind of *zero-paperwork* approach that the standard library takes in many places. For example, the `net/http` package has a nice, defaults-oriented way to make HTTP requests:

```go
resp, err := http.Get("https://example.com")
```

Of course, in more sophisticated programs we would probably want to customise certain things about the HTTP client's behaviour, such as timeouts. In that case, where we *want* to do some paperwork, we can:

```go
client := http.Client{
    Timeout: 10 * time.Second,
}
resp, err := client.Get("https://example.com")
```

This way, we get the best of both worlds: sensible default behaviour with no paperwork, or customisable behaviour with *minimal* paperwork. That's a result.

A good rule to remember in this context, coined by researcher Alan Kay, is "Simple things should be simple, but complex things should be possible." It captures our user-focused, paperwork-reducing approach perfectly.

A simple line counter

Let's apply what we've learned so far to write a slightly more useful program: one that counts the number of lines in its input and prints the result to its output. Again, this is straightforward to write directly as a `main` package:

```go
func main() {
    lines := 0
    input := bufio.NewScanner(os.Stdin)
    for input.Scan() {
        lines++
    }
    fmt.Println(lines)
}
```

(Listing count/1)

Of course, this isn't the only way to count lines, or even the best. If we wanted to spend memory to buy speed, for example, we could read the entire input with `io.ReadAll`, and count the newlines in the resulting slice with `bytes.Count`.

Focus on behaviour, not implementation

Personally, I don't need this program to be any faster than it is, and I don't like programs whose memory footprint scales linearly with their input. It's too easy to crash such programs by sending them a lot of data, either deliberately or accidentally.

Any time you take input, assume it will be of arbitrary size, larger than available memory. Process it in bite-size chunks, rather than all at once. If we only need a line at a time, we should only *read* a line at a time.

In any case, the actual implementation doesn't matter here: we can treat it as a black box. Indeed, it *should* be trivially replaceable, shouldn't it? We're primarily concerned here with the user experience and friendliness of the package.

How should we go about turning it into an importable package? Try it yourself first.

GOAL: Design and implement a line-counting package, test-first, in the same way we did with `hello`. Include a `main.go` that uses your package to build a line-counting CLI tool.

HINT: Remember, you don't need to solve the problem of how to count lines—we've already done that using `bufio.Scanner` in listing count/1.

The challenge here is how to turn this behaviour into an importable *package* (shall we call it `count`?). Once we have that, we can easily write a zero-paperwork `main` function to use it.

We'll also need to be able to test it, and that suggests we won't want to read directly from `os.Stdin`. Instead, think about a counter object that could be configured with some `Input`, just as the "hello printer" object had a field to specify its `Output`.

Actually, the line counter is really similar to the hello printer in many respects, so if you're not sure how to get started, have a look at listing hello/4 for inspiration.

One possible first version

SOLUTION: As usual, let's start with a test. What would a test for the line counter look like? Again, let's refer to listing hello/4 for some inspiration. Here's the test for the hello printer:

```
func TestPrintPrintsHelloMessageToOutput(t
*testing.T) {
    t.Parallel()
    buf := new(bytes.Buffer)
    p := hello.NewPrinter()
    p.Output = buf
```

```
    p.Print()
    want := "Hello, world\n"
    got := buf.String()
    if want != got {
        t.Errorf("want %q, got %q", want, got)
    }
}
```

(Listing hello/4)

We'll be doing something very similar in the test for the line counter. For example, we could call `NewCounter` to get a counter, and then set its `Input` to a `bytes.Buffer`. We'll need to be able to "pre-load" the buffer with some test input —a string containing a few line breaks should be fine.

The next question is how we get the results. For users of a command-line tool, it would be fine if the counter simply printed its result to the standard output (or any `io.Writer`, to make testing easier).

But that wouldn't be convenient for those who want to consume our package in their *own* programs. They'd like the line count to be returned as a Go number—that is, as an `int`.

And that's more flexible in general: after all, if we have an `int`, it's easy to print it out when we want to. But if all you can do is print to some writer, it's a lot trickier to turn that data back into a number.

So let's call our counting method `Lines`, since that's what it returns: the number of lines counted in the input.

It sounds as though we've argued ourselves into something like the following test:

```
func TestLinesCountsLinesInInput(t *testing.T) {
    t.Parallel()
    c := count.NewCounter()
    c.Input = bytes.NewBufferString("1\n2\n3")
    want := 3
    got := c.Lines()
    if want != got {
        t.Errorf("want %d, got %d", want, got)
    }
}
```

(Listing count/2)

And here’s the corresponding package:

```
package count

import (
    "bufio"
    "io"
    "os"
)

type counter struct {
    Input io.Reader
}

func NewCounter() *counter {
    return &counter{
        Input: os.Stdin,
    }
}

func (c *counter) Lines() int {
```

```go
    lines := 0
    input := bufio.NewScanner(c.Input)
    for input.Scan() {
        lines++
    }
    return lines
}

func Main() {
    fmt.Println(NewCounter().Lines())
}
```

(Listing count/2)

Note that all we need to do to use this package from a command-line program is to call `count.Main`. It doesn't take any parameters, or need any configuration. It just does the right thing. How convenient!

```go
func main() {
    count.Main()
}
```

(Listing count/2)

What about configuration?

To recap, we've said that practical programs often need some kind of configuration in order to be flexible. A useful pattern in Go is to have some kind of object—some struct, that is—with fields that we can set. For example, with the hello printer we can set the `Output`, and with the line counter we can change its `Input`.

This is fine, but how well would it scale if there were *lots* of things to configure? It might be annoying to have to write

separate assignment statements, one for every struct field we want to set, if there were dozens of such fields.

We *could* make these values extra parameters to the constructor, but as we've seen, there are several problems with that idea. Go doesn't allow us to omit arguments to a function, so users would have to do annoying paperwork such as passing a "mystery `nil`". Let's rule that approach out of court, and keep thinking.

Config structs don't solve the problem

A common, but not very satisfactory pattern, is to create some *config struct* type and pass that to the constructor instead. For example:

```
type Config struct {
    Input io.Reader
    Output io.Writer
}

func MakeCounterWithConfig(config Config) *counter
{
    c := NewCounter()
    c.Input = config.Input
    c.Output = config.Output
    return c
}
```

But this seems a bit pointless. We already *have* a struct that contains the config information: the `counter` struct itself. Adding another struct type doesn't help users, it only creates *more* paperwork. And now we have to laboriously copy all the values from one struct into the other.

It's fine to just have users set fields directly on the object itself, as we've seen with the hello printer and the line counter. Most of the time, indeed, this is the best solution, especially if we can provide sensible defaults using a constructor.

But, just for fun, can we think of anything better? Is there a way we could configure many different things about our object, but set them all in a single call to the constructor, without having to do any paperwork?

An elegant option API

Using our Zen pre-mountaineering technique again, let's try to write roughly the code that we wish were possible, and then figure out how to *make* it possible.

Imagine, for example, that we could write something like this:

```
c := count.NewCounter(
    count.WithInput(os.Stdin),
    count.WithOutput(os.Stdout),
    ... // Maybe more options here
)
```

In other words, `NewCounter` can apparently take *any* number of arguments, each of which configures something different about the counter.

We could have done that with ordinary parameters, of course, but, crucially, we wouldn't be able to *omit* any of them. That's not the case here, because we also want to be able to write, for example:

```
c := count.NewCounter()
```

So, is this possible?

Okay, so what's an "option"?

First, let's look at these arguments to `NewCounter`. Clearly there's some magic going on here, because we already said that Go doesn't allow us to omit arguments to a function. That's true, except in one rather special case.

Recall that in the first chapter, we looked at the implementation of `fmt.Println`, and this was its signature:

```
func Println(a ...any) (n int, err error)
```

The `...` in the parameter list means "any number of". Indeed, `Println` takes as many, or as few, arguments as we care to give it:

```
fmt.Println()
fmt.Println("hello!")
fmt.Println("eggs", 61, "new pence a dozen")
```

That's the magic of `...` at work. The technical way to put this is to say that `Println` is *variadic*, meaning that it accepts a variable number of arguments (including none at all). And that's just what we want with `NewCounter`, too.

But what *type* will these arguments be? We can only choose one type, and it must apply to all our arguments.

For example, `Println` declares its arguments as type any, which means exactly what you'd expect: literally any type of value is allowed. That makes sense, because `Println` can print any random junk we care to send it.

But that wouldn't work for us here. For example, this wouldn't be meaningful:

```go
c := count.NewCounter("eggs")
```

Right? We can't just have users pass in random junk and expect it to do anything sensible to the counter. In fact, the arguments to `NewCounter` represent something specific: let's call them *options*.

An "option", we'll say, is something that sets some *field* of the counter to some *value*. For example, the `WithInput` option sets the input field to something.

How could that work? Well, let's simply wave a wand and imagine that there's a type `option` that will take care of this for us. So now at least we can write the beginning of `NewCounter`:

```go
func NewCounter(opts ...option) *counter {
    c := &counter{
        input:  os.Stdin,
        output: os.Stdout,
    }
```

So far, so good. We still don't know what kind of type `option` is, but maybe we can ask instead: what would we *do* with an option?

Well, we'd *apply* it to the counter. In fact, the word "apply" is a clue that maybe an option could be a function. What about a function that takes the `*counter` itself as a parameter, and sets the appropriate field to the value the user wants?

We can receive any number of such options, so a `range` loop seems appropriate. For each option, then, we would call it as a function, passing it the counter. After we've applied all the options, the counter will be set up correctly.

```
for _, opt := range opts {
    opt(c)
}
```

This seems promising, because no matter what the option actually does, its function signature is always the same. We can now define our `option` type as simply this:

```
type option func(*counter)
```

Options are functions

At the end of our `range` loop, then, we will have applied all the options we received, and the fully-configured `counter` will be all ready to return.

So here's what that looks like:

```
func NewCounter(opts ...option) *counter {
    c := &counter{
        Input:  os.Stdin,
        Output: os.Stdout,
    }
    for _, opt := range opts {
        opt(c)
    }
    return c
}
```

The pieces are falling neatly into place. The only missing piece, in fact, is how we create these `option` functions in the first place. Should we make the user supply them as function literals? That sounds like paperwork:

```
// Don't make me write this!
c := count.NewCounter(
```

```
    func (c *count.Counter) {
        c.Input = inputBuf
    },
    func (c *count.Counter) {
        c.Output = outputBuf
    }
)
```

The only things the user is actually *supplying* here are `inputBuf` and `outputBuf`; the rest is boilerplate.

Let's eliminate that by providing an *option constructor* for each field that we can set. In other words, rather than have to write the option function themselves, users can simply call a function to create it.

Here's a suitable constructor:

```
func WithInput(input io.Reader) option {
    return func(c *counter) {
        c.Input = input
    }
}
```

Now all users have to write is, for example:

```
c := count.NewCounter(
    count.WithInput(inputBuf),
)
```

How delightful!

"Always valid" fields

A nice consequence of this approach is that now, since users don't ever need to set the struct fields directly, we can make them unexported. Now that the only way to set the

fields is by calling some function, we can also take advantage of that to *validate* the settings.

For example, we might want to ensure that the input and output are never `nil`, which would cause a panic if the program tried to use them. That's great, and we can certainly check those values, but what if they fail the check? In general, what should we do when *invalid* options are supplied?

We could silently ignore them, but that doesn't sound right. The Go-like thing to do here would be to return an error, so let's do that:

```
func WithInput(input io.Reader) option {
    return func (c *counter) error {
        if input == nil {
            return errors.New("nil input reader")
        }
        c.input = input
        return nil
    }
}
```

Since the only way users can set this field is by calling our validating `WithInput` function, we can now be sure that the value of `c.input` is *always valid*. This is much better than having to check it every time we use it, and safer, too.

This needs a little tweak to our `option` type, but we can do that:

```
type option func(*counter) error
```

And since our `option` function now returns `error`, we'll need to receive and check that error in the apply-options

loop in `NewCounter`.

That, in turn, means that `NewCounter` also needs to return `error` (along with `counter`). And that makes sense: if you supply options that happen to be invalid for some reason, then `NewCounter` should tell you about it.

Here's the updated constructor:

```
func NewCounter(opts ...option) (*counter, error)
{
    c := &counter{
        input:  os.Stdin,
        output: os.Stdout,
    }
    for _, opt := range opts {
        err := opt(c)
        if err != nil {
            return nil, err
        }
    }
    return c, nil
}
```

(Listing count/3)

Some internal paperwork

Now that `NewCounter` returns `error`, we'll also need to update our `Main` convenience function. Previously, this was all we needed:

```
func Main() {
    fmt.Println(NewCounter().Lines())
}
```

Now we’ll have to receive the `counter` and `error` values from `NewCounter`, then call the `Lines` method on the `counter`. But what should we do with that error value? Ignore it?

```
func Main() {
    // I have a bad feeling about this...
    c, _ := NewCounter()
    fmt.Println(c.Lines())
}
```

We might salve our guilty conscience by saying “Well, I don’t pass any options here, so it can never return `error` in that case.” But who says? Actually, if `NewCounter` returns an error when given no options, that’s the kind of *disastrous* bug that we’d absolutely want to know about straight away, isn’t it?

Handling internal errors

Any time you find yourself thinking “It’s okay to ignore this error, because with the program as it currently stands, the error can never be non-nil”, think again.

First, you could be wrong about that. With a sufficiently complex program, you’re almost guaranteed to be wrong about that.

Second, the program is bound to change in the future, and the person changing it might not know that returning an error from `NewCounter` could blow something up in a function far, far away.

Ignoring errors based on unstated and possibly faulty assumptions is like leaving a land mine in the code for someone to step on later; most likely your future self. After all, there are more ways for things to go wrong than for

them to go right, so it’s totally fine for the majority of our code to be about error handling. Indeed, it’s statistically inevitable.

If ignoring the error from `NewCounter` is unacceptable, what can we do instead? Return it?

```
func Main() error {
    c, err := NewCounter()
    if err != nil {
        return err
    }
    fmt.Println(c.Lines())
}
```

But the whole reason for writing `Lines` in the first place was to create a “minimal `main`” with no annoying paperwork. Well, we just created some new paperwork:

```
func main() {
    err := count.Main()
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
}
```

If the only person who will ever call `Lines` can’t do anything useful with the error except log it and crash, we may as well do that directly:

```
func Main() {
    c, err := NewCounter()
    if err != nil {
        panic(err)
    }
```

```
    fmt.Println(c.Lines())
}
```

(Listing count/3)

There is no useful information we can give the user, because the user can't fix our program, and this is definitely an internal program bug. That's exactly what `panic` is for: reporting unrecoverable internal program bugs.

It's not generally a good idea to call `panic` outside of the `main` package, because if somebody is using your package to build their own application, they don't want it unexpectedly panicking. You're not in a position to judge what is and isn't a recoverable error for *them*, so the right thing to do with errors in general is to return them to the caller and let them decide.

However, `Main` is such a CLI-specific function that we can regard it as, essentially, a delegated equivalent of the real `main`. I don't love this kind of semi-hidden panic, but I prefer it to making the user handle errors that aren't their fault and can never happen anyway.

Many software engineering decisions, indeed, amount to choosing the least worst of a bunch of unsatisfactory solutions. Sorry about that.

A line counter with options

So here's the `count` package as it currently stands:

```
package count

import (
    "bufio"
    "errors"
```

```
	"io"
	"os"
)

type counter struct {
	input  io.Reader
	output io.Writer
}

type option func(*counter) error

func WithInput(input io.Reader) option {
	return func(c *counter) error {
		if input == nil {
			return errors.New("nil input reader")
		}
		c.input = input
		return nil
	}
}

func WithOutput(output io.Writer) option {
	return func(c *counter) error {
		if output == nil {
			return errors.New("nil output writer")
		}
		c.output = output
		return nil
	}
}

func NewCounter(opts ...option) (*counter, error)
{
```

```
    c := &counter{
        input:  os.Stdin,
        output: os.Stdout,
    }
    for _, opt := range opts {
        err := opt(c)
        if err != nil {
            return nil, err
        }
    }
    return c, nil
}

func (c *counter) Lines() int {
    lines := 0
    input := bufio.NewScanner(c.input)
    for input.Scan() {
        lines++
    }
    return lines
}

func Main() {
    c, err := NewCounter()
    if err != nil {
        panic(err)
    }
    fmt.Println(c.Lines())
}
```

(Listing count/3)

Here’s the test:

```go
package count_test

import (
    "bytes"
    "testing"

    "github.com/bitfield/count"
)

func TestLinesCountsLinesInInput(t *testing.T) {
    t.Parallel()
    inputBuf := bytes.NewBufferString("1\n2\n3")
    c, err := count.NewCounter(
        count.WithInput(inputBuf),
    )
    if err != nil {
        t.Fatal(err)
    }
    want := 3
    got := c.Lines()
    if want != got {
        t.Errorf("want %d, got %d", want, got)
    }
}
```

(Listing count/3)

And here's the `main.go`:

```go
package main

import (
    "github.com/bitfield/count"
)
```

```go
func main() {
    count.Main()
}
```

(Listing count/3)

This *functional options* pattern is becoming popular in Go, and you may well have encountered it before, but perhaps it wasn't always quite clear what problem it was solving. Approaching it this way, by degrees, and considering the various alternatives, helps make it easier to understand *why* we use functional options, and why they work the way they do.

It does require a bit of extra machinery on the package side, but since you can use essentially the same pattern for every Go package you ever write, it's a nice tool to have in the box.

Methodical options

Another way to use functions to set options, instead of passing them as arguments to the constructor, is to add methods on the object itself.

For example, to set the line counter's input reader, we could have written a method like this:

```go
func (c *counter) WithInput(input io.Reader)
*counter {
    c.input = input
    return c
}
```

How would we call it? Well, perhaps something like this:

```
c := count.NewCounter()
    .WithInput(os.Stdin)
    .WithOutput(os.Stdout)
    ... // maybe more options
```

We'd need to change `NewCounter` to return just a `*counter`, and no `error`, but that's okay: if `NewCounter` no longer needs to take any arguments, then it no longer needs to return `error`.

Arguably, using methods to set our options—we might call them *methodical options*—is no less clear than the functional style. And we don't need to repeat the package prefix for each option, which is nice.

You'll see this style used in some Go tools, but I can't say I'm especially keen on it myself. Since we can only return the counter itself from each of these methods, we can't also return `error`; that would break the method chaining. So we can't validate the arguments to option methods.

There's a more subtle problem, too. The user isn't obliged to call these methods before using the object: they can call them at any time. That *might* be okay, or it might not. Will your object handle unexpected changes to its configuration after it's started doing things? Well, it had better!

If you *want* users to be able to change the configuration of the object while using it, then clearly this is the way to go, and that wouldn't be possible with functional options. On the other hand, it's easier to write some programs if we can guarantee that the configuration won't suddenly change underneath us.

Going further

Here's an idea you might like to try get some practice with these techniques.

- Write a Go tool that can search its input for lines containing a given string, and print them to its output.

  For example, if the search string is `hello`, and you feed the program some text file, the program should print out all the lines in the file that contain `hello` as a substring.

  You'll find one possible solution in listing `match/1`.

3. Arguments

> *As far as the customer is concerned, the interface **is** the product.*
> —Jef Raskin, “The Humane Interface”

There are many useful programs that operate only on their standard input and output, such as the line counter from the previous chapter.

However, we could expand the range of behaviour available if we had a way to pass the program *arguments* on the command line.

For example, right now we have to run our `count` program by piping text into it, like this:

```
cat foo.go | count
```

But if you didn’t already know this, you *might* assume that the program is supposed to be passed the name of some file on the command line. For example:

```
count foo.go
```

Right now, that command would simply do nothing but pause forever, waiting for data on its standard input. It seems a shame to punish users for a completely understandable (indeed, predictable) mistake. One thing about great designs is that they're very intuitive.

Designing the behaviour

Good programs make themselves hard to misuse. The simplest, most obvious thing someone might try should *just work*. So, if the user supplies a filename on the command line, the program should do the right thing with it.

How can we arrange that?

Deciding the scope

As usual, let's start by trying to describe the behaviour we want in *words*, and then refine that description into the form of a test. We might start with something like:

> "If a filename is specified on the command line, the counter should count the lines in the specified file. Otherwise, it should count lines from standard input as before."

Perhaps the thought already occurred to you that it would be handy to specify *multiple* files on the command line, or to use the shell's globbing facility (so that we could write `count *.go`, for example).

Yes, it would, but let's rein in our ambition for a moment. A good question to ask when you're trying to write any program is "What simpler version of this problem could I solve first?"

If we can solve the problem for *one* file, we can presumably extend that solution to multiple files. So let's proceed by easy stages.

First, if the test is going to supply command-line arguments to the `count` package, how is it going to do that?

Testing CLI arguments

As you probably know, a Go program's command-line arguments are available to it as `os.Args`, in the form of a slice of strings. So the `count` package could just look at `os.Args` and interpret any arguments as input filenames. That's straightforward.

But how are we to supply such arguments in a *test*? That might be a little more tricky.

Our first idea might be to *set* `os.Args`. It's a variable: specifically, an exported package-level variable in the `os` package, so we have "permission" to set it, if you like. But is that a good idea?

Almost certainly not. We determined in an earlier chapter that mutating global state is a recipe for disaster. That certainly applies here. For example, we may well want to run several parallel tests, each of which would need to set `os.Args` to a different value, and these would interfere with each other.

So setting `os.Args` directly is a no-no. What can we do instead?

We already constructed some *option* functions in previous chapters to control the behaviour of the counter object, such as `WithInput` to set its input reader. Suppose there were some option like `WithInputFromArgs`?

This idea seems promising, because the option could take a `[]string` as the parameter. This matches the type of `os.Args`, so we'll be able to pass that value when running the program for real.

But in the test, we can pass any strings we want, to simulate the various command-line arguments that a user might provide.

A first attempt

Let's try to rough out the test, and see if we run into any design issues. We know that the test must call `NewCounter`, since that's the only way it can pass options.

Let's start with that function call, which must come roughly in the middle of the test, and then work both backwards (setting up the necessary world) and forwards (checking the results against expectations).

So we know there'll be a function call like this in the body of the test:

```
c, err := count.NewCounter(
    count.WithInputFromArgs(args),
)
```

Working backwards from this, we can see that we need some variable `args` to pass to `WithInputFromArgs`. We've said we want to be able to pass a single filename, and have the counter read input from the specified file. So first of all we'll need that file to exist; let's create it. Where?

Test data files

It's conventional in Go to put test data files in a subfolder named `testdata`, so we'll create that folder first, and then,

within it, let’s create a suitable file.

Creating the test data

We could call it anything, but it’ll be helpful to name the file after the test *case* that it represents. So something about the number of lines it contains would be logical. How many lines should we add to the file?

An *empty* file would be pretty pointless, so if zero lines is not enough, what about just one line? Well, it’s possible to imagine faulty line counting code that would *always* report 1, no matter the number of input lines. That’s less likely for two lines, but since the lines themselves needn’t be very long, let’s stretch out a bit and give ourselves three lines to play with.

Having decided on a three-line test file (the lines “1”, “2”, and “3” would be fine, though it shouldn’t matter for our purposes what the lines actually contain), we now know what to name it: `three_lines.txt` (for example).

Name a test fixture for the case it demonstrates. When we have many test files for many cases, this will be very helpful, because we’ll know which file belongs to which case.

Using the data in a test

There’s no more world setup needed, so we can turn now to the remainder of the test. As before, we should set our `want` to 3, call `c.Lines` to get our `got` value, and compare them. So here’s the completed test:

```
func TestWithInputFromArgs_SetsInputToGivenPath(t
*testing.T) {
    t.Parallel()
```

```
    args := []string{"testdata/three_lines.txt"}
    c, err := count.NewCounter(
        count.WithInputFromArgs(args),
    )
    if err != nil {
        t.Fatal(err)
    }
    want := 3
    got := c.Lines()
    if want != got {
        t.Errorf("want %d, got %d", want, got)
    }
}
```

(Listing count/4)

By the way, even though `"testdata/three_lines.txt"` looks like a Unix-specific file path, with components separated by a forward slash, it nevertheless works on all platforms. Although Windows has traditionally used a backslash as its default path separator, a forward slash works too. So there's no need to clutter up your code with calls to `filepath.Join`, for example.

The failing test

We should now be at the point where the test fails to compile because the relevant method is undefined. Let's see:

```
./count_test.go:12:3: undefined:
count.WithInputFromArgs
```

The next step, again following the workflow, is to write a null implementation of `WithInputFromArgs`, so that we can see the test fail:

```go
func WithInputFromArgs(args []string) option {
    return func (c *counter) error {
        return nil
    }
}
```

This is straightforward: the test code already requires that the function takes a `[]string` parameter and return `option`. That in turn requires that we return a suitable function literal, that must at least return `nil`. And, for now, the least is what we're going to do.

Once again applying our precognitive powers, we feel that the test should fail, but why exactly? Thinking it out, we can see that since the option returned by `WithInputFromArgs` does nothing at all, the line counter has no way of knowing what file to read from, so by default it will presumably read from standard input. As we're not supplying any standard input, the result should be a count of zero lines, as against the test's expectation of 3.

Let's find out if our prediction is correct by running `go test`:

```
--- FAIL:
TestWithInputFromArgs_SetsInputToGivenPath (0.00s)
    count_test.go:20: want 3, got 0
FAIL
```

If we saw anything else here, we'd be somewhat puzzled (a test pass would be even more surprising). But everything seems in order.

GOAL: Make this test pass.

Implementing file-reading

HINT: The simplest way I can think of to implement this behaviour is to have the `WithInputFromArgs` option open the specified file and attach it to the counter's `input` field.

Then the counter doesn't have to do anything special when it starts counting. For example, it doesn't have to worry about opening the file—that's already been taken care of. In the "real" program (that is, when users run the command, rather than when we run tests), the input will be `os.Stdin`, which is always already open.

SOLUTION: Here's my version:

```
func WithInputFromArgs(args []string) option {
    return func (c *counter) error {
        f, err := os.Open(args[0])
        if err != nil {
            return err
        }
        c.input = f
        return nil
    }
}
```

This passes the test, so in theory, we're done here. But naïve tests don't always tell the whole story. While the program behaves as specified, there are at least two problems with this code.

GOAL: Can you spot both problems? (If you can find more than two, you get extra credit.)

HINT: What happens if `args` is empty? Also, when does `f` get closed?

Empty slice checking

SOLUTION: Problem one is that we assume the `args` slice contains at least one element. If it doesn't, the code will panic, because it references `args[0]`. Indeed, it's never safe to reference a slice index unless you know that the slice contains that many elements. I call this a *YOLO slice access*; "you only live once" can be a good strategy for some decisions, but usually not in programming.

In the test, we passed a one-element slice, so this issue was accidentally hidden. How can we fix this problem, test-first?

GOAL: Fix this problem, test-first.

HINT: We have, in fact, just thought of a new behaviour, that we hadn't previously realised we needed. Specifically, if we pass a nil or empty slice to `WithInputFromArgs`, the program should not panic. Good programs don't panic, and nor do good programmers.

What should happen instead? Well, we started out by wanting to let users pass filenames on the command line, and we've done that. But if they don't happen to give any filenames as arguments, it makes sense for the program to just read from standard input, as before.

More specifically, we can say that if the `WithInputFromArgs` option is supplied to `NewCounter`, but there are no arguments, the counter should simply ignore the option and use its existing configured input. That'll be the standard input unless it's already been overridden by some `WithInput` option.

Testing the "no args" behaviour

SOLUTION: Now that we have a clear idea about the behaviour we want, we can express it in the form of a test. Something like this:

```
func TestWithInputFromArgs_IgnoresEmptyArgs(t
*testing.T) {
    t.Parallel()
    inputBuf := bytes.NewBufferString("1\n2\n3")
    c, err := count.NewCounter(
        count.WithInput(inputBuf),
        count.WithInputFromArgs([]string{}),
    )
    if err != nil {
        t.Fatal(err)
    }
    want := 3
    got := c.Lines()
    if want != got {
        t.Errorf("want %d, got %d", want, got)
    }
}
```

(Listing count/4)

What's going on here? First, we set up some input consisting of a buffer containing three lines. Then, we call `NewCounter` and use `WithInput` to set the counter's input to our buffer.

But we *also* pass `WithInputFromArgs` with an empty slice of strings, simulating what happens when a user runs the program with no command-line arguments. We already know that this will panic with the current implementation, and this test proves it:

```
--- FAIL: TestWithInputFromArgs_IgnoresEmptyArgs
(0.00s)
```

```
panic: runtime error: index out of range [0] with
length 0
[recovered]
```

In order to make the test pass, we need to prevent that panic, and also ensure that if no arguments are supplied, the counter's input is not changed.

GOAL: Make this test pass.

HINT: To avoid panicking when we reference an `args[0]` that doesn't exist, we need some kind of `len` check beforehand. What should we do in the zero-length case? Well, nothing. If there are no arguments, there's nothing for us to do, so we can just return `nil`. This makes the test pass without panicking.

SOLUTION: We can add a check like this to the beginning of the function returned by `WithInputFromArgs`:

```
if len(args) < 1 {
    return nil
}
```

Closing files and other resources

We can turn now to the second problem: the file `f` is never closed, meaning that the program leaks a resource (in this case a file handle). Normally we would defer calling `f.Close` as soon as we established that we successfully opened the file, but that won't work here.

Since the program exits after fully reading the file, thus releasing all its resources in any case, this might not seem

like a big deal. But, remember, we're writing packages, not programs.

We should always treat resource leaks as *potentially* serious, then, because we don't know how other developers will be using our code. It might be in a long-running program, where running out of resources would eventually cause it to crash.

Updating the user interface

Let's save that interesting problem for an end-of-chapter boss challenge. For now, let's put the finishing touches to our updated program.

Now that we can accept text for line-counting from both the standard input and from filenames supplied on the command line, we'll need to change our `Main` function to include that behaviour:

```
func Main() {
    c, err := NewCounter(
        WithInputFromArgs(os.Args[1:]),
    )
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    fmt.Println(c.Lines())
}
```

Why do we pass `os.Args[1:]` rather than just `os.Args`? The first element of `os.Args` is always the pathname of the running binary, which we're not interested in, so we can skip it.

Instead of calling `panic` in case of error, we now report it to the standard error writer (`os.Stderr`) and terminate the program with exit status 1, indicating an error.

Some user testing

Let's run the program and check that it works as expected. First, counting lines from standard input, as before:

```
echo hello | go run ./cmd/count
```

```
1
```

Now we can try out passing a filename on the command line:

```
go run ./cmd/count testdata/three_lines.txt
```

```
3
```

Finally, let's see what happens if we both pass a filename *and* send it text on standard input. Which one will take precedence?

```
echo hello | go run ./cmd/count
testdata/three_lines.txt
```

```
3
```

It turns out that a supplied filename causes standard input to be ignored, which makes sense, and matches what users would expect.

Setting exit status

It might have occurred to you that the *error behaviour* of our program is slightly different now that it takes command-line arguments. The only error the program needed to worry

about before was if `NewCounter` was called with some invalid option, such as a nil input reader.

And, since `NewCounter` was only called internally, within the package, we felt justified, though not ecstatic, about using `panic` to signal this kind of internal programming mistake.

True, there could also have been an error while reading the input stream, but in this case `input.Scan` would have returned `false`, and we would have finished scanning and reported the number of lines successfully read up to that point, if any. That’s not unreasonable, but now things are getting a bit more complicated.

Now, for example, the user can supply the name of a file on the command line that *doesn’t exist*, or at least isn’t readable for some reason. It’s not sensible to silently ignore a problem like this, because that can’t be what the user wants. We need to report the error so that they can decide what to do.

As you may know, programs return an integer *exit status* value to the operating system when they’re done. Conventionally, this is zero if everything’s okay, or greater than zero otherwise. Any non-zero value indicates an error, and programs are free to use different values to signal different kinds of problem if they want to.

The programs we’ve written so far have *implicitly* returned a zero exit status, because that’s what happens when the `main` function returns. If you want to specify a non-zero exit status, that’s done by calling the `os.Exit` function with the value you want.

In fact, we used this already, in the `Main` function in listing count/4, to terminate the program with exit status 1 if there’s an error opening the user’s specified file. But this

isn't really ideal, because now people using our package just have to *know* that calling `Main` can exit the program.

They might not want that behaviour; for example, they might want to try to recover from certain errors, perhaps by prompting the user to enter a different filename. Anyway, we'd prefer `Main` not to arrogate this decision to itself, but instead *always* return to `main`, whether there's an error or not.

Let's make a small change to `Main` to allow for this. We'll have it return an `int` representing the exit status, and then `main` can pass that to `os.Exit` (or do something else if it wants to).

```
func Main() int {
    c, err := NewCounter(
        WithInputFromArgs(os.Args[1:]),
    )
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        return 1
    }
    fmt.Println(c.Lines())
    return 0
}
```

(Listing count/4)

And here's the updated `main` function that calls it:

```
func main() {
    os.Exit(count.Main())
}
```

(Listing count/4)

Very simple, but it strikes a nice balance between flexibility and paperwork.

Test scripts

We saw earlier in this chapter that, when a program needs to read its command-line arguments, they will come ultimately from `os.Args`. But, because this is difficult to *test*, we designed the `count` package to take an arbitrary `[]string` as its argument to `WithInputFromArgs`.

When we call the `NewCounter` function from `Main`, we *pass* it `os.Args` as the value of this slice. In the tests, though, we're free to pass whatever strings we like as the program "arguments".

This is great, and it's always a good idea to decouple our packages in this way so that we make them more flexible, and less dependent on being used in a very specific way.

However, it's not always possible or convenient to do this decoupling, and there's a limit to how well we can simulate the use of a package from the command line when all we can do is call its functions.

Running the program as a binary

But are we restricted to just calling functions? Could we, for example, execute the program as a binary, passing it arbitrary arguments, from a Go test, and see what it produces?

Of course, because we can do anything with Go. We could, for example, write a test that executes the Go compiler on our `main.go`, producing a binary, and *then* executes that binary in the way we want.

This is all perfectly possible using the `os/exec` package. We can execute programs as subprocesses, from our tests, and obtain their standard output or standard error streams as `io.Reader` values, for example.

There's some plumbing involved, though, and it's annoying to have to repeat it all for every program we write, simply to execute our `main` package as a binary and see its output.

Isn't there a better way? Hasn't some public-spirited citizen, for example, contributed a package to the universal library that would help us here?

Introducing `testscript`

Indeed they have. The `testscript` package, originally written by the Go team to test the go tool itself, has been adapted and extended to become a general-purpose utility for testing *any* command-line tool. Let's see how it works.

Using `testscript`, we can frame our tests as simple text files that look very like shell scripts. Here's an example:

```
stdin three_lines.txt
exec count
stdout '^3\n$'

-- three_lines.txt --
this input
contains
three lines
```

(Listing count/5)

The line beginning `stdin` sets the standard input for any subsequent commands to come from the file

`three_lines.txt`. Then the `exec count` line executes the `count` command (that is, our line counting program).

We then assert something about the expected output, using `stdout`: specifically, that it contains "3", which is what the program should produce if it's working correctly.

This script, in other words, *is* a test, hence the name 'testscript'. There are two ways it could fail. First, executing the `count` command could return a non-zero exit status. Second, its output might not contain the expected text.

Invoking test scripts

How do we run this script as part of the tests executed by `go test`, then? All we need to do is to add a Go test like this:

```
func Test(t *testing.T) {
        t.Parallel()
        testscript.Run(t, testscript.Params{
            Dir: "testdata/script",
    })
}
```

(Listing count/5)

Given everything we've said about writing useful test names, you might be surprised that this test is called, simply, `Test`. That's because it doesn't assert any behaviour by *itself*: it exists merely to run our test scripts.

Each of these scripts will be a subtest of the parent test, and the name of that subset is derived from the filename of the script. So, for example, if our script file is named:

```
count_counts_lines_from_stdin.txtar
```

then its behaviour, as formatted by `gotestdox`, will be:

```
✔ Count counts lines from stdin
```

And notice that the parent test doesn't do anything except call `testscript.Run`, passing a `Params` object to configure it. In this case the only parameter we set is the directory in which to look for test scripts: `testdata/script`.

So, let's put our script file in the `testdata/script` folder, and name it something like `count_counts_lines_from_stdin.txtar` (the `.txtar` extension is important, but the name isn't).

Let's try running the test:

```
--- FAIL: Test/count_counts_lines_from_stdin
(0.00s)
    testscript.go:429: > stdin three_lines.txt
      > exec count
      [exec: "count": executable file not found in
$PATH]
      FAIL:
testdata/script/count_counts_lines_from_stdin.txta
r:2:
      unexpected command failure
```

Hmm. It looks like the exec line in the script has failed, not because the `count` command returned a non-zero exit status, but because there *is* no `count` command in our path. That makes sense: we haven't compiled it yet!

Defining custom commands

Note that we could quite happily have executed any external command that *does* exist in our path, including any command we could run from a terminal. Sometimes that

can be useful, but it's not what we need here, because we want to run our `count` program *as though* it were an external binary, without actually having to create one.

To do that, we need one more bit of code:

```
func TestMain(m *testing.M) {
    os.Exit(testscript.RunMain(m,
map[string]func() int{
        "count": count.Main,
    }))
}
```

(Listing count/5)

The name `TestMain` is special to Go, because this function is always executed first when we run `go test`, before any tests are run. Its purpose is to set up any text fixtures or other things that need to exist for the tests to work. In this particular case, we're using it to call `testscript.RunMain`. So what does *that* do?

It takes a map argument defining any "custom commands" we want to make available to test scripts. In this case, we want to define the `count` command, by associating it with the `count.Main` function. We're saying that if some test script calls `exec count`, then the `count.Main` function should be executed as an independent binary, in a subprocess, just as if it were a "real" external command.

The delegate `Main` function

The signature of this "delegate main" function is fixed by `RunMain`: it must take no arguments and return `int`, indicating the command's exit status. By a total coincidence, that's exactly what we have:

```
func Main() int {
```

So *now* we should be able to run the test and have it pass:

```
PASS
```

Reassuring! Just to make sure it's really running `Main`, let's change that function to return 1 instead of 0:

```
func Main() int {
    // Houston, we've had a problem.
    return 1
}
```

Now the test should fail, and we should also see exactly why:

```
--- FAIL: Test/count_counts_lines_from_stdin
(0.01s)
    testscript.go:429: > stdin three_lines.txt
      > exec count
      [exit status 1]
      FAIL:
testdata/script/count_counts_lines_from_stdin.txta
r:2:
      unexpected command failure
```

The `[exit status 1]` tells us that `Main` is really being called, and the `unexpected command failure` tells us that the test failed because the `count` command didn't return a zero exit status. Nice.

The `stdout` assertion

Let's see what happens if the input file contains, for example, *four* lines instead of three. We'll edit the test script

so that the part specifying the contents of the input file now has an extra line:

```
...
-- three_lines.txt --
this input
contains
three lines
plus one more
```

The filename is a lie! More to the point, the test should fail. Let’s see:

```
...
> exec count
[stdout]
4
> stdout '^3\n$'
FAIL:
testdata/script/count_counts_lines_from_stdin.txta
r:3:
no match for `^3\n$` found in stdout
```

Now the `exec` assertion succeeds, because the `count` command does indeed return zero exit status, but it’s the `stdout` assertion that fails this time. The command’s output didn’t contain the expected text, because it printed “4” instead.

With just this very simple and lightweight mechanism, we can test our CLI tool *as* a CLI tool, without needing a lot of complicated plumbing, or even having to build a binary. That’s very handy.

Testing arguments

What about when we give `count` a command-line argument representing a filename? Could we test that behaviour using a script, too? Certainly:

```
exec count three_lines.txt
stdout '^3\n$'

-- three_lines.txt --
this input
contains
three lines
```

(Listing count/5)

Note that we didn't use the `stdin` directive this time, so if the `count` command were to try to read from standard input, there wouldn't be any. Instead, it *must* open and read the file named on its command line for it to produce the correct output.

There's a whole lot more we can do with `testscript`, and indeed you can read a whole chapter about it in The Power of Go: Tests, but we've already learned enough to radically simplify our tests for the `count` tool.

In the next chapter, we'll see how to extend the behaviour of `count` further, using a special kind of command-line argument: *flags*.

Going further

Here's a suggestion for exploring the ideas discussed in this chapter.

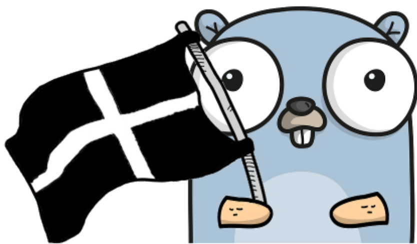
- Add support to the line counter for supplying multiple filenames on the command line. Ensure that all files are

closed after reading.

You’ll find one possible solution in listing `count/6`.

4. Flags

> *We used to sit around in the Unix room saying, “What can we throw out? Why is there this option?” It’s often because there is some deficiency in the basic design. Instead of adding an option, think about what was forcing you to add that option.*
> —Doug McIlroy, “Ancestry of Linux—How the Fun Began”



With APIs, as in other areas of life, simplicity is always the best plan. The smaller your API, the easier it is to use (and the more likely that it will *be* used). The ideal command-line tool, according to the Unix philosophy, does one thing, and does it well.

Commands

We would like our tools to be so simple and focused that they don’t need a lot of options and switches to control their behaviour. A tool that does one thing is easier to use than one that does two things, or N things. But we are often compelled, against our better instincts, to add extra modes and features to our programs that necessitate a more complex API.

The Unix way

Our long-suffering line counter program needs no additional behaviours to be useful, but let's give it some, purely for the purposes of demonstration. Suppose that, like some hard-working authors, we are anxious to know how many *words* we have written so far, rather than simply how many lines.

The most Unix-like (and therefore the most Go-like) way to do this would in fact to write *two* separate programs: one that counts words, and another that counts lines.

The two programs can share almost all their code, of course, since they'll import it from the `count` package, but from the user's point of view, they'll be two different commands.

Let's see how to do that.

Multiple `main` packages

As you know, every Go binary you build requires its own `main` package, and since you can't have two different `main` packages in the same folder, that means we'll need a separate folder for each binary we want to build. Each will contain the `main.go` code for that specific command.

We already have a `main.go` in the `cmd/count` subfolder, that runs the line counter:

```
func main() {
    os.Exit(count.Main())
}
```

(Listing count/4)

We're now envisaging two commands, one that counts lines (which we already have), and one that counts words (which we'll need to write). They can't both be called `count`, so

suppose we rename the line-counting program to `lines`? Let's rename its subfolder accordingly:

```
mv cmd/count cmd/lines
```

A `words` command

It seems reasonable, then, to call our new word-counting command `words`, so let's create a new subfolder for it:

```
mkdir cmd/words
```

Now we can create `cmd/words/main.go`. Since the two programs are so similar in behaviour, we'd like to make their `main` functions as similar as possible.

Our existing `main` function for the `lines` command delegates all its functionality to `count.Main`, for ease of testing. We'll need a similar function for `words`, and they can't *both* be called `Main`. What to do?

Let's rename the existing `count.Main` to `count.MainLines`:

```
func main() {
    os.Exit(count.MainLines())
}
```

(Listing count/7)

Following this pattern, we can create a `cmd/words/main.go` that calls `MainWords` instead:

```
func main() {
    os.Exit(count.MainWords())
}
```

(Listing count/7)

This looks all right, so what should `MainWords` do? Well, again, let's see what we have in `MainLines`:

```
func MainLines() int {
    c, err := NewCounter(
        WithInputFromArgs(os.Args[1:]),
    )
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        return 1
    }
    fmt.Println(c.Lines())
    return 0
}
```

(Listing count/7)

With some pragmatic (nay, shameless) code duplication, let's simply copy this to `MainWords`, substituting `c.Words` for `c.Lines`:

```
func MainWords() int {
    c, err := NewCounter(
        WithInputFromArgs(os.Args[1:]),
    )
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        return 1
    }
    fmt.Println(c.Words())
    return 0
}
```

(Listing count/7)

If we're really bothered about the duplication, we can always refactor this code later, but let's focus for now on producing a *working* program, rather than a beautiful one.

A test for word counting

We now need to write `Words`. Let's start with a test, and again we'll get a head start by looking at the existing test for `Lines`:

```
func TestLinesCountsLinesInInput(t *testing.T) {
    t.Parallel()
    inputBuf := bytes.NewBufferString("1\n2\n3")
    c, err := count.NewCounter(
        count.WithInput(inputBuf),
    )
    if err != nil {
        t.Fatal(err)
    }
    want := 3
    got := c.Lines()
    if want != got {
        t.Errorf("want %d, got %d", want, got)
    }
}
```

(Listing count/7)

GOAL: Write a test for `Words` in a similar way.

HINT: You can start by literally copying and pasting the test for `Lines`, and making the appropriate changes. Can you see what to do?

SOLUTION: Although we could construct a suitable test just by changing `c.Lines` to `c.Words`, let's make it a little more interesting by adding some extra words to our simulated input in `inputBuf`. We'll make it six words in total, meaning that our `want` value also becomes 6:

```
func TestWordsCountsWordsInInput(t *testing.T) {
    t.Parallel()
    inputBuf := bytes.NewBufferString("1\n2
words\n3 this time")
    c, err := count.NewCounter(
        count.WithInput(inputBuf),
    )
    if err != nil {
        t.Fatal(err)
    }
    want := 6
    got := c.Words()
    if want != got {
        t.Errorf("want %d, got %d", want, got)
    }
}
```

(Listing count/7)

GOAL: Add a null implementation of `Words` and check that the test fails with the following output:

```
want 6, got 0
```

HINT: Be careful not to write the real implementation of `Words` too quickly. Before we do that, we should make sure that this test really *tests* anything. And to do *that*, we need to see what it reports when `Words` returns the wrong answer.

The easiest wrong answer to return is zero, and that's what the suggested test output says. But you can use whatever wrong answer you like. The only thing that wouldn't work would be 6. Other than that, get creative!

SOLUTION: Well, here's the null implementation I would write:

```
func (c *counter) Words() int {
    return 0
}
```

If this doesn't fail the test, what would?

Okay, now that we've proved we have a working bug detector, let's go ahead and implement `Words` for real.

GOAL: Write the real implementation of `Words`.

HINT: How should we approach this? Well, you know the drill by now: start with what we have already.

```
func (c *counter) Lines() int {
    lines := 0
    input := bufio.NewScanner(c.input)
    for input.Scan() {
        lines++
    }
    for _, f := range c.files {
        f.(io.Closer).Close()
    }
    return lines
}
```

(Listing count/7)

What needs to change here to count words instead of lines? One idea that isn't terrible is to use something like `strings.Split` on each line to count the number of words in it, and add it to the running word total. But there's a shortcut.

By default, a `bufio.Scanner` scans lines, but we can make it scan words by calling `input.Split` to set the scanner's split function to `ScanWords`, before we start scanning.

SOLUTION: I like the `bufio.Scanner` solution, because it's kind of symmetrical with the way `Lines` works. So the only substantive difference with this method is that we need to set the split function to `ScanWords`, as hinted:

```
func (c *counter) Words() int {
    words := 0
    input := bufio.NewScanner(c.input)
    input.Split(bufio.ScanWords)
    for input.Scan() {
        words++
    }
    for _, f := range c.files {
        f.(io.Closer).Close()
    }
    return words
}
```

(Listing count/7)

With the test now passing, we're ready to try out our new `words` command:

```
echo hello world | go run ./cmd/words
```

```
2
```

Pretty neat. Let’s make sure the existing `lines` program still works as expected:

```
echo hello world | go run ./cmd/lines
```

```
1
```

Updating the test scripts

This looks promising, so we can now modify the `testscript` tests we wrote earlier to take account of our two new commands, `lines` and `words`.

First, we’ll update our `TestMain` function to register these two commands, so that they’re available in our scripts:

```
func TestMain(m *testing.M) {
    os.Exit(testscript.RunMain(m,
map[string]func() int{
        "lines": count.MainLines,
        "words": count.MainWords,
    }))
}
```

(Listing count/7)

Our existing “counts lines from stdin” script only needs the command name updating:

```
stdin three_lines.txt
exec lines
stdout '^3\n$'
```

```
-- three_lines.txt --
this input
contains
three lines
```

And we have a new command, `words`, that deserves its own test script:

```
stdin five_words.txt
exec words
stdout '^5\n$'

-- five_words.txt --
this input
contains
five words
```

And, since we added support for specifying filenames as command-line arguments, let’s make sure we test that for both commands:

```
exec lines input1.txt input2.txt
stdout '^3\n$'

exec words input1.txt input2.txt
stdout '^10\n$'

-- input1.txt --
this input
contains two lines
-- input2.txt --
and this has just one
```

It’s worth pausing for a moment to reflect on what we’ve done here. We took an existing program that counted lines,

and turned it into *two* programs, one that counts lines, and one that counts words.

Suppose we hadn't read the earlier chapters of this book, and we had just implemented that behaviour in the `main` function. When we duplicated the `main` function, we would have duplicated the *entire* code for the line counter.

That's not a disaster, but it is inconvenient. If we wanted to add some extra behaviour that applies to both commands, we would have had to add it in two places. If we'd needed to fix a bug, we would have had to fix it in two places.

For a real, useful, production program, multiply that by lots of features and lots of bugs over lots of years (and lots of programmers). It just doesn't scale.

But we didn't do that. Instead, we put all the substantive behaviour into the `count` package, and made the `main` function for the `lines` command about as small as it could possibly be.

So when we added the `words` command, we were able to re-use just about everything we needed: the `counter` struct, with its constructor and functional options, its multiple file handling, and so on.

Yet we didn't need to add any confusing command-line flags to switch between counting lines and counting words. The delegate `Main...` function for each command is quite short, straightforward, and to the point. That makes it easy to understand and easy to maintain.

So, when we want our program to have different behaviours that nevertheless share a lot of code, very much the cleanest and most user-friendly way to do that is to create

two separate commands, each with its own `Main...` function.

That technique, though, is only practical when we're using the "write packages, not programs" philosophy. And that's no coincidence. Because an importable package makes it easy for users to create their own programs using our code, it also makes it easy for *us* to create new programs with it.

Flags

The "one command per behaviour" approach makes perfect sense with something like the counter example. But practical programs often have many different behaviours, and it would be awkward to manage dozens of binaries, one for each possible combination that users might want.

Introducing flags

So it's usual for programs like this to instead let users specify the behaviours they want using command-line arguments. For example, we could have implemented the counter program as a single command that accepted a *verb* as argument, like this:

```
count lines input.txt
```

or

```
count words input.txt
```

That's not terrible; it's just a bit more typing for users, and of course we'd need to handle the situation where they specify an *unknown* verb, or don't specify one at all. But it could be done.

However, most programs also use command-line arguments for other things. In the case of the counter program, it accepts filenames, for examples. If we mix verbs in with these, how is the program to know whether a given argument represents a verb, or the name of some input file? What if it could be both, for example?

So it's common to distinguish arguments that are intended to control *behaviour* using some special format, such as prefixing them with a hyphen. You're probably already familiar with hyphenated flags from such commands as:

```
go test -v
```

Here, the leading hyphen identifies `-v` as a flag, not just the name of some Go package or source file. The specific meaning of *this* flag is to tell `go test` to be *verbose*: that is, to print more detailed output than it would otherwise.

Adding a `-lines` flag

Suppose we added a flag to control the behaviour of our counter program, then; what would that look like? Let's decide, arbitrarily, that the *default* behaviour will be to count words, but you can enable line-counting mode by specifying the `-lines` flag:

```
count input.txt
```

(counts words)

```
count -lines input.txt
```

(counts lines)

We could add *another* flag, `-words`, to count words, but there's no need. Users don't want to type any more than

they have to. There are only two possible behaviours, so we only need one flag to choose between them.

So, we have a fair idea of the behaviour we want. How can we express that in the form of a test? Let's consider the test script we used for a previous version of the `count` program:

```
stdin three_lines.txt
exec count
stdout '^3\n$'

-- three_lines.txt --
this input
contains
three lines
```

(Listing count/5)

We'll start by changing this to reflect the fact that we now want `count` on its own, with no flags, to count *words* in its input:

```
exec count input.txt
stdout '^5\n$'

-- input.txt --
this input
contains
five words
```

(Listing count/8)

And we can now add a new script showing that, with the `-lines` flag, the program instead counts lines, not words:

```
exec count -lines input.txt
stdout '^3\n$'

-- input.txt --
this input
contains
three lines
```

(Listing count/8)

Now that we only have one binary, we'll go back to our original scheme from listing count/5. We need just one `main.go`, in the `cmd/count` folder:

```
func main() {
    os.Exit(count.Main())
}
```

(Listing count/8)

And we're defining a single `count` command in `TestMain`:

```
func TestMain(m *testing.M) {
    os.Exit(testscript.RunMain(m,
map[string]func() int{
        "count": count.Main,
    }))
}
```

(Listing count/8)

Let's run the tests and see what happens. We don't expect them to pass, since we haven't added the flag-handling code yet. However, we would like to confirm that they fail for the right reasons:

```
go test

--- FAIL: Test/count_counts_lines_with_lines_flag
(0.01s)
    testscript.go:429: > exec count -lines
input.txt
      [stderr]
      open -lines: no such file or directory
      [exit status 1]
      FAIL:
testdata/script/count_counts_lines_with_lines_flag
.
      txtar:1: unexpected command failure
--- FAIL: Test/count_counts_words_by_default
(0.02s)
    testscript.go:429: > exec count input.txt
        [stdout]
        3
        > stdout '^5\n$'
        FAIL:
testdata/script/count_counts_words_by_default.txta
r:
        2: no match for `^5\n$` found in stdout
```

The word-counting test fails because our program still defaults to its old line-counting mode, so it gives the wrong answer for count `input.txt`:

```
no match for `^5\n$` found in stdout
```

The line-counting test fails for a more interesting reason. We gave the `-lines` flag on the command line, but the program currently has no way of interpreting this as anything other than the name of a file to read:

```
open -lines: no such file or directory
```

So it doesn't count anything. Fair enough. Now let's see what we need to do to get these tests passing.

Implementing the behaviour

Do we need to change anything about the `Counter` struct, its constructor, its options, or its methods? Not really: everything we need is there. The only choice the program has to make is whether to print the result of `c.Lines`, or `c.Words`, based on the flag.

Let's look at the `Main` function we had before:

```
func Main() int {
    c, err := NewCounter(
        WithInputFromArgs(os.Args[1:]),
    )
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        return 1
    }
    fmt.Println(c.Lines())
    return 0
}
```

(Listing count/4)

We'll need to modify this so that it can detect whether the `-lines` flag is supplied on the command line. If it is, we should call `c.Lines`, just as we do now, but if it isn't, we should call `c.Words` instead.

First, how can we detect the presence (or absence) of the `-lines` flag? We could read `os.Args`, of course, but it turns out we don't need to. The standard library has our back, as usual. The `flag` package will do exactly what we want:

```go
func Main() int {
    lineMode := flag.Bool("lines", false, "Count
lines, not words")
    flag.Parse()
    c, err := NewCounter(
        WithInputFromArgs(flag.Args()),
    )
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        return 1
    }
    if *lineMode {
        fmt.Println(c.Lines())
    } else {
        fmt.Println(c.Words())
    }
    return 0
}
```

Actually, we didn't need to add a lot of code to achieve the behaviour we want. Here's the first thing we added, at the top of `Main`:

```go
lineMode := flag.Bool("lines", false, "Count
lines, not words")
```

We call the function `flag.Bool` to declare a new boolean flag, whose name is `"lines"`, and whose default value is `false`. We also add a helpful note explaining to users what this flag means.

The result of `flag.Bool` is assigned to the variable `lineMode`, which we'll use to find out if "line mode" is on or off, depending on the flag.

We don't know the actual state of the flag yet, since we haven't *parsed* the program's arguments to look for flags. That won't happen until we call this function:

```
flag.Parse()
```

This means "Dear `flag` package, please parse the program's command-line arguments, and use them to set the value of any flags I've previously defined." After this call, then, we should expect `lineMode` to reflect whether or not the user supplied `-lines` as an argument.

But we don't need to know that yet, because we have to construct the `Counter` regardless. Because the user might also have supplied some filenames on the command line, we still need to call `WithInputFromArgs`. However, we don't pass it `os.Args[1:]`, as we did before, because that might contain the word `-lines`. And that shouldn't be interpreted as a filename, or we'll get the test failure we saw previously.

What we'd really like instead is a way to get all the *non-flag* arguments. In other words, having called `flag.Parse` to extract all the arguments that refer to flags, we want to ask "what's left?"

That's exactly what `flag.Args` does:

```
WithInputFromArgs(flag.Args()),
```

Finally, having constructed the counter `c`, we can use it to count words (or lines). And that depends on the value of `lineMode`:

```
if *lineMode {
    fmt.Println(c.Lines())
} else {
```

```
    fmt.Println(c.Words())
}
```

Why is `lineMode` a pointer, rather than a plain old `bool` value? Well, `flag.Parse` needs to be able to modify each of our defined flag variables when we call it. So, somewhere under the hood, it must maintain a list of pointers to those variables. Thus, all flag variables are pointers, so we use the * operator to dereference them and get the value we need.

One important thing to note is that the `flag` package stops parsing as soon as it sees a non-flag argument. So you can't put flags after arguments:

```
count input.txt -lines
```

```
open -lines: no such file or directory
```

Help and usage information

Whenever your program takes flags and arguments, or even when it doesn't, it's nice to provide a help message for users who aren't sure what to do. Typically, programs will recognise a `-h` or `--help` flag for this purpose.

Indeed, this is built into the `flag` package, and we get this behaviour for free:

```
count -h
```

```
Usage of count:
  -lines
        Count lines, not words
```

Pretty neat! Still, we can improve on this a bit. It doesn't say what the program actually *does*, and it's never safe to assume that users know this. They may just be wondering

what this weird `count` binary is on their system, who installed it, and why.

It also doesn't document that the program takes command-line arguments. Let's see what we can do. A little spelunking into the `flag` package shows us that the function actually being called here is `flag.Usage`. Here it is:

```
var Usage = func() {
    fmt.Fprintf(CommandLine.Output(), "Usage of
%s:\n", os.Args[0])
    PrintDefaults()
}
```

Why did they make this a `var` declaration, rather than simply defining `func Usage...`? For the very good reason that we can *assign* to it. Let's assign our own function that's a little more informative:

```
flag.Usage = func() {
    fmt.Printf("Usage: %s [-lines] [files...]\n",
os.Args[0])
    fmt.Println("Counts words (or lines) from
stdin (or files).")
    fmt.Println("Flags:")
    flag.PrintDefaults()
}
```

The assignment needs to come before we call `flag.Parse`, because it's this parsing that may trigger the printing of the usage message. Here's the result:

```
count -h

Usage: count [-lines] [files...]
Counts words (or lines) from stdin (or files).
```

```
Flags:
  -lines
        Count lines, not words
```

It doesn't matter how great the program is if no one can figure out how to use it. A little effort like this to make our programs helpful and friendly can go a long way.

Going further

Here's a suggestion for exploring the ideas discussed in this chapter.

- Add a new flag to the line counter program that makes it capable of counting *bytes* instead of words or lines.

  If both the byte-counting flag and the line-counting flag are supplied simultaneously, the program should report an error.

  You'll find one possible solution in listing `count/9`.

5. Files

> *Plan 9 has persistent objects—they're called "files".*
> —Ken Thompson, quoted in Plan 9 Fortunes Files

Many Go programs are about manipulating files in some way; we've already seen in a previous chapter how to open a file and read data from it. We might also be called upon to *write* to files, and also to manage files themselves as data: create them, delete them, rename them, list them, and so on.

Let's take the writing problem first, and see what's involved in building a Go package that deals with writing to files.

Writing to files

The first question to ask is whether the program itself needs to deal with files at all. As we've seen, the Unix shell can helpfully read data from files and send it to a program's standard input. When a program only needs to deal with data as a stream of bytes, the most Unix-like answer is to

read from standard input, and let the operating system take care of the file handling.

Similarly, when producing data that needs to be *written* to a file, the simplest and most flexible way is for the program to write to its own standard output stream. This can be redirected by the shell to the file where it's needed (or even the input of some other program, for example).

The art of judicious logging

Incidentally, the same applies to *logging*, which is a special case of writing data. Many programmers spend a good deal of time and energy on deciding what information to log, where to log it, how to log it, which package to use, and so on. Most of this effort is probably wasted. Well-designed applications don't need to log much information, if any. What they do need to say can be said to standard output (or standard error, as appropriate), and this gives the maximum flexibility for the program operator to decide where this data should be sent.

It's always worth asking, before you decide to log some information, "What question about the program's behaviour does this log message answer?" For example, the answer might be "It tells me that there was some error". That's fine, and the best place to write that message is usually to the program's standard error stream. After all, that's where the user is most likely to see it.

On the other hand, if you can't think of any worthwhile question that the log message would help to answer, then you don't need the message. If what you want is aggregate information about how many times the program does something, or how often, emitting *metrics* (using the Prometheus format, for example) is likely to be better than logging.

Similarly, if you want to analyse the program's performance, or understand why it misbehaved in a certain situation, your best option is probably to use *tracing* (such as OpenTelemetry data) instead of logging.

That said, let's look at some non-logging-related uses for writing to files, and see what kind of API and testing patterns emerge.

Testing a `WriteToFile` function

Let's assume that we're writing *useful* data to a file, then, and not just suffering from a nasty case of logorrhoea. This will be a fun thing to test. We can start by trying to build a simple function that creates a file and writes some arbitrary data to it.

We'd like to start with a test, of course, so what behaviour should we test? Well, we already have a rough description of the required behaviour in the form of a user story: the program creates a specified file and writes the specified data to it. Let's work from the middle out, as before. What sort of function would it make sense to call here? Perhaps:

```
writer.WriteToFile(path, data)
```

The package name `writer` sounds sensible: it's short, clear, and describes the primary concern of the package. Similarly, `WriteToFile` expresses the specific behaviour of the function we're trying to write. It may turn out, in practice, that a slightly different name makes more sense, but that's okay: we've seen already in this book that our process often results in quite a few changes along the way, and they're usually for the better.

What does the function need to take as arguments? Clearly, at least the pathname of the file to create, and the data to

write to it. What would be the appropriate types for these things? We usually operate on pathnames as strings, and arbitrary data as a `[]byte`, so we'll need to set these up at the beginning of the test.

Now that we know what `WriteToFile` takes, we can ask what, if anything, it returns. The first thing to ask about any function's results, regardless of what else it returns, is whether it should return `error`. The way to answer that is to ask another question: "*Can* there be an error?"

Designing errors out of existence

If there's no readily imaginable situation where the function could encounter an error, then the answer is no. Similarly, if it's safe for the function to ignore errors, or it's able to handle any errors that occur without troubling the caller with them, then it need not return `error`.

The best kind of function is one that doesn't return `error`, because then you don't have to check it. Wherever possible, consider designing your API such that it doesn't need to return `error` for a given operation. An example might be deleting a file: if the specified file doesn't exist, should the delete function return an error?

Arguably, no: the required behaviour is that, after calling the function, the file should not exist. Well, if the file *does* exist, the function can delete it, but if it doesn't, then the function can succeed by simply doing nothing!

> *The best way to eliminate exception handling complexity is to define your APIs so that there are no exceptions to handle: **define errors out of existence**. This may seem sacrilegious, but it is very effective in practice.*
> —John Ousterhout, "A Philosophy of Software Design"

In other words, don’t just return an error because it’s there. Instead, try to design functions that don’t *need* to return errors.

Another approach is what’s called *crash-only programming*, and as the name suggests, it involves not trying to *handle* errors at all, but instead just letting the program crash if it encounters them. This sounds rather reckless at first, but can be a sensible idea in some cases. Modern software orchestration systems such as Kubernetes can automatically restart any program that exits unexpectedly.

Wise program designers should assume the program will eventually crash anyway, whether they intended it to or not. The program should be able to cope with crashing and being restarted without losing any important state or user data.

Looking for inspiration

What action on error makes sense here? Well, crashing wouldn’t be helpful, for one thing. This is a package, so we don’t want to take that choice away from the user. We discussed `panic` in a previous chapter and pointed out that unilaterally exiting the program is unfriendly behaviour for any package. The same applies here.

We also can’t design errors out of existence in this case: if we’re unable to create the file, or unable to write all the data to it, the user needs to know that. While we might be able to mitigate errors somehow, by retrying, saving data elsewhere, logging any lost data, or whatever, that’s not a responsibility we should take on ourselves here.

It’s always a good idea to look to the standard library for inspiration; can we find anything similar? Well, the standard `Write` method for anything that implements `io.Writer` looks like this:

```
Write(p []byte) (n int, err error)
```

As you can see, it returns two things: an error, and an integer value `n` that represents the number of bytes successfully written. The semantics of `Write` require it to return a non-nil error if `n` is less than `len(p)`: that is, if it did not in fact write all the bytes supplied.

Should we do the same here? As usual, we have conflicting principles to weigh. On the one hand, the *Principle of Least Surprise* (otherwise known as the “don’t make me think” rule) says that things should work the way people expect them to work. If a convention exists, for example, we should follow it.

On the other hand, as the saying goes, everything should be as simple as possible, but no simpler. That is, all else being equal, simplicity should usually win.

The job of this function is to write some data to a file, and the only thing it needs to report is whether or not it succeeded. So let’s have `WriteToFile` return just `error` to indicate that.

What are we really testing?

So are we done? If we’ve called `WriteToFile` and checked that there’s no error, is there anything more we need to test? In other words, would something like this be good enough?

```
func TestWriteToFile_ReturnsNoError(t *testing.T)
{
    t.Parallel()
    err := writer.WriteToFile("data", []byte{1, 2,
3})
    if err != nil {
```

```
        t.Fatal(err)
    }
}
```

Of course you're too smart to answer "Yes", because you know I wouldn't be asking the question in that case. So, if this *isn't* good enough, why not?

Actually, you'll come across a lot of Go tests that look more or less just like this example. It's a symptom of what we may call "test-last development": in other words, it was likely written afterwards, as a box-ticking exercise.

This test isn't wrong, as such, but the real problem is that it clearly assumes *the function is already correct*. And that's what a test is supposed to establish in the first place, so something must be missing.

Indeed, it's always worth asking of any test, "What are we really testing here?" Let's ask it now.

What's the most important behaviour of `WriteToFile`, actually? That it returns a `nil` error? No, because in that case we could pass the test by writing simply:

```
func WriteToFile(path string, data []byte) error {
    return nil
}
```

And that can't be right!

The real implementation of `WriteToFile`, it turns out, needs to do something a bit more challenging than just returning `nil`. It needs to actually write some data to the file.

Let's try to flesh out this description of the behaviour to be very specific, until it starts to sound like a test.

When we call `WriteToFile` and it returns a `nil` error, then the specified file should exist and it should contain the specified data.

If we were to express this as a brief sentence (as produced by `gotestdox`, for example), we might say something like:

```
✔ WriteToFile writes given data to file
```

That gives us the name of our test, which is always a good start:

```
func TestWriteToFile_WritesGivenDataToFile(t
*testing.T) {
```

It sounds a bit redundant: `WriteToFile` writes to file. Because of course it does! Actually, this is the sign of a really well-named function. Its name perfectly expresses what the function is supposed to do. Naturally, the name of the *test* reflects that too.

So we'll certainly be calling `WriteToFile` and checking the error, but we won't stop there. We'll also test that we can successfully open the resulting output file, *and* that it contains exactly the bytes we want.

To do that, we can use `os.ReadFile` to open and read the file's contents as `got`. Now, how should we compare them with the `want` data?

The `go-cmp` module

Since both `want` and `got` are of type `[]byte`, we could use `bytes.Equal` to compare them, but let's use the more sophisticated [`go-cmp`] module, which has a handy "diff" feature. In the event that the slices are not the same, we'll be able to see exactly where and how they differ.

We'll need to add an import for the `go-cmp/cmp` package, since it's not part of the standard library. So here's the complete test package:

```
package writer_test

import (
    "os"
    "testing"

    "github.com/bitfield/writer"
    "github.com/google/go-cmp/cmp"
)

func TestWriteToFile_WritesGivenDataToFile(t
*testing.T) {
    t.Parallel()
    path := "testdata/write_test.txt"
    want := []byte{1, 2, 3}
    err := writer.WriteToFile(path, want)
    if err != nil {
        t.Fatal(err)
    }
    got, err := os.ReadFile(path)
    if err != nil {
        t.Fatal(err)
    }
    if !cmp.Equal(want, got) {
        t.Fatal(cmp.Diff(want, got))
    }
}
```

And since Go doesn't automatically download a new dependency when it's added, we'll need to run `go mod tidy`

to do this:

```
go: finding module for package
github.com/google/go-cmp/cmp
go: found github.com/google/go-cmp/cmp in
github.com/google/go-cmp v0.5.9
```

If we run the test now, we should expect to see a failure on trying to open the output file, because we know we won't have created it. And that's exactly what happens:

```
--- FAIL: TestWriteToFile_WritesGivenDataToFile
(0.00s)
    writer_test.go:21: open
testdata/write_test.txt: no such
    file or directory
```

GOAL: Make this test pass.

HINT: The Go standard library can do an amazing amount of heavy lifting for us, especially when dealing with things like files. Have a look into the os package and see if you can find something that would make this job easy.

Implementing a `WriteToFile` function

SOLUTION: Here's the simplest implementation of `WriteToFile` I can think of:

```go
func WriteToFile(path string, data []byte) error {
    return os.WriteFile(path, data, 0o600)
}
```

Again, we could have opened the file for writing, deferred closing it, and written the byte data to it, but `os.WriteFile` does all this with one function call. It also creates the file if it

doesn’t exist, and truncates it if it does, which is the behaviour we want.

File permissions

By the way, that otherwise mysterious `0o600` argument to `os.WriteFile` specifies the permission bits that the created file should have. You probably know that in Unix (and Unix-like systems) files have some metadata bits governing who’s allowed to do what to a file: read it, write it, execute it. It’s convenient to express these with a literal in *octal* (base 8) notation, which starts with `0o`, indicating “octal”.

The next three octal digits indicate permissions on the file for its owner, group, and everyone, respectively. The value `0o600`, in particular, means “read and write (6) for owner, no access (0) for group, no access (0) for everyone.”

We have to specify some permission value to `os.WriteFile`, syntactically, but we don’t really care that much what it is. While read and write permission for *us* makes sense, we don’t need to grant access to any other users, so this value seems a sensible choice.

It’s possible that users of our `writer` package might want to use it to create files that other users can read. In that case, we could always provide a `WriteToPublicFile` function, or something similar. But it’s sensible to default to creating *private* files, because even if we warn users not to use this package for sensitive information, it’s a good bet that sooner or later someone will.

When the directory doesn’t exist

Let’s see if the test is now passing:

```
--- FAIL: TestWriteToFile_WritesGivenDataToFile
(0.00s)
    writer_test.go:20: open
testdata/write_test.txt: no such
    file or directory
```

That’s weird. It’s now failing because `WriteToFile` is returning a “no such file” error, which means that `os.WriteFile` is also returning that error. But isn’t `WriteFile` supposed to create the file if it doesn’t exist?

Yes, it is, but if the *directory* it’s supposed to be in doesn’t exist, it doesn’t go so far as to create that too. We don’t really want to be in the business of creating extensive file trees on the user’s computer. If they’ve asked for a file to be written to a directory that doesn’t exist, that’s probably a mistake we should let them know about.

Let’s create `testdata` manually so that the test can run:

```
mkdir testdata
```

Now all is well:

```
PASS
ok      writer  0.185s
```

Is there anything that’s not quite satisfactory about our existing test? Once you have a test passing, it’s always a good idea to review it and see if it can be improved in any way.

Well, one characteristic of good unit tests is that they should be *idempotent* (Latin for “same effect”): they should always do the same thing, and they should always leave the world in the same state. In other words, tests shouldn’t have side-effects.

This one has a side-effect, though, doesn't it? It creates the file `testdata/write_test.txt`. This is a problem, not just because we're polluting the code repository with uncommitted changes, but because it could mask bugs in `WriteToFile`.

A disastrous bug

I'll explain what I mean. Since the output file has now been created and it contains the expected data, then as long as no one deletes it, the test will always pass.

In other words, it's no longer a test at all, since `WriteToFile` could still pass even if it's subsequently changed to do nothing at all:

```
func WriteToFile(path string, data []byte) error {
    return nil
}
```

Will running the tests now catch this *regression* bug? Nope:

```
PASS
ok      writer  0.148s
```

Oh dear. This is pretty bad. We no longer have an effective test, but the worst of it is that it still *looks* like we do. Even the code coverage tool will show that `WriteToFile` is covered by tests (which is a useful reminder that coverage isn't everything, or indeed anything much).

Code coverage only proves that the code gets *called*, after all, not that it works. It's a bit like the useless test we saw earlier that only checks the function's `error` result, forgetting to also check that it actually did something! So this won't do.

GOAL: Fix this regrettable situation, the right way. (Be careful.)

HINT: Well, the *wrong* way to fix this would be to have the test delete the output file after it's created! Why's that wrong, though?

Don't worry, most programmers would make the same mistake, and it's a very natural one. Our first instinct on finding a bug is to fix the bug, but that would be to miss an important step.

We're aiming to build a self-testing program that contains its own bug detectors in the form of tests. So if we find a bug, that means we *also* have a bug in our bug detector. The problem wasn't caught by any test, so the first thing we need to do is fix *that* problem.

Depending on the bug, we may need to add a new test specifically for it, or we may instead be able to extend some existing test. For example, we can do that in this case, by extending the test for `WriteToFile` to check whether it cleans up after itself properly.

Right now, the test just calls `WriteToFile` and checks that the output file exists afterwards. As we now realise, that's not good enough. If that file *ever* gets created, the test will always pass, even when `WriteToFile` does nothing.

So the first thing we should do in the test is check that the file *doesn't* exist before it's supposed to. In other words, it's not good enough just to check our *postconditions* after calling `WriteToFile`. In order to know that `WriteToFile` actually did anything, we also need to check the *preconditions* beforehand.

SOLUTION: So, the first thing we should do is check whether the file already exists, and fail if that's the case.

This isn't testing `WriteToFile`, as such; it's just a sanity check for the test itself. But that's worth having: insane tests don't help anyone.

We've said that the test should be idempotent, leaving the world in exactly the same state as it was. If that's not the case, then we've failed to clean something up, so let's check:

```
func TestWriteToFile_WritesGivenDataToFile(t
*testing.T) {
    t.Parallel()
    path := "testdata/write_test.txt"
    _, err := os.Stat(path)
    if err == nil {
        t.Fatalf("test artifact not cleaned up:
%q", path)
    }
    defer os.Remove(path)
    want := []byte{1, 2, 3}
    err := writer.WriteToFile(path, want)
    if err != nil {
        t.Fatal(err)
    }
    got, err := os.ReadFile(path)
    if err != nil {
        t.Fatal(err)
    }
    if !cmp.Equal(want, got) {
        t.Fatal(cmp.Diff(want, got))
    }
}
```

To check whether the file exists beforehand, we call `os.Stat`, which requests metadata about a file. In this case we don't care about the metadata, so we ignore that result with the blank identifier _. We only care that there should be an error trying to get it, because the file shouldn't exist at this point in the test.

Having ensured the file doesn't exist *before* the test, we then need to ensure it doesn't exist *after* the test, by calling `defer os.Remove(path)`.

Why do we need to *defer* this, instead of just calling it at the end of the test? Because the test could exit at several different points before it reaches that call. As you know, `defer` executes the supplied call when the surrounding function exits, however and whenever that may be.

Using `t.TempDir`

This is all right, but a decent chunk of the test code is taken up by paperwork around checking and removing this file. Can't we do better? What we'd like is to write to a file that would *automatically* remove itself after the test.

The `t.TempDir` method can do exactly this for us. It will create, and return the path to, a temporary folder, which will be deleted once the test is complete. So now we don't need to check for un-cleaned-up artifacts, because there won't be any.

Here's the resulting test:

```
func TestWriteToFile_WritesGivenDataToFile(t
*testing.T) {
    t.Parallel()
    path := t.TempDir() + "/write_test.txt"
    want := []byte{1, 2, 3}
```

```
    err := writer.WriteToFile(path, want)
    if err != nil {
        t.Fatal(err)
    }
    got, err := os.ReadFile(path)
    if err != nil {
        t.Fatal(err)
    }
    if !cmp.Equal(want, got) {
        t.Fatal(cmp.Diff(want, got))
    }
}
```

(Listing writer/1)

Instead of *checking* our preconditions, as in the previous version, this goes one better, by taking *control* of the preconditions. Calling `t.TempDir` creates a new directory that didn't exist before, so it *can't* have any test junk left in it. And, as a bonus, it's also self-cleaning.

We can't always use a temporary directory like this, though. Sometimes the job of a function is to create files in some specific place where other files might also exist, for example. So when we *do* have to manually check and clean up test artifacts, at least now we know how.

Finishing the job

Is this good enough? Are we done *now*? Well, not quite.

One thing we haven't done is test that `WriteToFile` returns an error when it should. I mean, we know it *does*, because we already saw that happen by accident, but that's not the same as having a test for it.

And now we know an easy way to trigger such an error: passing `WriteToFile` a path containing a directory that doesn't exist. That sounds like a straightforward test to write:

```
func
TestWriteToFile_ReturnsErrorForUnwritableFile(t
*testing.T) {
    t.Parallel()
    path := "bogusdir/write_test.txt"
    err := writer.WriteToFile(path, []byte{})
    if err == nil {
        t.Fatal("want error when file not
writable")
    }
}
```

(Listing writer/1)

What else haven't we covered? Well, one interesting question to ask is what should happen if the file already exists when we try to write to it.

One option would be to return an error, but is this really an error situation? After all, `WriteToFile`'s job is to make sure that the file exists, and contains the given data. It doesn't really matter whether it existed beforehand or not, in this case.

Let's design this error out of existence, then. We'll say simply that if the output file already exists, `WriteToFile` should *clobber* it: that is, overwrite it with the new data.

GOAL: Write a test to demonstrate that `WriteToFile` clobbers existing files.

HINT: When writing a test, we start with the behaviour we want: what's *supposed* to happen if the code works correctly. Then, to turn that into a Go test, we ask "In what circumstances should the test *fail*, then?"

We're saying that if a file with the same name already exists, our `WriteToFile` function should replace it entirely. So how could that go wrong?

Well, one bug could be that we don't do anything at all, leaving the existing file unchanged. Another is that we simply *append* the data to what's there already, meaning that the file ends up containing both sets of data.

Of course, we can always imagine other bugs: we *prepend* the new data, or interleave it with the old data, or we delete the file altogether, or leave it empty, or wipe the user's computer and install Windows for Workgroups 3.11, or generate an infinite number of digits of π, or… well, you get the point.

Let's try to write a test that covers as many of these different classes of bugs as possible. One thing that they all have in common is that, if present, we won't end up with a file containing "1, 2, 3" (or whatever our input data was).

And we already have code to test that, so all we really need to do is write some *other* data to the file beforehand.

It's clobbering time

SOLUTION: Here's my version:

```
func TestWriteToFile_ClobbersExistingFile(t
*testing.T) {
    t.Parallel()
    path := t.TempDir() + "/clobber_test.txt"
```

```
    err := os.WriteFile(path, []byte{4, 5, 6},
0o600)
    if err != nil {
        t.Fatal(err)
    }
    want := []byte{1, 2, 3}
    err = writer.WriteToFile(path, want)
    if err != nil {
        t.Fatal(err)
    }
    got, err := os.ReadFile(path)
    if err != nil {
        t.Fatal(err)
    }
    if !cmp.Equal(want, got) {
        t.Fatal(cmp.Diff(want, got))
    }
}
```

(Listing writer/1)

In this test, we actually *want* the file to exist beforehand, and to contain some data, so that we can check whether `WriteToFile` handles that situation correctly. So we write some arbitrary bytes to it.

One way for `WriteToFile` to misbehave might be to refuse to write to an existing file, returning an error, and this test would catch that. Another might be to *append* to the file instead of truncating it. This test would also catch that, because the final file would contain `4, 5, 6, 1, 2, 3` instead of just `1, 2, 3`.

It may well be that in years to come someone looks at `WriteToFile` and thinks "I could implement this a much

better way, by doing so and so," but doesn't realise that the contract requires this clobbering behaviour, for example. The test will catch that error right away. It also serves to *document* the required behaviour; a wise coder will read all the tests covering the unit of interest before attempting to refactor it, for exactly this reason.

Ensuring permissions

We said earlier that, since we're obliged to tell `os.WriteFile` what permissions we want the created file to have, mode `0o600` is a reasonable choice. It makes the file readable and writable by the user running the program, but completely inaccessible to everyone else, even for reading.

Actually, this reminds us that having the file end up with the right permissions is part of the required behaviour of `WriteToFile`, and we haven't tested it. Let's fix that now.

How can we check the permissions on an existing file? We've used `os.Stat` already in this chapter to get a file's metadata, and at the time we weren't interested in the metadata itself, only whether or not we could get it.

This time we *are* interested in the metadata: specifically, the permissions. We can add the following code to the original `WriteToFile` test to check them:

```
stat, err := os.Stat(path)
if err != nil {
    t.Fatal(err)
}
perm := stat.Mode().Perm()
if perm != 0o600 {
```

```
    t.Errorf("want file mode 0o600, got 0%o",
perm)
}
```

If calling `os.Stat` *fails*, meaning the file doesn't exist, then something's gone very wrong with the test and we should bail out. But assuming we get `stat` successfully, we can then ask it for the file's permissions, which should be `0o600`.

It doesn't really matter whether we do this before or after checking the file's contents: we need both the permissions and the contents to be correct for the test to pass.

It *does* pass, which is reassuring. But is this a waste of time? Aren't we just testing that `os.WriteFile` behaves correctly, which we shouldn't do? We asked the same question about testing the clobbering behaviour, and it's the same answer: the tests should test the required behaviour, *regardless of how `WriteToFile` is currently implemented*.

What's the worst that could happen?

Another way to think about what's worth testing is to ask "what could be wrong?" In other words, is it possible that we could implement `WriteToFile` in such a way that the file ends up with incorrect permissions? It certainly is! We could simply pass the wrong permission value to `os.WriteFile`.

If it could be wrong, it needs testing, and this is one way to test it. But there's another, more subtle way that `WriteToFile` could behave incorrectly. Suppose, as in the clobbering test, that the specified file already exists beforehand. But suppose also that it has *different permissions*: say, `0o644` (read and write for user, read for everyone else).

What happens to those permissions when we call `WriteToFile`? Actually, it's not clear from looking at our code. We know that `os.WriteFile` clobbers the *contents* of an existing file, because we already tested that. But does it change the *permissions* of an existing file, if they're not what we specify?

If not, this would be a serious potential security hole, wouldn't it? It would make us vulnerable to a *pre-population attack*. If a malicious user were able to create the file beforehand with open permissions (for example `0o644`), and our program were to write our private data to the file leaving the permissions unchanged, the attacker could then read it and learn our secrets.

This kind of thing happens all the time, and it's not the way you want your company to get on the six o'clock news. Let's write a test that checks for this vulnerability, then. What could we do?

GOAL: Write a test that checks `WriteToFile` is not vulnerable to a pre-population attack that leaves the file with insecure permissions.

HINT: To test something like this, we can do exactly what the attacker would do: create the file in advance with open permissions. Then we'll call `WriteToFile` and see whether the permissions are still open afterwards. If so, that's a fail.

A security leak

SOLUTION: Here's what that could look like:

```
func TestWriteToFile_ChangesPermsOnExistingFile(t
*testing.T) {
    t.Parallel()
```

```
    path := t.TempDir() + "/perms_test.txt"
    // Pre-create empty file with open perms
    err := os.WriteFile(path, []byte{}, 0o644)
    if err != nil {
        t.Fatal(err)
    }
    err = writer.WriteToFile(path, []byte{1, 2,
3})
    if err != nil {
        t.Fatal(err)
    }
    stat, err := os.Stat(path)
    if err != nil {
        t.Fatal(err)
    }
    perm := stat.Mode().Perm()
    if perm != 0o600 {
        t.Errorf("want file mode 0o600, got 0o%o",
perm)
    }
}
```

(Listing writer/1)

Let's run it and see what happens.

```
want file mode 0o600, got 0o644
```

Oh my goodness! We just gave away the secret formula for Coke, or whatever. Can it really be that `os.WriteFile` leaves existing permissions unchanged? What does the documentation say?

> *WriteFile writes data to the named file, creating it if necessary. If the file does not exist, WriteFile creates it*

}

> *with permissions perm (before umask); otherwise WriteFile truncates it before writing, without changing permissions.*
> (os.WriteFile)

Without changing permissions. Well, there it is. A good Go programmer should be thoroughly familiar with the standard library, but she might be excused for overlooking or forgetting about this somewhat obscure edge case. That's another great reason for writing tests, of course.

GOAL: Get this test passing.

HINT: It's clear that we can't rely on `os.WriteFile` to set the desired permissions if the file already exists, so we may as well set the permissions explicitly ourselves, *after* calling `WriteFile`. That way, it doesn't matter whether the file already existed or not.

We can do this by calling `os.Chmod` ("change mode"). That could fail, of course, but happily it returns `error`, and so does `WriteToFile`, so there's a concise way to handle this error. Can you see what to do?

SOLUTION: Here's one I prepared earlier:

```
func WriteToFile(path string, data []byte) error {
    err := os.WriteFile(path, data, 0o600)
    if err != nil {
        return err
    }
    return os.Chmod(path, 0o600)
}
```

([Listing writer/1](#))

Whatever the file permissions end up being after the call to `WriteFile`, we ensure that they're set correctly before the function returns.

It's still possible that the attacker might be able to read at least some data between the time `WriteFile` starts executing and the time `Chmod` closes the permissions, though. So for secret data that *really* matters, we'd need to implement writing to files at a lower level, and that's beyond the scope of this book.

This is a useful little reminder, though, that behaviour intended to be helpful (leaving existing permissions unchanged) can lead to dangerous vulnerabilities when you're not aware of it, or don't test for it.

It's worth knowing that `os.MkdirAll` suffers from a similar desire to be helpful: its job is to create the directory that you specify, along with all the parent directories that don't already exist. For example, if you specified the path `/a/b/c`, and `/a` existed but `b` and `c` didn't, it would create `b` and `c` for you.

Since directories have permissions too, `MkdirAll` also takes a permission value to apply to any directories it creates. But, just like `WriteFile`, if any of the directories already exist, it leaves their current permissions intact. This is another potential security hole that you can close by explicitly testing for the permissions of all files or directories that your program creates.

When you see `os.WriteFile` or `os.MkdirAll` in other people's programs, it's worth asking "Does this explicitly check the permissions afterwards, or is it vulnerable to a pre-population attack?"

Going further

Here are some mini-projects you might like to tackle to explore these ideas further.

- Add a CLI to the `writer` package that lets users create a named file with a configurable size, whose bytes are all zeroes.

  For example, a command like the following:

  ```
  writefile -size 1000 zeroes.dat
  ```

  would create the file `zeroes.dat` containing exactly one thousand zeros.

  You might like to compare your solution with mine in [`writer/2`](writer/2).

6. Filesystems

> *Computer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty.*
> —Donald Knuth, “Computer Programming as an Art”

In previous chapters we’ve read data from files, and written data to them. We haven’t worried too much about exactly how those files are stored, treating them instead as simple abstractions with byte-oriented interfaces: `io.Reader` and `io.Writer`.

That’s fine, but Go programmers are often called upon to write tools and utilities that manipulate files directly on disk (or the equivalent of disk, such as cloud storage).

Files and filesystems

Because the exact semantics of working with files depends on your operating system, the Go standard library provides many useful facilities in the `os` package. For example, you

can work directly with files via the `os.File` type, which represents a disk file as seen by the operating system.

This is great when all you need to do is operate on some individual file, but we're often concerned with *collections* of files. Let's take a moment to talk about how these are stored and addressed, at least in Unix-like operating systems.

What even is a file?

There are lots of different data storage technologies, including spinning magnetic disks, solid-state disks, RAM disks, and so on. As programmers, we almost never need to worry about any of these implementation details. One of the reasons Unix has been so successful is that it gives us some very simple abstractions to deal with files:

- We can *address* a file using a string called a *pathname*. This uniquely specifies some file and how to find it, relative to some *filesystem*.

- Having found the file we want, we can access its data using very simple *read*, *write*, and *seek* operations.

You're already familiar with reading and writing files, and *seeking* is the process of moving the current read/write position to some specific offset in the file. For example, if you have a 100GiB data file and you only want the last byte of it, you don't have to read all the preceding bytes. You can seek directly to one byte before the end, and then read it. In practice, we're not usually working at such a low level of abstraction in Go programs.

Organising files

On the other hand, we're often very concerned with how to address files: finding them, creating them, deleting them, renaming them, and so on. So how are files organised? The simplest imaginable way would be a *flat file system*: each file has a unique name, and that's it.

Flat file systems aren't very convenient, because of the uniqueness constraint, and there's often a fairly small limit on the length of filenames, which in turn limits the number of files we can have on a given disk or filesystem.

A much more flexible scheme is the *hierarchy*: grouping files into collections called *directories* or *folders*. Now filenames only have to be unique within their containing folder, and since folders can contain folders, this structure is recursive. Here's an example:

```
notes.txt
letters/
    bank.txt
    archive/
        bank.txt
```

There are several files here, some with the same name, but that's allowed because they're not in the same folder: each file still has a unique *path*. The files and folders form a *tree* structure. At the top level (the *root* of the tree), there's a file `notes.txt`, and a folder `letters`.

The `letters` folder in turn contains the file `bank.txt`, and another folder, `archive`. We say that `archive` is a *subfolder* of `letters`. This contains another file `bank.txt`, and so on.

We can describe this whole structure as a tree, but also refer to individual *subtrees* within it, such as the subtree rooted at `letters`. The files, folders, and relationships between them, are collectively called a *filesystem*. Usually,

a filesystem corresponds to some physical disk or storage system.

It’s pretty easy to implement folders, by the way: we can give certain files a special marker or piece of metadata that tells the operating system it’s a folder, instead of a regular file. This special file can contain a list of the files (and folders) contained in the folder, each with the necessary information to tell the OS how to find the data associated with that file.

As a user of such a filesystem, how do we address individual files and folders? In Unix-like systems, pathnames are usually written by specifying each folder’s name, in descending order from the root, separated by a slash (`/`). For example, these are the pathnames of the text files shown above:

```
/notes.txt
/letters/bank.txt
/letters/archive/bank.txt
```

A simple file finder

Suppose we have been tasked with writing a tool that will count the number of Go source files contained in some tree (for example, a project repository). How might we go about that using the operations available in the `os` package?

The argument to the program will be the pathname of some folder. For example:

```
/Users/john/code/games/lander
```

The first thing the program would need to do would be to get the list of all files in this folder (remembering that

subfolders are also a kind of file). To do this, we can use `os.ReadDir`:

```
files, err :=
os.ReadDir("/Users/john/code/games/lander")
```

Since this path might not exist or not be readable by us, we need to handle a potential error from `ReadDir`. Assuming there is none, we now have a `files` variable containing a slice of `os.DirEntry` values. Each of these contains metadata about a specific file.

We need to inspect the name of each file in turn, so we could `range` over this slice and use the handy `path.Ext` function to check if a file's extension is `.go`, and increment some counter value if the filename matches. Something like this:

```
func countGoFiles(folder string) (count int) {
    files, err := os.ReadDir(folder)
    if err != nil {
        return 0
    }
    for _, f := range files {
        if path.Ext(f.Name()) == ".go" {
            count++
        }
    }
    return count
}
```

Great! Are we done?

Handling folders recursively

Well, not quite: we only counted the Go files in the target folder. What if it contains subfolders with Go files in, or subfolders within subfolders? We'll need an extra step to detect if a given file is in fact a folder. We can call `f.IsDir` to ask if this is the case.

And what should we do if we find a folder? We could call `os.ReadDir` on it to get the list of *its* contents, but there's no limit to how deep this could go. Instead, we would do better to put our code in some function that we could call recursively, to match the recursive nature of the hierarchy.

If this function finds a folder, it can call itself to get the count of Go files within the tree rooted at that folder, and so on. Eventually, having recursed into all the folders in the tree, the final result will be the total number of Go files.

Suppose we had a tree like this, for example:

```
testdata/
    tree/
        file.go
        subfolder/
            subfolder.go
        subfolder2/
            another.go
            file.go
```

There are four files whose names end in `.go`, so when we run our file finder against this tree, we would expect the output "4".

Something like this might work:

```
func main() {
    fmt.Println(countGoFiles("testdata/tree", 0))
}
```

```go
func countGoFiles(folder string, count int) int {
    files, err := os.ReadDir(folder)
    if err != nil {
        // skip
        return count
    }
    for _, f := range files {
        if f.IsDir() {
            count =
countGoFiles(folder+"/"+f.Name(), count)
        }
        if path.Ext(f.Name()) == ".go" {
            count++
        }
    }
    return count
}
```

(Listing findgo/1)

This is a little tricky to write, and even trickier to read. It's also not very efficient: each of those recursive function calls eats up a little bit of stack memory, so the deeper the hierarchy, the bigger the memory footprint of our program.

In practice it would take a fair bit more code to make this program really robust and flexible, never mind portable. And we'd hate to have to write all of this every time we wanted to do some operation on a file tree. Can't we do better?

Filesystems and `io/fs`

Dealing with file trees of arbitrary depth is so common that the Go standard library provides an abstraction to make this

easier: the *filesystem*. The `io/fs` package defines an interface `fs.FS` that represents a tree of files.

In principle, any set of objects addressable by hierarchical *pathnames* can be represented by an `fs.FS`. A tree of disk files is the obvious example, but if we design our program to operate on an `fs.FS` value, it can also process ZIP and `tar` archives, Go modules, arbitrary JSON, YAML, or CUE data, or even Web resources addressed by URLs.

The `fs.FS` abstraction provides some really helpful facilities for working with these data structures, especially when we need to recursively *walk* filesystem trees, selectively performing operations on their nodes.

Let's refer to an `fs.FS` value as a "filesystem" from now on, keeping in mind that the term also has the more general meaning we discussed earlier in this chapter. How can we create a filesystem representing the tree of files rooted at a specific disk folder, then?

Opening a folder as an `fs.FS` is straightforward. We can do this by calling `os.DirFS`:

```
fsys := os.DirFS("testdata/tree")
```

Notice that there's no error result to handle here. That's because we haven't actually done any disk operations yet; we've just created the abstraction representing the file tree rooted at `testdata/tree`. If there doesn't happen to be such a path, or we're not allowed to read it, well, too bad: we'll find that out when we try to do something that involves reading it.

An `fs.FS` by itself doesn't do much. You might be surprised by how small its method set is:

```go
type FS interface {
    Open(name string) (File, error)
}
```

(io/fs)

The only thing we can do by calling methods on the filesystem, then, is to `Open` some path within it. For example:

```go
f, err := fsys.Open("file.go")
```

If this path doesn't correspond to any existing file, then there will be some error (specifically, an `fs.PathError`). But if there isn't, then we successfully obtained some `f` of type `fs.File`. Great! So what's that?

```go
type File interface {
    Stat() (FileInfo, error)
    Read([]byte) (int, error)
    Close() error
}
```

(io/fs)

Again, a very simple interface. All we can do with an `fs.File` is *stat* it (ask for its metadata, such as its name or permissions), or use it as an `io.ReadCloser`, just like a regular `os.File` pointer.

One interesting thing we can't do is *write* to it. Filesystems in Go are read-only, which is fine. A Go proverb says:

> *The bigger the interface, the weaker the abstraction.*

In other words, if we're interested in operations on files *as part of a tree structure*, then `fs.FS` is the ideal abstraction

for us: it’s not cluttered up with stuff related to writing, modifying, moving, deleting, or changing permissions on files.

Once we know a file’s pathname, we can always use the existing `os` machinery to modify or write to it. So when you’re dealing with trees of files, use `fs.FS` rather than trying to write the recursion code yourself.

Matching files by name

So what filesystem-level operations might we want to do? With our file-counting program in mind, one useful tool is `fs.Glob`, which takes a filesystem and a pathname containing wildcards (for example, `"*.go"`) and returns the names of all the matching files:

```
matches, err := fs.Glob(fsys, "*.go")
```

An error here would indicate that the pattern is invalid in some way, while `matches` is a slice of strings containing the pathnames of the matching files. The *glob* patterns are very similar to those understood by the Unix shell; for example, in the command `ls *.go`.

Great! Does that mean we can write our file-counting program as simply as this?

```
func main() {
    fsys := os.DirFS("testdata/tree")
    matches, err := fs.Glob(fsys, "*.go")
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    fmt.Println(len(matches))
}
```

Not quite. `fs.Glob` interprets a glob like `*.go` as referring to all the `.go` files in a specific folder (for example, the root folder of the filesystem). So if we used this glob to count Go files, we'd miss any files that aren't in the root folder:

```
go run main.go

1
```

Couldn't we use another `*` to represent any folder, so that `*/*.go` would find all Go files in all subfolders? No, because while that glob would match Go files in `subfolder` or `subfolder2`, for example, it *wouldn't* match files in the root folder, or in any subfolders of subfolders. The `*` wildcard can only match non-separator characters in a pathname. We can't write a recursive glob, in other words.

Some languages and tools use a *double star* wildcard for this: for example, `**/*.go` would match all Go files anywhere in the filesystem. Alas, `fs.Glob` and the other Go standard library functions that use wildcards don't support this syntax. We'll have to keep thinking.

Walking the tree

Since a filesystem is recursive in nature, the actual recursion operation is always the same. In principle, the standard library could do this for us, and all we'd need to supply is the specific code to execute for each file or folder we find.

The `fs.WalkDir` function does exactly this. It takes a filesystem and some starting path within it, and recursively *walks* the tree, visiting every file and folder (in lexical order; that is, alphabetically).

For each one it finds, it calls some function that you provide, passing it the pathname. For example:

```
fs.WalkDir(fsys, ".", myFunc)
```

With our example tree, `myFunc` would be called once for each of the following pathnames:

```
.
file.go
subfolder
subfolder/subfolder.go
subfolder2
subfolder2/another.go
subfolder2/file.go
```

The first path, `.`, is the root folder of the filesystem itself.

So what could `myFunc` do with these paths? Well, it could look at each path and increment a counter if it ends in `.go`. This sounds promising. Let's see if we can write `myFunc`.

The function we pass to `fs.WalkDir` must have a signature that matches the type `fs.WalkDirFunc`:

```
func(path string, d DirEntry, err error) error
```

([io/fs](io/fs))

When this function gets called, the `path` parameter will be the pathname of the current file, while the `DirEntry` contains the file's metadata. We'll see what `err` is for in a moment.

First, let's write a suitable `WalkDirFunc` for our file counter. It just needs to check the path and increment its counter if it decides it's found a Go file:

```
func matchGo(p string, d fs.DirEntry, err error)
error {
    if filepath.Ext(p) == ".go" {
        count++
    }
    return nil
}
```

We can now pass `matchGo` as the function argument to `fs.WalkDir`:

```
fs.WalkDir(fsys, ".", matchGo)
```

In practice, and to avoid making `count` a global variable, we wouldn't write some named function `matchGo` and then refer to it. We would pass it as an anonymous *function literal* instead, and this is very idiomatic in Go:

```
var count int
fsys := os.DirFS("testdata/tree")
fs.WalkDir(fsys, ".", func(p string, d
fs.DirEntry, err
    error) error {
    if filepath.Ext(p) == ".go" {
        count++
    }
    return nil
})
fmt.Println(count)
```

(Listing findgo/2)

So what's that `error` parameter about? It lets you control what happens if there's an error reading some folder. For example, suppose we don't have read permission to get the contents of `subfolder`, so when `WalkDir` tries to open it

there'll be an `fs.ErrPermission` error. In this case, `WalkDir` would call our function with `"./subfolder"` and the error, and the function could decide what to do about it.

The simplest thing it could do would be to just return the error immediately, in which case `WalkDir` would stop walking and return the same error itself. That's usually not what we want, though. We wouldn't want a single permission error to crash a 17-hour file scanning job a few minutes before the end, for example, forcing us to start all over again.

Instead, we can return the special error value `fs.SkipDir`, meaning "skip this folder, but carry on walking". We don't handle errors in our example program, so it will always just skip any folders it can't read, which isn't completely unreasonable. You might need more sophisticated error handling in production programs, though.

A file-finding tree-walker

It looks like using a filesystem and `fs.WalkDir` will work for our file-finding program, so let's see how to turn it into a full-fledged, well-tested Go package.

To do that, let's expand our ambitions a bit. Counting files can be useful, but it seems a shame to go to all the trouble of *finding* the files, only to throw away everything but the *number* of files we found.

Suppose users wanted to get a *list* of those files; well, it's bad luck for them, if all they have is the value of `count`. They'd have to walk the tree all over again.

On the other hand, if we *have* the list of files, it's very easy to count them: just use the built-in `len` function. Finding files

is the more general problem, so let’s try to solve that in a useful way.

Starting at the top

As usual, let’s first think about the `main` function we’d like to write, with absolutely minimal paperwork. Something like this would be nice:

```
func main() {
    paths := findgo.Files(os.Args[1])
    for _, p := range paths {
        fmt.Println(p)
    }
}
```

(Listing findgo/3)

It wouldn’t actually be that simple in practice, since we’d need to check that `os.Args[1]` exists, report errors, and so on. But the CLI isn’t the point of this example, and we’ve covered all that machinery in previous chapters, so let’s take it as read for now, and see how `findgo.Files` would work.

It would need to take the pathname of some folder as its argument, and it would walk the tree rooted at that folder finding Go files, in the way that we’ve already done as a proof of concept. Let’s write a test for that.

```
func TestFilesCorrectlyListsFilesInTree(t
*testing.T) {
    t.Parallel()
    want := []string{
        "file.go",
        "subfolder/subfolder.go",
```

```
        "subfolder2/another.go",
        "subfolder2/file.go",
    }
    got := findgo.Files("testdata/tree")
    if !cmp.Equal(want, got) {
        t.Error(cmp.Diff(want, got))
    }
}
```

(Listing findgo/3)

We'll copy our example tree of files into `testdata/tree`, so the test has something to work on. The test is saying that if we call `Files` with this path, in which there are four Go files, it should return the expected slice of strings. Over to you to make this work.

GOAL: Implement `Files`.

HINT: Well, we've done most of the work already, haven't we? We can take the code from our `main.go` proof of concept and move it straight into the `findgo` package.

All we need to change is that, instead of incrementing a counter every time we find a file, we append its path to a slice instead. That slice is the result we should return once `fs.WalkDir` is done.

SOLUTION: Code lifted and shifted!

```
func Files(path string) (paths []string) {
    fsys := os.DirFS(path)
    fs.WalkDir(fsys, ".", func(p string, d
fs.DirEntry, err error) error {
```

```go
        if filepath.Ext(p) == ".go" {
            paths = append(paths, p)
        }
        return nil
    })
    return paths
}
```

(Listing findgo/3)

Excellent! The program works perfectly on our little test tree. But we can imagine that a program with more complicated logic *might* run into problems, especially in large and complicated filesystems. How could we test cases like that?

The `fs.FS` abstraction

One problem with tests that use files on disk is that disk access is, in general, extremely slow (at least, compared to accessing data in memory). Old-school spinning disks have a pretty high *latency*: it takes a long time to start sending data, because the read head has to physically travel to a specific place on the disk first.

Solid-state disks are faster, or at least have lower latency, but the data still has to get to the CPU over some relatively low-bandwidth link, such as a USB bus. The technology is always changing, but it’s a reasonable rule of thumb that we can access data in memory an order of magnitude or two faster than data on disk—that is to say, 10-100x faster.

This is the kind of speedup that every programmer dreams of, so can we achieve it here? In general, could we test filesystem operations on some in-memory data structure instead of having to access a disk?

As we saw earlier in this chapter, `fs.FS` is an interface. Specifically, an interface with the method `Open`. This effectively maps some slash-separated pathname to an `fs.File`, where `fs.File` is some file-like object that supports statting, reading, and closing. Such small interfaces make for powerful abstractions.

If we think a little more generally, we could treat a filesystem as a kind of key-value database. The keys here are pathnames, and each key is associated with a unique value representing a file. Specifically, this is a *path-value* database, because the keys obey a hierarchical pattern. In our example, the key `subfolder/subfolder.go` maps to whatever data is represented by the `subfolder.go` file.

Any path-value map is a "filesystem"

In fact, there are lots of things that fit this abstraction. The Web itself is a kind of path-value database: a URL is a hierarchical, slash-separated pathname that maps to some specific data. A JSON, YAML, or CUE document has the same implicit structure. So does a Go import path, or the files contained in a ZIP archive.

In principle, we could use an `fs.FS` to access any of these things. As long as there's some concept of a hierarchical pathname mapped to a piece of data, the filesystem abstraction can be useful.

We could even imagine implementing `fs.FS` with some simple data type based on a Go map. All we'd need to do is give it an `Open` method, that maps string keys to some value that satisfies `fs.File`. Since `fs.File` is also a small interface, that's easy to do.

This would be very useful for testing any code that works with filesystems. While a tree of files on disk is a perfectly

good `fs.FS`, it's a very slow one compared to a Go map, or anything else that lives in memory.

Instead, we could create simple pieces of test data, such as our Go file tree, by using a map literal. And the test would execute very fast, because it wouldn't need to make any disk accesses: everything would be in memory.

The `fstest.MapFS` filesystem

As it happens, we don't need to implement such a type ourselves, because the standard library already provides one: `fstest.MapFS`. This satisfies the `fs.FS` interface, and it's easy to construct MapFS-based filesystems in code.

Let's see how to rewrite our test for `Files` using a MapFS instead of regular disk files.

```
func TestFilesCorrectlyListsFilesInMapFS(t
*testing.T) {
    t.Parallel()
    fsys := fstest.MapFS{
        "file.go":                {},
        "subfolder/subfolder.go": {},
        "subfolder2/another.go":  {},
        "subfolder2/file.go":     {},
    }
    want := []string{
        "file.go",
        "subfolder/subfolder.go",
        "subfolder2/another.go",
        "subfolder2/file.go",
    }
    got := findgo.Files(fsys)
    if !cmp.Equal(want, got) {
        t.Error(cmp.Diff(want, got))
```

```
    }
}
```

(Listing findgo/4)

A MapFS is some kind of Go map, as the name implies. Specifically, it's a map of `string` (the pathname) to `*MapFile` values.

In this test, we first set up our variable `fsys` as a MapFS literal where each of the keys represents one of our test files in `testdata/tree`.

Notice that we haven't included the folders `subfolder` and `subfolder2` as explicit keys in this map. We could, but it's not necessary, because they're implied by the pathnames of the files within them. For example, the path `subfolder2/file.go` implies a containing folder `subfolder2`. The MapFS implementation will "simulate" that folder for us if it needs to.

A `MapFile` is the "fake file" object that satisfies `fs.File`, and it has fields for storing data as a `[]byte`, file permissions, and so on. In other words, everything we're allowed to do with an `fs.File`, we can do with a `MapFile`.

As it happens, the code under test here only needs to look at the pathnames, so the `MapFile` values don't matter at all. We can leave them empty, so each element of this map is just the empty struct `{}`.

When you need to test some function that takes a filesystem, it's faster to give it a MapFS than a real tree of files on disk, because we eliminate so many relatively slow I/O operations.

What should happen when we call `Files` with this filesystem? Well, just the same as before: it should recursively walk the filesystem looking for `.go` files and return the slice of matching paths. The result should be just the same as it was with the tree of real files on disk.

Indeed, `Files` doesn't know that it's *not* looking at disk files, and that's the point. We've generalised it to something that simply queries a path-value database.

We'll need to make a couple of minor changes to both `Files` and our previous `TestFiles...` function to make this work. Can you see what to do?

GOAL: Get all tests passing.

HINT: We need to update `Files` to take an `fs.FS` as its parameter instead of a pathname. And since we're *receiving* the filesystem now, we needn't open it ourselves using `os.DirFS`, so we can remove that call.

Adding a filesystem to our API

SOLUTION: Here's the modified `Files` function:

```
func Files(fsys fs.FS) (paths []string) {
    fs.WalkDir(fsys, ".", func(p string, d
fs.DirEntry, err error) error {
        if filepath.Ext(p) == ".go" {
            paths = append(paths, p)
        }
        return nil
    })
    return paths
}
```

(Listing findgo/4)

Next, we'll update the other test: the one that calls `Files` on a disk-based filesystem. Instead of just passing the root path, it will now need to call `os.DirFS` itself to create the filesystem:

```
func TestFilesCorrectlyListsFilesInTree(t
*testing.T) {
    t.Parallel()
    fsys := os.DirFS("testdata/tree")
    want := []string{
        "file.go",
        "subfolder/subfolder.go",
        "subfolder2/another.go",
        "subfolder2/file.go",
    }
    got := findgo.Files(fsys)
    if !cmp.Equal(want, got) {
        t.Error(cmp.Diff(want, got))
    }
}
```

(Listing findgo/4)

We don't really need this test anymore, since it just replicates what the MapFS-based test does, only a lot more slowly.

But before we delete it, it would be interesting to know *how* much slower it is. Let's do a little science.

Timing potentially slow operations

Go provides excellent *benchmarking* facilities for measuring performance, in the same standard `testing` package we've

been using. Before we see how to use them to write a benchmark for `Files`, let's think about the problem a little. What's actually involved in measuring the speed of some function?

If we didn't have built-in benchmarking facilities, and we wanted to measure the time elapsed during some potentially slow operation, what could we do?

One approach might be something like this:

```
start := time.Now()
slowOperation() // could take a while
elapsed := time.Since(start)
fmt.Println(elapsed)
```

By finding the current *wall-clock time* before and after the operation, and subtracting one from the other, we get the elapsed time. Fine. But how reliable is that figure?

Not all that reliable, in fact: your computer is doing a lot of other things at the same time as running this program, and any of these might cause the operation to be faster or slower.

In other words, the environment is *noisy*, causing *jitter* in our time measurements: if we repeat the same operation many times, we'll get results that differ by essentially random amounts. If the jitter is large compared to the total time we're measuring, that's a problem.

To avoid this, it's a good idea to run the operation some large number of times (say a million), and then divide the total by that figure to get the *mean* time per operation. Hopefully the noise will mostly cancel out, giving us a more accurate result. So we'll call the function inside a loop that executes many times.

But how many times? If the function is very fast, we should run it very many times to get a statistically useful answer. But if it’s slow, running it many times would make the test take a long time. So the number of times we need to run it depends on how long it actually takes.

We could imagine determining the number of loop iterations by trial and error, first calling the function a small number of times to see how long that takes, optionally running it more times, and so on, until we’ve reached some overall time limit (ten seconds, let’s say).

Writing benchmark functions

Once again, the standard library can take over this workload for us. All we need to do in a benchmark function is write a loop that calls our function N times, where N is some variable that can be set automatically by the test machinery. Here’s what that looks like:

```
func BenchmarkFilesOnDisk(b *testing.B) {
    fsys := os.DirFS("testdata/tree")
    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        _ = findgo.Files(fsys)
    }
}
```

(Listing findgo/4)

Just as test function names begin with the word `Test`, benchmark function names begin with the word `Benchmark`. And just as tests take a `*testing.T` parameter that controls the test execution, benchmarks take an analogous `*testing.B` parameter that controls the benchmark execution. We’ll see exactly how that works in a moment.

We first want to measure how long it takes `Files` to search a disk-based filesystem, so we create one using `os.DirFS` in the same way as the regular test.

The stopwatch is running from the moment the benchmark function starts. But since we don't want the call to `os.DirFS` to be included in the benchmark timing, we then call `b.ResetTimer` to reset the clock to zero: "start timing from now!"

And now we write the loop we discussed earlier, to call our function a variable number of times. In fact, it will loop `b.N` times, where `b.N` is controlled by the benchmark machinery so as to make the overall time around ten seconds, to give a more reliable result.

To run this benchmark, use the command:

```
go test -bench .

goos: darwin
goarch: amd64
pkg: findgo
cpu: Intel(R) Core(TM) i7-3615QM CPU @ 2.30GHz
BenchmarkFilesOnDisk-8    5434  333303 ns/op
PASS
```

Because a benchmark is just a special kind of test, we're using the `go test` command, which will also run our regular tests. This makes sense, because there's no point benchmarking anything unless the tests are passing: a function can be arbitrarily fast if it doesn't have to be *correct*.

However, we also supply the `-bench .` flag, asking `go test` to also run any benchmarks whose names match the regular expression ".". Since this "dot" wildcard matches any

character, this will run all benchmarks defined in the package.

The output gives us some contextual information about the system we're running the test on (a `darwin` kernel, signifying macOS, on the `amd64` architecture, specifically an Intel Core i7 clocked at 2.3GHz).

It then reports the results of the benchmark:

```
BenchmarkFilesOnDisk-8    5434  333303 ns/op
```

The `-8` following the benchmark name indicates that 8 CPU cores were available to Go, in case we need to know that: it might be important for concurrent programs.

The next number, `5434`, indicates how many times the loop executed overall, and thus how many times `Files` was called.

Finally, the figure `333303` tells us that the mean time per operation (that is, per call to `Files`) was 333,000 nanoseconds and change. That is to say, about 333 microseconds.

Seems pretty fast! Can we do better using the `MapFS`-based filesystem? Let's find out:

```
func BenchmarkFilesInMemory(b *testing.B) {
    fsys := fstest.MapFS{
        "file.go":                {},
        "subfolder/subfolder.go": {},
        "subfolder2/another.go":  {},
        "subfolder2/file.go":     {},
    }
    b.ResetTimer()
```

```
    for i := 0; i < b.N; i++ {
        _ = findgo.Files(fsys)
    }
}
```

(Listing findgo/4)

Here's the result:

```
go test -bench .

...
BenchmarkFilesOnDisk-8    5492   353633 ns/op
BenchmarkFilesInMemory-8  91056  13483 ns/op
```

Looking at the last column, we can see that the disk-based benchmark again took around 350 microseconds, while the memory-based one took only 13 microseconds. That's about 25 times faster, or about one order of magnitude: roughly what we expected.

So was it worth it, just to save 0.3 milliseconds when running tests? Well, we're all busy people, but not so busy that we couldn't spare a millisecond or two. If you think about the impact of a 25x speedup on large test suites, though, eliminating as much unnecessary disk I/O as possible in this way makes a lot of sense.

We also save the disk space, of course, and arguably it's clearer to see the file tree structure directly in the test code, rather than having to look at the `testdata` folder. So a MapFS is a neat tool to have available whenever we want to test code that operates on filesystems.

Taking `fs.FS` makes APIs more flexible

There's nothing stopping you from writing your own `fs.FS` implementation, and it's quite straightforward. Indeed, whenever you're writing Go code to deal with data that could in principle be addressed as a path-value tree, you might like to consider accepting an `fs.FS` as input, or making your data type satisfy `fs.FS` itself. It all helps to make your packages more flexible, useful, powerful, and friendly.

We can see the effect of this with our file-finder example. Initially, because it took a disk pathname, the only thing we could use it to search was a disk-based filesystem. Now that we've updated it to accept `fs.FS`, it can operate on anything satisfying that interface. Our test can pass it a `MapFS` and it works just fine.

So what else would work? We mentioned earlier some examples of other things that satisfy `fs.FS`. Just for fun, let's try `Files` with a filesystem derived from a ZIP archive: after all, it should work, shouldn't it?

First, let's zip up our test `findgo` folder and its contents using the `zip` command. If you don't have that command, you can use anything that creates standard ZIP files, including the macOS Finder's "Compress" action.

```
cd testdata

zip -r files.zip tree/

adding: tree/ (stored 0%)
adding: tree/subfolder/ (stored 0%)
adding: tree/subfolder/subfolder.go (stored 0%)
adding: tree/subfolder2/ (stored 0%)
adding: tree/subfolder2/another.go (stored 0%)
```

```
adding: tree/subfolder2/file.go (stored 0%)
adding: tree/file.go (stored 0%)
```

All these files are empty, which is why zipping them doesn't seem to save much space, but that's not the point: we just want a ZIP file to play with. Now, how do we open it as a filesystem?

Helpfully, Go provides facilities for reading ZIP files in the standard library package `archive/zip`, so here's our test:

```
func TestFilesCorrectlyListsFilesInZIPArchive(t
*testing.T) {
    t.Parallel()
    fsys, err :=
zip.OpenReader("testdata/files.zip")
    if err != nil {
        t.Fatal(err)
    }
    want := []string{
        "tree/file.go",
        "tree/subfolder/subfolder.go",
        "tree/subfolder2/another.go",
        "tree/subfolder2/file.go",
    }
    got := findgo.Files(fsys)
    if !cmp.Equal(want, got) {
        t.Error(cmp.Diff(want, got))
    }
}
```

(Listing findgo/4)

We call `zip.OpenReader` with the pathname of our test ZIP file, and the result is a value that satisfies `fs.FS`, so we can

pass it directly to `Files`. And, of course, it gives us the correct answer:

```
PASS
```

Just by using the `fs.FS` abstraction instead of talking directly to the disk, we made our `Files` function work with ZIP archives, with no extra effort. And if you expose this type in your own APIs, other people will be able to use *your* `fs.FS` values in their own code.

If your function operates on trees of files, or sets of path-value pairs, then, consider taking an `fs.FS` parameter instead of a root path.

Going further

If you want some ideas for ways to get more practice with the ideas we've talked about, here is a mini-project suggestion:

- Write a Go package that finds all files in a given filesystem older than a specified duration (for example, 30 days). That is, files whose "last modified" date is 30 days or more before the present.

  Test your package using a `MapFS`. Check that it works on a disk filesystem too.

  You might like to compare your solution with that shown in listing `older/1`.

7. Commands

> *This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.*
> —Doug McIlroy, quoted in Peter Salus's "A Quarter Century of Unix"

In previous chapters we've used Go to create executable commands that users can run on their systems, such as `words` or `lines`. That's great, but what if we wanted to run such a command from within a Go program itself? What would that look like?

The exec package

For example, suppose we want to run the command `ls` to list the files in the current directory. When we ask the shell to run that command, it makes a call to the operating system kernel to start a new process. There's more work involved in this than you might think.

What even is a process?

}

The kernel first allocates some memory to store the process's context and other housekeeping information about it. Then it *loads* the specific executable file we requested (something like `/bin/ls`), which means copying its bytes from disk into memory. Finally it starts *executing* the machine code instructions at the beginning of the program.

So the kernel-level API for running a command needs to take at least the path to the executable, and in practice there are a few more arguments.

On Unix-like systems, there's some *environment* (a set of key-value pairs storing information the program might need) for the process, possibly some *command-line arguments*, and also some *file descriptors*: byte streams allowing the program to read input and write output in a standard way.

This sounds like a lot of paperwork, and while we *could* do it (using the `os.StartProcess` standard library call), we'd prefer not to. So it's good to know that there's a higher-level API provided by the `os/exec` package.

Let's see how that works in a Go program with our `ls` example:

```
package main

import (
    "os/exec"
)

func main() {
    cmd := exec.Command("/bin/ls")
    cmd.Run()
}
```

First we call `exec.Command` with the path to some executable (`/bin/ls`). If the `ls` executable is in a different place on your system, you'll need to use a different path here (use `whereis ls` to find it).

This returns a value of type `*exec.Cmd`, and we assign it to the variable `cmd`. Note that nothing has actually *happened* so far: we haven't started a new process yet, just created the abstraction we'll use to control it.

Calling the command's Run method is what actually causes the kernel to create a new process, load the `ls` executable, and run it. If we run this Go program, then, we should see just one file listed: the `main.go` file containing its source code. Let's try.

```
go run main.go
```

Hmm. Nothing happened. In fact, the `ls` command worked perfectly well, but we have no way of knowing that, because we didn't see its output.

Managing command output

We said earlier that the Unix process model includes input and output file descriptors: the equivalent of `os.Stdin` and `os.Stdout` (and `os.Stderr`) in Go. And an `exec.Cmd` has fields for these to be attached, but we didn't attach anything, so the output of `ls` went… nowhere.

We'll need to attach something to `cmd.Stdout` in order to see that output. If we attach `os.Stdout`, for example, we should see it printed to the terminal:

```go
func main() {
    cmd := exec.Command("/bin/ls")
    cmd.Stdout = os.Stdout
```

```
    cmd.Run()
}
```

```
go run main.go
```

```
main.go
```

That’s more like it! In practice, when we’re running external commands from a Go program, we usually want to capture that output somehow: for example, in a `bytes.Buffer`. But this is enough to prove the concept.

Can we do more? How about passing arguments to the command, for example?

It turns out that `exec.Command` is variadic: it takes any number of strings after the first, and interprets them as arguments to be passed to the command.

```
func main() {
    cmd := exec.Command("/bin/ls", "-l",
"main.go")
    cmd.Stdout = os.Stdout
    cmd.Run()
}
```

Here’s the output:

```
-rw-r--r--  1 john  staff  127 30 Sep 15:46
main.go
```

When not to use exec

Although it makes a neat demo, we wouldn’t actually want to run the `ls` command from a Go program: that functionality is much better implemented in Go itself, and we’ve seen how to do it in the chapter on filesystems.

Similarly, there's no benefit in *shelling out* (that is, invoking a subprocess) to run standard file-management commands such as `mkdir`, `touch`, `rm`, and so on. All of these things are either already available in the standard library, or easy to write in Go. Don't shell out to an external command whose functionality is easily replicated in native Go code. That just adds an unnecessary dependency.

Suppose we wanted to do something a bit more advanced, though, such as triggering a Kubernetes deployment with `kubectl`, or creating a cloud virtual machine with the `gcloud`, `aws`, or `az` commands. Would this be a good case for running external commands from Go?

Not really. Apart from the dependency issue that we already discussed, complicated programs like `kubectl` tend to change over time. New flags and verbs are added, old ones deprecated, and the behaviour of the program can change radically.

A new version of `kubectl` could easily cause your Go program to stop working properly. And since we usually want the output from such tools, we have to *parse* it to get the information we need, and that's equally fragile.

Tiny changes in the output format of external commands can completely break programs that rely on them, and this kind of thing isn't easy to test automatically.

A better approach is to look for a Go package that provides the same, or similar functionality to the command-line tool. Often, the CLI tool itself will use that package to implement what it does.

For example, this is the complete `main` function for the `kubectl` tool, which I'm sure we will all agree does a lot:

```go
func main() {
    command := cmd.NewDefaultKubectlCommand()
    if err := cli.RunNoErrOutput(command); err !=
nil {
        // Pretty-print the error and exit with an
error.
        util.CheckErr(err)
    }
}
```

(kubectl.go)

It doesn't look like there's much code here, but that's the point: this minimal `main` function just calls into the accompanying `cli` package to do the actual work. And if `kubectl` can use that package, so could we.

All the important machinery of `kubectl`, then, is in Go packages that we can import and use in our own programs, which is sensible. Anything the `kubectl` binary can do, we can do too, using pure Go, and without fragile dependencies on external commands.

We've worked hard throughout this book to design programs with a "minimal `main`", and now that we're looking at other people's programs from the user's point of view, we can see why that's so valuable. `main` can't be imported into any program, so it shouldn't do anything that might be useful to others.

When to use exec

There are legitimate use cases for running external commands from Go. For example, the *purpose* of the Go program might be to run such a command, perhaps to automate some process that's currently done manually.

It may not be practical or desirable to completely replace this command with a Go program, or at least not yet, so the compromise option is to execute it from Go for the time being.

And there isn’t always a convenient Go SDK or package that we can use to get the same functionality that the command provides. For example, the command might be written in C, or use a C shared library. If this is the only implementation of the necessary behaviour that exists, then we have no choice but to use it, or write our own.

While we can use the cgo interoperability layer to call C functions from Go, that’s always a last resort, and greatly complicates our programs. In such a case, it would be better to use exec to run some command instead, despite the disadvantages that we’ve already discussed.

Migrating from shell scripts to Go

It’s sometimes said that the internet is held together with shell scripts and duct tape, and it’s certainly true that a great many organisations depend on some rather fragile and ancient (and almost certainly buggy) shell scripts. While the shell is a great tool for one-shot tasks and rapid prototyping, it’s not ideal for developing robust, maintainable software over the long term.

Why use Go to run commands?

It’s possible to write and maintain relatively robust shell programs, but it’s uphill work. Just use Go instead. Go is a much better language for this job, so many important programs that were once made of duct tape are now being migrated to Go.

When the program is a script that simply executes a bunch of Unix commands, for example, the first step on the migration path may well be to replace it with a Go program that uses exec to run the same bunch of commands.

That may not sound like a big improvement, but we can do a lot from this position. We can write unit tests, and use Go for things like text processing, instead of relatively awkward `grep`, `awk`, and `sed` commands.

I yield to no one in my admiration of things like `awk`, and even Perl: they're terrific at doing what they do. What they *don't* necessarily do well is communicate data to and from other tools, except in the very simple byte-oriented way provided by the Unix API.

In other words, we can pipe lines of text in and out of tools, but we can't easily do complex filtering, logic, or manipulation. That's a job for a programming language. Luckily, we have one.

A command wrapper in Go

When we need the behaviour provided by proprietary or other closed-source programs, especially if they're only available as executable binaries, we may have no alternative but to use exec.

The `pmset` command

For example, suppose we want to get the current battery charge status on a Mac. We can do this easily on the command line with the OS-specific `pmset` command:

```
pmset -g ps
```

```
Now drawing from 'AC Power'
 -InternalBattery-0 (id=10879075) 98%; charging;
    0:42 remaining present: true
```

Getting this information from a Go program is not so straightforward, though. How does `pmset` do it? Looking at the C source code for this command, we find that it gets the data by making a system call to the macOS kernel.

While we *could* make the same system call from a Go program, this is likely to involve a lot of paperwork; probably more than we'd care to do just to get the battery status.

Although we don't want to use external commands when it's easy to replicate their functionality in Go, that's not the case here. Running the `pmset` command makes our Go program much simpler than it would be if we tried to implement the same behaviour ourselves.

Since the `pmset` command is part of macOS, it's fairly safe to assume we'll be able to execute it on any Mac. To put it another way, executing the command is no *less* portable or reliable than making the equivalent system call. And the output doesn't look too difficult to parse.

What can we test?

Let's see how to approach writing a `pmset` *wrapper* in Go. This will be a Go package that users can import and call some function to get the current battery status, without having to worry about how that actually works under the hood.

If you're not using macOS, just substitute some equivalent command, such as `acpi` on Linux, or `powercfg` on Windows, and make the appropriate tweaks to the code as you go.

The logic will be roughly the same whatever command you're using.

Go ahead and create a new folder for this project (you might call it `battery`, for example). Now, what's the first test we should write?

Suppose what we want is some Go function that will give us the current charge status of the battery. You might find it difficult to think of any test that you could write for this: how can you know in advance what the correct battery charge status will *be*?

Any function that returns dynamic information based on some external conditions can't be tested by comparing its result directly against expectation. We need to think harder.

Throughout this book we've been asking the question: *what behaviour are we really testing?* So what's the answer here?

It's tempting to answer "The function correctly gets the battery status", but that's wrong. The "getting the battery status" behaviour isn't actually *in* our code; it's in the `pmset` command. So what does *our* code do, then?

Breaking down behaviour into chunks

The behaviour we're really testing in this case is in two parts:

1. We execute the `pmset` command with the correct arguments
2. We correctly parse the `pmset` output to get the battery status

When you frame it this way, it's easier to see how to test these two chunks of behaviour, isn't it? The "output" of the

first one is simply a string representing the appropriate `pmset` command line.

The “input” of the second behaviour is some string representing the kind of text that `pmset` prints out when you run it, and its output is whatever battery information we manage to parse from it.

In between these two steps there’s some hidden stuff that’s outside our code: basically, everything `pmset` does. And we’re not interested in testing `pmset`, so we can focus entirely on the two things *our* code does.

In this case, the first behaviour is trivial, since the required command line is always the same: it’s just `pmset -g ps`. No need to test that we can produce this string. We can imagine situations where this could be more complicated, and would need testing, but let’s consider this one solved for now.

An otherwise hard-to-test function can often be broken down into sub-behaviours, each of which can be refactored out to a testable function. We can then trivially *compose* these behaviours into a single function that users can call.

Parsing command output

The second behaviour needs a little more thought. First, we’ll need some test input. The best test data is always the *real* data, so let’s use `pmset` to generate it, and *pipe* the output to a file in the `testdata` folder.

```
mkdir testdata

pmset -g ps >testdata/pmset.txt
```

As we saw earlier, the result will be something like this:

```
Now drawing from 'AC Power'
 -InternalBattery-0 (id=10879075) 98%; charging;
0:42
remaining present: true
```

So that’s the input to our parsing function; what output do we want? Well, there are several useful pieces of information we *could* glean from this text, such as the AC power status, the battery ID, its charge percentage, and the charging time remaining.

That’s too much fun for one chapter. Let’s just focus on the charge percentage for now: if we can extract that, presumably we can extract the other things the same way. We’d like to avoid being tempted into solving more problems than we strictly need to.

Without worrying yet about exactly *how* we’re going to get the charge number, we can certainly write a test for the function that does it. Take a minute and try to sketch out a suitable test.

GOAL: Write a test for the “parse `pmset` output for battery charge percentage” behaviour. Use the real output from `pmset -g ps` as your test data.

HINT: First, what input will the function need to take? Well, what we’ll *have* is a string, since the output from `pmset` is plain text, so a string parameter sounds reasonable.

What about outputs? The most useful thing would be a *number* representing the battery charge percentage, so we could have the function return an `int`. `pmset` doesn’t report fractional percentages, so `int` will be fine.

But, thinking ahead a little, we can see that there might be other information we'll want to extract from the data in future. Can we leave room in our API for this?

We'd like to avoid changing the signature of our function, as this would break any user code that calls it. Instead, it'll be convenient to have some struct type representing all the information we want to know about the battery status. We can always add new fields to it, without making breaking changes to our public API.

So let's define a `want` variable of this struct type, whose only field (so far) is the charge percentage. To be unambiguous, then, let's name it `ChargePercent`.

And the rest should be straightforward: call the function, and compare the result we `got` against `want` in our now-familiar way.

Testing the parsing function

SOLUTION: Here's my attempt at this:

```
func TestParsePmsetOutput_GetsChargePercent(t
*testing.T) {
    t.Parallel()
    data, err := os.ReadFile("testdata/pmset.txt")
    if err != nil {
        t.Fatal(err)
    }
    want := battery.Status{
        ChargePercent: 98,
    }
    got, err :=
battery.ParsePmsetOutput(string(data))
    if err != nil {
```

```
        t.Fatal(err)
    }
    if !cmp.Equal(want, got) {
        t.Error(cmp.Diff(want, got))
    }
}
```

(Listing battery/1)

This looks pretty much like most of the other tests we've written, which makes it straightforward to understand. We read the test data, call our parse function, and compare `want` against `got`.

What's `want`? Our `pmset` output text shows a charge percentage of 98, so that's the `ChargePercent` we should expect.

`ParsePmsetOutput` needs to return an `error` as well as the `Status` struct, because clearly parsing *can* fail: we might not be able to make sense of the input if its format has changed, for example.

Parsing command output

Let's write a null implementation of `ParsePmsetOutput` and check that we get the expected test failure:

```
-       ChargePercent: 98,
+       ChargePercent: 0,
```

Good. This time, the test was the easy bit, and the *implementation* might be more challenging. Let's see.

GOAL: Implement `ParsePmsetOutput`.

Clarifying the problem

HINT: This needs a little thought. Let’s clarify the problem statement. We’re expecting input similar to our test data:

```
Now drawing from 'AC Power'
 -InternalBattery-0 (id=10879075) 98%; charging;
0:42
remaining present: true
```

And we want to be able to extract the charge percentage: in this case, the value 98. How could we do that?

It’s not a case of simple string matching, unfortunately. We don’t know the value in advance, so we can’t search for it.

The value isn’t always at the same byte position in the text, either, so we can’t just read bytes 63-64, or something like that. We can’t even rely on counting words to get to the text we want, since presumably the part about “drawing from ‘AC Power’” could change to something else.

What could we do?

SOLUTION: Some programmers, when confronted with a problem like this, think “I know, I’ll use regular expressions!” (Now they have two problems.)

What kind of regular expression would work here? Well, the thing we want is a *sequence of digits*, but there’s more than one such sequence in our example. Can we narrow it down a bit more?

Writing a regular expression

On closer inspection, it seems that there’s only *one* digit sequence followed by a `%` character, so we can try to match that.

Here’s a regular expression that will match one or more digits followed by a `%`:

```
[0-9]+%
```

What can we do with that in Go? Here’s an example:

```
r := regexp.MustCompile("[0-9]+%")
result := r.FindString("the charge is 98%")
```

Our real string is different, but this simpler version will make it easier to see what’s going on. First, we *compile* the regular expression, which is like a mini-program, to get some value `r` that we can use. Then, we call its `FindString` method with our input text.

`FindString` returns the first string that completely matches the specified regular expression, so in this case that’s:

```
98%
```

Great, but we only want the 98, not the trailing `%`. We need to *match* the `%` character, to make sure we’ve got the right digit sequence. But once we *have* it, we then want to extract only the digits from it.

We can do this by adding parentheses to our regular expression, to create a *capturing group*:

```
([0-9]+)%
```

If this expression matches a chunk of text, then we’ll be able to extract just the contents of the capturing group: the part inside parentheses. Here’s what that looks like:

```
r := regexp.MustCompile("([0-9]+)%")
matches := r.FindStringSubmatch("the charge is
98%")
```

Whereas `FindString` only returns the entire matched string, `FindStringSubmatch` returns a slice of strings. The first element of this slice is the entire matched string, as before, but the second is the contents of the capturing group.

If we had more groups in the regular expression, then there would be extra elements in this slice, each holding the contents of the corresponding group. But as it happens, we only have one group, so the `matches` slice contains exactly two elements:

```
["98%", "98"]
```

The second element is the one we want, and we can use the index expression `matches[1]` to get it. Once we have that string, we feel confident that we can use something like `strconv.Atoi` to turn it into an integer.

Now we have a plan. So how can we put all this together to implement `ParsePmsetOutput`?

Using the regexp

It happens that the `MustCompile` operation is relatively expensive, compared to the actual string matching. But compilation only needs to be done once per program run, so we can do it at package level, in a `var` statement.

If the call to `MustCompile` were instead inside our parsing function, then the regexp would be recompiled every time the function is called, which is unnecessary. Once we *have* the compiled value, we can use it to match against any number of input strings relatively cheaply.

Here’s my version of the function, then:

```
var pmsetOutput = regexp.MustCompile("([0-9]+)%")

func ParsePmsetOutput(text string) (Status, error)
{
    matches :=
pmsetOutput.FindStringSubmatch(text)
    if len(matches) < 2 {
        return Status{}, fmt.Errorf("failed to
parse pmset \
            output: %q", text)
    }
    charge, err := strconv.Atoi(matches[1])
    if err != nil {
        return Status{}, fmt.Errorf("failed to
parse charge \
            percentage: %q", matches[1])
    }
    return Status{
        ChargePercent: charge,
    }, nil
}
```

(Listing battery/1)

Having used the compiled `pmsetOutput` regexp to test the input for matches, we now need to check whether or not it actually matched. If it did, we should have the contents of the capturing group, so there will be at least two elements in `matches`. We can check `len(matches)` and return an error if this isn’t the case.

Note that we include the output that we couldn’t parse: just saying “failed to parse” would be no help, either to the user

or the developer. Including the unparsable text in the error message takes only a few more keystrokes, but makes a big difference to the usefulness of the error.

Now we know that we successfully extracted the string of digits we need, we can call `strconv.Atoi` to turn it into an integer value. This returns `error`, because not all strings represent valid integers. We feel *pretty* sure that ours will, but again, let's not take chances. So we check that error too.

Finally, we have the `int` value we need, so we can construct a suitable `Status` literal and set its `ChargePercent` field to the value we calculated.

Let's try the test again:

```
PASS
```

Mischief managed! We can now feel confident that, given some genuine `pmset` output, we can extract the battery status from it.

We can turn our attention, then, to the other part of the task: *getting* that output in the first place. We can imagine some function `GetPmsetOutput` that runs `pmset` using `exec` and returns its output as a string.

But there's a problem. If we're not allowed to execute external commands in a unit test, how can we possibly write a unit test for such a function?

We can't, clearly. But unit tests aren't the only kind of tests we can write.

Integration tests

A unit test, as the name implies, tests some *unit* of your code, such as a function, usually in isolation. Its job is to verify that the function's *logic* is correct, so it tries to avoid using any external dependencies, such as commands.

The job of an *integration* test, on the other hand, is to test what happens when we *do* use those external dependencies. It validates our assumptions about how they work: for example, that we execute the right command in the right way.

Why isolate integration tests?

Why make a distinction between these two kinds of tests? Well, unit tests need to be fast and lightweight, because we run them very often, and the only time they should fail is when something's wrong with our code.

We don't need to run integration tests so often, though, because the only way they could break is if something external changed: the `pmset` command was updated or removed, for example.

Unit tests check a program's behaviour given certain assumptions about external dependencies. Integration tests check those assumptions are still *correct*. So it doesn't matter if integration tests are relatively slow, or use external dependencies, as long as we can figure out a way to avoid running them every time we run unit tests.

Build tags

One common way to control the execution of integration tests is to use a *build tag*.

A build tag is a special comment in a Go file that prevents it from being compiled unless that tag is defined. There are

some *predefined* build tags: for example, the tag `windows` is defined if we're building on Windows. Similarly, `darwin` and `linux` indicate macOS and Linux.

Here's what specifying a build tag looks like in a Go file. It needs to be the first line in the file, and be followed by a blank line before the package declaration:

```
//go:build darwin

package ...
```

We could use this mechanism to provide OS-specific implementations of a certain piece of code, for example. We would write the macOS version in a Go file protected by the `darwin` build tag, the Linux version protected by `linux`, and so on.

But we can also define arbitrary build tags of our own. Suppose we put the following at the beginning of a new Go file:

```
//go:build integration

package battery_test
```

Now this file will be ignored by the Go tools unless the `integration` tag is defined, and under normal circumstances it won't be, because it's not one of the predefined tags. If the file contains tests, they won't be run by the `go test` command. If it contains implementations, they won't be built by `go build`, and so on.

So suppose we write some test for `GetPmsetOutput` and we put it in its own Go file, protected by the `integration` build

tag. It won’t be run by `go test`, which is what we want, but then how *do* we run it when we want to?

The answer is that you can supply arbitrary build tags to the `go test` command, so we would run something like this:

```
go test -tags=integration
```

Now that tag is defined, so the file will become visible to Go, and the test will be run. But if we don’t supply that tag on the command line, it won’t.

Testing the command runner

What integration test should we write? We know we’re going to call `GetPmsetOutput`, and that that function will execute the `pmset` command. Fine. So what can we test about it?

Well, another way to phrase that question is to ask “What could fail?” Clearly, running the command could fail for a number of reasons, most likely that the command doesn’t exist or has a different path.

If we ran this test on some non-macOS machine, for example, it would presumably fail. So that’s one thing we can check. In that case we’d get an error something like this:

```
fork/exec /usr/bin/pmset: no such file or
directory
```

If this happens in the test, we could do some work to make the test failure friendlier. For example, we could check the value of `runtime.GOOS`, which tells us what operating system Go thinks it’s running on. If it’s something other than `darwin`, we could report a failure like “This test will

only work when run on macOS and when `/usr/bin/pmset` is available.”

That’s getting a bit fancy, though, so for now we’ll just skip the test, using `t.Skip`, if we can’t run the `pmset` command for whatever reason.

There’s one other sanity check we need. Even if `pmset` is available, on machines without batteries (a Mac mini, for example), it won’t report anything useful to us. So if we don’t find the string `InternalBattery` in the output from `pmset`, we should also skip this test.

Now it’s reasonably safe to call the function under test, so what should we test about it? `GetPmsetOutput` returns a string, so we could test that the string isn’t empty, but that doesn’t prove much. Can we do more? We don’t know in advance what the string actually *is*, so we can’t simply compare it against a string literal.

But there’s something else we *can* do with it. We can pass it to `ParsePmsetOutput`. After all, if we executed the `pmset` command correctly, then its output *should* be parseable without error, shouldn’t it?

So we can check the error value, but is there anything useful we can test about the resulting `Status` struct? Not really, it turns out.

We *could* look at the `ChargePercent` field and see if it has the default value `0`, indicating it hasn’t been set. But the battery might really *be* 0% charged, in which case we’d get a bogus test failure for code that’s actually correct.

In any case, we don’t actually need to check the result at all. If `ParsePmsetOutput` doesn’t return error, then the

result is valid by definition. And we know that function works, because it has its *own* test.

We're not *testing* the parsing here, just using it to check that `pmset` returned something parseable. If this is indeed the case, then we feel we can't have messed up *too* badly in executing it. So we've done enough for now.

Here's the test, then:

```
//go:build integration

package battery_test

import (
    "testing"

    "github.com/bitfield/battery"
)

func TestGetPmsetOutput_CapturesCmdOutput(t
*testing.T) {
    t.Parallel()
    data, err := exec.Command("/usr/bin/pmset", "-
g", "ps").
        CombinedOutput()
    if err != nil {
        t.Skipf("unable to run 'pmset' command:
%v", err)
    }
    if !bytes.Contains(data,
[]byte("InternalBattery")) {
        t.Skip("no battery fitted")
    }
```

```
    text, err := battery.GetPmsetOutput()
    if err != nil {
        t.Fatal(err)
    }
    status, err := battery.ParsePmsetOutput(text)
    if err != nil {
        t.Fatal(err)
    }
    t.Logf("Charge: %d%%", status.ChargePercent)
}
```

(Listing battery/1)

Even if there's no error from parsing the `pmset` output, it would be nice to *know* what value is actually being parsed, so we pass that to `t.Logf`. The output from this won't be printed unless the test fails, or unless we run `go test` with the `-v` flag to enable verbose mode.

Running the command

If we supply a null implementation of `GetPmsetOutput`, we should be able to run this test and see it fail:

```
go test -tags=integration

--- FAIL: TestGetPmsetOutput_CapturesCmdOutput
(0.00s)
    battery_integration_test.go:19: failed to
        parse pmset output: ""
```

Capturing output

Good. Now, how do we implement `GetPmsetOutput`? You already know how to use `exec.Command` to create a command object, and `Run` to run it. How can we get its output as a Go value?

In our `ls` example, we set the `Stdout` field on the command to send its output somewhere. We could use something like a `strings.Builder` to capture it, but there's an easier way.

The command object has an `Output` method that runs the command and returns its output as a string. In fact, it'll be handy to get the standard error stream, too.

We can get both standard output and standard error in one string, by calling `CombinedOutput`. Here's what that looks like:

```
func GetPmsetOutput() (string, error) {
    data, err := exec.Command("/usr/bin/pmset -g
ps").
        CombinedOutput()
    if err != nil {
        return "", err
    }
    return string(data), nil
}
```

(Listing battery/1)

This should pass the test, so let's try:

```
go test -tags=integration
```

```
fork/exec /usr/bin/pmset -g ps: no such file or
directory
```

Oops. The command line is correct, as a single string, but that's not actually how you *pass* it to `exec.Command`.

It needs to be broken up into individual strings, one for each command-line argument:

```
data, err := exec.Command("/usr/bin/pmset", "-g",
"ps").
    CombinedOutput()
```

Good thing we wrote that integration test, or we might not have spotted that bug until we ran the program for real.

Let's try again:

```
go test -tags=integration -v

...
    battery_integration_test.go:21: Charge: 100%
--- PASS: TestGetPmsetOutput_CapturesCmdOutput
(0.02s)
```

Not only does this pass, but thanks to the `-v` flag, we can spy on the charge percentage value we passed to `t.Logf`:

```
Charge: 100%
```

There's more we could do with this package, of course, but it's a good start. Some programmer out there will be glad that we provided a straightforward way for them to get the battery status in Go code. Otherwise, they'd have to do what we did: work out what command to run, exactly how to invoke it, parse its output, and so on.

When to import a third-party package

It's worth saying a word or two about what to do when you have some problem that isn't trivially solved by the standard library, like the battery status example. This happens a lot, so how should we think about the right way to proceed?

The first thing would be to look for some existing third-party package (on `pkg.go.dev`, for example). If there's already a pure Go package that does exactly what we need, then that's wonderful—providing its licence allows us to use it, naturally.

It sometimes happens, though, that the best we can find is something that's *close* to what we want, but not a perfect match. We may need to do a fair bit of paperwork to get the package to solve our problem, and at that point it's worth questioning whether we should import the package at all.

If we can import a package to save writing ten lines of Go code, but it takes ten lines to *use* the package, then there's no real benefit from importing it. It might be better just to copy the piece of code that we need directly into our program.

If importing some package makes your program simpler, in other words, then import it. If it doesn't, don't.

We need to keep in mind that imports aren't free. Every extra dependency for your program complicates the code, slows the build process, and adds to the list of things that could break your build for one reason or another.

Maybe the package pushed a breaking upgrade, or introduced a critical bug, or maybe it was just deleted altogether. Every import is a potential point of failure.

Even if we're copying code rather than importing it, we're still introducing a foreign object into our program that could contain bugs. If it doesn't have its own tests that we can copy, we should cover it with tests.

We can't just take the attitude that if something is on GitHub or StackOverflow, it must be fine. That's very much

not the case, as a brief inspection of those sources will confirm.

Go adoption is growing rapidly, after all, so statistically most Go programmers must be relative beginners. By extension, most Go code we find in the wild won't be all that good.

That doesn't mean we should never use "found code": it just means we shouldn't assume it's necessarily a model of good Go style, or even that it works at all.

Going further

If you feel excited and inspired to write some interesting Go programs that execute external commands, or even if you don't, here's one suggestion.

- Write a command in Go that times how long it takes to execute some *other* command. For example, you could use it like this:

  ```
  howlong sleep 1
  (time: 1.007s)

  howlong backup.sh
  ...
  (time: 1h14m2s)
  ```

  Make sure the substantive functionality is part of an importable package, so other people can use your code in their own programs.

  If you get stuck, take a sneak peek at my suggested solution in listing `howlong/1`.

8. Shells

> *Individual people remain largely unprogrammable. They do things that the most imaginative programmers do not expect. And they want things that those programmers cannot anticipate.*
> —Scott Rosenberg, “Dreaming in Code”

In the previous chapter we saw how to run a command from Go using the exec package. It’s not too hard to write a very basic shell along these lines. What *is* a shell, actually?

Essentially, a shell reads lines from its input and executes them as commands. Well, we know how to do both of those things, so let’s have a little fun, and try to write the world’s least featureful shell.

A simple shell

A shell is a good example of the kind of program that seems intrinsically hard to test at first, because it’s *interactive*: it gets input from the user, and sends output back. How can you supply input in a test, or verify output? We’ve covered

some techniques in this book that can help solve these problems.

Defining some behaviour

But it’s easy to fall into the trap of trying to write a test before you’re completely clear in your own mind what the behaviour should be. Let’s establish that firmly by writing down some sensible assumptions about what a shell should do:

1. It should read a line of input and interpret it as a command line.
2. It should execute that command line and print the output, along with any errors.
3. If the input line is empty, the shell should do nothing.
4. If the input line consists only of the end-of-file character (EOF), the shell should exit (perhaps with a cheerful message).

This is specific enough that we can at least feel more comfortable about writing some tests. Not *completely* comfortable, perhaps, but it’s a start.

If you’re not sure how to test something, it’s usually because you don’t yet know exactly how it should behave. And that’s very often the case.

It’s not impossible that we could eventually write some test that covers all the specified behaviour, but that probably won’t be a useful test for helping us *implement* that behaviour. We’d like to break it up into smaller chunks, so that we can start to build up the machinery we need, piece by piece, test-first.

Identifying the first test

An important software design skill is *breaking the problem down* into units of behaviour that we can envisage testing and implementing independently.

Even though we're concerned with testing behaviour, not functions, it can be helpful to ask "Is there any piece of behaviour here that might be implemented as a self-contained function?"

If you can't see how to test the whole program, that is, you can ask instead "Well, what part of it *could* I test?"

Even if we can only find one thing, that will help us get started. Suppose we start with the first behaviour: interpreting an input line as a command.

We know that, whatever else it does, the program will need to be able to take some string representing a command line, and use the exec package to create a runnable `Cmd` object from it. That sounds like something we could write a test for.

GOAL: Write a test for a function `CmdFromString` that takes a string and returns the corresponding `*exec.Cmd` value. The function should be able to handle space-separated command-line arguments.

HINT: Again, even this simple-seeming test presents us with some interesting little problems. The first puzzle is, what's our `want`?

Suppose we have some input like `/bin/ls -l main.go`, as in our earlier example, and our function returns an appropriately-configured `*exec.Cmd`. How can we check that it's correct?

We'd like to compare the result of `CmdFromString` against some `*exec.Cmd` object that the test expects, but that might not be straightforward. What can you come up with?

Comparing the incomparable

SOLUTION: We might start by trying an idea something like this:

```
func TestCmdFromString_CreatesExpectedCmd(t
*testing.T) {
    t.Parallel()
    input := "/bin/ls -l main.go"
    want := exec.Command("/bin/ls", "-l",
"main.go")
    got := shell.CmdFromString(input)
    if want != got {
        t.Errorf("want %v, got %v", want, got)
    }
}
```

This looks reasonable, but doesn't work:

```
want /bin/ls -l main.go, got /bin/ls -l main.go
```

Even though they print the same, it's clear that `want` and `got` aren't comparing equal. What's going on?

We know they're pointers, so rather than the general-purpose `%v`, let's use the special `%p` verb for pointers, to see what they actually are:

```
t.Errorf("want %p, got %p", want, got)
```

This makes it clear why they're not equal, and why comparing pointers is meaningless:

```
want 0xc0000ec000, got 0xc0000ec160
```

They point to different locations in memory. What we really wanted was to compare the *structs they point to*, not the pointers themselves. Suppose we use the * operator to dereference both values:

```
if *want != *got {
```

But this doesn’t work either:

```
invalid operation: *want != *got (struct
containing []string
cannot be compared)
```

Just as it’s not meaningful to compare two pointers, it’s also not meaningful to compare two slices, because slices are, like pointers, just references to some memory location where the real data is stored. So comparing structs that contain slice fields isn’t allowed.

Fortunately, we don’t need to examine the whole `Cmd` struct, just its `Args` field. This is a slice, but we know how to use `cmp.Equal` to compare slices:

```
func TestCmdFromString_CreatesExpectedCmd(t
*testing.T) {
    t.Parallel()
    input := "/bin/ls -l main.go"
    want := []string{"/bin/ls", "-l", "main.go"}
    cmd := shell.CmdFromString(input)
    got := cmd.Args
    if !cmp.Equal(want, got) {
        t.Error(cmp.Diff(want, got))
    }
}
```

With a null implementation of `CmdFromString` that simply returns an empty command, we should now see a test failure, and so we do:

```
- "/bin/ls",
+ "",
- "-l",
- "main.go",
```

GOAL: Make this test pass by implementing `CmdFromString`.

Parsing user input

HINT: What does `CmdFromString` need to do? Given some string like `"/bin/ls -l main.go"`, we want it to return the result of:

```
exec.Command("/bin/ls", "-l", "main.go")
```

This is really just splitting up the string by spaces, isn't it? We can use `strings.Fields` to do that:

```
args := strings.Fields(input)
```

Now we have a slice of strings `args`, containing each element of the input line.

We know the pathname will be the first element: `args[0]`. The arguments will be all the remaining elements, `args[1:]`. By now, seeing a slice expression like that should give you pause for thought.

Any time you write a line of code, you should ask yourself "Under what circumstances would this not work?" Well, `args[1:]` wouldn't work here if `args` is empty.

So *can* the result of `strings.Fields` be an empty slice, in fact? Good question. One does not simply refer to a slice element that may or may not exist. Let's check the documentation for `strings.Fields`:

> *Fields splits the string s around each instance of one or more consecutive whitespace characters, as defined by unicode.IsSpace, returning a slice of substrings of s or an empty slice if s contains only whitespace.*
> (strings.Fields)

So `args` can be empty if `input` contains only spaces, or is empty itself. In that case, would it even make *sense* to construct a `Cmd`? No. It wouldn't do anything when executed.

Should we just return `nil` if the input is empty? No, because calling Run on a nil command would panic. So it looks like `CmdFromString` will need to return `error` in this case.

Let's update the test, then:

```
func TestCmdFromString_CreatesExpectedCmd(t
*testing.T) {
    t.Parallel()
    cmd, err := shell.CmdFromString("/bin/ls -l
main.go\n")
    if err != nil {
        t.Fatal(err)
    }
    want := []string{"/bin/ls", "-l", "main.go"}
    got := cmd.Args
    if !cmp.Equal(want, got) {
        t.Error(cmp.Diff(want, got))
    }
}
```

(Listing shell/1)

To make sure the function *does* return error on empty input, let's add a new test for that:

```
func TestCmdFromString_ErrorsOnEmptyInput(t
*testing.T) {
    t.Parallel()
    _, err := shell.CmdFromString("")
    if err == nil {
        t.Fatal("want error on empty input, got
nil")
    }
}
```

(Listing shell/1)

SOLUTION: To pass *this* test we'll have to check `len(args)`, and return an error if there isn't at least one element. Assuming there are enough elements, though, we can pass them to `exec.Command` and return the result:

```
func CmdFromString(input string) (*exec.Cmd,
error) {
    args := strings.Fields(input)
    if len(args) < 1 {
        return nil, errors.New("empty input")
    }
    return exec.Command(args[0], args[1:]...), nil
}
```

(Listing shell/1)

Why `args[1:]...`? We might instead have tried to write just:

```
return exec.Command(args[0], args[1:]), nil
```

which *looks* reasonable, but doesn't work:

```
cannot use args[1:] (value of type []string) as
string value
in argument to exec.Command
```

The result of `args[1:]` is a slice, but `exec.Command` doesn't *take* a slice: it's variadic, so it takes any number of *strings* instead. Therefore, we need to use the *unroll* operator "`...`" to pass the individual elements of `args[1:]` as separate arguments.

Prototyping

We have one component that we know we'll need, but it's still not entirely clear where it will fit into the program as a whole, or what else we might need. Let's do some prototyping so that we can work out what we even *want*, before trying to build it.

A pseudocode outline

When you're not sure about the overall structure of a program, it's a good idea to start by sketching out roughly what the shell program will look like, in so-called *pseudocode*:

```
repeat forever:
    read input line
    parse command
    execute command
    print output or error
```

We're not quite ready to write a test for this, because we need to flesh out a few more details. So let's use a

technique I call, slightly tongue-in-cheek, *main-driven development*.

We'll write some `main` function that does roughly what we want, because we're not *sure* yet exactly what we want. We need to see what it looks like in action, and probably tweak a few details.

If you're not sure what the test should test, try writing a throwaway version of the function first, and test it. Then throw it away. Let's use that tactic here.

GOAL: Write a `main.go` that implements the basic shell loop. Nothing fancy, just enough to get something on the screen, so we can poke and prod at it.

HINT: Well, we know how to read lines from standard input, using a `bufio.Scanner`: we did that back in the chapters about the line counter. Let's start with a loop like that, and then add in a call to `CmdFromString`, to turn the user's input into a command object that we can run.

We also know how to run commands and get their combined output, using `CombinedOutput`, as we did for the battery status command in the previous chapter. See if you can put these pieces together to create the beginnings of a shell.

Main-driven development

SOLUTION: Here's my first attempt at the program (in `cmd/shell/main.go`):

```
func main() {
    input := bufio.NewScanner(os.Stdin)
    for input.Scan() {
        line := input.Text()
```

```
        cmd, err := shell.CmdFromString(line)
        if err != nil {
            continue
        }
        out, err := cmd.CombinedOutput()
        if err != nil {
            fmt.Println("error:", err)
        }
        fmt.Printf("%s", out)
    }
}
```

First, we construct a `bufio.Scanner`, so that we can read the user's input line by line. As usual, we loop over it using `input.Scan`.

Next, we use our existing `CmdFromString` function to parse the command line and create a runnable `Cmd` object.

Finally, we use the `CombinedOutput` method to run the command *and* get its standard output and standard error as a string, which we print, along with any error.

It's pretty basic, but it should do *something*. Let's give it a try:

```
go run ./cmd/shell
```

Hmm, nothing. It's waiting for input, but if we didn't know that, we might just think it wasn't working.

Let's make a note to add in a suitable *prompt* character to make it more user-friendly.

What happens if we type in a command?

```
echo hello
```

```
hello
```

Nice! What about some command that doesn’t exist, though; will we see an error?

```
bogus
```

```
error: exec: "bogus": executable file not found in
$PATH
```

That looks sensible. What else shall we try?

Our “spec” says that empty lines should be ignored, so let’s see what happens when we enter one. Nothing. Excellent.

Finally, what happens when we close the input stream (which we can do by typing Ctrl-D)? We’d like the shell to shut down cleanly, ideally with a polite farewell message, but instead it just exits silently. That’s okay; we can work on the politeness a little.

Feedback from user testing

That was an excellent *user testing* session. It’s always a great idea to get your program in front of real users as early as possible in its development. It’s easier to make changes at this stage than later on, when we’ll have a lot more code to modify.

The moment you actually use the program, you’ll see a lot of problems with it. So don’t delay this moment until it’s too late to fix them. Instead, get to a runnable program as quickly as you can, and then run it. As soon as you become a *user* of your own software, you immediately spot usability

problems with it that you don’t see when you’re just looking at the code.

Let’s make some changes based on our user testing feedback, then. We need to handle empty lines correctly, and add a shutdown message:

```
func main() {
    fmt.Print("> ")
    input := bufio.NewScanner(os.Stdin)
    for input.Scan() {
        line := input.Text()
        cmd, err := shell.CmdFromString(line)
        if err != nil {
            continue
        }
        out, err := cmd.CombinedOutput()
        if err != nil {
            fmt.Println("error:", err)
        }
        fmt.Printf("%s", out)
        fmt.Print("\n> ")
    }
    fmt.Println("\nBe seeing you!")
}
```

(Listing shell/1)

This looks much better. Let’s run a command again:

```
go run ./cmd/shell
```

```
> ls
cmd
go.mod
go.sum
```

```
shell.go
shell_test.go
test.txtar
```

Nice! Let's try quitting:

```
> ^D
Be seeing you!
```

Indeed.

It's precisely this sort of minor, but important, tweaking that's much easier to do in `main`. By the time we've moved on to some large and fragile test that depends critically on every newline and space, changing it will be painful.

Actually, it's easiest to write this kind of *transcript* test when the program's more or less finished. By this point the exact details of input and output should be more or less set. If we should make some future change to the program that causes it to behave differently, the transcript test will catch it.

A stateful shell session

The next test will be interesting, because there are no more obvious little units like `CmdFromString` to extract. But it's also not straightforward to turn our proof-of-concept `main` function into something testable.

What would we like to write?

Let's approach the problem from both ends at once. Firstly, by writing the `main.go` we'd *like* to write. Secondly, we'll try to figure out how we *could* test the behaviour that we have.

The minimal `main` would be something like this, we imagine:

```go
package main

import (
	"os"

	"github.com/bitfield/shell"
)

func main() {
	os.Exit(shell.Main())
}
```

(Listing shell/2)

This is great, and we can test this CLI from a user's point of view with `testscript`, as we did with the line counter in earlier chapters. But such *end-to-end* tests, while useful, are always a bit limited. We can only interact with the program in the way that a user would: by "typing" input and looking at the response.

And when such a test fails, all it tells us is that *something*'s wrong (which is still valuable)—but not what. To know that, we'd need smaller, more focused tests that can call Go functions directly and check their behaviour.

So what would that look like for `Main`? Are there any potential problems with calling it from a Go test? Well, for one thing, it needs to read user "input". Reading from `os.Stdin`, while it works in the real program, won't do much in the test: since there *is* no input, the scanner will just wait until the test times out.

Similarly, the shell will try to print results, and we don't want them to go to `os.Stdout` when that happens in a test.

Tests should never produce any output on the terminal except failure messages. So we'll need a way to configure the shell with at least input and output streams, as we did for the line counter. And let's add a standard error stream, too, just for the fun of it.

In the line counter program, we created a special type of *object* to hold the configuration and current state of the counter, and we called it, straightforwardly, `Counter`.

What object would make sense?

So what's the object here? We could call it a `Shell`, but that doesn't seem *quite* right. What we're modelling is really the state of a specific user's interaction with the shell. They start the shell, see a prompt, type a command, see the result, perhaps repeat this for a few more commands, and finally exit. So let's call our object a `Session`.

If there's a session object, then that also implies some constructor function: `NewSession`, let's say. And since merely *constructing* the session shouldn't necessarily start it running, let's also plan on giving it a Run method to do this.

So what would `Main` do, given these facilities? It would first need to construct a new session with the right configuration, and then run it. That sounds pretty simple, so let's just go ahead and write it. If it's not correct, we'll soon find out.

```
func Main() int {
    session := NewSession(os.Stdin, os.Stdout,
os.Stderr)
    session.Run()
    return 0
}
```

(Listing shell/2)

We didn't bother with functional options here; you can add them if you want to. Indeed, we could have made the `NewSession` API simpler still, by having it take no arguments at all, and just letting users set the respective fields on the `Session` struct themselves, if they want to change the defaults.

There's no one right way to do this (*that's* something you won't hear from most software engineers). It really is up to you. My advice would be to make things as simple as they can possibly be while still being useful. It's easy to make a program more complicated later, if it turns out you need to: it's much more difficult to make it simpler.

So, let's start putting together the building blocks we need: namely, `NewSession` and Run. It makes sense to start with a test for `NewSession`, so over to you for this part.

GOAL: Write a test for NewSession.

Designing the `Session` object

HINT: The requirements for `NewSession` are that we create some kind of struct to represent the session, and it needs to be customisable with different input, output, and error streams.

It sounds like a straightforward `want` and `got` comparison should do the trick, but, as usual when writing tests, 90% of the problem is just figuring out what our `want` is!

What do you think?

SOLUTION: Here's my attempt:

```
func TestNewSession_CreatesExpectedSession(t
*testing.T) {
    t.Parallel()
    want := shell.Session{
        Stdin:  os.Stdin,
        Stdout: os.Stdout,
        Stderr: os.Stderr,
    }
    got := *shell.NewSession(os.Stdin, os.Stdout,
os.Stderr)
    if want != got {
        t.Errorf("want %#v, got %#v", want, got)
    }
}
```

(Listing shell/2)

It doesn’t actually matter what *values* we assign to `Stdin`, `Stdout`, and `Stderr`, so long as they satisfy the required interfaces. We may as well use the predefined os streams, then, since the test won’t actually cause anything to be read or written to them.

It might seem like this isn’t testing a lot: we just check that `NewSession` has indeed configured the session the way we asked for. But that’s okay. Don’t be afraid, in general, to write tests for functions that seem to have very simple behaviour. Those are the best kind of functions, after all.

We feel this function should be pretty easy to write, too, so let’s have a go.

GOAL: Implement `NewSession`.

HINT: We already did the heavy lifting in the test (which is one reason I like writing tests first). The struct definition practically writes itself!

Not quite, though, so you'll need to give it a little help. Once we have the `Session` type we need, then we can write `NewSession` to return an instance of it, configured as requested.

SOLUTION: Just as in our previous programs, we're looking at defining some kind of struct type to represent our session object. We need a `Session` struct with suitable fields (their types are implied by the test) and the constructor can return a pointer to a literal of this type.

So here's something that would pass the test:

```
type Session struct {
    Stdin          io.Reader
    Stdout, Stderr io.Writer
}

func NewSession(in io.Reader, out, errs io.Writer)
*Session {
    return &Session{
        Stdin:  in,
        Stdout: out,
        Stderr: errs,
    }
}
```

(Listing shell/2)

Running a session

Since merely *creating* a session doesn't actually do anything, once we have the `Session` object we'll need to give it a Run method, to start things happening.

Testing Run might be a little more tricky. Try this yourself first, and then we'll work through it together.

GOAL: Write a test for Run.

HINT: We'll need to *call* Run in the test, but hold on: once we've done that, it won't return until the session is complete and the "user" has "exited".

So we'll need to supply some simulated user input, representing one or more commands. When the session has consumed it all, Run should return.

We'll also need to check the *output* of the session, so let's construct a buffer to receive it in the same way that we've done in previous tests.

Testing the `Run` method

SOLUTION: Here's a first cut:

```
func TestRunProducesExpectedOutput(t *testing.T) {
    t.Parallel()
    in := strings.NewReader("echo hello\n\n")
    out := new(bytes.Buffer)
    session := shell.NewSession(in, out,
io.Discard)
    session.Run()
    want := "> hello\n> > \nBe seeing you!\n"
    got := out.String()
    if !cmp.Equal(want, got) {
        t.Error(cmp.Diff(want, got))
```

```
    }
}
```

(Listing shell/2)

Let’s break this down.

Because Run is supposed to have an interactive session with the user, and expects to get the user’s input from a reader, we construct a `strings.Reader` containing the command `echo hello` (plus a couple of newlines).

If all goes well, the shell should read the user’s command, run it, and send the output to the session’s configured output writer (in this case, the buffer `out`). We can read that output afterwards to see what happened.

So what’s our `want` here? This is a bit tricky. The first thing the shell should print is the prompt character, >, followed by a space. It will then try to read some input.

The first line of input will be `echo hello`. This should cause the `echo` command to print the string `hello`, followed by a newline, and we should then see the prompt character again.

The next line of input is empty, which should result in another prompt character and space. When the shell tries to read more input, it will hit the end of our `strings.Reader` buffer, which should trigger the goodbye message.

So our `want` string contains the sum of these expected outputs:

```
want := "> hello\n> > \nBe seeing you!\n"
```

You might expect to see the user's command here, too, but think about how most shells work: you type in your command, and it prints the output. It doesn't print your command *back* to you: you can see what you just typed, because it's already in your terminal window.

So the only *output* the shell should produce here is its prompts, the result of the fake user's `echo` command, and the goodbye message.

If we wanted to test a longer and more complicated session, it would be a good idea to compare it against a *golden file*, that is, a text file containing the exact output we want. We'll come back to golden files later in this book.

This test is correct, then, as it stands, but there's still a problem. Can you see what it is?

GOAL: Work out what is wrong (or at least not ideal) with this version of the `Run` test.

Dependencies on external commands

HINT: It's always a good idea to avoid external dependencies in tests, especially when they're not obvious. Can you see the hidden external dependency here?

It's `echo`. Running the test with this input will cause `Run` to actually execute the `echo hello` command, if it can… but *can* it?

It's common nowadays for automated CI systems to run tests in *scratch* containers. These don't contain the binaries for `cat`, `echo`, and so on. Indeed, a true scratch container contains nothing at all, hence "building from scratch".

It’s not safe to assume, then, when running Go tests, that *any* external program exists. Even if it does, we still don’t really want to run external programs in unit tests.

Running commands is relatively slow. Also, the commands could have unwanted side effects, and we already know tests shouldn’t have side effects.

But unless we supply some commands for the shell to execute, we won’t know if that behaviour really works. So what can we do?

We need some way that the shell can tell us it *understood* a line of input as a command, but without actually *running* it. We could write an integration test, as we did in the chapter on commands, but let’s think more creatively. What can you come up with?

A dry-run mode

SOLUTION: One way to do this is to provide some kind of *dry-run mode*. Many command-line tools have a mode like this: they will tell you what they *would* have done, without actually doing it. This can be very helpful for real users, as well as for testing.

Suppose we added a `bool` field on the session object named `DryRun`, for example. We can set that field in the test, and the Run method can check it before actually running any command.

If `DryRun` is set, what should happen instead of running the command? Well, for testing purposes it’ll be helpful if the shell simply prints the command it *would* have executed. That way, we can check it by looking at the output buffer.

GOAL: Update the test to use `DryRun`.

}

HINT: Well, we've imagined a field named `DryRun` on the session object, and we've said it sounds like a boolean. In the test, we want dry run mode to be active, so that commands *aren't* run for real, just echoed to the output instead.

We can't set a field on the session object before it exists, so the change to the test will need to come after the call to `NewSession`. If we set the field to `true`, and then call Run, what output should we expect?

It won't be `hello`, as it would have been if we'd really run the echo command. Instead, we should see the whole command (`echo hello`) printed to the session's output writer.

Can you figure it out?

SOLUTION: Here's the updated test:

```
func TestRunProducesExpectedOutput(t *testing.T) {
    t.Parallel()
    stdin := strings.NewReader("echo hello\n\n")
    stdout := new(bytes.Buffer)
    session := shell.NewSession(stdin, stdout,
io.Discard)
    session.DryRun = true
    session.Run()
    want := "> echo hello\n> > \nBe seeing you!\n"
    got := stdout.String()
    if !cmp.Equal(want, got) {
        t.Error(cmp.Diff(want, got))
    }
}
```

(Listing shell/3)

Note that we now expect to see the `echo hello` command itself echoed to the output. We could imagine that if the shell did more sophisticated things with its input, such as expanding wildcards or variable references, this would be a good way to see (and test) the results.

We *haven't* made `DryRun` an argument to `NewSession`, though; why not? The answer is that real users wouldn't normally want to enable dry-run mode, so they'd always have to supply a "mystery `false`":

```
shell.NewSession(in, out, err, false)
```

We already know this kind of thing is an API smell. Accordingly, we make the `DryRun` field exported, and set it in the test using a direct assignment.

Implementing `Run`

We're now in a position to implement the Run method: we can move the existing code from `main`, make the necessary changes, and add the dry-run check. Over to you!

GOAL: Implement Run.

HINT: We don't need to worry about writing any new code for looping, reading lines from the user and turning them into commands, or even executing them. We already have all that stuff in listing shell/1, and we can lift and shift it into Run.

Your job, effectively, is to turn our old `main` function into a Run method on the session object, and also add the `DryRun` functionality so that we can call it in test mode.

In other words, if `DryRun` is true, then instead of calling `CombinedOutput` to run the user's command, we should just print the input line to the output and continue.

As usual, the test will tell you when you've got it right, so see what you can do.

SOLUTION: Here's my version:

```
func (s *Session) Run() {
    fmt.Fprintf(s.Stdout, "> ")
    input := bufio.NewScanner(s.Stdin)
    for input.Scan() {
        line := input.Text()
        cmd, err := CmdFromString(line)
        if err != nil {
            fmt.Fprintf(s.Stdout, "> ")
            continue
        }
        if s.DryRun {
            fmt.Fprintf(s.Stdout, "%s\n> ", line)
            continue
        }
        output, err := cmd.CombinedOutput()
        if err != nil {
            fmt.Fprintln(s.Stderr, "error:", err)
        }
        fmt.Fprintf(s.Stdout, "%s> ", output)
    }
    fmt.Fprintln(s.Stdout, "\nBe seeing you!")
}
```

(Listing shell/3)

It's relatively long and complicated, but that's okay: we at least have working code, as defined by the tests. That gives us the luxury of time to refactor it if we want to, since we already solved the *user's* problem. As a bonus, we'll know straight away if our refactoring breaks anything: the tests will tell us!

What's still missing

It's time to run the updated program and try some more advanced things, to see if we have all the facilities we'd expect from a shell.

Globbing

One thing we'd expect to be able to do is to refer to a bunch of files at once, using a *glob* expression, involving a wildcard. For example:

```
go run ./cmd/shell

> cat *.go
error: exit status 1
cat: *.go: No such file or directory
```

That's weird: there are some files in this folder that should be matched by `*.go`. Why can't we see them?

Well, it turns out that globbing—expanding wildcards such as * into a list of filenames—isn't actually done by `cat` and other commands. This is something the shell itself would normally do, and it would then pass the list of matching files as individual arguments to `cat`.

But we haven't implemented that behaviour yet, so that's one for our wishlist of future user stories.

Redirection

Let’s try something else: using the familiar shell *redirect* feature. For example, we’d expect to be able to redirect the output of a command such as echo by using the > operator to send it to a file. Here goes:

```
> echo hello >tmp.txt
hello >tmp.txt
```

Well, that is actually what we asked for, but it’s not quite what we expected!

Our shell doesn’t know about redirection, so it just ran the echo command with the literal string `hello >tmp.txt`. echo did what it was told and echoed that string to the output. So we should also add redirection to our wishlist.

Piping

What about chaining commands together with the `|` (pipe) symbol? This should send the output of one command to the input of the next. For example:

```
> grep Test shell_test.go | wc -l
error: exit status 2
grep: |: No such file or directory
grep: wc: No such file or directory
shell_test.go
```

We wanted the output of the `grep` command—all the lines in `shell_test.go` matching the string `Test`—to be piped into the `wc` command, to count the matches.

What actually happened was that `grep` received three extra arguments: `|`, `wc` and `-l`. The latter is a valid flag to `grep`, coincidentally, but it interpreted `|` and `wc` as the names of files. Since no such files exist, `grep` complains and bails out.

Quoting

One other feature familiar from shells like `zsh` and `bash` is being able to *quote* filenames, or other arguments, that contain spaces. Without quoting, each word would be interpreted as a separate argument.

Let’s see what happens when we try to create a file with spaces in the name:

```
> touch "filename with spaces"
> ls
"filename
spaces"
with
```

Hmm. We actually created three separate files instead. Handling quoted arguments will need a little more work: another one for the wishlist.

Nor do we have any kind of line-editing, command history, string manipulation, environment variables, arithmetic, or a way to configure the prompt.

We’re also missing facilities for *job control*: for example, interrupting a running command, or running commands in *background*. But we have a decent proof of concept, and while it wouldn’t exactly be delightful to use this shell as our Unix environment, we *could* use it.

It’s not that we particularly need a new *shell*, anyway. But any tool that needs to interact with users could use similar machinery, couldn’t it? It’s quite common for command-line programs to have some kind of interactive, shell-like interface, and it makes for a simpler API.

Something else that standard shells offer is a *scripting* capability. Actually, we already have this in a limited form. Since our shell reads commands from standard input, we

could use *another* shell to pipe the contents of a script file into it.

However, users wouldn't be able to write very complicated programs: our simple shell doesn't have any *control flow* constructs such as `if` or `for`. So there's no way to write loops or logic at the moment.

That's okay, though, since we're focusing on a tool for interactively running commands, rather than implementing a scripting language. And while shell scripts are great for one-off tasks, they're not the best way to write robust and maintainable programs for use in production.

For that, we need a real programming language; fortunately, we have an excellent one to hand. In the next chapter, we'll see how to use Go to implement some of the kind of tasks that have traditionally been done with shell scripts.

Going further

It's fun to write shells, and while the world certainly doesn't need another shell, who cares what the world thinks? Programming is enjoyable for its own sake, to a certain kind of person.

If you're that kind of person, and given that you're reading this, I suspect you are, here's one idea you could work on:

- Add a *transcript* feature to the shell, so that it automatically saves the session to a file in the current directory named `transcript.txt`. The file should contain both the user's input and the shell's output, so that it more or less mirrors exactly what the user saw in the terminal.

This could be useful for audit purposes, for example, or for recording what you do so that you can review it later. A transcript is also a good way to document system procedures: you can show readers exactly what they should type and what they should expect to see in response.

You can see one possible way to do this in listing `shell/4`.

9. Pipelines

> *That was the big revelation to me when I was in graduate school—when I finally understood that the half page of code on the bottom of page 13 of the Lisp 1.5 manual was Lisp in itself. These were "Maxwell's Equations of Software"! This is the whole world of programming in a few lines that I can put my hand over.*
> —Alan Kay, "ACM Queue: A Conversation with Alan Kay"

So far in this book, we've focused on *applications*: software that directly solves user problems. But not all software engineering is about writing applications.

Developers also need *tooling*: programs and services to automate everyday tasks like configuring servers and containers, running builds and tests, deploying their applications, and so on.

And, of course, all of this relies on the underlying *system* software such as the OS kernel, networking stack, cluster orchestrators, and what-not. Somebody has to write (and maintain) that stuff, too.

A realistic operations task

How suitable is Go for such software, then? We know that Go has many advantages for general programming: it's fast, scalable, maintainable, and developers can be productive with it pretty quickly. What about using Go for tooling and systems code, though?

Well, in these domains, what really matters about software is its correctness and reliability. Can Go help here?

Well, maybe. Because Go is a compiled language with a strong type system, the compiler can do a lot to help us write correct programs. Go also surfaces runtime errors in a way that makes them hard (though, sadly, not impossible) to ignore. In turn, this helps us produce robust, reliable programs.

Matching and counting log lines

Let's think about a typical devops-type task such as counting the lines in a log file that match a certain string ("error", for example). What's a good way to write this code?

Most experienced Unix users would probably write some kind of shell *pipeline* to do this. A pipeline is a set of commands joined at their inputs and outputs, so data can flow between them, with the overall job control handled by the shell.

For example:

```
grep error log.txt | wc -l
```

The `grep` command filters its input, producing only lines that contain "error", while `wc -l` simply counts the number of lines it receives.

The overall effect of this pipeline is to print the number of lines in `log.txt` that contain "error". The shell makes it easy to compose individual commands like `grep` and `wc` in this way to do virtually anything you want.

A quick shell spell

But shell wizards can do much more. For example, suppose we have a web server access log to analyse. Here's a typical line from such a log: it contains the client's IP address, a timestamp, and various information about the request.

```
203.0.113.17 - - [30/Jun/2019:17:06:15 +0000] "GET /
HTTP/1.1"
    200 2028 "https://example.com/ "Mozilla/5.0..."
```

And suppose we want to find the top ten most frequent visitors to our website, by IP address. How could we do that?

Each line represents one request, so we'll need to count the lines for each IP address, sort them in descending order of frequency, and take the first ten.

A shell one-liner like this would do the job:

```
cut -d' ' -f 1 log.txt | sort | uniq -c
    | sort -rn | head
```

This extracts the IP address from the first column of each line (`cut`), counts the number of unique values (`uniq -c`), sorts them in descending numerical order (`sort -rn`), and shows the first ten (`head`).

Since virtually all Unix commands can accept data on standard input, and write results to standard output, some very complex pipelines that can be constructed this way, using only the shell's simple pipe operator.

The shell and its associated *userland* tools, it turns out, make this kind of work easy, and that's no surprise. The particular dialect understood by the shell is not a general-purpose programming language like Go; rather, it's a *domain-specific* language optimised for writing system tools and one-off *scripts*, like our visitor-counting example.

Solving the problem with Go

Let's try to write a similar kind of program in Go, then, just for fun, and we'll see how easy or hard it is compared to doing the same job with the shell's language.

GOAL: Write a Go command that reads a log file in this format, counts the number of requests made by each IP address, and prints a table of the top ten IP addresses by number of requests. Use the example log file in listing visitors/1 as input.

HINT: While the task is well-defined, this program is by no means easy to write in Go. It's certainly hard to make it as concise as the shell version, but don't worry about that. Just focus on getting the frequency data we need, by any means, fair or foul.

When it's not clear right away how to solve a problem, try breaking it down into smaller sub-problems first. For example, we might break down the tasks this way:

1. Open the file and scan it line by line.

2. Extract the IP address from the line.

3. Increment some counter for this IP address.

4. When done reading the file, sort the results by frequency.

5. Print the final frequency table.

Taking these tasks one at a time is easier psychologically, because we can focus on a comparatively small problem. As we check off more and more tasks, it also helps boost our confidence that we'll get to the end goal.

Try this approach and see how far you can get.

SOLUTION: Something like this might do the job:

```
package main

import (
    "bufio"
    "fmt"
    "log"
    "os"
    "sort"
    "strings"
)

func main() {
```

```
    f, err := os.Open("log.txt")
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    defer f.Close()
    input := bufio.NewScanner(f)
    uniques := map[string]int{}
    for input.Scan() {
        fields := strings.Fields(input.Text())
        if len(fields) > 0 {
            uniques[fields[0]]++
        }
    }
    type freq struct {
        addr  string
        count int
    }
    freqs := make([]freq, 0, len(uniques))
    for addr, count := range uniques {
        freqs = append(freqs, freq{addr, count})
    }
    sort.Slice(freqs, func(i, j int) bool {
        return freqs[i].count > freqs[j].count
    })
    fmt.Printf("%-16s%s\n", "Address", "Requests")
    for i, freq := range freqs {
        if i > 9 {
            break
        }
        fmt.Printf("%-16s%d\n", freq.addr, freq.count)
    }
}
```

(Listing visitors/1)

There are several problems with this program, not least that it's pretty complicated. You might like to test your code-reading skills by

figuring out how it works, but I'm by no means recommending it as a model of Go style.

It's just a quick, untested, hack, and that's partly the point. In devops work we're often required to solve problems quickly, rather than elegantly. The server could be on fire *now*, and we need to figure out which IP address is burning it down. It's much quicker and easier to do that with the shell, and the resulting program is much simpler.

So this isn't a very satisfactory result for us Go fans. If Go is so great, why doesn't it seem to be a good fit for this problem? What kind of code would we like to write instead?

In what language would this be easy?

Given the nature of the problem, we'd prefer to express the solution as a *pipeline*, just like the shell program. How could we express that in Go? What about something like this?

```
File("log.txt").Column(1).Freq().First(10).Stdout()
```

In other words, read the file `log.txt`, take its first column, sort by frequency, get the first ten results, and print them to standard output.

Not only is this extremely concise, it's arguably even clearer than the shell pipeline. So, can we make it real? Let's have a think.

Programs as pipelines

Throughout this book, we've used the "Zen mountaineering" technique of software design: write the code we *wish* we could write, and then derive the necessary architecture to make that code possible.

Well, then, how shall we tackle this problem as Zen mountaineers? We've done the first part already: we have a program that looks nice. So what else would we need to build in order for this program to work as written?

One notable thing about the program is that it seems to consist mainly of a series of *method calls*. For example:

```
... .Column(1).Freq() ...
```

`Column` is clearly a method on something here, as is `Freq`: the dot notation tells us that. But methods on what kind of type, exactly?

A fluent API

It makes sense to assume that each of these methods will form independent, reusable stages, and that we could put them together in any order to compose the operations we want. So what does that imply?

Well, `Column`, `Freq`, and friends would all have to be methods on the same type for that idea to work. Let's call it `Pipeline`, and assume it'll be some kind of struct. If every method on a pipeline also *returns* a pipeline, then we can chain together as many of these method calls as we want.

This pattern is sometimes called a *fluent* API, in the sense that data "flows" through the pipeline, with the output of one method forming the input to the next. It's just like the shell pipeline that we saw at the beginning of this chapter. This sounds promising!

A question occurs to us at this point, though: what happens when there's an error?

Errors in sequenced operations

Clearly pipeline stages *can* have errors. For example, `File("log.txt")` could fail because the file doesn't exist or we can't read it. Normally we'd have the method return an `error` result to deal with this.

But the `File` method can't return an error result here, because then we wouldn't be able to chain method calls together. If we're going to call `Column` on the result of `File`, then `File` can only return one value, not two.

If `File` can only return a pipeline, then what happens if that pipeline is invalid, and we then call some method on it like `Column`? We don't want to blow up the program with a panic. We'll need some way for `Column` to know if it actually has some valid data to read, or not.

To explain how this could work, let's think about a simpler program for a moment. Suppose you have some `io.Writer`, perhaps representing a file, and you need to write many short pieces of data to it.

Every individual write operation can potentially return an error, so you'd have to check each one in turn. Something like this:

```
func write(w io.Writer) error {
    metadata := []byte("hello\n")
    _, err := w.Write(metadata)
    if err != nil {
        return err
    }
    _, err = w.Write(metadata)
    if err != nil {
        return err
    }
    _, err = w.Write(metadata)
    if err != nil {
        return err
    }
    _, err = w.Write(metadata)
    if err != nil {
        return err
    }
    return nil
}
```

Repetition isn't a problem in itself, but is *this* repetition really necessary? Do we actually need to check the error from each individual `Write` operation? Or do we really only care about whether an error happened *somewhere* in the sequence?

We take exactly the same action on every error, and it's always to bail out and return the error. It seems like we should be able to do that just once, instead of after every write operation.

Could we just check the final value of `err`, then, after all the writes have finished? No, because it's possible that *some* write failed, but subsequent writes succeeded. The `err` variable would have been overwritten with `nil` by the time we check it.

It would *look* like everything was okay, except that we would have silently lost some of the user's data. And that's a problem. What can we do about this?

GOAL: Think of a way to solve this problem that eliminates the repeated `if err != nil` blocks. Implement your idea and see if it works.

HINT: If we're not going to check the error result after each `Write`, then we need to make it safe for `Write` to be called even when there's been a previous error. The only safe thing to do in such a situation is nothing, so let's just return.

But how can `Write` know that a *previous* call to `Write` encountered an error? Only by checking the value of some variable that persists across multiple calls to the function. But what?

We know that a global variable is always the wrong answer, whatever the question. So what if we stored the error result in some struct, and then made `Write` a *method* on that struct instead?

Now `Write` can look at the error field on its receiver, and if it's not `nil`, just return without doing anything. Nice.

Assuming there was no previous error, though, we can go ahead and do the write. *That* could fail, of course, and if that happens, we know what to do: store the resulting error on the receiver so that future calls to `Write` will see it and bail out.

Can you go ahead and implement this scheme?

An error-safe writer

SOLUTION: As we discussed, "remembering" the previous error status would be easy if `Write` were a method on some struct. And such a struct would also be the ideal place to store the destination writer, so that we don't have to keep passing it to `Write` again every time.

We could call this struct type an "error-safe writer": one that behaves normally until there's an error, at which point all subsequent writes are skipped. After the user's done with all their writes, they can check the error field on the struct to see if there was a problem at any point during the sequence.

Let's start with the struct type. How about something like this:

```
type safeWriter struct {
    w     io.Writer
    Error error
}
```

(Listing safewriter/1)

We can now add `Write` as a method on this new type. We know exactly what it needs to do, because we already worked it out: check the error, do the write, and save the error.

Here goes:

```
func (sw *safeWriter) Write(data []byte) {
    if sw.Error != nil {
        return
    }
    _, err := sw.w.Write(data)
    if err != nil {
        sw.Error = err
    }
}
```

(Listing safewriter/1)

Let's refactor our original `write` function to use the safe writer, removing the now-unnecessary error checks:

```
func write(w io.Writer) error {
    metadata := []byte("hello\n")
    sw := safeWriter{w: w}
    sw.Write(metadata)
    sw.Write(metadata)
    sw.Write(metadata)
    sw.Write(metadata)
    ...
    return sw.Error
}
```

(Listing safewriter/1)

Pretty sweet. Now that we know what to do, it's almost trivial to implement a "safe-whatever" when we need it. Some people complain that Go is a bad language because they end up writing `if err != nil` a lot, but really, that's up to them. It's not a requirement of the language, as we can see.

Putting the pieces together

We now know one way to eliminate errors from our pipeline, but we still have a bit more thinking to do. It's not really clear yet how the methods on `Pipeline` can actually work.

How does data flow from one method to another?

It looks like each method reads data from *somewhere*, but where? Let's work backwards from the end of the pipeline. Here's our example again:

```
File("log.txt").Column(1).Freq().First(10).Stdout()
```

The last method call in the chain is `Stdout`, which we suppose will write the contents of the pipe to standard output. What is "the contents of the pipe"? Logically, it must be the output of the *previous* stage: `First(10)`.

So what's that? Well, `First` outputs the first N lines of the contents of *its* pipe; that is, the output of the previous stage, `Freq`. And so on.

It looks like each method just *reads* from something in the `Pipeline` struct. But what? Well, `io.Reader` would make sense, wouldn't it?

Let's write some code. We'll start with something that looks relatively easy, like `Stdout`. Naturally, we'll start with a test.

GOAL: Write a test for `Stdout`.

Testing the end of the pipeline

HINT: There's something unusual about `Stdout`, compared to the other methods in the sample program: what is it? Interestingly, it's *not a pipeline stage*.

In other words, it doesn't return a `Pipeline` for further methods to be called on. Instead, it's the *end* of a pipeline. It reads the final contents of the pipe, after it's filtered through all the methods in the chain.

What happens to that data? Well, the name `Stdout` suggests that it should go to standard output. That's the behaviour we need to test, but it'll be easier to do that if the pipeline has some kind of "output writer" that we can configure.

So we'll construct a `Pipeline` with some data in it, set its `Output` to a buffer we prepared earlier, and call `Stdout`. We can then check the contents of the buffer to see if they're what we expected.

But what *do* we expect? We need a pipeline containing some text, so that there's something to write. How would we create such a pipeline?

What we'd *like* is some function that creates a pipeline *from* a given string: `FromString`, perhaps. Let's pretend such a function exists, and keep thinking. What else would the test need to do?

Well, we've said there's some `Error` field on the pipeline that enables the "safe writer" pattern. Actually, in this case it's a "safe reader", but the principle is exactly the same.

What should we expect about the final value of the `Error` field? We can assume that writes to our test buffer will succeed, so in that

case `Error` should be `nil`.

Finally, we can check the contents of the buffer against the string we started with. Sound reasonable? Have a go!

SOLUTION: Here's my version of the test:

```
func TestStdoutPrintsMessageToOutput(t *testing.T) {
    t.Parallel()
    want := "Hello, world\n"
    p := pipeline.FromString(want)
    buf := new(bytes.Buffer)
    p.Output = buf
    p.Stdout()
    if p.Error != nil {
        t.Fatal(p.Error)
    }
    got := buf.String()
    if !cmp.Equal(want, got) {
        t.Errorf("want %q, got %q", want, got)
    }
}
```

(Listing pipeline/1)

Things are starting to come into clearer focus. We're ready to start adding the necessary bits and pieces for the test to compile.

GOAL: Get this test compiling!

Breaking ground

HINT: First, we need to create a `pipeline` module, a `pipeline.go` declaring `package pipeline`, and so on.

We'll need a `Pipeline` struct type, with an `Error` field, and we also need an `Output` field so that we can attach it to our test buffer.

To construct the test pipeline, we'll need to provide a `FromString` function. We know the pipeline's methods will be pointer methods,

so it makes sense for `FromString` to return a pointer to `Pipeline`.

Finally, to run the test we'll have to supply at least a null implementation of `Stdout`.

SOLUTION: Here's what that could look like:

```
package pipeline

import "io"

type Pipeline struct {
    Output io.Writer
    Error  error
}

func FromString(s string) *Pipeline {
    return nil
}

func (p *Pipeline) Stdout() {}
```

This compiles, so we're ready to run the test:

```
--- FAIL: TestStdoutPrintsMessageToOutput (0.00s)
panic: runtime error: invalid memory address or nil
pointer
dereference [recovered]
panic: runtime error: invalid memory address or nil
pointer
dereference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x0
pc=0x112c228]
...
```

Whoops! Let's see which test line is panicking:

```
p.Output = buf
```

That makes sense: `FromString` currently just returns `nil`, so `p` is `nil`, and and we can't dereference a nil pointer. What do we need to add to prevent the test from panicking, then?

Not much, it turns out. All we need to do is have `FromString` return a pointer to an empty `Pipeline`, instead of `nil`:

```
func FromString(s string) *Pipeline {
    return &Pipeline{}
}
```

Now the test fails as expected:

```
want "Hello, world\n", got ""
```

See if you can fill in the missing code.

GOAL: Get this test passing.

Getting the test passing

HINT: In order for the test to pass, `buf` needs to contain the string `Hello, world\n`. So how would that string get into the buffer?

The answer is that `Stdout` must write it there: that is, it must write the data to the pipeline's `Output`. But how would it know what string to write?

Since `Stdout` takes no arguments, the only place the data could come from is some field on the `Pipeline`. And we said earlier that `io.Reader` would be a sensible type for this.

SOLUTION: Let's add a suitable field to store the reader, then:

```
type Pipeline struct {
    Reader io.Reader
    Output io.Writer
    Error  error
}
```

(Listing pipeline/1)

Next, we'll have the `FromString` function create a `strings.Reader` out of the supplied string:

```
func FromString(s string) *Pipeline {
    return &Pipeline{
        Reader: strings.NewReader(s),
    }
}
```

Now we're ready to implement `Stdout`. When you want to copy the contents of a reader to a writer, `io.Copy` is a neat solution:

```
func (p *Pipeline) Stdout() {
    io.Copy(p.Output, p.Reader)
}
```

(Listing pipeline/1)

This passes the test, but we're not quite done with `Stdout` yet: we haven't made it error-safe. See what you can do.

GOAL: Make `Stdout` error-safe.

Adding error safety

HINT: As usual, we'll start by adding a failing test. What should it look for? Well, if we have some pipeline whose `Error` field is set—we can set it ourselves to some non-`nil` value—then `Stdout` should do nothing!

Translating that into a test, we could say, suppose we have a pipeline created from some string (it doesn't matter what). If we then set an error on the pipeline object and call its `Stdout` object, it should produce no output—that is, the buffer it's writing to should be empty.

Updating `Stdout` to check the error and short-circuit is straightforward: it's just like our `safeWriter`'s `Write` method from earlier on.

Over to you, then. Get piping!

SOLUTION: We could write the test something like this:

```
func TestStdoutPrintsNothingOnError(t *testing.T) {
    t.Parallel()
    p := pipeline.FromString("Hello, world\n")
    p.Error = errors.New("oh no")
    buf := new(bytes.Buffer)
    p.Output = buf
    p.Stdout()
    got := buf.String()
    if got != "" {
        t.Errorf("want no output from Stdout after error,
\
            but got %q", got)
    }
}
```

(Listing pipeline/1)

This fails, not surprisingly, because we haven't done that bit yet:

```
want no output from Stdout after error, but got "Hello,
world\n"
```

Let's add the error-checking code:

```
func (p *Pipeline) Stdout() {
    if p.Error != nil {
        return
    }
    io.Copy(p.Output, p.Reader)
}
```

(Listing pipeline/1)

We now have all tests passing, so let's pause and reflect on the progress we've made. Just by reasoning backwards from the sample code, we've designed a package, its core struct type, a way to get input into the pipeline and output out of it, and we've deduced several important things about the way it needs to work.

In fact, we never really needed to solve any hard puzzles, did we? Each stage of the design followed logically from the previous one, more or less. We didn't need to come up with brilliant leaps of intuition at any point: instead, all we needed to do was *think clearly* and work methodically towards the goal.

If you find yourself with a hard problem, stop, and renegotiate an easier problem. In other words, if we're writing complicated code, something's gone wrong. We shouldn't congratulate ourselves on having written such a hairy program: we should instead be wondering what's wrong with our basic approach.

This point is well illustrated by reading certain parts of the standard library, for example. It all looks so simple and straightforward! We search in vain for brilliancies, but the code just plods onwards, step by obvious step, doing one simple thing after another. By the end we feel as though we must have missed something: maybe the *clever* code was in some other file.

But there is no clever code. Instead, the cleverness of the standard library, and other well-written programs, is at a higher level: in the design. Everything is very carefully and thoughtfully structured so that when you get down to the level of individual functions or statements, the code follows clearly and elegantly, even necessarily, from the design.

This simplicity is characteristic of really excellent programs, and it doesn't arise by accident. It was the goal all along. I call this *obviousness-oriented programming*, and it's the only kind of OOP that I think is worthwhile.

If you can't see any clever code, then you're looking at a clever design. Let's see if we can apply the same kind of thinking to the rest of our pipeline program.

Trying it out

As we saw in the previous chapter, it's always a good idea to get your code in front of users as soon as possible, so that they can tell

you it’s not what they want. Also we need to see that the code works correctly when it’s run *by* users, not just by tests.

Tests always fail to capture *something* important about the production environment: we just don’t know what, until we try the program out for real.

Hello, world

Let’s write a program that uses `pipeline` to do something. It doesn’t have to be anything fancy. We’ll try just printing a string to the standard output:

```
package main

import "github.com/bitfield/pipeline"

func main() {
    pipeline.FromString("hello, world\n").Stdout()
}
```

This is so simple, it can hardly go wrong, can it?

```
go run main.go

panic: runtime error: invalid memory address or nil
pointer
dereference [signal SIGSEGV: segmentation violation
code=0x1
addr=0x18 pc=0x105a8bd]

goroutine 1 [running]:
io.WriteString({0x0, 0x0}, {0x108322c, 0xc})
        /usr/local/go/src/io/io.go:314 +0x7d
...
```

The world strikes back

Whoops! Looks like I spoke too soon. Why would there be a panic calling `WriteString` on our output? Well, if the output was `nil`, for example.

Let's look again at where we create the Pipeline struct in FromString:

```
func FromString(s string) *Pipeline {
    return &Pipeline{
        Reader: strings.NewReader(s),
    }
}
```

Looks like we never set an Output on the pipeline at all. In the tests, we set it to a *bytes.Buffer, necessarily, since we want to capture it. But when we call FromString from our main.go, we get a pipeline with a nil output, hence the crash.

This is a good example of a bug that's hard to catch in tests: failing to set a default value for something that is overriden by a test! We could have written ever more elaborate tests for this, of course, but the quickest way to find this bug is simply to run a real program and watch it crash.

Setting defaults with a constructor

We could fix this bug by setting Output to os.Stdout in the FromString function, but let's be intelligent about it. Since we'll eventually want to create pipelines from several different sources, there will be multiple places in the code where we could forget to set a default Output writer.

Instead, let's write *one* piece of code to create a Pipeline, and get it right. Then we can use it in every function that creates a pipe source.

```
func New() *Pipeline {
    return &Pipeline{
        Output: os.Stdout,
    }
}
```

(Listing pipeline/1)

Great. Now we can rewrite FromString in terms of this new constructor function:

```
func FromString(s string) *Pipeline {
    p := New()
    p.Reader = strings.NewReader(s)
    return p
}
```

(Listing pipeline/1)

This should fix our panic:

```
go run main.go
```

```
hello, world
```

That's more like it. We now have a robust "new pipeline" function we can use throughout the package, and we don't need to worry about forgetting to set that `Output` field. Any time you find a bug caused by forgetting to do something, consider restructuring the code so that you *can't* forget to do it, because you don't have to remember.

Reading data from files

In the sample program, the pipeline needs to read text from a file, not a string. Since we already have a function `FromString`, let's call the corresponding function that creates a pipeline from a file `FromFile`, for symmetry. How could we implement it?

GOAL: Implement the `FromFile` function, test-first.

HINT: We already have a test that ensures the pipeline contains some specified text, so we can use that as a starting point.

The only difference here is that the text will come from a file, rather than a string. To check it, we won't need to call `Stdout`; we can read directly from the pipeline's `Reader` instead (using `io.ReadAll`, for example).

Since that can fail, we'll also need to set the error field on the pipeline in this case (and don't forget to test this behaviour too).

SOLUTION: Here's a test that looks reasonable. If we create a file in the `testdata` folder containing the expected text, then opening it with `FromFile` and reading the resulting pipeline should produce the same text we started with:

```
func TestFromFile_ReadsAllDataFromFile(t *testing.T) {
    t.Parallel()
    want := []byte("Hello, world\n")
    p := pipeline.FromFile("testdata/hello.txt")
    if p.Error != nil {
        t.Fatal(p.Error)
    }
    got, err := io.ReadAll(p.Reader)
    if err != nil {
        t.Fatal(err)
    }
    if !cmp.Equal(want, got) {
        t.Errorf("want %q, got %q", want, got)
    }
}
```

(Listing pipeline/1)

Clearly there can *be* an error opening a file, but the semantics of the `FromFile` function demand that it returns only `*Pipeline`, not `error`, as we've already seen.

Instead, the pipeline itself will hold any error in its `Error` field. Let's write the "invalid input" test now:

```
func TestFromFile_SetsErrorGivenNonexistentFile(t
*testing.T) {
    t.Parallel()
    p := pipeline.FromFile("doesnt-exist.txt")
    if p.Error == nil {
        t.Fatal("want error opening non-existent file,
got nil")
    }
}
```

(Listing pipeline/1)

Here’s a null implementation of `FromFile` to get the tests compiling:

```
func FromFile(pathname string) *Pipeline {
    return New()
}
```

Another pipeline explosion

What happens if we run the tests now? Alas, there’s a panic at this line:

```
got, err := io.ReadAll(p.Reader)
```

And that makes sense, because `p.Reader` was never set to anything, so it still has the default value for any interface type: that is, `nil`.

Is this the same kind of bug as we found with the user testing session earlier in this chapter when we forgot to set a default `Output` on the pipeline? Not really, because while there’s a sensible choice for a default output (`os.Stdout`), that doesn’t apply to *inputs*.

If the user tries to create a pipeline without setting its input, for example, we want them to be alerted to that fact right away. Defaulting to `os.Stdin` would just leave their program apparently hanging. In fact, it would be waiting for some input on the terminal, but there would be nothing to indicate that. It might take quite a while for the user to figure out what’s gone wrong.

So in this case we don’t want to add a default `Reader` to the constructor function `New`. That would hide program errors, whereas we want to expose them; and a panic is pretty hard to ignore.

Instead, while we’re stubbing out our `FromFile` function to validate the test, we’ll set a valid `Reader` on the pipeline that just doesn’t happen to contain anything:

```
p := New()
p.Reader = strings.NewReader("")
```

```
return p
```

This eliminates the panic:

```
--- FAIL: TestFromFile_SetsErrorGivenNonexistentFile
(0.00s)
    pipeline_test.go:48: want error opening non-existent
file,
    but got nil
--- FAIL: TestFromFile_ReadsAllDataFromFile (0.00s)
    pipeline_test.go:40: want "Hello, world\n", got ""
```

The failure of the “error” test makes sense, because we never actually set the pipeline’s error status. The failure of the “valid input” test also makes sense, because we know the pipeline doesn’t contain what it’s supposed to. We haven’t written any code to actually read the data yet.

We can now fill in the missing pieces:

```
func FromFile(pathname string) *Pipeline {
    f, err := os.Open(pathname)
    if err != nil {
        return &Pipeline{Error: err}
    }
    p := New()
    p.Reader = f
    return p
}
```

(Listing pipeline/1)

Normally we would want to defer a call to `f.Close` here, but that won’t work: the file needs to stay open. If we close it as soon as `FromFile` returns, then we won’t be able to read anything from it later. We’ll touch on a possible solution to this problem later in the chapter.

We’ve now implemented both ends of the pipeline, so to speak. We have some *sources* that put data in, `FromFile` and`FromString`. We

also have a *sink*, `Stdout`, that takes it out. Now, how could we add a method in the *middle* of a pipeline?

Filtering data

Most pipelines take data from a source, pass it through one or more *filters*, or intermediate stages, and send it to some sink. Let's try to write a filter, then, and see how it works out.

Extracting columns

The first filter method in our example program was `Column`, which extracts a specified column from the input.

For example, suppose we have some input lines consisting of whitespace-separated data:

```
1 2 3
1 2 3
1 2 3
```

Filtering this data through a pipeline stage like `Column(2)`, for example, should eliminate all but the second field of the input (that is, the second *column*, if we think of this as a table). The result:

```
2
2
2
```

That sounds like a test we could write. Let's try.

GOAL: Write a test, or tests, for `Column`.

HINT: We know how to create a new pipeline from a given string, and we can have the test call `Column(2)` on it, for example. What next? How do we check if `Column` did the right thing?

We could, as before, create a buffer, set it as the pipeline's output, and call `Stdout`, but that's tiresome. Since we'll often want the pipeline's output as a string in real programs, let's add a `String` sink method, too, so that we can use it in this test.

Once you've done that, writing the `Column` test becomes easier. We can create a pipeline containing some text in columns (separated by whitespace). We can then use `Column` to filter that data by column, and finally call `String` to check that we got the result we expect.

A `String` sink

SOLUTION: So, let's push the `Column` test onto our mental stack for a moment while we work on the tests for `String`. Something like this, for example:

```
func TestStringReturnsPipeContents(t *testing.T) {
    t.Parallel()
    want := "Hello, world\n"
    p := pipeline.FromString(want)
    got, err := p.String()
    if err != nil {
        t.Fatal(err)
    }
    if !cmp.Equal(want, got) {
        t.Errorf("want %q, got %q", want, got)
    }
}

func TestStringReturnsErrorWhenPipeErrorSet(t *testing.T)
{
    t.Parallel()
    p := pipeline.FromString("Hello, world\n")
    p.Error = errors.New("oh no")
    _, err := p.String()
    if err == nil {
        t.Error("want error from String when pipeline has
\
            error, but got nil")
    }
}
```

(Listing pipeline/1)

In other words, if the pipeline contains "Hello, world", then calling `String` should produce that same string, and no error. Alternatively, if the pipeline has an error, then `String` should return an empty string plus the error.

Great. That's not too hard to implement:

```
func (p *Pipeline) String() (string, error) {
    if p.Error != nil {
        return "", p.Error
    }
    data, err := io.ReadAll(p.Reader)
    if err != nil {
        return "", err
    }
    return string(data), nil
}
```

([Listing pipeline/1](#))

You might ask, why return `error` from a method like `String`, given that we already have an `Error` field on the pipeline? Why not just return a string, and let the user check `p.Error` manually?

Well, they might forget to check, and it's very idiomatic for Go functions to return "something and error" to remind them. We'll adapt that convention for our purposes and have the pipeline *as a whole* return something and error.

Testing `Column`

Now that we have an easy way to get the pipeline contents as a string, we can finish writing `TestColumn`:

```
func TestColumnSelectsColumn2of3(t *testing.T) {
    t.Parallel()
    input := "1 2 3\n1 2 3\n1 2 3\n"
    p := pipeline.FromString(input)
    want := "2\n2\n2\n"
    got, err := p.Column(2).String()
    if err != nil {
```

```
        t.Fatal(err)
    }
    if !cmp.Equal(want, got) {
        t.Error(cmp.Diff(want, got))
    }
}
```

([Listing pipeline/1](https://github.com/bitfield/tpg-tools2/tree/main/pipeline/1))

We create a pipeline with our example input, call `Column(2)` on it, and check that we get exactly the second column of input returned from `String` (and no error).

Speaking of errors, do we need a “what if there’s an error” test, too? We surely do, because the required behaviour of `Column` if the pipeline has an error is to do nothing.

Unless we explicitly check for a pipeline error, though, it will do *something*, and that’s wrong. Let’s add a new test, then:

```
func TestColumnProducesNothingWhenPipeErrorSet(t
*testing.T) {
    t.Parallel()
    p := pipeline.FromString("1 2 3\n")
    p.Error = errors.New("oh no")
    data, err := io.ReadAll(p.Column(1).Reader)
    if err != nil {
        t.Fatal(err)
    }
    if len(data) > 0 {
        t.Errorf("want no output from Column after error,
but \
            got %q", data)
    }
}
```

([Listing pipeline/1](https://github.com/bitfield/tpg-tools2/tree/main/pipeline/1))

There’s a little wrinkle here that needs further explanation. Why can’t we just call `String` on the pipeline to see its result?

Because `String` *itself* does nothing when the pipeline has an error. So `String` effectively hides the error-handling behaviour of any previous stages.

We need to check that `Column` has produced no output, though, so what can we do? We can read everything from the pipeline's `Reader`. This should return exactly nothing (and no error).

Testing that a function does nothing might seem weird, but sometimes that's the most important thing a function can do.

Are we done with testing `Column`? Well, not quite. There's *another* "invalid input" case to test.

Validating arguments

Since `Column` takes a numeric argument representing the column to cut, that *argument* could be invalid. It doesn't make sense to ask for `Column(0)` or `Column(-1)`, for example, so what should happen in that case?

The `Column` method itself can't return an error, as we know. So if we supply an invalid argument to `Column`, we should expect to find the pipeline's `Error` field to be set afterwards. And if that's so, then reading data from the pipe should produce nothing, whatever the input was.

Let's try:

```
func
TestColumnSetsErrorAndProducesNothingGivenInvalidArg(t
*testing.T) {
    t.Parallel()
    p := pipeline.FromString("1 2 3\n1 2 3\n1 2 3\n")
    p.Column(-1)
    if p.Error == nil {
        t.Error("want error on non-positive Column, but
got nil")
    }
    data, err := io.ReadAll(p.Column(1).Reader)
    if err != nil {
```

```
        t.Fatal(err)
    }
    if len(data) > 0 {
        t.Errorf("want no output from Column with invalid
col, but \
            got %q", data)
    }
}
```

(Listing pipeline/1)

To get these tests compiling, we'll need a null implementation of `Column`. To avoid panics, let's use `FromString` to create a pipeline containing just the string bogus:

```
func (p *Pipeline) Column(col int) *Pipeline {
    return FromString("bogus")
}
```

This looks good, and all three tests are now failing as expected:

```
TestColumnSelectsColumn2of3: string(
        -   "2\n2\n2\n",
        +   "bogus",
          )
TestColumnProducesNothingWhenPipeErrorSet: want no output
from
Column after error, but got "bogus"
TestColumnSetsErrorAndProducesNothingGivenInvalidArg:
want error
on non-positive Column, but got nil
```

Now it's over to you again to make these tests pass.

GOAL: Implement `Column`.

Implementing `Column`

HINT: Let's first deal with the error-handling behaviour. The tests require us to short-circuit and set the pipeline's `Error` if the column argument is zero or negative, and we can do that.

Secondly, if there's already an error on the pipeline, we should set the pipeline's reader to the empty string and return. That's straightforward.

The main behaviour of `Column`, though, is to extract the specified column from the input. How could we do that?

We can read the input line by line, and we know how to do that using `bufio.Scanner`. We also know we can get individual columns with `strings.Fields`. So what do we do with each column value once we've extracted it?

One idea is to write it to a buffer, which will then serve as the reader for the pipeline we return. Can you turn this scheme into working code?

SOLUTION: Here's my attempt:

```
func (p *Pipeline) Column(col int) *Pipeline {
    if p.Error != nil {
        p.Reader = strings.NewReader("")
        return p
    }
    if col < 1 {
        p.Error = fmt.Errorf("bad column %d: must be
positive",
            col)
        return p
    }
    result := new(bytes.Buffer)
    input := bufio.NewScanner(p.Reader)
    for input.Scan() {
        fields := strings.Fields(input.Text())
        if len(fields) < col {
            continue
        }
        fmt.Fprintln(result, fields[col-1])
    }
    return &Pipeline{
```

```
        Reader: result,
    }
}
```

(Listing pipeline/1)

The `script` package

If you implemented `Column` a different way, that's completely fine. We've said all through this book that, *provided the tests are correct*, the exact implementation you choose doesn't matter. If it behaves as it should, reads clearly, and is as simple as possible (but no simpler), then it's good enough to ship.

Was this a waste of time?

You might be objecting by now that we seem to have written a long and complicated package purely so that we could use it to write a short and simple program. *You've got some attitude, mister*. But isn't that exactly what good software engineering is about, actually: creating useful abstractions?

The plain-ol'-Go version of the access log program only did one very specialised job, so in that sense writing it was a waste of time. It's no help to us with any other kind of problems.

By contrast, writing our `pipeline` package was an excellent use of time. We can now use the facilities it provides to write not only the access log counter, but any number of other programs, in a very clear and simple way.

Writing a one-off program is an expense, but writing a reusable package is an *investment*. For example, if we implemented a few more sources (`Stdin`) and a few more filters (`First`, `Last`, `Match`, `Replace`), we'd be well on the way to being able to replace a good many shell scripts with Go programs.

For things the `pipeline` package doesn't yet do, we could add a filter method that sends the data through any external Unix command. That gives us some flexibility and enables a gradual migration from shell script to pure Go program.

Introducing `script`

There is (not by coincidence) a Go package that implements the ideas described in this chapter, along with many others:

- https://github.com/bitfield/script

`script` is designed precisely to make it easy to write Go programs that chain together operations into a pipeline, in the same way that shell scripts do. It works in more or less the way we've discussed in this chapter, with a few extra bells and whistles.

For example, `script` pipelines automatically close their input files after reading, in the same sort of way that we discussed in the chapter on arguments.

You can use `script` to construct the sort of simple one-off pipelines that would otherwise require the shell, or special-purpose tools that need to do text filtering.

Some simple one-liners

Here's a simple version of the Unix `cat` command, for example. Like `cat`, it concatenates all the files you give it as arguments, and sends the result to its output:

```go
package main

import (
    "fmt"
    "os"

    "github.com/bitfield/script"
)

func main() {
    script.Args().Concat().Stdout()
}
```

Well, that was easy! We can also use `script` to write a basic version of the `grep` command:

```
script.Stdin().Match(os.Args[1]).Stdout()
```

Or something like echo, for example:

```
script.Args().Join().Stdout()
```

What about `ls`? Listing files is easy, but let’s do a little more. We’ll use a regular expression to suppress “dotfiles”: files whose names begin with `.`, such as `.zshrc`.

```
dotFiles := regexp.MustCompile(`^\.`)
script.ListFiles(".").RejectRegexp(dotFiles).Stdout()
```

There’s a very useful Unix command `xargs`, which lets us execute a given command repeatedly, once for each line of input. We can do something similar to this using `script`’s ExecForEach method:

```
script.ListFiles("*").ExecForEach("echo {{.}}").Stdout()
```

For each filename produced by `ListFiles("*")`, this will run the command echo with that filename as its argument. Because `ExecForEach` takes a Go template string, you can use it to construct some fairly complex commands.

More sophisticated programs

As you’d expect, `script` also lets us concisely express the kind of programs we’ve considered in this chapter, such as the access log analyser:

```
script.Stdin().Column(1).Freq().First(10).Stdout()
```

We’re not limited to getting data only from files or commands. We can make HTTP requests too:

```
script.Get("https://wttr.in/London?format=3").Stdout()
// Output:
// London: Sunny +13°C
```

That’s great for simple GET requests, but suppose we want to *send* some data in the body of a POST request, for example:

```
script.Echo(data).Post(URL).Stdout()
```

If, as is common, the data we get from an HTTP request is in JSON format, we can use JQ queries to interrogate it:

```
data, err := script.Do(req).JQ(".[0] | {message:
.commit.message, \
    name: .commit.committer.name}").String()
```

Concurrent pipeline stages

We can also run external programs and get their output:

```
script.Exec("ping 127.0.0.1").Stdout()
```

Note that Exec runs the command concurrently: it doesn't wait for the command to complete before returning any output. That's good, because this `ping` command will run forever (or until we get bored).

Instead, when we read from the pipe using `Stdout`, we see each line of output as it's produced:

```
PING 127.0.0.1 (127.0.0.1): 56 data bytes
64 bytes from 127.0.0.1: icmp_seq=0 ttl=64 time=0.056 ms
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.054 ms
...
```

Custom filter functions

If there doesn't happen to be a built-in `script` method that does what we want, we can just write our own, using `Filter`:

```
script.Echo("hello world").Filter(func (r io.Reader, w
io.Writer) error {
    n, err := io.Copy(w, r)
    fmt.Fprintf(w, "\nfiltered %d bytes\n", n)
    return err
}).Stdout()
// Output:
// hello world
// filtered 11 bytes
```

The function we supply to `Filter` takes just two parameters: a reader to read from, and a writer to write to. The reader reads the previous stages of the pipe, as you might expect, and anything written to the writer goes to the *next* stage of the pipe.

If our filter function returns some error, then, just as with the `Do` example, the pipe's error status is set, and subsequent stages become a no-op (that is, they short-circuit without doing anything).

Filters run concurrently, so the pipeline can start producing output before the input has been fully read, as it did in the `ping` example. In fact, most built-in pipe methods, including `Exec`, are implemented *using* `Filter`.

The aim of software engineering, as we've discussed, is not merely to write programs to solve specific problems, but to create a new kind of "language" in which it's easy and natural to solve a general *class* of problems. It makes sense, then, that `script` should largely be written in itself.

Solving problems

Let's apply the `script` package to some of the problems we've already solved together in this book, and see if it helps at all. What about the line counter, for example? Could we shorten that a little?

```
lines, err := script.Args().Concat().CountLines()
```

Well, this isn't bad. Of course the earlier program that we developed together is much better. It has a few more facilities, such as counting words or bytes, for example, and is generally more helpful and user-friendly.

But if we just want to quickly count some lines in a bunch of files, this does the job, and we can get it working with very little time and effort. Which is the point, of course.

The Go file finder is a lot smaller this way, too:

```
goFiles := regexp.MustCompile(".go$")
script.FindFiles(".").MatchRegexp(goFiles).Stdout()
```

And, of course, we already know what the visitor-counting program looks like:

```
script.File("log.txt").Column(1).Freq().First(10).Stdout()
```

You get the idea. We wouldn't necessarily want to write all our important tools this way, but `script` is useful for quick-and-dirty little scripts that we might otherwise write as shell one-liners.

It also gives us some handy convenience methods that let us write more sophisticated systems tools in Go, without quite so much boilerplate required for straightforward things like reading a file and matching text.

The landscape of simple programs

The real takeaway of this chapter, then, is not that you should use the `script` package for writing devops tools and shell-like scripts, though naturally you're welcome to do so if you like.

The point is rather that when plain Go doesn't provide a convenient way to solve some problem, you yourself can use it to implement a "language" that does.

In this case, we used Go to provide the language of Unix-style pipelines. But we could have chosen any architecture we wanted to suit the problem.

If Go doesn't already provide the tool you need, use Go to build that tool, then use it. A great way to discover such an architecture is to write the easy, simple code that you'd *like* to be able to write, and then reason backwards from it to the program architecture that makes it possible.

Going further

If you found this chapter easy going, try this extra-credit challenge:

- Use the `script` package to write a program that counts the number of non-blank lines of Go code in a project. It should recursively find all Go files in the tree rooted at the current

directory, count their lines (ignoring any blank lines), and report the final total.

For example, you might run it something like this:

```
loc
```

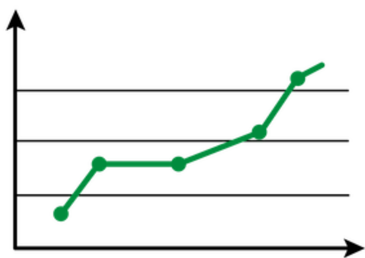
```
You've written 719 lines of Go in this project. Nice work!
```

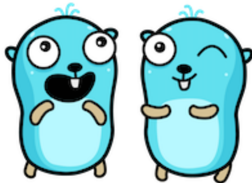
You should find that `script` has the necessary facilities to do this calculation in a single pipeline. Alternatively, you could extend the `pipeline` package developed in this chapter so that you can use it to implement this program.

If you’d like some inspiration, have a look at the suggested implementation in listing `loc/1`.

10. Data

A parser for things
Is a function from strings
To lists of pairs
Of things and strings
—Graham Hutton, “Programming in Haskell”





All programs are about data, in some sense. Some of the data in our programs is transient: it only exists while the program is running (sometimes not even that long). But we often need *persistent* data, too: data that stays around even after the program has exited.

For example, a to-do list app that couldn’t save its data wouldn’t really be much use. You’d have to keep it running all the time, and if it ever crashed, or your computer rebooted, you’d lose all your reminders. (Some people might consider this a feature, not a bug.)

Marshalling

So we need to be able to send data from a program out into the world in some way, perhaps into a disk file, or some cloud storage, or a relational database, or some kind of service that consumes it.

We need a way of putting our data into a format that can exist outside our program. Let's call this process *marshalling*, and the inverse operation *unmarshalling*.

We could marshal Go data into many formats: plain text, base64-encoded text, SQL results or queries, TCP/IP packets, and so on. Whatever the target format, the idea is basically the same.

Serialisation

The simplest imaginable way to format data for transmission is as a stream of bytes. This kind of marshalling is called *serialisation*, and we've already seen in this book that byte-stream abstractions can be powerful: `io.Reader`, for example.

So how should we represent Go values as bytes? The answer is that it doesn't matter, as long as both the producer and the consumer can agree on how to do it.

If we're serialising data to be read back into our own program later, or if we're sending it to another Go program, we can use an efficient "Go-aware" byte format. The standard library provides such an encoding, called `gob`.

The `gob` package

The `encoding/gob` package can serialise most kinds of Go values to bytes, with a few exceptions. Serialising a function or a channel value isn't allowed, but that wouldn't really make sense anyway.

A file-based data store

Let’s play with gob by using it to store some arbitrary data in a file. Suppose we write a package implementing a little key-value store, for example, that can persist its data to disk.

What kind of API would make sense to users of such a package? What would they do first?

Well, they’d need to be able to *open* a store whose data lives in a particular file, so we could write that something like this:

```
s, err := kv.OpenStore("data.bin")
```

Sounds reasonable, doesn’t it? There could *be* an error, so `OpenStore` should return it, and if there isn’t, we should also get an object representing the store.

What if the file doesn’t exist? Well, that’s bound to be the case when we create the store for the very first time, so this doesn’t seem like an error situation. Instead, we can just create and return a new empty store, ready for use. We needn’t even write anything to disk, since there’s nothing to write yet.

Testing data persistence

To test `OpenStore`, we could open a store file we prepared earlier, and check that the store contains exactly the data that we put in it to start with. This sounds great! But how do we create such a file?

We could do it manually, by figuring out the gob encoding format, typing bytes into a hex editor, and so on, but that sounds like hard work. Thinking ahead, we’ll need *saving*

data to be part of the API, too. Once we can do that, it'll be easy to create a suitable file for testing `OpenStore`.

Suppose there were some Save method on the store, then, that marshalled its data via gob to a disk file. Then, when we wanted to make sure the store's contents are saved, we could write simply:

```
err := s.Save()
```

We shouldn't need to pass the file path to Save: after all, presumably we already supplied it to `OpenStore`. It sounds as though the store should remember its own path, and be prepared to save data to it on demand.

Setters and getters

Fair enough. But we can't test much by saving and loading empty files: we need to get some data into the store somehow. We've said it's a key-value store, so we could imagine writing something like:

```
s.Set("key", "value")
v, ok := s.Get("key")
fmt.Println(v, ok)
// Output:
// value true
```

Of course, users might want to store and retrieve data of any type, but let's stick to strings for the time being. Solve the simpler problem first.

And, if you think about it, this "setting and getting" behaviour is independent of any disk files. It can happen purely in memory. And, since we'll need it to test disk operations later, let's build this part first.

GOAL: Write tests for sensible `Set` and `Get` behaviour, on a store created by `OpenStore` with a dummy path (we needn't actually save or load any data for these tests).

A sensible key-value API

HINT: There are several behaviours here, so we'll need a few different tests. Maybe something like:

1. When we query a key that doesn't exist, `Get` returns the empty string and `false`.

2. When we query a key that *does* exist, `Get` should return its associated value, along with `true`.

3. When we `Set` a key to a given value, we should be able to get that value back again by querying the key with `Get`.

4. If a key already exists with a certain value, but we use `Set` to associate it with a new value, then subsequent calls to `Get` should return the updated value.

The easiest of these tests to write is probably number 1, because it doesn't need the store to contain any data. Fine. Let's start there.

First, we'll need a store. Let's assume that the `store` type itself is not exported, so we can't just create a literal of that struct outside its home package.

This is probably a good thing, since a `store` with no path wouldn't be able to save its data. To prevent this situation arising, we'll mandate that the only way to get a store is to call `OpenStore`.

How about something like this, then?

```
func TestGetReturnsNotOKIfKeyDoesNotExist(t
*testing.T) {
    t.Parallel()
    s, err := kv.OpenStore("dummy path")
    if err != nil {
        t.Fatal(err)
    }
    _, ok := s.Get("key")
    if ok {
        t.Fatal("unexpected ok")
    }
}
```

(Listing kv/1)

Can you go ahead and write the other tests?

Testing the key-value machinery

SOLUTION: It’s surprising how much design work we end up doing, just by writing these tests, isn’t it? Even if we threw the tests away later, it would still have been worth writing them, just because it encouraged us to think about the API from the user’s point of view.

Let’s copy the test we have and use it as the basis for a new test for case 2, getting a key that *does* exist.

This time, we’ll need to *set* the key to something before querying it. Here’s a version that could work:

```
func TestGetReturnsValueAndOKIfKeyExists(t
*testing.T) {
    t.Parallel()
    s, err := kv.OpenStore("dummy path")
    if err != nil {
```

```
        t.Fatal(err)
    }
    s.Set("key", "value")
    v, ok := s.Get("key")
    if !ok {
        t.Fatal("not ok")
    }
    if v != "value" {
        t.Errorf("want 'value', got %q", v)
    }
}
```

(Listing kv/1)

We make progress! What about case 3, setting a key that doesn't exist? Well, we sort of already tested that just now, by using `Set` in the test for `Get`.

If `Set` didn't do anything, then the `Get` test could hardly pass. And if we were to write a separate test for `Set`, then it would look more or less exactly like the one we just wrote. That seems silly, so let's skip it.

The final test case is that `Set` overwrites any existing value with a new one. That seems straightforward:

```
func TestSetUpdatesExistingKeyToNewValue(t
*testing.T) {
    t.Parallel()
    s, err := kv.OpenStore("dummy path")
    if err != nil {
        t.Fatal(err)
    }
    s.Set("key", "original")
    s.Set("key", "updated")
```

```go
    v, ok := s.Get("key")
    if !ok {
        t.Fatal("key not found")
    }
    if v != "updated" {
        t.Errorf("want 'updated', got %q", v)
    }
}
```

(Listing kv/1)

Before we even worry about saving and loading, then, let's get this basic key-value machinery working.

GOAL: Get these tests passing.

Implementing the store

HINT: We know we need some `store` struct type, and we know it has to have some field that can hold the key-value data. A map sounds logical, doesn't it? Specifically, a `map[string]string` would do exactly what we need.

`OpenStore` will need to at least initialise this to an empty map, though it doesn't need to do anything else to pass the tests we have right now.

`Set` and `Get` are both easy to write if the store contains a map, since we can use the built-in map operations:

```go
s.data[key] = value
...
v, ok := s.data[key]
return v, ok
```

Can you see what to do?

SOLUTION: Let's start with the `store` struct itself. It doesn't yet need to hold anything except the key-value data:

```
type store struct {
    data map[string]string
}
```

(Listing kv/1)

Here's `OpenStore`, then, whose only job (so far) is to initialise the `data` map:

```
func OpenStore(path string) (*store, error) {
    return &store{
        data: map[string]string{},
    }, nil
}
```

(Listing kv/1)

And here's `Set` and `Get`:

```
func (s *store) Set(k, v string) {
    s.data[k] = v
}


func (s store) Get(k string) (string, bool) {
    v, ok := s.data[k]
    return v, ok
}
```

(Listing kv/1)

It's not complicated, but it's wise to get this stuff thoroughly ironed out before trying to do anything fancy like serialising the data to gob. Well done! So, what's the next step?

Adding persistence

Well, to make this a *persistent* data store we'll need to add a Save method to the store, and we'll also need to extend `OpenStore` so that it actually reads this saved data, if any.

Just as with `Set` and `Get`, these operations effectively test each other, so that helps. See what you can do:

GOAL: Write tests for the saving and loading behaviour, and make them pass.

HINT: We could imagine writing a test that creates a store, puts some data in it, saves it, and then checks the contents of the resulting file against what it should be.

That's not wrong, though it might be a little awkward to write, because we don't know what order the map keys will be in, for example. The real problem with a test like this is that, fundamentally, *we don't care about the bytes on disk*.

What are we really testing here?

In other words, what's the important behaviour of Save that we want to test? Not that it produces a specific sequence of bytes for given data. What really matters here is that, whatever bytes it produces, *we can read them back* and recover our original data.

After all, that's why we were doing this: to be able to *persist* the data. The exact *wire format*—the specific bytes—don't matter, and indeed we could have used any serialisation scheme we wanted (JSON, for example).

If we can put some data into a store, save it, open a *new* store from the same file, and get back the data we started with, then we'll be happy.

Even if we wrote a golden-file test for Save, we'd still have to test the loading behaviour of `OpenStore` too, so why not combine the two into a single test?

See what you can do.

An end-to-end persistence test

SOLUTION: Here's the test, then:

```
func TestSaveSavesDataPersistently(t *testing.T) {
    t.Parallel()
    path := t.TempDir() + "/kvtest.store"
    s, err := kv.OpenStore(path)
    if err != nil {
        t.Fatal(err)
    }
    s.Set("A", "1")
    s.Set("B", "2")
    s.Set("C", "3")
    err = s.Save()
    if err != nil {
        t.Fatal(err)
    }
    s2, err := kv.OpenStore(path)
    if err != nil {
        t.Fatal(err)
    }
    if v, _ := s2.Get("A"); v != "1" {
        t.Fatalf("want A=1, got A=%s", v)
    }
    if v, _ := s2.Get("B"); v != "2" {
        t.Fatalf("want B=2, got B=%s", v)
    }
    if v, _ := s2.Get("C"); v != "3" {
```

```
        t.Fatalf("want C=3, got C=%s", v)
    }
}
```

(Listing kv/1)

We open a store at a temporary path, put three key-value pairs into it, and save it. Then we open that path as a new store and check what's in it. Nice and simple.

It might seem a bit basic, but really, it's hard to imagine serious bugs in Save or `OpenStore` that wouldn't be caught here. If Save didn't write anything, or missed out some keys, or mixed up different keys and values, then we'd know because the `Get` checks would fail.

Similarly, if `OpenStore` garbled the data on loading somehow, then `Get` would return different results. So this *end-to-end* test is quite good value.

Of course, if it ever fails, it'll tell us there's a bug in either Save or `OpenStore` (or both), but not which. If this bothers you, by all means write golden-file tests to check the serialisation and deserialisation code separately.

Saving and loading

Let's go ahead and write Save:

```
func (s store) Save() error {
    f, err := os.Create(s.path)
    if err != nil {
        return err
    }
    defer f.Close()
    return gob.NewEncoder(f).Encode(s.data)
}
```

([Listing kv/1](#))

It’s pleasantly easy to use gob. All we need to do is create a `NewEncoder` on some writer (the file `f`, in this case), and pass our data to its Encode method. It returns an error, which we simply pass back to the caller without comment.

What about `OpenStore`? Right now, all it does is create a new store with an empty map, and that store has no way of remembering its disk path. We’ll need to do a little more:

```
type store struct {
    path string
    data map[string]string
}

func OpenStore(path string) (*store, error) {
    s := &store{
        path: path,
        data: map[string]string{},
    }
    f, err := os.Open(path)
    if errors.Is(err, fs.ErrNotExist) {
        return s, nil
    }
    if err != nil {
        return nil, err
    }
    defer f.Close()
    err = gob.NewDecoder(f).Decode(&s.data)
    if err != nil {
        return nil, err
    }
    return s, nil
}
```

(Listing kv/1)

Because the file at `path` may contain data, we should try to open it. If this fails with a `fs.ErrNotExist` error, then there's no such file, but that's okay. It just means we're creating a new store that's never been saved before, so we can return it as-is.

If the file *does* exist, and there are no other errors opening it, we use gob to decode the disk bytes into `s.data`, and return the populated store.

This passes the test, which is encouraging. So are there any important bits we haven't covered? What don't we test?

Tightening up the tests

Well, we don't test that `OpenStore` returns an error if the store file *exists*, but can't be read. Why would that be? One possibility is that we don't have permission to read it.

So let's create a dummy file and set its permissions to `0o000`, meaning "no permissions for anyone". Trying to open that should fail:

```
func TestOpenStore_ErrorsWhenPathUnreadable(t
*testing.T) {
    t.Parallel()
    path := t.TempDir() + "/unreadable.store"
    if _, err := os.Create(path); err != nil {
        t.Fatal(err)
    }
    if err := os.Chmod(path, 0o000); err != nil {
        t.Fatal(err)
    }
    _, err := kv.OpenStore(path)
```

```
    if err == nil {
        t.Fatal("no error")
    }
}
```

(Listing kv/1)

What else? Well, we also don't test that `OpenStore` returns an error when gob can't decode the file's contents. That's easily arranged:

```
func TestOpenStore_ReturnsErrorOnInvalidData(t
*testing.T) {
    t.Parallel()
    _, err :=
kv.OpenStore("testdata/invalid.store")
    if err == nil {
        t.Fatal("no error")
    }
}
```

(Listing kv/1)

What would invalid data look like? Well, an empty file would be pretty invalid:

```
touch testdata/invalid.store
```

There's only one remaining loophole in our tests: we're not testing that Save returns an error when it can't save the data. Hold my beer!

```
func TestSaveErrorsWhenPathUnwritable(t
*testing.T) {
    t.Parallel()
```

```
    s, err :=
kv.OpenStore("bogus/unwritable.store")
    if err != nil {
        t.Fatal(err)
    }
    err = s.Save()
    if err == nil {
        t.Fatal("no error")
    }
}
```

(Listing kv/1)

In this case, the path is unwritable because the parent directory bogus doesn't exist (we assume). Trying to save the store to it should result in something like "no such file or directory".

Good work! We have a fully-functional persistent key-value store. And, since it doesn't care *how* the data gets persisted, you could use any kind of encoding or serialisation format you like. We just picked gob for this because it's fun and useful to know about.

JSON: a text data format

Let's turn to another kind of byte encoding, then. JSON is a widely-used format, so as working programmers we'll need to know how to read and write JSON-encoded data with Go programs.

The `json` package

The standard library API for JSON handling is very similar to that for gob:

```
import "encoding/json"
...
err := json.NewEncoder(w).Encode(x)
...
err = json.NewDecoder(r).Decode(&x)
```

JSON data is usually represented by UTF-8-encoded text, meaning that it's human-readable (for some values of "human"):

```
{
    "john": {
        "age": 29,
        "hobbies": [
            "physics",
            "reading"
        ]
    }
}
```

What's the point of serialising to text, rather than a more space-efficient binary format such as gob? Well, it's useful to be able to read the encoded data by eye, which we can't really do with gob.

We can see, for example, that this is some kind of mapping from a string (`john`) to some kind of struct, with a numeric age field and a hobbies field that is a slice of strings. And we can see the values of all those fields.

The whitespace and newlines help with readability, but in fact they're ignored by JSON. They just allow us to *pretty-print* the data for better readability.

JSON is a useful auxiliary language

JSON isn't Go-aware, but on the other hand its syntax is sufficiently general to suit most programming languages and data types. This perhaps partly explains JSON's wide adoption.

It makes sense, then, that the primary use for JSON is as an *auxiliary language* for communicating data between different kinds of applications and computer systems. For example, web-based APIs commonly accept request data formatted as JSON, and send responses in the same format.

It's also becoming increasingly common to use JSON as an input/output data format for command-line tools, and that's a nice way for us to tie it in with the focus of this book.

As we discussed in the chapter on pipelines, the traditional Unix tools communicate with each other using either plain text, or more generally, byte streams.

A byte stream, though, lacks any kind of structure or type information. We can easily *format* a Go struct as text using the `fmt` package, but it's not so easy to *parse* arbitrary text into Go data, as we found in the chapter on commands.

A JSON output option

JSON, though, has at least some basic notions of structure: strings, numbers, slices, and structs. Many tools, such as `kubectl` and `terraform`, already offer a JSON output option for this reason.

This makes it much easier, in turn, to write tools that consume or parse their output. There are some powerful tools that can operate on arbitrary JSON, such as the invaluable `jq`:

- https://stedolan.github.io/jq/

And, as we saw in the chapter on pipelines, programs using the `script` package can also process JSON data using `jq` queries.

Go itself has a JSON output mode for tests, which can be useful for implementing things like automated test runners (such as `gotestdox`, which we encountered in the first chapter):

```
go test -json
```

In a previous chapter we wrote a Go wrapper for the `pmset` command to get the system's battery status. This would certainly have been easier if `pmset` had an option to output JSON instead of plain text.

It makes sense, then, to add a JSON output option to our own tools. Let's try.

Adding JSON to the `battery` package

Suppose in some future version of the `battery` package, we parsed *all* the data produced by `pmset`, instead of just the battery charge percentage. Could we then serialise it to JSON for output?

By the way, there's an interesting project called `jc`, which acts as a "JSONiser" for various popular Unix utilities. It parses the specific command's output format, just as we did for `pmset`, and then formats the result as JSON:

- https://github.com/kellyjonbrazil/jc

Could we do something similar for `pmset`? Suppose we had a `Battery` struct like this:

```
batt := battery.Battery{
    Name:             "InternalBattery-0",
    ID:               10813539,
    ChargePercent:    100,
    TimeToFullCharge: "0:00",
    Present:          true,
}
```

We know how to take flags, so we could take a `-j` flag to our tool requesting JSON output. How would we generate it?

Producing JSON strings

We know how to encode JSON to a writer, but that isn't quite what we want here. The user would have to pass us the writer to write to, which is awkward:

```
batt.WriteJSONTo(os.Stdout)
```

In any case they might not be ready to actually print the value yet. They might just want the JSON data to interpolate in some document they're generating, such as a report.

In the "normal" (non-JSON) mode, our output would be a string, so it makes sense that our JSON mode should behave the same way.

Suppose there were a method on the `Battery` type named `ToJSON`, then, that returns the data as a JSON-encoded string. Should it also return `error`? What do you think?

What could go wrong?

Clearly there can *be* an error encoding values to JSON. One such error would be trying to encode an invalid type such as a function or channel. Alternatively, we could have a *cyclic*

structure: a struct that contains a pointer to itself, for example. This can't be represented in JSON.

It seems unlikely that we'd make this kind of mistake in defining our `Battery` struct, though. Even if we did, there's nothing users could do about it, so sending them an error wouldn't be helpful.

This is one of the rare cases where a *panic* is appropriate. Remember, a panic signals an unrecoverable *internal* program error: it's the programmer's fault, not the user's. And that's certainly the case if the `Battery` struct is so malformed as to be un-JSON-able.

Now let's see if you can rough out a suitable test.

GOAL: Write a test for `ToJSON`.

HINT: We've said that the API for `ToJSON` is something like "method on `Battery`, returns `string`". So if we create some `Battery` object and then call its `ToJSON` method, we should be able to predict the exact string we want as the result.

The JSON string might be a bit unwieldy to write directly in our code as a literal, so if you like, you can read it from a `testdata` file instead.

See what you can do!

Testing `ToJSON`

SOLUTION: Here's my version:

```
func TestToJSON_GivesExpectedJSON(t *testing.T) {
    t.Parallel()
    batt := battery.Battery{
```

```
        Name:                 "InternalBattery-0",
        ID:                   10813539,
        ChargePercent:        100,
        TimeToFullCharge: "0:00",
        Present:              true,
    }
    wantBytes, err :=
os.ReadFile("testdata/battery.json")
    if err != nil {
        t.Fatal(err)
    }
    want := string(wantBytes)
    got := batt.ToJSON()
    if !cmp.Equal(want, got) {
        t.Error(cmp.Diff(want, got))
    }
}
```

(Listing battery/2)

By the way, you don't have to laboriously construct the golden file by hand. You can take the lazy option instead: just put the string bogus in the file, so that it fails against the null implementation of `ToJSON`. Then implement the method for real and see what it actually produces.

If the result looks correct to you, then copy it into the golden file. Needless to say, you'd better be sure you've got it right, but this is a useful technique for constructing test data and keeping it in sync with your program.

Over to you again now to make this test pass.

GOAL: Implement `ToJSON`.

HINT: You already know how to marshal a Go object to JSON by creating a `NewEncoder` on some writer, then passing the object to Encode. And, if you like, you can implement `ToJSON` this way.

The problem is that we'd need to create a writer, which we'll only end up throwing away, because what we *really* wanted was a string. In that situation, we could use an alternate API provided by the `json` package:

```
data, err := json.Marshal(object)
```

There's no intermediate `Encoder` here: instead, `Marshal` turns the given object directly into a `[]byte` (unless there's an error, of course). Once we have the bytes, it's easy to turn them into a string.

Implementing `ToJSON`

SOLUTION: Let's make this a bit more interesting, and extend our `Battery` struct type to store a bit more information. For example:

```
type Battery struct {
    Name             string
    ID               int64
    ChargePercent    int
    TimeToFullCharge string
    Present          bool
}
```

(Listing battery/2)

How should we create the JSON string we want? As we discussed in the hint section, we could create a buffer and encode the JSON to it, then return the contents of the buffer as a string.

That’s not terrible, but using the alternate `json.Marshal` API results in a simpler implementation:

```
func (b Battery) ToJSON() string {
    output, err := json.Marshal(b)
    if err != nil {
        panic(err)
    }
    return string(output)
}
```

(Listing battery/2)

As we discussed, there can only be a marshalling error here if we’ve messed up the struct definition somehow, so `panic` is quite appropriate for that case.

Pretty-printing with indentation

The string produced by `ToJSON` looks like this:

```
{"Name":"InternalBattery-
0","ID":10813539,"ChargePercent":100,
"TimeToFullCharge":"0:00","Present":true}
```

That’s valid JSON, for sure, but it’s a bit dense. A pretty-printed version would be preferable for display to humans. What can we do?

There’s a handy variant of `json.Marshal` called `MarshalIndent`, which will do this for us:

```
output, err := json.MarshalIndent(b, "", "  ")
```

The two otherwise mysterious arguments are the *prefix* and the *indent* strings to use: here, we’ve selected no prefix and an indent of two spaces.

The result looks like this:

```
{
  "Name": "InternalBattery-0",
  "ID": 10813539,
  "ChargePercent": 100,
  "TimeToFullCharge": "0:00",
  "Present": true
}
```

In tests, we usually want to compare JSON strings while ignoring differences due to whitespace. A good way to do this is to *normalise* them both, by unmarshalling them and then re-marshalling, eliminating any irrelevant whitespace.

Querying JSON output

Now that we can generate JSON from our battery status tool, let's see what we can do with that output.

We can pipe it through `jq`, for example, and query it for the charge percentage:

```
battery | jq ".ChargePercent"
100
```

Because the data now has *structure*, rather than just being raw text, this is a great way of adding value to any command-line tool.

YAML: a less verbose JSON

Simplicity is a recurring theme of this book, and simple tools don't need much configuration. They won't take lots of flags or switches, because their behaviour is simple and obvious. If they're so complex as to need a *configuration file*, things have gone really wrong.

But some programs *do* take configuration files, and if we were writing a Go tool to replace them, we'd probably need to start by accepting the same kind of configuration.

Also, some tools need to process configuration data for *other* tools, because that's their job. Such data is usually formatted as YAML, which is a bit less verbose than JSON:

```
# my global config
global:
  scrape_interval: 15s
  evaluation_interval: 30s
  # scrape_timeout is set to the global default
(10s).

  external_labels:
    monitor: codelab
    foo: bar
```

This is part of an example config file for the Prometheus monitoring system. As you can see, comments are allowed, indentation defines structure, and quotes are not required.

Parsing YAML in Go

How could we parse something like the Prometheus config example into a Go program? Let's try.

GOAL: Write a test for a function that parses the example YAML file.

HINT: Okay, we have some bytes, and we need to deserialise them into a Go value. What type of Go value would best represent this data, then?

One possibility is `map[string]any`, because any YAML (or JSON) document can be represented this way. But that's not really *parsing*, just remoulding one big ball of mud into another. Mud-wrestling has its appeal, no doubt, but we can do better than this.

Instead, let's think about what structure is implied by the data itself. Is it a list of things? No, it's *one* thing: a Prometheus configuration. Let's call that type `prom.Config`.

It looks like a struct, so what fields will it need? There seems to be one top-level field, identified by the key `global`. Its value looks like another struct: let's call it `GlobalConfig`.

This struct has three fields: a "scrape interval", an "evaluation interval", and some "external labels". The first two look like `time.Durations`, and the third is a list of something… but what?

It seems to be a list of string key-value pairs. So it sounds like the Go type `map[string]string` would model this data nicely.

There's also a comment that's worth noting:

```
# scrape_timeout is set to the global default
(10s).
```

It looks like there's also some duration field `ScrapeTimeout` on `GlobalConfig`, defaulting to 10 seconds. We'll need to expect this value in the test, even though it's not explicitly set in the data.

Suppose we had some function that reads a YAML file like this one, and turns it into a `Config` struct. Let's call the function `ConfigFromYAML`. The next decision is about its

signature: what does it need to take as a parameter? What does it need to return?

All we need to take, in this case, is the path to the YAML file, and we'll want to return the parsed `Config` struct and error. See what you can do.

Designing the API with a test

SOLUTION: We might start with a test for `ConfigFromYAML` looking something like this:

```
func TestConfigFromYAML_CorrectlyParsesYAMLData(t
*testing.T) {
    t.Parallel()
    want := prom.Config{
        Global: prom.GlobalConfig{
            ScrapeInterval:     15 * time.Second,
            EvaluationInterval: 30 * time.Second,
            ScrapeTimeout:      10 * time.Second,
            ExternalLabels: map[string]string{
                "monitor": "codelab",
                "foo":     "bar",
            },
        },
    }
    got, err :=
prom.ConfigFromYAML("testdata/config.yaml")
    if err != nil {
        t.Fatal(err)
    }
    if !cmp.Equal(want, got) {
        t.Error(cmp.Diff(want, got))
    }
}
```

(Listing prom/1)

Defining our types

As usual, to get the test compiling, we need to define the types we've mentioned:

```
type Config struct {
    Global GlobalConfig
}


type GlobalConfig struct {
    ScrapeInterval     time.Duration
    EvaluationInterval time.Duration
    ScrapeTimeout      time.Duration
    ExternalLabels     map[string]string
}
```

If we've got everything right, then our test should fail against a null implementation of `ConfigFromYAML`, and so it does.

Now, how should we implement the function for real?

The `go-yaml` package

There isn't a standard library package that deals with YAML, but fortunately for us, there's a good third-party equivalent:

- https://github.com/go-yaml/yaml

Decoding

The decoding API for `yaml` is very similar to that of the gob and `json` packages. We create a `yaml.Decoder` on some stream, and then pass its Decode method a pointer to a variable of the right type (in this case, `Config`).

Something like this should work:

```
func ConfigFromYAML(path string) (Config, error) {
    f, err := os.Open(path)
    if err != nil {
        return Config{}, err
    }
    defer f.Close()
    config := Config{}
    err = yaml.NewDecoder(f).Decode(&config)
    if err != nil {
        return Config{}, err
    }
    return config, nil
}
```

Does this pass? Let’s find out:

```
go test

prom.Config{
        Global: prom.GlobalConfig{
  -         ScrapeInterval:     s"15s",
  +         ScrapeInterval:     s"0s",
  -         EvaluationInterval: s"30s",
  +         EvaluationInterval: s"0s",
  -         ScrapeTimeout:      s"10s",
  +         ScrapeTimeout:      s"0s",
  -         ExternalLabels:
    map[string]string{"foo": "bar", "monitor":
"codelab"},
  +         ExternalLabels:     nil,
        },
    }
```

When unmarshalling doesn’t work

When unmarshalling doesn't work, it's usually either because the fields are unexported, or because the field names don't match.

Unexported fields aren't visible outside the package where they're defined, so they wouldn't be visible to the `yaml` package, but there are no unexported fields here: we can rule that problem out.

So it looks as though the Go field names don't match the YAML data. `ScrapeInterval` isn't the same as `scrape_interval`, for example. But Go uses camel case, so we don't want to change the Go field name.

We need a way to tell the `yaml` package that it should associate the Go `ScrapeInterval` field with the YAML `scrape_interval` field. Where could we put that information?

The format of struct tags

The Go designers have thoughtfully provided *struct tags* for this purpose. A struct tag is a string literal following a field declaration. For example:

```
type bogus struct {
    name string "Hi, I'm a struct tag!"
}
```

The Go compiler allows, but ignores, these tags. We can use them for whatever we want. In practice, they're mostly used for supplying serialisation hints to packages like `yaml`.

Here's a struct tag giving the required YAML field name hint for the `ScrapeInterval` field:

```
ScrapeInterval  time.Duration
`yaml:"scrape_interval"`
```

(Listing prom/1)

See if you can figure out the others, based on this example.

Setting defaults

Does that fix the test? Almost:

```
-       ScrapeTimeout:      s"10s",
+       ScrapeTimeout:      s"0s",
```

Ah yes, we forgot something. We need to set the default value for `ScrapeTimeout` before the YAML parsing begins:

```
config := Config{
    GlobalConfig{
        ScrapeTimeout: 10 * time.Second,
    },
}
```

(Listing prom/1)

And we have a passing test!

Eliminating config structs

So that’s how to parse arbitrary YAML data. What about the case where the YAML represents configuration for our *own* program? What should the API for that look like?

Well, we *wouldn’t* want to write something like this, for reasons we’ve discussed earlier in this book:

```
config, err := ConfigFromYAML("config.yaml")
... // handle error
```

```go
client, err := NewClient(config)
... // handle error
```

Config structs are an anti-pattern, because they make users do paperwork to construct them, only to pass them right back to us.

We already know how to do better than this. We can write, for example:

```go
client, err := NewClient(
    WithConfigFromYAML("config.yaml"),
)
... // handle error
```

Presumably there *is* some `Config` struct under the hood, if only to parse the YAML into, but it's hidden away. Users don't need to know about it.

Going further

If you've battled your way through this chapter so far, destroying all monsters, then try this end-of-level boss challenge:

- Extend the simple key-value store in listing kv/1 to add a command-line tool (or tools) that can add, update, or query data from the store, including the ability to dump all the keys and values.

  You can stick with the gob encoding, or use JSON instead, or something else. That's up to you, and the program *shouldn't* care about the specific wire format, beyond being able to read and write it. Feel free to write golden-file tests if you want to, though.

  You can find one possible solution in listing kv/2.

11. Clients

> *In general, it is best to assume that the network is filled with malevolent entities that will send in packets designed to have the worst possible effect.*
> —RFC 1122

So far in this book we've confined ourselves to writing tools that do things on the local machine, ignoring the wider world of resources available to us on the network. The internet can be a scary place, as we all know, so let's don our fireproof clothing and venture forth to see what's out there.

A simple weather client

For almost any general task that you can imagine doing, there's a public API that provides it: converting dates from the Hebrew calendar to the Gregorian, generating random numbers with laser beams, or downloading arbitrary-sized placeholder images of bacon (mmm, bacon).

Indeed, you can find a list of hundreds of such public APIs in this excellent GitHub repo:

- https://github.com/public-apis/public-apis

What do we need?

The bacon one is tempting, but let's try writing a *weather* client first. Suppose we just want to be able to get a short summary of the current weather conditions at our location:

```
Clouds 11.2ºC
```

We could use our tool to show the current weather conditions in our terminal prompt, or menu bar, for example. Where can we get that data from?

Let’s pick a weather API from that big list on GitHub. OpenWeatherMap isn’t a bad choice:

- https://openweathermap.org/api

Browsing through their documentation, we can see that there are lots of APIs and endpoints relating to weather, so we’ll pick the simplest one that can give us the current conditions at a specified location:

- https://openweathermap.org/current

We’ll need an *API key*, or token, which we can get by signing up to the OpenWeatherMap site and entering an email address. It doesn’t cost anything to use the API; the key just helps prevent the service being abused.

Kicking the tyres

Let’s first of all see if we can get the data we want using a web browser or a command-line tool like `curl`, before we worry about trying to do it with Go.

The API documentation gives a really nice example, showing what URL to call, and how to include your location and API key:

```
api.openweathermap.org/data/2.5/weather?q={city name}&appid=
{API key}
```

Wouldn’t it be great if every project’s documentation started with an example of how to use it? Word to the wise.

There are lots of ways to specify our location, but let’s try the “city, two-letter country code” format:

```
curl "https://api.openweathermap.org/data/2.5/weather?q=\
London,UK&appid=XXX"
```

You’ll need to substitute your own API key for XXX in this URL. Using one I registered earlier, I get the following response:

```
{"coord":{"lon":-0.1257,"lat":51.5085},"weather":[{"id":801,
"main":"Clouds", "description":"few clouds","icon":"02n"}],
"base":"stations","main":{"temp":282.47,"feels_like":281.12,
"temp_min":280.57,"temp_max":284.01,"pressure":996,"humidity":

77},"visibility":10000,"wind":{"speed":2.57,"deg":240},
"clouds":{"all":20},"dt":1635787388,"sys":{"type":2,"id":201,
"country":"GB","sunrise":1635749645,"sunset":1635784437},
"timezone":0,"id":2643743,"name":"London","cod":200}
```


Now that we know what HTTP request to make, we can try making it from a Go program.

GOAL: Write a Go program to make an OpenWeatherMap request, and print the response.

HINT: Let's use main-driven development to write the simplest possible Go program that can get weather data. Our program needs to do a few things:

1. Get the API key value from the user
2. Make the HTTP request
3. Print the response body

Environmental credentials

How should we have the user supply their API key? Passing credentials on the command line is a bad idea, because your command line is usually visible to other users via something like ps, and it will be saved in your shell history.

Putting the key in a file feels like paperwork, and we want to be able to run the program non-interactively, so we can't prompt for it either.

Instead, we can put the key in an *environment variable*: that way, the user can set it in their `.zshrc`, or however they configure their environment.

Making a GET request

How should we make the HTTP request? There's a nice low-paperwork API for this in the standard library `http` package:

```
resp, err := http.Get(URL)
```

Here, `resp` is the API's response, and we can look at its `StatusCode`, or read the contents of its Body.

Are you feeling inspired?

SOLUTION: Here's the least I can possibly do that solves this problem:

```
package main

import (
    "fmt"
    "io"
    "net/http"
    "os"
)

const BaseURL = "https://api.openweathermap.org"

func main() {
    key := os.Getenv("OPENWEATHERMAP_API_KEY")
    URL := fmt.Sprintf("%s/data/2.5/weather?
q=London,UK&appid=%s",
        BaseURL, key)
    resp, err := http.Get(URL)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    defer resp.Body.Close()
    io.Copy(os.Stdout, resp.Body)
}
```

(Listing weather/1)

We've hard-wired the location `London,UK` into the URL here for now; we'll worry about how to get the user's location later on.

Initial user testing

Let's try the program and see what happens:

```
go run main.go
```

```
{"cod":401, "message": "Invalid API key. Please see
http://openweathermap.org/faq#error401 for more info."}
```

Oops, I forgot to set the environment variable, so my API key was being sent as the empty string. We can improve on this error message, though, can't we? Something like this would be more polite:

```
Please set the environment variable OPENWEATHERMAP_API_KEY.
```

Computers should always be respectful when addressing humans: after all, who pays the electricity bill around here?

We're also not yet checking the HTTP response status, which we'll need to do. Anything other than `http.StatusOK` will need to be handled somehow.

I've made a note of those nits to fix in the next iteration, so let's set that variable and try again:

```
export OPENWEATHERMAP_API_KEY=xxx

go run main.go

{"coord":{"lon":-0.1257,"lat":51.5085},
... // lots more JSON
2643743,"name":"London","cod":200}
```

A second pass

Let's add in some checks for the missing API key, and a non-OK response status, and try again:

```
func main() {
    key := os.Getenv("OPENWEATHERMAP_API_KEY")
    if key == "" {
        fmt.Fprintln(os.Stderr, "Please set the environment \
            variable OPENWEATHERMAP_API_KEY.")
        os.Exit(1)
    }
    URL := fmt.Sprintf("%s/data/2.5/weather?q=London,\
        UK&appid=%s", BaseURL, key)
    resp, err := http.Get(URL)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
```

```
        os.Exit(1)
    }
    defer resp.Body.Close()
    if resp.StatusCode != http.StatusOK {
        fmt.Fprintln(os.Stderr, "unexpected response status",
            resp.Status)
        os.Exit(1)
    }
    io.Copy(os.Stdout, resp.Body)
}
```

(Listing weather/2)

All this behaviour will eventually be covered by tests, naturally, but we can check it manually for now.

Parsing JSON responses

What can we write a test for? One thing we know we'll need to do is to parse that JSON response into a Go value, which we can then print as a string.

So suppose there were some function `ParseResponse` that turned the raw JSON data from OpenWeatherMap into a Go struct. Could we write a test for that?

GOAL: Write a test for `ParseResponse`, using the real JSON data from OpenWeatherMap.

Testing `ParseResponse`

HINT: The first thing to do is put our sample JSON data in a file in `testdata`; that's our test input. Next, what's our `want`? Some struct, but with what fields?

Let's leave the temperature aside for a moment and start by getting the one-word weather summary as a string. Can you see what to do?

SOLUTION: Here's a first attempt at this test:

```
func TestParseResponse_CorrectlyParsesJSONData(t *testing.T)
{
    t.Parallel()
```

```go
    data, err := os.ReadFile("testdata/weather.json")
    if err != nil {
        t.Fatal(err)
    }
    want := weather.Conditions{
        Summary: "Clouds",
    }
    got, err := weather.ParseResponse(data)
    if err != nil {
        t.Fatal(err)
    }
    if !cmp.Equal(want, got) {
        t.Error(cmp.Diff(want, got))
    }
}
```

([Listing weather/3])

If we create a `weather` module, add the sample JSON data in `testdata`, define the `Conditions` struct, and provide a null implementation of `ParseResponse`, we should have a failing test:

```
-    Summary: "Clouds",
+    Summary: "",
```

That makes sense: we're not actually parsing anything yet. How should we do that?

To decode or to unmarshal?

We know two different JSON decoding APIs: `json.Decoder` and `json.Unmarshal`. Which is the right choice here?

In principle, we usually prefer the more efficient *streaming* API, `json.Decoder`. But the byte slice API, `json.Unmarshal`, might actually be better in this case. Why?

Consider what happens if there's an error parsing the JSON for some reason. The user experience might be something like:

```
weather London,UK

unexpected end of JSON input
```

This isn't terribly helpful: we'd like to *see* the JSON that couldn't be parsed, but we can't.

If we use `json.Decoder` on a stream, then by the time we hit a problem, the data will have already been consumed in the decoding. We won't have it available for printing.

On the other hand, if we used `json.Unmarshal`, we'd have all the data in the form of a byte slice. If we can't parse it for some reason, we'll be able to print the complete response, to help with debugging.

A temporary struct type

The next thing we need to know is what kind of struct to unmarshal the JSON data *into*. But wouldn't we just use a `Conditions`?

No, because that wouldn't work. We've defined `Conditions` as a struct containing a single string field `Summary`. But we can see from the OpenWeatherMap data that the schema implied by the JSON is something quite different.

Let's take a closer look at the test data. The JSON specifies some `coord` field that we don't care about, and then something that looks more promising:

```
"weather": [
  {
    "id": 801,
    "main": "Clouds",
    "description": "few clouds",
    "icon": "02d"
  }
],
```

So the struct implied by this data has a top-level field `weather`, containing an *array* (that is, a slice) of something; what? Some struct with a string field `main`—we want that—and some other fields we don't.

We're going to need a Go struct definition matching this schema, just as we did in the YAML parsing example in the previous chapter.

This type isn't the same as the `Conditions` struct we already defined, and that's okay. Its only purpose is to serve as a temporary holding area for the data parsed from the response. I call this kind of thing an *adapter struct*: it exists only to adapt the API's schema to our own.

The API's schema isn't suitable for use in our own program, because it contains mostly irrelevant information, and what *is* relevant isn't in a form that'll be convenient for us to work with. So we'll define some transient struct just for decoding purposes: let's call it `OWMResponse`.

The response struct

We can infer all the details of the necessary type by looking at the JSON. It's this:

```
type OWMResponse struct {
    Weather []struct {
        Main string
    }
}
```

(Listing weather/3)

We didn't bother to name the inner struct, the one that forms the elements of the `Weather` array, because we don't need to.

We also didn't add any struct tags, again because they're not needed. Indeed, we don't even need to *export* this type, because users won't refer to it.

But let's make it exported for now: people using our package might want to write their own tests that *do* create values of this struct.

Now that we have the test, the JSON data, and the struct type required to decode it, we're ready to go ahead and write `ParseResponse`.

GOAL: Write `ParseResponse`.

Implementing `ParseResponse`

HINT: Here's the general plan: we'll set up a variable of type `OWMResponse`, ready to receive the data, and we'll call `json.Unmarshal` to do the unmarshalling.

If the parsing is successful, we'll have a valid `OWMResponse`, and we can extract the data we want from it. Specifically, we'll want the value of the `Main` field of the first element of the `Weather` slice, and we'll put that value into the `Summary` field of our result.

Can you get the test to pass?

SOLUTION: Here's a first attempt:

```
func ParseResponse(data []byte) (Conditions, error) {
    var resp OWMResponse
    err := json.Unmarshal(data, &resp)
    if err != nil {
        return Conditions{}, fmt.Errorf(
            "invalid API response %s: %w", data, err)
    }
    conditions := Conditions{
        Summary: resp.Weather[0].Main,
    }
    return conditions, nil
}
```

What could go wrong?

Not bad, but we can see from the fact that there are two `return` statements that we need another test for "invalid input" behaviour. The simplest JSON document that won't parse successfully into an `OWMResponse` is an empty `[]byte`.

Let's write a test using this invalid data, proving that `ParseResponse` returns an error in this case. This should work:

```
func TestParseResponse_ReturnsErrorGivenEmptyData(t
*testing.T) {
    t.Parallel()
    _, err := weather.ParseResponse([]byte{})
    if err == nil {
        t.Fatal("want error parsing empty response, got nil")
    }
}
```

([Listing weather/3]())

While we're playing the "what could go wrong?" game, a closer look at this version of `ParseResponse` reveals another potential issue. Can you spot it?

GOAL: Find the bug in `ParseResponse` and add a test that demonstrates it, then fix it.

Other kinds of invalid data

HINT: We know that referring to a slice element without a protective `len` check can panic. So we need to write a test that *will* cause `ParseResponse` to panic unless it checks the slice length first.

Can you design some input data that would have this effect?

SOLUTION: Here's the test, first of all:

```
func TestParseResponse_ReturnsErrorGivenInvalidJSON(t
*testing.T) {
    t.Parallel()
    data, err := os.ReadFile("testdata/weather_invalid.json")
    if err != nil {
        t.Fatal(err)
    }
    _, err = weather.ParseResponse(data)
    if err == nil {
        t.Fatal("want error parsing invalid response, got
nil")
    }
}
```

(Listing weather/3)

Next, we'll need to write some new test data. We want something that's valid JSON, but whose `Weather` array contains no elements.

We can arrange that by copying the *valid* data and setting its `weather` field to an empty array:

```
"weather": [],
```

(Listing weather/3)

To pass this test, we'll need to add the requisite `len` check, with a suitably helpful error message. Here's the result:

```
func ParseResponse(data []byte) (Conditions, error) {
    var resp OWMResponse
    err := json.Unmarshal(data, &resp)
    if err != nil {
```

```
        return Conditions{}, fmt.Errorf(
            "invalid API response %s: %w", data, err)
    }
    if len(resp.Weather) < 1 {
        return Conditions{}, fmt.Errorf("invalid API response
%s: \
            want at least one Weather element", data)
    }
    conditions := Conditions{
        Summary: resp.Weather[0].Main,
    }
    return conditions, nil
}
```

(Listing weather/3)

What are we really testing here?

The internet is full of malevolent entities and bad data, so our parsing code is pretty cautious. If the JSON is empty, incomplete, invalid, or has the wrong schema, we detect that and give the user the raw data to help diagnose the problem.

Even if the JSON is syntactically valid and matches the schema, the data still might not be *well-formed* because it doesn't provide any `Weather` elements. We also handle that specific problem informatively.

Neither of these things would be our *fault*, to be sure, but they can still happen. Some programmers might be anxious about testing such situations, retaining a vague memory of being told "not to test other people's code".

But we're not testing OpenWeatherMap. We're testing what our program does when OpenWeatherMap misbehaves. In other words, don't test other people's code: test that your code does the right thing when *theirs* doesn't.

Back to a running program

Our earlier version of the program in `func main` ended like this, dumping a load of raw JSON into the user's surprised face:

```
io.Copy(os.Stdout, resp.Body)
```

We can do better than that now, because we have a `ParseResponse` function to make sense of the data. To extract it from the supplied `resp.Body`, we'll use our friend `io.ReadAll`:

```
data, err := io.ReadAll(resp.Body)
if err != nil {
    fmt.Fprintln(os.Stderr, err)
    os.Exit(1)
}
conditions, err := weather.ParseResponse(data)
if err != nil {
    fmt.Fprintln(os.Stderr, err)
    os.Exit(1)
}
fmt.Println(conditions)
```

What does the output look like now?

```
go run main.go
```

```
{Clouds}
```

Not bad!

Constructing the request URL

Another chunk of behaviour that we could pull out into a function and unit-test is constructing the request URL, given the user's location and API key:

```
URL := fmt.Sprintf("%s/data/2.5/weather?
q=London,UK&appid=%s",
    BaseURL, key)
```

A `FormatURL` function

We'll need to be able to supply the location, too. Let's say there's some function `FormatURL` that takes the base URL, location, and key, and returns a string containing the complete request URL.

This sounds like something we could write a test for. Let's try.

GOAL: Write a test for `FormatURL`.

HINT: We have a clear idea what the request URL should be, given our base URL, location, and key. So we can write a straightforward `want` and `got` comparison. Can you turn this idea into a test?

SOLUTION: This will get us started:

```
func TestFormatURL_ReturnsCorrectURLForGivenInputs(t
*testing.T) {
    t.Parallel()
    baseURL := weather.BaseURL
    location := "Paris,FR"
    key := "dummyAPIKey"
    want := "https://api.openweathermap.org/data/2.5/weather?
\
        q=Paris,FR&appid=dummyAPIKey"
    got := weather.FormatURL(baseURL, location, key)
    if !cmp.Equal(want, got) {
        t.Error(cmp.Diff(want, got))
    }
}
```

(Listing weather/3)

Having checked this test against the null implementation of `FormatURL`, we can now fill in the real code:

```
func FormatURL(baseURL, location, key string) string {
    return fmt.Sprintf("%s/data/2.5/weather?q=%s&appid=%s",
        baseURL, location, key)
}
```

(Listing weather/3)

We’re saying that the location, instead of being hard-wired to London, is now an argument to `FormatURL`. So how will the user supply that information when they run the program?

Getting the location as input

We could imagine some `-location` flag on the command line, like this:

```
weather -location London,UK
```

But that doesn't seem quite right. What if they omit the flag? There's no sensible default location that we could choose.

Actually, since the location is compulsory, it should be an *argument*, not a flag:

```
weather London,UK
```

Indeed, if the user does run the program without an argument, we can be helpful and show them what to do. Let's define a usage message to be shown in this case:

```
const Usage = `Usage: weather LOCATION

Example: weather London,UK`
```

Now we'll detect this case in `main` and show the message:

```
if len(os.Args) < 2 {
    fmt.Println(Usage)
    os.Exit(0)
}
location := os.Args[1]
URL := weather.FormatURL(weather.BaseURL, location, key)
```

(Listing weather/3)

Refactoring the remaining code

We've already written `FormatURL` and `ParseResponse`, so the only parts remaining to be refactored are making the request and checking the response status.

How would we call those from `main`? Maybe something like this:

```
URL := weather.FormatURL(weather.BaseURL, location, key)
data, err := weather.MakeAPIRequest(URL)
if err != nil {
    fmt.Fprintln(os.Stderr, err)
    os.Exit(1)
}
conditions, err := weather.ParseResponse(data)
```

A paperwork-reducing `GetWeather` function

This is *okay*, but not *great*. It feels a bit fussy. Why get a URL from some function only to pass it back to some other function to actually make the request?

Similarly, why get the `data` from some function only to pass it back to another function for parsing? We're doing too much paperwork.

What we'd *like* is to call some function that only needs the location and the key:

```
conditions, err := weather.GetWeather(location, key)
```

In general, don't give users paperwork just so they can pass it back to you. Keep the paperwork to yourself.

Testing against a local HTTP server

Testing `GetWeather` is interesting, because we don't want it to call the real OpenWeatherMap API. We need it to call some local HTTP server instead, that will respond to any GET request with our test JSON data.

The standard library `httptest` package has a nice, low-paperwork way to do this:

```
ts := httptest.NewTLSServer(http.HandlerFunc(
    func(w http.ResponseWriter, r *http.Request) {
        http.ServeFile(w, r, "testdata/weather.json")
    }))
defer ts.Close()
```

A simple `httptest` example

Let's start by writing a very simple test that creates an `httptest` server like this, calls it using `http.Get`, and checks that the status code is OK. Have a try.

GOAL: Write a simple test along these lines.

HINT: Once we've started the `httptest` server, we need to know its URL in order to make a request to it. Happily, the server itself contains that information, in the field `ts.URL`.

Can you figure it out?

SOLUTION: We might start by trying something like this:

```
func TestHTTPGet_SuccessfullyGetsFromLocalServer(t
*testing.T) {
    t.Parallel()
    ts := httptest.NewTLSServer(http.HandlerFunc(
        func(w http.ResponseWriter, r *http.Request) {
            http.ServeFile(w, r, "testdata/weather.json")
        }))
    defer ts.Close()
    resp, err := http.Get(ts.URL)
    if err != nil {
        t.Fatal(err)
    }
    defer resp.Body.Close()
    want := http.StatusOK
    got := resp.StatusCode
    if !cmp.Equal(want, got) {
        t.Error(cmp.Diff(want, got))
    }
}
```

But this doesn't work:

```
2021/11/03 15:13:21 http: TLS handshake error from 127.0.0.1:
62009: remote error: tls: bad certificate
--- FAIL: TestHTTPGet_SuccessfullyGetsFromLocalServer (0.22s)
    weather_test.go:72: Get "https://127.0.0.1:62008": x509:
    certificate signed by unknown authority
```

One way around this might be to use a non-TLS server for testing, but that's a mistake. Sure, we trust the test server, and we're talking to it over a local link. But the `weather` code will need to talk to *untrusted* servers (no offence, OpenWeatherMap) over insecure networks.

In other words, the real program will always use TLS connections, so let's test it the same way. The trust problem is easily solved, it turns out.

A trustful TLS client

We got a "signed by unknown authority" error when connecting to our test server because its certificate (not surprisingly) is self-signed. We wouldn't want to trust a self-signed certificate from some random server

on the internet, but this is different: we control the test server, so it's okay. How can we persuade our HTTP client to trust it, then?

Well, the test server itself can provide us with a suitably trustful client:

```
client := ts.Client()
resp, err := client.Get(ts.URL)
```

If we make this change, the test passes. Great! Let's force it to fail by setting `want` to something that's not `http.StatusOK`:

```
want := http.StatusTeapot
```

This fails as expected:

```
-   418,
+   200,
```

A `weather` client object

We still have a problem: we don't have a way to pass the test server's URL to `GetWeather`. It will, presumably, use `FormatURL` to construct the URL itself, incorporating the OpenWeatherMap base URL.

How can we inject the test URL instead? We could make `BaseURL` a global variable instead of a constant, and set it from the test, but that feels wrong, and we wouldn't be able to parallelise our tests.

A familiar pattern

We're saying, in fact, that up to now we've effectively been using some *default OpenWeatherMap client*, and now we'd like one that we can customise.

This is a familiar pattern:

```
c := weather.NewClient(key)
```

The default client should talk to the real OpenWeatherMap URL, but we'll be able to override that for testing purposes:

```
c.BaseURL = ts.URL
```

Now that we have a client object, it makes sense that we should get the weather by calling some method on it. `GetWeather`, perhaps:

```
conditions, err := c.GetWeather(location)
```

Refactoring the tests to use our client

We'll need to update this test, since `FormatURL` now also becomes a method on the client:

```
func TestFormatURL_ReturnsCorrectURLForGivenInputs(t
*testing.T) {
    t.Parallel()
    c := weather.NewClient("dummyAPIKey")
    location := "Paris,FR"
    want := "https://api.openweathermap.org/data/2.5/weather?
\
        q=Paris,FR&appid=dummyAPIKey"
    got := c.FormatURL(location)
    if !cmp.Equal(want, got) {
        t.Error(cmp.Diff(want, got))
    }
}
```

(Listing weather/4)

GOAL: Get the `FormatURL` test passing again.

Writing the client constructor

HINT: What information does the client struct need to store? First, the API key, which will be passed in to the constructor. Next, for test purposes, we'll want to be able to set both the base URL and an arbitrary HTTP client.

Can you add the necessary fields to the struct type, and set them to sensible values in `NewClient`?

SOLUTION: Let's try this:

```
type Client struct {
    APIKey  string
    BaseURL string
    HTTPClient *http.Client
}
```

Here's the constructor:

```
func NewClient(apiKey string) *Client {
    return &Client{
        APIKey:  apiKey,
        BaseURL: "https://api.openweathermap.org",
        HTTPClient: &http.Client{
            Timeout: 10 * time.Second,
        },
    }
}
```

(Listing weather/4)

Refactoring `FormatURL` is straightforward. We're not adding any new behaviour, just moving paperwork around:

```
func (c Client) FormatURL(location string) string {
    return fmt.Sprintf("%s/data/2.5/weather?q=%s&appid=%s",
        c.BaseURL, location, c.APIKey)
}
```

(Listing weather/4)

We're back to passing tests, which is good. There's nothing wrong with making refactorings that break tests (indeed, those are the best kind, because they're improving your API).

But once we *have* failing tests, our top priority is to fix them. This will be much harder to do later, because we'll no longer be clear about exactly what change broke them.

Never let the code drift more than about five minutes away from passing tests. That way, you still have a fighting chance of getting them passing again.

Refactoring `GetWeather` as a client method

We need to do one more thing: check that `GetWeather` returns the expected weather data.

Testing `GetWeather`

See if you can put all these pieces together now to write a test for `GetWeather`.

GOAL: Write a test for `GetWeather`.

HINT: We can adapt the simple HTTP test we wrote earlier, adding the new handler, and creating a weather client configured to call it.

Since we control the data, we also know the exact `Conditions` struct that we should get back. Can you see what to do?

SOLUTION: Here's the test:

```
func TestGetWeather_ReturnsExpectedConditions(t *testing.T) {
    t.Parallel()
    ts := httptest.NewTLSServer(http.HandlerFunc(
        func(w http.ResponseWriter, r *http.Request) {
            http.ServeFile(w, r, "testdata/weather.json")
        }))
    defer ts.Close()
    c := weather.NewClient("dummyAPIKey")
    c.BaseURL = ts.URL
    c.HTTPClient = ts.Client()
    want := weather.Conditions{
        Summary: "Clouds",
    }
    got, err := c.GetWeather("Paris,FR")
    if err != nil {
        t.Fatal(err)
    }
    if !cmp.Equal(want, got) {
        t.Error(cmp.Diff(want, got))
    }
}
```

(Listing weather/4)

Once you've got this test completed and filled in the necessary null implementation of `GetWeather`, we should able to see it fail:

```
-   Summary: "Clouds",
+   Summary: "",
```

All we need to do now is move the remaining code from `main` to `GetWeather`, and update it to use the new client struct.

Implementing `GetWeather`

We don't need to do any clever problem-solving here, just careful refactoring:

```
func (c *Client) GetWeather(location string) (Conditions,
    error) {
    URL := c.FormatURL(location)
    resp, err := c.HTTPClient.Get(URL)
    if err != nil {
        return Conditions{}, err
    }
    defer resp.Body.Close()
    if resp.StatusCode != http.StatusOK {
        return Conditions{}, fmt.Errorf("unexpected response
\
            status %q", resp.Status)
    }
    data, err := io.ReadAll(resp.Body)
    if err != nil {
        return Conditions{}, err
    }
    conditions, err := ParseResponse(data)
    if err != nil {
        return Conditions{}, err
    }
    return conditions, nil
}
```

(Listing weather/4)

A convenience wrapper

What will all this paperwork reduction leave us with in `main`? Let's take a look:

```
func main() {
    if len(os.Args) < 2 {
        fmt.Println(Usage)
        os.Exit(0)
```

```
    }
    key := os.Getenv("OPENWEATHERMAP_API_KEY")
    if key == "" {
        fmt.Fprintln(os.Stderr, "Please set the environment \
            variable OPENWEATHERMAP_API_KEY.")
        os.Exit(1)
    }
    location := os.Args[1]
    c := weather.NewClient(key)
    conditions, err := c.GetWeather(location)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    fmt.Println(conditions)
}
```

Can we do more? We know that whenever we have a client object, users will appreciate being able to call a convenience wrapper around an implicit default client.

They'd *like* to be able to call a single function, such as `weather.Get`, that constructs and uses the client for them, transparently. Let's write that:

```
func Get(location, key string) (Conditions, error) {
    c := NewClient(key)
    conditions, err := c.GetWeather(location)
    if err != nil {
        return Conditions{}, err
    }
    return conditions, nil
}
```

(Listing weather/4)

Adding a `Main` function

We can also lift and shift the code from `main` into a `weather.Main` function, in the same way that we did for previous CLI tools:

```
func Main() {
    if len(os.Args) < 2 {
        fmt.Println(Usage)
```

```
        return 0
    }
    key := os.Getenv("OPENWEATHERMAP_API_KEY")
    if key == "" {
        fmt.Fprintln(os.Stderr, "Please set the environment \
            variable OPENWEATHERMAP_API_KEY.")
        return 1
    }
    location := os.Args[1]
    conditions, err := Get(location, key)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        return 1
    }
    fmt.Println(conditions)
    return 0
}
```

(Listing weather/4)

That gets us to the minimal main we want:

```
package main

import (
    "os"

    "github.com/bitfield/weather"
)

func main() {
    os.Exit(weather.Main())
}
```

(Listing weather/4)

Adding temperature support

We now have the weather machinery working end to end for the *summary* data, so let's now add in the missing *temperature* feature. See what you can do:

GOAL: Add temperature handling.

HINT: Extending our `Conditions` struct with a `float64` field to represent the temperature seems sensible. How would we add support for this, test-first? Can you figure it out by applying what you've learned so far?

Parsing temperature data

SOLUTION: Let's start by modifying the `ParseResponse` test to expect the temperature value from the test data:

```
want := weather.Conditions{
    Summary:     "Clouds",
    Temperature: 284.1,
}
```

If we add the necessary field to the struct definition, the test fails as expected, because we're not parsing the temperature data yet:

```
    Summary:     "Clouds",
-   Temperature: 284.1,
+   Temperature: 0,
```

To get this working, we need to tell the JSON decoder where to find the temperature information, by adding its schema to our adapter struct:

```
type OWMResponse struct {
    Weather []struct {
        Main string
    }
    Main struct {
        Temp float64
    }
}
```

Now there should be a `resp.Main.Temp` field on the decoded value in `ParseResponse`. We can use that to set the temperature in the computed `Conditions` struct:

```
conditions := Conditions{
    Summary:     resp.Weather[0].Main,
    Temperature: resp.Main.Temp,
}
```

The test now passes, and we can also adjust the `want` value in the `GetWeather` test to expect the same temperature. That should pass without any further changes.

More user testing

Let's try running the program for real to see what the output looks like:

```
go run ./cmd/weather London,UK
```

```
{Clouds 285.22}
```

We can definitely improve on this, can't we? Firstly, we don't need to see the curly braces generated by `fmt.Println`. Let's use `Printf` to improve the formatting a little:

```
fmt.Printf("%s %.1f\n", conditions.Summary,
    conditions.Temperature)
```

This is a little nicer:

```
Clouds 285.2
```

Handling quantities with units

But what *units* is that temperature in? The API docs say:

> *Temperature is available in Fahrenheit, Celsius and Kelvin units… Temperature in Kelvin is used by default.*
> —https://openweathermap.org/current#data

Unless they're physicists, users are most likely to want temperature information in Celsius or Fahrenheit, so what should we do?

Actually, units are just a matter of presentation, aren't they? We can store the temperature *internally* in whatever units we want: kelvin will be just fine, and that's what we happen to have.

In general, don't convert values back and forth between different unit systems in your code. Pick a lane, preferably SI units, and stick to it. You only need to convert units when it comes to displaying values to users.

Presenting temperatures in Celsius

Let's update the weather client to be able to report the temperature in degrees Celsius. What API would make sense for this?

We could make users call some unit-conversion function (`KelvinToCelsius`, for example), but that feels a little paperworky.

What we'd *like* to write is something like:

```
conditions.Temperature.Celsius()
```

See what you can do!

GOAL: Test and implement the `Celsius` method.

HINT: It's a classic `want` and `got` situation, isn't it? Testing the behaviour of the `Celsius` method is easy, then, but implementing it isn't as simple as defining a method on `float64`, because that's not allowed.

Can you see how to get around this? Once you know the trick, it's quite simple.

SOLUTION: We might start by writing a test like this:

```
func TestCelsiusCorrectlyConvertsFahrenheitToCelsius(t
*testing.T) {
    t.Parallel()
    input := 274.15
    want := 1.0
    got := input.Celsius()
    if !cmp.Equal(want, got) {
        t.Error(cmp.Diff(want, got))
    }
}
```

As expected, this gives us a compile error:

```
input.Celsius undefined (type float64 has no field or method
Celsius)
```

No problem: let's define this method. It's not hard to write:

```
func (t float64) Celsius() float64 {
    return t - 273.15
}
```

But this still doesn't compile:

```
invalid receiver float64 (basic or unnamed type)
```

Defining a `Temperature` type

We can’t define methods on other people’s types, including built-in types, which makes sense: that would *modify* the type, and that’s not allowed. Instead, we need to define our own type, and add a method to it:

```
type Temperature float64

func (t Temperature) Celsius() float64 {
    return float64(t) - 273.15
}
```

(Listing weather/5)

To make our test compile, we just need to explicitly convert our `input` value to the new `Temperature` type:

```
func TestCelsiusCorrectlyConvertsFahrenheitToCelsius(t
*testing.T) {
    t.Parallel()
    input := weather.Temperature(274.15)
    want := 1.0
    got := input.Celsius()
    if !cmp.Equal(want, got) {
        t.Error(cmp.Diff(want, got))
    }
}
```

(Listing weather/5)

And now all is well. Let’s update our `Main` method to use `Celsius` when displaying the temperature:

```
fmt.Printf("%s %.1fºC\n", conditions.Summary,
    conditions.Temperature.Celsius())
```

Here’s what that looks like when we run the program for real:

```
Sunny 12.0ºC
```

Tackling more complex APIs

The code we've developed in this chapter is a good starting point for building a client for more or less any API. Let's look briefly at some of the ways we might need to extend it to cope with more complex APIs.

Request data

We may need to send request data as a JSON-encoded body, rather than embedded in the URL. In this case it's probably a good idea to define some `APIRequest` adapter struct, as we did with `OWMResponse`.

To make sure we're marshalling the request data correctly, the `httptest` handler can check it by *unmarshalling* it and comparing it with the original value. For APIs with multiple endpoints with different types of requests and responses, we may need multiple adapter structs.

"CRUD" methods

One common pattern for APIs that manage some kind of external resource is the set of methods known as *CRUD*: Create, Read, Update, and Delete. It makes sense to map each of these to a corresponding Go method on the client object.

There's usually some unique ID associated with the resource, so that we can specify the one we want. Usually the API assigns an ID, so `Create` should return it, while Read, `Update`, and `Delete` should take it as a parameter.

You can often write a single test for all the CRUD methods at once. It should create a resource, read it to check it was created properly, update it, and read it again to make sure the update worked. The test can then delete the resource, and make sure that a final read fails because it no longer exists.

Last words

Well, here we are at the end already. Thanks for taking this journey with me!

We began this book by describing the "universal library" of Go, and redefining our own role as software developers to see ourselves as contributors to this library, instead of merely writing one-off tools to solve our immediate problems. In other words, we're writing packages, not just programs.

This shift in mindset automatically produces higher quality software: easier to use, more flexible, more reliable. We’re doing better work because we’re just *thinking* about it more, and I hope you’ll agree that this actually makes the whole development process more fun, not less.

However, there’s one more thing we should keep in mind. The game designer Sid Meier, of “Civilization” fame, had a wise and useful rule for himself:

> *Make sure the player is the one having fun.*

Without that rule, he said, it would be too easy for him to add game features that seemed like fun to *implement*, but that might end up not being so much fun for people to actually *play*.

Writing software can be very enjoyable and rewarding (if you’re doing it right), but we should be careful not to forget that the user is the reason we’re doing it. Good design is not just a matter of putting more things in, but also of knowing what to leave *out*.

Features that seem cool or interesting to us, for example, might well mean nothing to users, or just serve to make the software more complicated and annoying to operate. Conversely, features that are important to them might seem boring or difficult to us, so we could easily overlook or shirk them.

Programming *is* fun, and that’s as it should be, but having fun needn’t mean working in a slapdash or lazy way. Instead, it’s more fun when we know we’re doing good work. But, especially in the commercial world, there can be a lot of pressure on us to do quick, sloppy work, and prioritise fast shipping over quality.

That isn’t good business sense, though, if you take the longer-term view. It’s no good making your manager happy by making users miserable, because it’s the users who ultimately pay your salary (and your manager’s).

Quality is the best business plan, and the only sustainable one in the long run. That means making sure the users are the ones having fun.

So it’s a good idea to ask ourselves from time to time, “Is this software fun to use?” That is to say, is it simple, easy, intuitive, friendly, polite, responsive, useful, helpful, natural, and—just occasionally—*delightful*?

And if the answer is “not yet”, well, perhaps it needs a little more work.

Can you see what to do?

Going further

If you've worked your way through the whole book and you still want more of a challenge, then I salute you. You're clearly made of the right stuff. Here's one suggestion for exploring further.

- Currently, locations with spaces in the name are not handled correctly, because we only read the first space-separated argument.

  Extend the weather client so that the entire command line is treated as the location. So, for example, you could run a command line:

  ```
  weather new york,us
  ```

  If there are multiple locations that match the request, OpenWeatherMap just returns one, but it may not be the one the user wanted. Have the client report what *actual* location it's giving the weather for, to avoid any confusion.

  You can see one possible solution to this in listing `weather/6`.

About this book

Who wrote this?

John Arundel is a Go teacher and mentor of many years experience. He's helped literally thousands of people to learn Go, with friendly, supportive, professional mentoring, and he can help you too. Find out more:

- Learn Go remotely with me

Feedback

If you enjoyed this book, let me know! Email go@bitfieldconsulting.com with your comments. If you didn't enjoy it, or found a problem, I'd like to hear that too. All your feedback will go to improving the book.

Also, please tell your friends, or post about the book on social media. I'm not a global mega-corporation, and I don't have a publisher or a marketing budget: I write and produce these books myself, at home, in my spare time. I'm not

doing this for the money: I'm doing it so that I can help bring the power of Go to as many people as possible.

That's where you can help, too. If you love Go, tell a friend about this book!

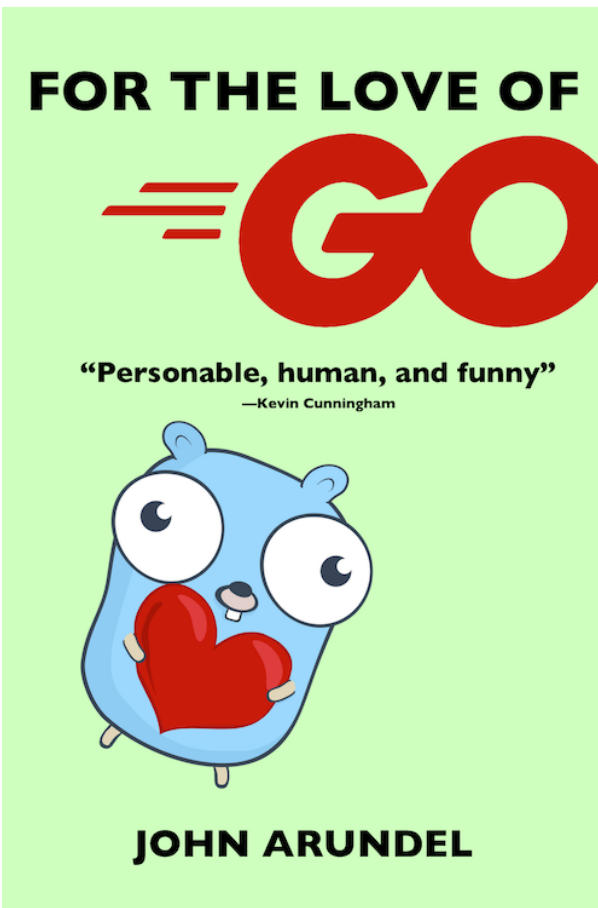
Mailing list

If you'd like to hear about it first when I publish new books, or even join my exclusive group of beta readers to give feedback on drafts in progress, you can subscribe to my mailing list here:

- Subscribe to Bitfield updates

For the Love of Go

For the Love of Go is a book introducing the Go programming language, suitable for complete beginners, as well as those with experience programming in other languages.



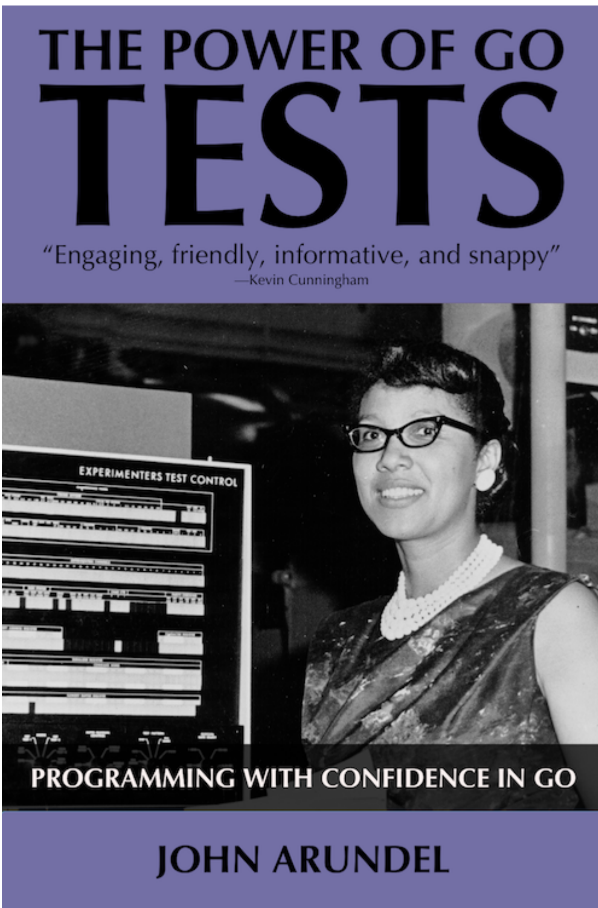
If you've used Go before but feel somehow you skipped something important, this book will build your confidence in the fundamentals. Take your first steps toward mastery with this fun, readable, and easy-to-follow guide.

Throughout the book we'll be working together to develop a fun and useful project in Go: an online bookstore called Happy Fun Books. You'll learn how to use Go to store data about real-world objects such as books, how to write code to manage and modify that data, and how to build useful and effective programs around it.

The Power of Go: Tests

What does it mean to program with confidence? How do you build self-testing software? What even is a test, anyway?

The Power of Go: Tests answers these questions, and many more.



Welcome to the thrilling world of fuzzy mutants and spies, guerilla testing, mocks and crocks, design smells, mirage tests, deep abstractions, exploding pointers, sentinels and six-year-old astronauts, coverage ratchets and golden files, singletons and walking skeletons, canaries and smelly suites, flaky tests and concurrent callbacks, fakes, CRUD methods, infinite defects, brittle tests, wibbly-wobby timey-wimey stuff, adapters and ambassadors, tests that fail only at midnight, and gremlins that steal from the player during the hours of darkness.

> *If you get fired as a result of applying the advice in this book, then that's probably for the best, all things*

> *considered. But if it happens, I'll make it my personal mission to get you a job with a better company: one where people are rewarded, not punished, for producing software that actually works.*

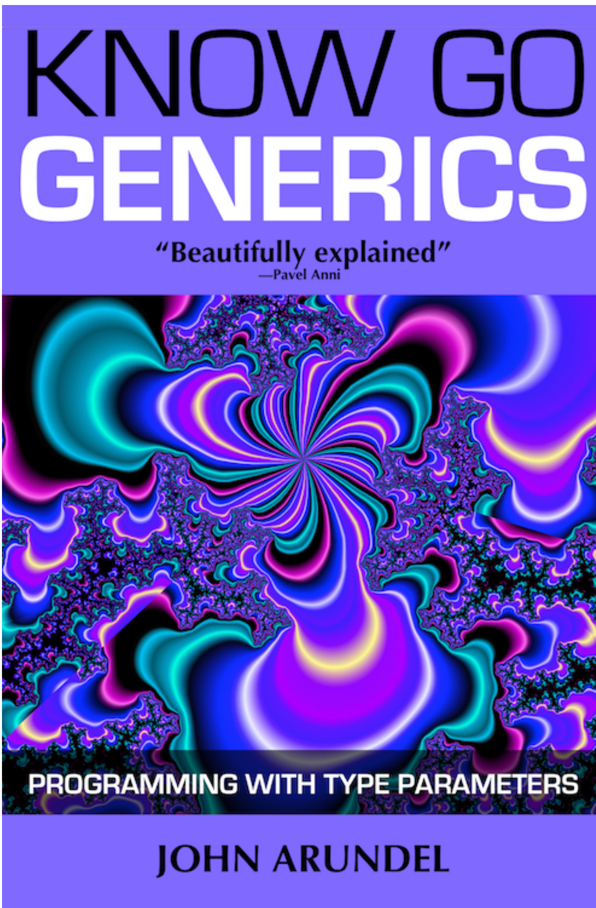
Go's built-in support for testing puts tests front and centre of any software project, from command-line tools to sophisticated backend servers and APIs. This accessible, amusing book will introduce you to all Go's testing facilities, show you how to use them to write tests for the trickiest things, and distils the collected wisdom of the Go community on best practices for testing Go programs. Crammed with hundreds of code examples, the book uses real tests and real problems to show you exactly what to do, step by step.

You'll learn how to use tests to design programs that solve user problems, how to build reliable codebases on solid foundations, and how tests can help you tackle horrible, bug-riddled legacy codebases and make them a nicer place to live. From choosing informative, behaviour-focused names for your tests to clever, powerful techniques for managing test dependencies like databases and concurrent servers, The Power of Go: Tests has everything you need to master the art of testing in Go.

If that sounds interesting, there's a sneak preview of the first chapter at the end of *this* book!

Know Go: Generics

Go beyond the basics, and master the new generics features introduced in Go 1.18. Learn all about type parameters and constraints in Go and how to use them, with this easy-to-read but comprehensive guide.



If you're new to Go and generics, and wondering what all the fuss is about, this book is for you! If you have some experience with Go already, but want to learn about the new generics features, this book is also for you. And if you've been waiting impatiently for Go to just get generics already so you can use it, don't worry: this book is for you too!

You don't need an advanced degree in computer science or tons of programming experience. Know Go: Generics explains what you need to know in plain, ordinary language, with simple examples that will show you what's new, how the language changes will affect you, and exactly how to use generics in your own programs and packages.

As you'd expect from the author of For the Love of Go and The Power of Go: Tools, it's fun and easy reading, but it's

also packed with powerful ideas, concepts, and techniques that you can use in real-world applications.

Further reading

You can find more of my books on Go here:

- Go books by John Arundel

You can find more Go tutorials and exercises here:

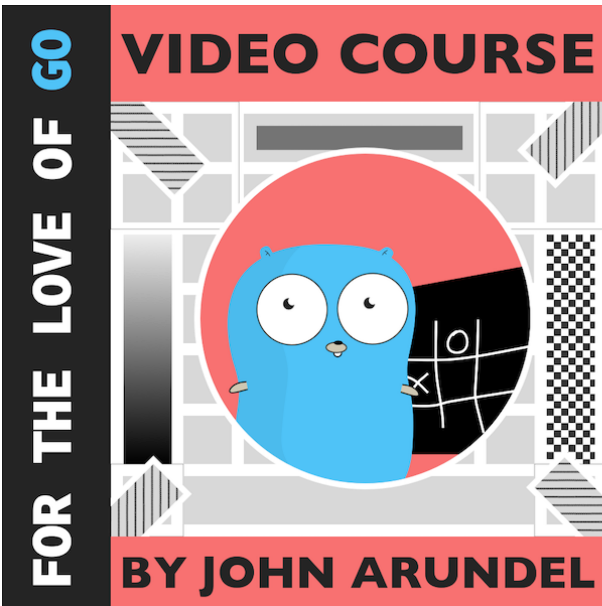
- Go tutorials from Bitfield

I have a YouTube channel where I post occasional videos on Go, and there are also some curated playlists of what I judge to be the very best Go talks and tutorials available, here:

- Bitfield Consulting on YouTube

Video course

If you're one of the many people who enjoys learning from videos, as well as from books, you may like the video course that accompanies the 'For the Love of Go' book:



- For the Love of Go: Video Course

Credits

Gopher images by the magnificent egonelbre and MariaLetta, except:

- “Gopher with flag” image copyright 2019 by danielb42, used under the MIT Licence.

Acknowledgements

The first and most important debt of gratitude I owe to my students at the Bitfield Institute of Technology, my online software engineering school. Mentoring is always a two-way process, and I've learned a great deal from working with this wonderful group of people on a vast range of practical software projects in Go. If there are any useful lessons in this book, they're things I've learned through collaboration with some of the best software engineers I know.

Many thanks also to the hundreds of beta readers who willingly read early drafts of various editions of this book, and took the time to give me detailed and useful feedback on where it could be improved. A special mention is due to Salvador Cavadini, Ivan Fetch, Artem Baguinski, Jakub Jarosz, and Lee Gibson for their help with this. Thanks also to Sam Atkins for catching a bug in the examples.

If you'd like to be one of my beta readers in future, please go to my website, enter your email address, and tick the appropriate box to join my mailing list:

- https://bitfieldconsulting.com/

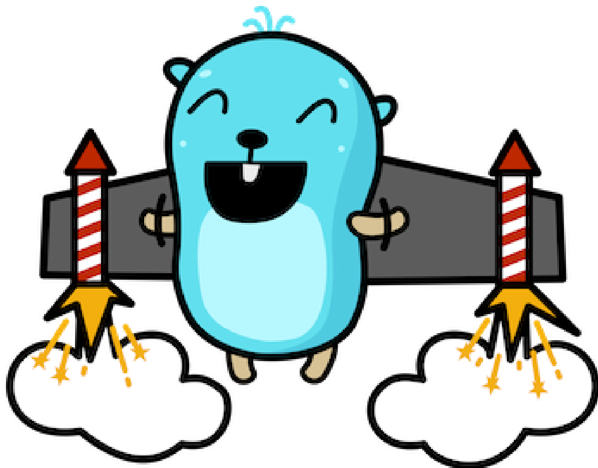
A sneak preview

If you're wondering where to go next after reading this book, here comes a sneak preview of the first chapter of The Power of Go: Tests. If you enjoy this chapter, please use the discount code T00LS2TESTS to get 25% off the price of the full book. Just go to the product page, add the book to your cart, and then enter the code at the checkout.

Happy fun reading!

1. Programming with confidence

> *It seemed that for any piece of software I wrote, after a couple of years I started hating it, because it became increasingly brittle and terrifying.*
> *Looking back in the rear-view, I'm thinking I was reacting to the experience, common with untested code, of small changes unexpectedly causing large breakages for reasons that are hard to understand.*
> —Tim Bray, "Testing in the Twenties"



When you launch yourself on a software engineering career, or even just a new project, what goes through your mind? What are your hopes and dreams for the software you're going to write? And when you look back on it after a few years, how will you feel about it?

There are lots of qualities we associate with good software, but undoubtedly the most important is that it be *correct*. If it doesn't do what it's supposed to, then almost nothing else about it matters.

Self-testing code

How do we know that the software we write is correct? And, even if it starts out that way, how do we know that the minor changes we make to it aren't introducing bugs?

One thing that can help give us confidence about the correctness of software is to write *tests* for it. While tests are useful whenever we write them, it turns out that they're especially useful when we write them *first*. Why?

The most important reason to write tests first is that, to do that, we need to have a clear idea of how the program should behave, from the user's point of view. There's some thinking involved in that, and the best time to do it is before we've written any code.

Why? Because trying to write code before we have a clear idea of what it should do is simply a waste of time. It's almost bound to be wrong in important ways. We're also likely to end up with a design which might be convenient from the point of view of the *implementer*, but that doesn't necessarily suit the needs of users at all.

Working test-first encourages us to develop the system in small *increments*, which helps prevent us from heading too far down the wrong path. Focusing on small, simple chunks of user-visible behaviour also means that everything we do to the program is about making it more valuable to users.

Tests can also guide us toward a good design, partly because they give us some experience of using our own

APIs, and partly because breaking a big program up into small, independent, well-specified modules makes it much easier to understand and work on.

What we aim to end up with is *self-testing code*:

> *You have self-testing code when you can run a series of automated tests against the code base and be confident that, should the tests pass, your code is free of any substantial defects.*
> *One way I think of it is that as well as building your software system, you simultaneously build a bug detector that's able to detect any faults inside the system. Should anyone in the team accidentally introduce a bug, the detector goes off.*
> —Martin Fowler, "Self-Testing Code"

This isn't just because well-tested code is more reliable, though that's important too. The real power of tests is that they make developers happier, less stressed, and more productive as a result.

> *Tests are the Programmer's Stone, transmuting fear into boredom. "No, I didn't break anything. The tests are all still green." The more stress I feel, the more I run the tests. Running the tests immediately gives me a good feeling and reduces the number of errors I make, which further reduces the stress I feel.*
> —Kent Beck, "Test-Driven Development by Example"

The adventure begins

Let's see what writing a function test-first looks like in Go. Suppose we're writing an old-fashioned text adventure game, like Zork, and we want the player to see something like this:

> *Attic*
> The attics, full of low beams and awkward angles, begin here in a relatively tidy area which extends north, south and east. You can see here a battery, a key, and a tourist map.

Adventure games usually contain lots of different locations and items, but one thing that's common to every location is that we'd like to be able to list its contents in the form of a sentence:

```
You can see here a battery, a key, and a tourist
map.
```

Suppose we're storing these items as strings, something like this:

```
a battery
a key
a tourist map
```

How can we take a bunch of strings like this and list them in a sentence, separated by commas, and with a concluding "and"? It sounds like a job for a function; let's call it `ListItems`.

What kind of test could we write for such a function? You might like to pause and think about this a little.

One way would be to call the function with some specific *inputs* (like the strings in our example), and see what it returns. We can predict what it should return when it's working properly, so we can compare that prediction against the actual result.

Here's one way to write that in Go, using the built-in `testing` package:

```go
func TestListItems_GivesCorrectResultForInput(t
*testing.T) {
    t.Parallel()
    input := []string{
        "a battery",
        "a key",
        "a tourist map",
    }
    want := "You can see here a battery, a key,
and a tourist map."
    got := game.ListItems(input)
    if want != got {
        t.Errorf("want %q, got %q", want, got)
    }
}
```

(Listing game/1)

Don’t worry too much about the details for now; we’ll deal with them later. The gist of this test is as follows:

1. We call the function `game.ListItems` with our test inputs.
2. We check the result against the expected string.
3. If they’re not the same, we call `t.Errorf`, which causes the test to fail.

Note that we’ve written this code as though the `game.ListItems` function already *exists*. It doesn’t. This test is, at the moment, an exercise in imagination. It’s saying *if* this function existed, here’s what we think it should return, given this input.

But it’s also interesting that we’ve nevertheless made a number of important design decisions as an unavoidable part of writing this test. First, we have to *call* the function,

so we’ve decided its name (`ListItems`), and what package it’s part of (`game`).

We’ve also decided that its parameter is a slice of strings, and (implicitly) that it returns a single result that is a string. Finally, we’ve encoded the exact behaviour of the function into the test (at least, for the given inputs), by specifying exactly what the function should produce as a result.

> *The original description of test-driven development was in an ancient book about programming. It said you take the input tape, manually type in the output tape you expect, then program until the actual output tape matches the expected output.*
> *When describing this to older programmers, I often hear, “Of course. How else could you program?”*
> —Kent Beck

Naming something and deciding its inputs, outputs, and behaviour are usually the hardest decisions to make about any software component, so even though we haven’t yet written a single line of code for `ListItems`, we’ve actually done some pretty important *thinking* about it.

And the mere fact of writing the test has also had a significant influence on the *design* of `ListItems`, even if it’s not very visible. For example, if we’d just gone ahead and written `ListItems` first, we might well have made it print the result to the terminal. That’s fine for the real game, but it would be difficult to test.

Testing a function like `TestItems` requires *decoupling* it from some specific output device, and making it instead a *pure function*: that is, a function whose result is deterministic, depends on nothing but its inputs, and has no side-effects.

Functions that behave like this tend to make a system easier to understand and reason about, and it turns out that there's a deep synergy between *testability* and good design, which we'll return to later in this book.

Verifying the test

So what's the next step? Should we go ahead and implement `ListItems` now and make sure the test passes? We'll do that in a moment, but there's a step we need to take first. We need some feedback on whether the *test* itself is correct. How could we get that?

It's helpful to think about ways the test could be *wrong*, and see if we can work out how to catch them. Well, one major way the test could be wrong is that it might not fail when it's supposed to.

Tests in Go pass by default, unless you explicitly make them fail, so a test function with no code at all would always pass, no matter what:

```
func TestAlwaysPasses(t *testing.T) {}
```

That test is so obviously useless that we don't need to say any more. But there are more subtle ways to accidentally write a useless test. For example, suppose we mistakenly wrote something like this:

```
if want != want {
    t.Errorf("want %q, got %q", want, got)
}
```

A value always equals itself, so this `if` statement will never be true, and the test will never fail. We might spot this just by looking at the code, but then again we might not.

I’ve noticed that when I teach Go to my students, this is a concept that often gives them trouble. They can readily imagine that the function itself might be wrong. But it’s not so easy for them to encompass the idea that the *test* could be wrong. Sadly, this is something that happens all too often, even in the best-written programs.

Until you’ve seen the test fail as expected, you don’t really have a test.

So we can’t be *sure* that the test doesn’t contain logic bugs unless we’ve seen it fail when it’s supposed to. When *should* the test fail, then? When `ListItems` returns the wrong result. Could we arrange that? Certainly we could.

That’s the next step, then: write just enough code for `ListItems` to return the wrong result, and verify that the test fails in that case. If it doesn’t, we’ll know we have a problem with the test that needs fixing.

Writing an incorrect function doesn’t sound too difficult, and something like this would be fine:

```
func ListItems(items []string) string {
    return ""
}
```

Almost everything here is dictated by the decisions we already made in the test: the function name, its parameter type, its result type. And all of these need to be there in order for us to call this function, even if we’re only going to implement enough of it to return the wrong answer.

The only real choice we need to make here, then, is what actual result to return, remembering that we want it to be *incorrect*.

What's the simplest incorrect string that we could return given the test inputs? Just the empty string, perhaps. Any other string would also be fine, provided it's not the one the test expects, but an empty string is the easiest to type.

Running tests with `go test`

Let's run the test and check that it does fail as we expect it to:

`go test`

```
--- FAIL: TestListItems_GivesCorrectResultForInput
(0.00s)
    game_test.go:18: want "You can see here a
battery, a key, and
    a tourist map.", got ""
FAIL
exit status 1
FAIL    game    0.345s
```

Reassuring. We *know* the function doesn't produce the correct result yet, so we expected the test to detect this, and it did.

If, on the other hand, the test had *passed* at this stage, or perhaps failed with some different error, we would know there was a problem. But it seems to be fine, so now we can go ahead and implement `ListItems` for real.

Here's one rough first attempt:

```
func ListItems(items []string) string {
    result := "You can see here"
    result += strings.Join(items, ", ")
    result += "."
```

```
    return result
}
```

(Listing game/1)

I really didn't think too hard about this, and I'm sure it shows. That's all right, because we're not aiming to produce elegant, readable, or efficient code at this stage. Trying to write code from scratch that's both correct *and* elegant is pretty hard. Let's not stack the odds against ourselves by trying to multi-task here.

In fact, the only thing we care about right *now* is getting the code correct. Once we have that, we can always tidy it up later. On the other hand, there's no point trying to beautify code that doesn't work yet.

> *The goal right now is not to get the perfect answer but to pass the test. We'll make our sacrifice at the altar of truth and beauty later.*
> —Kent Beck, "Test-Driven Development by Example"

Let's see how it performs against the test:

```
--- FAIL: TestListItems_GivesCorrectResultForInput
(0.00s)
    game_test.go:18: want "You can see here a
battery, a key, and
    a tourist map.", got "You can see herea
battery, a key, a
    tourist map."
```

Well, that looks *close*, but clearly not exactly right. In fact, we can improve the test a little bit here, to give us a more helpful failure message.

Using `cmp.Diff` to compare results

Since part of the result is correct, but part isn't, we'd actually like the test to report the *difference* between `want` and `got`, not just print both of them out.

There's a useful third-party package for this, [go-cmp](). We can use its `Diff` function to print just the differences between the two strings. Here's what that looks like in the test:

```
func TestListItems_GivesCorrectResultForInput(t
*testing.T) {
	t.Parallel()
	input := []string{
		"a battery",
		"a key",
		"a tourist map",
	}
	want := "You can see here a battery, a key,
and a tourist map."
	got := game.ListItems(input)
	if want != got {
		t.Error(cmp.Diff(want, got))
	}
}
```

([Listing game/2]())

Here's the result:

```
--- FAIL: TestListItems_GivesCorrectResultForInput
(0.00s)
    game_test.go:20:   strings.Join({
                "You can see here",
        -       " ",
                "a battery, a key,",
        -       " and",
```

```
                " a tourist map.",
        }, "")
```

When two strings differ, `cmp.Diff` shows which parts are the same, which parts are only in the first string, and which are only in the second string.

According to this output, the first part of the two strings is the same:

```
"You can see here",
```

But now comes some text that's only in the first string (`want`). It's preceded by a minus sign, to indicate that it's missing from the second string, and the exact text is just a space, shown in quotes:

```
-       " ",
```

So that's one thing that's wrong with `ListItems`, as detected by the test. It's not including a space between the word "here" and the first item.

The next part, though, `ListItems` got right, because it's the same in both `want` and `got`:

```
"a battery, a key,",
```

Unfortunately, there's something else present in `want` that is missing from `got`:

```
-       " and",
```

We forgot to include the final "and" before the last item. The two strings are otherwise identical at the end:

```
" a tourist map.",
```

You can see why it's helpful to show the *difference* between `want` and `got`: instead of a simple pass/fail test, we can see how close we're getting to the correct result. And if the result were very long, the diff would make it easy to pick out which parts of it weren't what we expected.

Let's make some tweaks to `ListItems` now to address the problems we detected:

```
func ListItems(items []string) string {
    result := "You can see here "
    result += strings.Join(items[:len(items)-1],
", ")
    result += ", and "
    result += items[len(items)-1]
    result += "."
    return result
}
```

(Listing game/3)

A bit ugly, but who cares? As we saw earlier, we're not trying to write beautiful code at this point, only correct code. This approach has been aptly named "Shameless Green":

> *The most immediately apparent quality of Shameless Green code is how very simple it is. There's nothing tricky here. The code is gratifyingly easy to comprehend. Not only that, despite its lack of complexity this solution does extremely well.*
> —Sandi Metz & Katrina Owen, "99 Bottles of OOP: A Practical Guide to Object-Oriented Design"

In other words, shameless green code passes the tests in the simplest, quickest, and most easily understandable way possible. That kind of solution may not be the *best*, as we've

said, but it may well be good enough, at least for now. If we suddenly had to drop everything and ship right now, we *could* grit our teeth and ship this.

So does `ListItems` work now? Tests point to yes:

`go test`

```
PASS
ok      game    0.160s
```

The test is passing, which means that `ListItems` is behaving correctly. That is to say, it's doing what we asked of it, which is to format a list of three items in a pleasing way.

New behaviour? New test.

Are we asking *enough* of `ListItems` with this test? Will it be useful in the actual game code? If the player is in a room with exactly three items, we can have some confidence that `ListItems` will format them the right way. And four or more items will probably be fine too.

What about just two items, though? From looking at the code, I'm not sure. It *might* work, or it might do something silly. Thinking about the case of *one* item, though, I can see right away that the result won't make sense.

The result of formatting a slice of *no* items clearly won't make sense either. So what should we do? We could add some code to `ListItems` to handle these cases, and that's what many programmers would do in this situation.

But hold up. If we go ahead and make that change, then how will we know that we got it right? We can at least have some confidence that we won't break the formatting for

three or more items, since the test would start failing if that happened. But we won't have any way to know if our new code correctly formats two, one, or zero items.

We started out by saying we have a specific job that we want `ListItems` to do, and we defined it carefully in advance by writing the test. `ListItems` now does that job, since it passes the test.

If we're now deciding that, on reflection, we want `ListItems` to do *more*, then that's perfectly all right. We're allowed to have new ideas while we're programming: indeed, it would be a shame if we didn't.

But let's adopt the rule "new behaviour, new test". Every time we think of a new behaviour we want, we have to write a test for it, or at least extend an existing passing test so that it fails for the case we're interested in.

That way, we'll be forced to get our ideas absolutely clear before we start coding, just like with the first version of `ListItems`. And we'll also know when we've written *enough* code, because the test will start passing.

This is another point that I've found my students sometimes have difficulty with. Often, the more experienced a programmer they are, the more trouble it gives them. They're so used to just going ahead and writing code to solve the problem that it's hard for them to insert an extra step in the process: writing a new *test*.

Even when they've written a function test-first to start with, the temptation is then to start extending the behaviour of that function, without pausing to extend the test. In that case, just saying "New behaviour, new test" is usually enough to jog their memory. But it can take a while to

thoroughly establish this new habit, so if you have trouble at first, you're not alone. Stick at it.

Test cases

We could write some new test functions, one for each case that we want to check, but that seems a bit wasteful. After all, each test is going to do exactly the same thing: call `ListItems` with some input, and check the result against expectations.

Any time we want to do the same operation repeatedly, just with different data each time, we can express this idea using a *loop*. In Go, we usually use the `range` operator to loop over some slice of data.

What data would make sense here? Well, this is clearly a slice of test *cases*, so what's the best data structure to use for each case?

Each case here consists of two pieces of data: the strings to pass to `ListItems`, and the expected result. Or, to put it another way, `input` and `want`, just like we have in our existing test.

One of the nice things about Go is that any time we want to group some related bits of data into a single value like this, we can just define some arbitrary `struct` type for it. Let's call it `testCase`:

```
func TestListItems_GivesCorrectResultForInput(t
*testing.T) {
    type testCase struct {
        input []string
        want  string
```

```
    }
    ...
```

(Listing game/4)

How can we refactor our existing test to use the new `testCase` struct type? Well, let's start by creating a slice of `testCase` values with just one element: the three-item case we already have.

```
...
cases := []testCase{
    {
        input: []string{
            "a battery",
            "a key",
            "a tourist map",
        },
        want:
        "You can see here a battery, a key, and a
tourist map.",
    },
}
...
```

(Listing game/4)

What's next? We need to loop over this slice of cases using range, and for each case, we want to pass its `input` value to `ListItems` and compare the result with its `want` value.

```
...
for _, tc := range cases {
    got := game.ListItems(tc.input)
    if tc.want != got {
        t.Error(cmp.Diff(tc.want, got))
```

```
    }
}
```

(Listing game/4)

This looks very similar to the test we started with, except that most of the test body has moved inside this loop. That makes sense, because we're doing exactly the same thing in the test, but now we can do it repeatedly for multiple *cases*.

This is commonly called a *table test*, because it checks the behaviour of the system given a table of different inputs and expected results. Here's what it looks like when we put it all together:

```
func TestListItems_GivesCorrectResultForInput(t
*testing.T) {
    type testCase struct {
        input []string
        want  string
    }
    cases := []testCase{
        {
            input: []string{
                "a battery",
                "a key",
                "a tourist map",
            },
            want:
            "You can see here a battery, a key,
and a tourist map.",
        },
    }
    for _, tc := range cases {
```

```
        got := game.ListItems(tc.input)
        if tc.want != got {
            t.Error(cmp.Diff(tc.want, got))
        }
    }
}
```

(Listing game/4)

First, let's make sure we didn't get anything wrong in this refactoring. The test should still pass, since it's still only testing our original three-item case:

```
PASS
ok      game    0.222s
```

Great. Now comes the payoff: we can easily add more cases, by inserting extra elements in the cases slice.

Adding cases one at a time

What new test cases should we add at this stage? We could add lots of cases at once, but since we feel pretty sure they'll all fail, there's no point in that.

Instead, let's treat each case as describing a new behaviour, and tackle one of them at a time. For example, there's a certain way the system should behave when given *two* inputs instead of three, and it's distinct from the three-item case. We'll need some special logic for it.

So let's add a single new case that supplies two items:

```
{
    input: []string{
        "a battery",
        "a key",
```

```
    },
    want: "You can see here a battery and a key.",
},
```

(Listing game/5)

The value of `want` is up to us, of course: what we want to happen in this case is a product design decision. This is what I've decided *I* want, with my game designer hat on, so let's see what `ListItems` actually does:

```
--- FAIL: TestListItems_GivesCorrectResultForInput
(0.00s)
    game_test.go:36:   strings.Join({
            "You can see here a battery",
        +   ",",
            " and a key.",
          }, "")
```

Not bad, but not perfect. It's inserting a comma after "battery" that shouldn't be there.

Now let's try to fix that. For three or more items, we'll always want the comma, and for two, one, or zero items, we won't. So the quickest way to get this test to pass is probably to add a special case to `ListItems`, when `len(items)` is less than 3:

```
func ListItems(items []string) string {
    result := "You can see here "
    if len(items) < 3 {
        return result + items[0] + " and " +
items[1] + "."
    }
    result += strings.Join(items[:len(items)-1],
", ")
```

```
    result += ", and "
    result += items[len(items)-1]
    result += "."
    return result
}
```

(Listing game/5)

Again, this isn't particularly elegant, nor does it need to be. We just need to write the minimum code to pass the current failing test case. In particular, we don't need to worry about trying to pass test cases we don't *have* yet, even if we plan to add them later:

Add one case at a time, and make it pass before adding the next.

The test passes for the two cases we've defined, so now let's add the one-item case:

```
{
    input: []string{
        "a battery",
    },
    want: "You can see a battery here.",
},
```

(Listing game/6)

Note the slightly different word order: "you can see here a battery" would sound a little odd.

Let's see if this passes:

```
--- FAIL: TestListItems_GivesCorrectResultForInput
(0.00s)
panic: runtime error: index out of range [1] with
```

```
length 1
[recovered]
```

Oh dear. `ListItems` is now panicking, so that's even worse than simply failing. In the immortal words of Wolfgang Pauli, it's [“not even wrong”].

Quelling a panic

Panics in Go are accompanied by a stack trace, so we can work our way through it to see which line of code is the problem. It's this one:

```
return result + items[0] + " and " + items[1] +
"."
```

This is being executed in the case where there's only one item (`items[0]`), so we definitely can't refer to `items[1]`: it doesn't exist. Hence the panic.

Let's treat the one-item list as another special case:

```
func ListItems(items []string) string {
    result := "You can see here "
    if len(items) == 1 {
        return "You can see " + items[0] + "
here."
    }
    if len(items) < 3 {
        return result + items[0] + " and " +
items[1] + "."
    }
    result += strings.Join(items[:len(items)-1],
", ")
    result += ", and "
    result += items[len(items)-1]
```

```
    result += "."
    return result
}
```

(Listing game/6)

This eliminates the panic, and the test now passes for this case.

Let's keep going, and add the zero items case. What should we expect `ListItems` to return?

```
{
    input: []string{},
    want:  "",
},
```

(Listing game/7)

Just the empty string seems reasonable. We could have it respond "You see nothing here", but it would be a bit weird to get that message every time you enter a location that happens to have no items in it, which would probably be true for most locations.

Running this test case panics again:

```
--- FAIL: TestListItems_GivesCorrectResultForInput
(0.00s)
panic: runtime error: index out of range [0] with
length 0 [recovered]
```

And we can guess what the problem is without following the stack trace: if `items` is empty, then we can't even refer to `items[0]`. Another special case:

```go
if len(items) == 0 {
    return ""
}
```

(Listing game/7)

This passes.

Refactoring

We saw earlier that it doesn't matter how elegant the code for `ListItems` looks, if it doesn't do the right thing. So now that it *does* do the right thing, we're in a good place, because we have options. If we had to ship right now, this moment, we could actually do that. We wouldn't be delighted about it, because the code is hard to read and maintain, but *users* don't care about that. What *they* care about is whether it solves their problem.

But maybe we don't have to ship right now. Whatever extra time we have in hand, we can now use to refactor this correct code to make it nicer. And, while there's always a risk of making mistakes or introducing bugs when refactoring, we have a safety net: the test.

The definition of "refactoring", by the way, is changing code without changing its *behaviour* in any relevant way. Since the test defines all the behaviour we consider relevant, we can change the code with complete freedom, relying on the test to tell us the moment the code starts *behaving* differently.

Since we have four different code paths, depending on the number of input items, we can more elegantly write that as a `switch` statement with four cases:

```
func ListItems(items []string) string {
    switch len(items) {
    case 0:
        return ""
    case 1:
        return "You can see " + items[0] + "
here."
    case 2:
        return "You can see here " + items[0] + "
and " +
            items[1] + "."
    default:
        return "You can see here " +
            strings.Join(items[:len(items)-1], ",
") +
            ", and " + items[len(items)-1] + "."
    }
}
```

(Listing game/8)

Did we break anything or change any behaviour? No, because the test still passes. Could we have written `ListItems` from the start using a `switch` statement, saving this refactoring step? Of course, but we've ended up here anyway, just by a different route.

In fact, all good programs go through at least a few cycles of refactoring. We shouldn't even try to write the final program in a single attempt. Instead, we'll get much better results by aiming for *correct* code first, then iterating on it a few times to make it clear, readable, and easy to maintain.

> *Writing is basically an iterative process. It is a rare writer who dashes out a finished piece; most of us work*

> *in circles, returning again and again to the same piece of prose, adding or deleting words, phrases, and sentences, changing the order of thoughts, and elaborating a single sentence into pages of text.*
> —Dale Dougherty & Tim O'Reilly, [“Unix Text Processing”]

Well, that was easy

No doubt there's more refactoring we could do here, but I think you get the point. We've developed some correct, reasonably readable code, with accompanying tests that completely define the behaviour users care about.

And we did it without ever having to really *think* too hard. There were no brain-busting problems to solve; we didn't have to invent any complicated algorithms previously unknown to computer science, or that you might be tested on in some job interview. We didn't use any advanced features of Go, just basic strings, slices, and loops.

That's a good thing. If code is hard to write, it'll be hard to understand, and even harder to debug. So we *want* the code to be easy to write.

> *Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?*
> —Brian Kernigan & P.J. Plauger, “The Elements of Programming Style”

If we find that writing the code is hard, we'll reduce the scope of our ambition so that we're solving some simpler problem instead. We'll keep simplifying the problem in this way until the code becomes easy and obvious. Then we can gradually build back up to the real problem we started with.

The key point is that we're always writing *code*, always testing our evolving ideas against a running program, instead of getting trapped into doing "big design up front": that never works.

> *My way of writing code is, you sculpt it, you get something as good as you can, and everything's subject to change, always, as you learn. But you climb this ladder of learning about your problem. Every problem's unique, so you have to learn about each problem, and you do something and get a better vantage point. And from that vantage point you can decide to throw it out. Code is cheap. But often it tells you what to do next.*
> —Andy Herzfeld, quoted in Scott Rosenberg's "Dreaming in Code"

At every stage in this process, we had the confidence that comes from having a good test. It caught our occasional missteps, showed us from moment to moment how close we were getting to the correct solution, and what still remained to be done.

As we generated new ideas during development, it was easy to add them as new test cases, and we only had to do a very little work to make them pass. Once we had all the cases passing, we were able to confidently refactor the entire function, without worrying about breaking anything or introducing subtle bugs.

If you're not used to this way of programming with tests, it might all seem a bit long-winded for a relatively simple function. But the process is a lot quicker to *do* than it is to explain step by step.

In a real-life programming situation, it would probably take me a couple of minutes to write this test, two or three

minutes to get it passing, and maybe another couple of minutes to refactor the code. Let’s generously say ten minutes is a reasonable time to build something like `ListItems` using this workflow.

Could we do it faster if we didn’t bother about the test? Perhaps. But *significantly* faster? I doubt it. To write a working `ListItems` from scratch that handles all these cases would take me a good ten minutes, I think. Indeed, it would actually be harder work, because I’d have to *think* very carefully at each stage and painstakingly work out whether the code is going to produce the correct result.

Even then, I wouldn’t be completely confident that it was correct without seeing it *run* a few times, so I’d have to go on and write some throwaway code just to call `ListItems` with some inputs and print the result. And it probably *wouldn’t* be correct, however carefully and slowly I worked. I would probably have missed out a space, or something. So I’d have to go back and fix that.

> *It will feel slow at first. The difference is, when we’re done, we’re really done. Since we’re getting closer to really done, the apparent slowness is an illusion. We’re eliminating most of that long, painful test-and-fix finish that wears on long after we were supposed to be done.*
> —Ron Jeffries, “The Nature of Software Development”

In other words, the test-first workflow isn’t slow at all, once you’re familiar with it. It’s quick, enjoyable, and productive, and the result is correct, readable, self-testing code.

Watching an experienced developer build great software fast, guided by tests, can be a transformative experience:

> *I grew a reporting framework once over the course of a few hours, and observers were absolutely certain it was*

a trick. I must have started with the resulting framework in mind. No, sorry. I’ve just been test driving development long enough that I can recover from most of my mistakes faster than you can recognize I’ve made them.
—Kent Beck, “Test-Driven Development by Example”

Sounds good, now what?

Maybe you’re intrigued, or inspired, by the idea of programming with confidence, guided by tests. But maybe you still have a few questions. For example:

- How does Go’s `testing` package work? How do we write tests to communicate intent? How can we test error handling? How do we come up with useful test data? What about the test cases we didn’t think of?

- What if there are bugs still lurking, even when the tests are passing? How can we test things that seem untestable, like concurrency, or user interaction? What about command-line tools?

- How can we deal with code that has too many dependencies, and modules that are too tightly coupled? Can we test code that talks to external services like databases and network APIs? What about testing code that relies on time? And are mock objects a good idea? How about assertions?

- What should we do when the codebase has no tests at all? For example, legacy systems? Can we refactor and test them in safe, stress-free ways? What if we get in trouble with the boss for writing tests? What if the existing tests are no good? How can we improve testing through code review?

- How should we deal with tests that are optimistic, persnickety, over-precise, redundant, flaky, failing, or just slow? And how can tests help us improve the design of the system overall?

Well, I’m glad you asked. Enjoy the rest of the book.