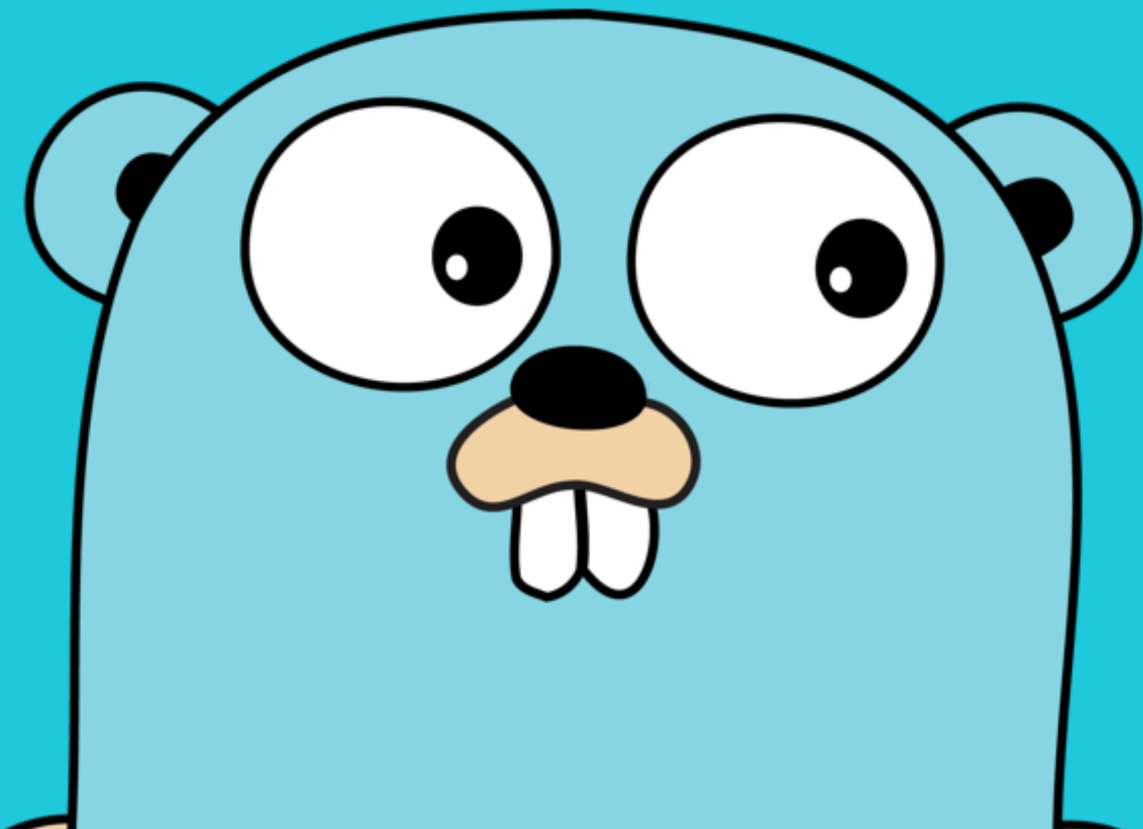


Production Go

Build modern, production-ready systems in Go



Herman Schaaf • Shawn Smith

Production Go

Build modern, production-ready web services in Go

Herman Schaaf and Shawn Smith

This book is for sale at <http://leanpub.com/productiongo>

This version was published on 2022-06-24



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2013 - 2022 Herman Schaaf and Shawn Smith

Contents

Introduction	i
Installing Go	1
Installation	1
Writing Code	1
Editor Integrations	3
(Optional) Linters and Correctness Tooling	5
Basics	7
Program Structure	7
Variables and Constants	9
Basic Data Types	12
Structs	17
Operators	19
Conditional Statements	22
Arrays	23
Slices	24
Maps	27
Loops	29
Functions	32
Exported Names	32
Pointers	33
Goroutines	34
Channels	36
Interfaces	39
Error Handling	42
Reading Input	43
Writing Output	44
Style and Error Handling	46
Style	46
Error Handling	50
Wrapping Up	51
Strings	52

CONTENTS

Appending to Strings	52
Splitting strings	54
Counting and finding substrings	55
Advanced string functions	57
Ranging over a string	61
Supporting Unicode	62
A very brief history of string encodings	62
Strings are byte slices	63
Printing strings	64
Runes and safely ranging over strings	66
Text Normalization	67
Summary	71
Concurrency	73
sync.WaitGroup	74
errgroup	76
Channels	78
Goroutines in web handlers	80
Pollers	81
Race conditions	83
Testing	89
Why do we need tests?	89
Writing Tests	90
Testing HTTP Handlers	94
Mocking	96
Generating Coverage Reports	100
Writing Examples	104
Benchmarks	108
A simple benchmark	108
Comparing benchmarks	112
Resetting benchmark timers	113
Benchmarking memory allocations	114
Modulo vs Bitwise-and	117
Tooling	120
Godoc	120
Go Guru	121
Race Detector	122
Go Report Card	134
Staticcheck	135
Go-fuzz	136

CONTENTS

govalidate	138
Pinning tools with Go modules	138
Security	140
Keep Go Up to Date	140
CSRF	140
HTTP Strict Transport Security (HSTS)	141
Content Security Policy (CSP)	143
SQL Injection	143
See Also	144
Going into Production	145
Continuous Integration & Continuous Deployment (CI/CD)	145
Deployment	146
Logging	146
Monitoring	146
Alerting	147
Further Reading	148
Acknowledgements	149
Licenses	149

Introduction

Why Go in Production?

If you are reading this book, we assume you are interested in running Go in a production environment. Maybe you dabble in Go on your side projects, but are wondering how you can use it at work. Or perhaps you've read a company blog post about converting their codebase to Go, which now has 3 times less code and response times one tenth of what they were before. Your mileage will vary when it comes to gains in productivity and efficiency, but we have generally found a switch to Go to be more than worthwhile. Our goal in writing this book is to provide the knowledge to write a production-ready service in Go. This means not only writing the initial implementation, but also reliably deploying it, monitoring its performance, and iterating on improvements.

Go is a language that allows for fast iteration, which goes well with continuous deployment. Although it is a statically typed language, it compiles quickly and can often be used as a replacement for scripting languages like Python. Many users report that when writing Go, once a program works, it continues to “just work”. This is due to the relatively simple design of the language, and the focus on readability rather than clever constructs.

In one project, we replaced existing APIs in PHP with equivalent functionality in Go. We saw performance improvements, including an order of magnitude reduction in response times, which led to both higher user retention and a reduction in server costs. We also saw developer happiness increase, because the safety guarantees in Go meant we could deploy changes regularly and safely.

This book is not meant for beginner programmers. We expect our audience to be knowledgeable of basic computer science topics and software engineering practices. Over the years, we have helped ramp up countless engineers who had no prior experience writing Go. Ideally this will be the book that people recommend to engineers writing Go for the first time, and who want to better understand the “right way” to write Go.

We hope this book will help guide you on your journey to running Go in production. It will cover many important aspects of running a production system, including topics not covered by most books on the language, like profiling the memory usage of a Go program, deploying and monitoring apps written in Go, and writing tests for web applications.

Feel free to skip around to chapters that seem more relevant to your immediate concerns or interests. We will do our best to keep the chapters fairly independent of one another in order to make that possible.

Installing Go

Installation

The [Go Downloads page](https://golang.org/dl/)¹ contains binary distributions of Go for Windows, Apple macOS, and Linux.

You can also find instructions for installing Go from source on their [Installing Go from source](https://golang.org/doc/install/source)² page. We recommend installing a binary release first, as there are extra steps necessary to install from source. You only need to install from source if you're planning to contribute changes upstream to the Go language.

Also, if you use Homebrew on macOS, you can install with `brew install go`.

Once installed, test the command by running:

```
$ go version
```

and you should see something like:

```
go version go1.15.6 darwin/amd64
```

Writing Code

You can write Go code anywhere on your filesystem. Let's assume that you have a folder in your home directory called `~/code`. Let's also assume that you use GitHub for hosting your source code. Make a new folder in `~/code` called `myapp`, then run:

```
go mod init github.com/[yourusername]/myapp.
```

You should have a `go.mod` file now with your package name in it. Now make a `main.go` file with the following contents:

¹<https://golang.org/dl/>

²<https://golang.org/doc/install/source>

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     fmt.Println("Hello, world")
7 }
```

and run it:

```
go run main.go
```

You should see the “Hello, world” output.

Next, let’s add a dependency with `go get`.

When you retrieve a package using the `go get` command, the source is downloaded into your `$GOPATH/pkg/mod` dir (note: if you haven’t set the `$GOPATH` environment variable explicitly, Go will use `$HOME/go` as your `$GOPATH`). A line for that import is also added to your `go.mod` file. For example, if you want to download the popular URL router library [gorilla/mux](https://github.com/gorilla/mux)³, you can run:

The `go get` command downloads the package’s source into your `$GOPATH/pkg/mod` dir (note: if you haven’t set the `$GOPATH` environment variable explicitly, Go will use `$HOME/go` as your `$GOPATH`). A line for that import is also added to your `go.mod` file. For example, if you want to download the popular URL router library [gorilla/mux](https://github.com/gorilla/mux)⁴, run:

```
go get github.com/gorilla/mux
```

and Go will download the source into (assuming the latest version is 1.8.0) `$GOPATH/pkg/github.com/gorilla/mux@v1.8.0`. You can now import that library in your code:

```
1 package main
2
3 import (
4     "fmt"
5     "net/http"
6
7     "github.com/gorilla/mux"
8 )
9
10 func HomeHandler(w http.ResponseWriter, r *http.Request) {
11     fmt.Fprintf(w, "Hello, world")
12 }
```

³<https://github.com/gorilla/mux>

⁴<https://github.com/gorilla/mux>


```
13
14 func main() {
15     r := mux.NewRouter()
16     r.HandleFunc("/", HomeHandler)
17     http.Handle("/", r)
18     http.ListenAndServe(":8080", r)
19 }
```

and you can run `go run main.go` again and `curl localhost:8080` should respond with `Hello, world`.

Your `go.mod` should now look like this:

```
1 module github.com/[yourusername]/myapp
2
3 go 1.15
4
5 require github.com/gorilla/mux v1.8.0
```

If you prefer to vendor your dependencies, you can run `go mod vendor` and Go will create a `vendor/` directory for you with the appropriate dependencies. When the Go toolchain recognizes that you have a `vendor/` directory it will choose to use that during compilation.

Editor Integrations

Since there are so many editors and IDEs out there, we will only briefly cover three common editors: Goland by JetBrains, vim and Sublime Text. We will link to relevant information for other popular editors at the end of the section.

For all editors, we recommend running `goimports` on save. Like the `gofmt` command, `goimports` formats your Go code, but it also adds or removes imports. If you reference `mux.NewRouter` in your code but have not yet imported `github.com/gorilla/mux`, `goimports` will find the package and import it. It also works for the standard library, which is especially useful for functions like `fmt.Println`. It is worthwhile to ensure your editor has the `goimports` plugin: it will speed up your Go development process.

Goimports can be installed with `go get`:

```
1 go get golang.org/x/tools/cmd/goimports
```

We now discuss some useful plugins for three common editors, in (arguably!) decreasing order of complexity.

GoLand

GoLand is a powerful and mature IDE for Go. It features code completion, the ability to jump to variable and type definitions, standard debugging tools like run-time inspection and setting break points, and more. A free 30-day trial can be downloaded from <https://www.jetbrains.com/go/>

Once installed, we recommend installing `goimports` on save. To do this, go to File -> Settings -> Tools -> File Watchers. Click the Plus (+) icon, and select “goimports”. Press OK. When you now save a Go file in the project, it should get formatted according to Go standards automatically.

Sublime Text

For Sublime Text 3, we recommend installing a package called [GoSublime](#)⁵. Sublime Text package control can install this package. Install package control with the commands provided at <https://packagecontrol.io/installation>. To open package control, press Ctrl+Shift+P on Windows and Linux, or Cmd+Shift+P on OS X. Then type “install package” and select “Package control: install package”. Now type “GoSublime” and choose the matching option. Finally, open the GoSublime settings by going to Preferences -> Package Settings -> GoSublime -> Settings-User. Make sure that GOPATH matches the path you configured earlier. Here are some typical settings:

```
1 {
2     // you may set specific environment variables here
3     // e.g "env": { "PATH": "$HOME/go/bin:$PATH" }
4     // in values, $PATH and ${PATH} are replaced with
5     // the corresponding environment(PATH) variable, if it exists.
6     "env": {"GOPATH": "$HOME/Code/go", "PATH": "$GOPATH/bin:$PATH" },
7
8     "fmt_cmd": ["goimports"]
9 }
```

vim

For vim, you will want to install [vim-go](#)⁶. Instructions can be found on that page for the various vim package managers.

Once you have installed vim-go, you can add the following line to your `.vimrc` file in order to run `goimports` on save:

```
let g:go_fmt_command="goimports"
```

⁵<https://github.com/DisposaBoy/GoSublime>

⁶<https://github.com/fatih/vim-go>

(Optional) Linters and Correctness Tooling

Linters and other types of tooling can help you catch common mistakes in Go code more quickly. By tightening the feedback loop between writing code and finding a bug, having such tools part of your regular workflow will also accelerate your journey to becoming an experienced Go programmer.

To install a suite of linters and tools to run on your code in one simple step, we recommend [golangci-lint](https://golangci-lint.run/usage/install/#local-installation)⁷. For installation instructions in your local environment, see <https://golangci-lint.run/usage/install/#local-installation>⁸.

golangci-lint allows you to run a variety of linters and tooling and conveniently combines all of their output into a standard format. We've found the `deadcode`, `ineffassign`, `staticcheck` and `misspell` (disabled by default, enable with `--enable=misspell`) linters to be particularly useful.

To run the default set of linters, run:

```
$ golangci-lint run
```

You may also want to disable all but a specific linter, as in the following example where we run only `errcheck`:

```
1 golangci-lint run --disable-all -E errcheck
```

While you may not want to require all the linters to pass in your build, one tool we recommend requiring is [go vet](https://golang.org/cmd/vet/)⁹. Using golangci-lint, it can be run like this:

```
1 golangci-lint run --disable-all -E govet
```

Go vet is a tool concerned with code correctness. It will find problems in your code such as using the wrong string formatting verb in a call to `Printf`:

```
1 package main
2
3 import (
4     "fmt"
5 )
6
7 func main() {
8     fmt.Printf("%d", "test")
9 }
```

This code compiles and runs, but if you do actually run it you'll see this:

⁷<https://github.com/golangci/golangci-lint>

⁸<https://golangci-lint.run/usage/install/#local-installation>

⁹<https://golang.org/cmd/vet/>

```
$ go run main.go
%!d(string=test)%
```

because `%d` is meant for printing integers, not strings. If you were to run `go vet` on the above code (or equivalently through `golanci-lint run`), you would see this warning:

```
$ go vet main.go
main.go:8: arg "test" for printf verb %d of wrong type: string
exit status 1
```

Another common issue `vet` will catch for you is the use of `printf` verbs inside a call to `Println`:

```
1 package main
2
3 import (
4     "fmt"
5 )
6
7 func main() {
8     fmt.Println("%d", "test")
9 }
```

Again this will compile and run fine, but the output would be this:

```
$ go run main.go
%d test
```

Calling `go vet` on this code will tell you:

```
$ go vet main.go
main.go:8: possible formatting directive in Println call
exit status 1
```

Basics

This chapter gives a quick run-through of Go's basic syntax, and the features that differentiate it from other languages. If you have never programmed in Go before, this chapter will give you the knowledge to start reading and writing simple Go programs. Even if you have programmed in Go before, we recommend that you still read this chapter. Through the examples, we will highlight common pitfalls in Go programs, and answer some questions that even experienced Go programmers might still have.

Program Structure

Go code is organized in packages containing one or more Go source files. When building an executable, we put our code into a package `main` with a single `func main`.

As mentioned in the Installation chapter, our Go code lives in `GOPATH`, which we're saying is `$HOME/go`. Let's say we want to write our first Go command, which randomly selects an item from a list and outputs it to `stdout`. We need to create a directory in our `GOPATH`, with a `main.go` file containing a package `main` and single `main` function:

```
$ mkdir -p $GOPATH/src/github.com/prodgopher/dinner
$ cd $GOPATH/src/github.com/prodgopher/dinner
```

In this directory we'll add our `main.go` with the following code:

Random dinner selection

```
1 package main
2
3 import (
4     "fmt"
5     "math/rand"
6     "time"
7 )
8
9 func main() {
10     dinners := []string{
11         "tacos",
12         "pizza",
```

```

13         "ramen",
14     }
15     rand.Seed(time.Now().Unix())
16     fmt.Printf("We'll have %s for dinner tonight!\n", dinners[rand.Intn(len(dinn\
17 ers))])
18 }

```

We can run our code with `go run main.go`. The other option is to build our code as an executable and run that. To do that, we first run `go build` to make sure that the code compiles. Then we run `go install`:

```

$ go build
$ go install
$ cd $GOPATH/bin
$ ./dinner
We'll have pizza for dinner tonight!

```

We can also run these commands from outside of our GOPATH, like so:

```

$ go build github.com/prodgopher/dinner
$ go install github.com/prodgopher/dinner

```

If we want to expose this functionality in a package so we can reuse it in other places, we need to add this functionality to a package. Let's say we want to return the name of the dinner, rather than the whole string, like "pizza", "ramen", etc. For convenience, let's reuse the same `github.com/prodgopher/dinner` directory. Remove the `main.go` file and create a new `dinner.go` file that looks like this:

Random dinner selection package

```

1 package dinner
2
3 import (
4     "math/rand"
5     "time"
6 )
7
8 func Choose() string {
9     dinners := []string{
10         "tacos",
11         "pizza",

```

```
12         "ramen",
13     }
14     rand.Seed(time.Now().Unix())
15
16     return dinners[rand.Intn(len(dinners))]
17 }
```

Now, somewhere outside of our `dinner` package directory (let's use our home folder), we'll invoke our new functionality in a file called `main.go`:

Using our random dinner selection package

```
1 package main
2
3 import (
4     "fmt"
5
6     "github.com/prodgopher/dinner"
7 )
8
9 func main() {
10     fmt.Println(dinner.Choose())
11 }
```

```
$ go run main.go
tacos
```

Now we have a convenient package for randomly selecting what to eat for dinner.

Variables and Constants

There are multiple ways to declare variables in Go. The first way, declaring a var with a given type, can be done like so:

```
var x int
```

With this type of declaration, the variable will default to the type's zero value, in this case 0.

Another way to declare a variable is like this:

```
var x = 1
```

Similar to the above method, but in this case we can declare the specific contents. The type is also implied.

Lastly, the short-hand variable declaration:

```
x := 1
```

This is probably the most common way, and the type is also implied like the above. Sometimes the `var` declaration method is used stylistically to indicate that the variable will be changed soon after the declaration. For example:

```
1 var found bool
2 for _, x := range entries {
3     if x.Name == "Gopher" {
4         found = true
5     }
6 }
```

One key difference between `var` and `:=` declarations is that the shorthand version (`:=`) cannot be used outside of a function, only `var` can. This means that variables in the global scope must be declared using `var`. This is valid:

Example using `var` to declare variable in global scope

```
1 package main
2
3 import "fmt"
4
5 var a = 1
6
7 func main() {
8     fmt.Println(a)
9 }
```

but this will not compile:

It is invalid to use shorthand variable declaration in the global scope

```
1 package main
2
3 import "fmt"
4
5 a := 1 // this is invalid, use var instead
6
7 func main() {
8     fmt.Println(a)
9 }
```

Running the above, we get the following error:

```
$ go run var_bad.go
# command-line-arguments
./var_bad.go:5: syntax error: non-declaration statement outside function body
```

The other subtle difference between `var`-declarations and shorthand-declarations occur when declaring multiple variables. The following code is valid,

```
1 var a, b = 0, 1 // declare some variables
2 b, c := 1, 2    // this is okay, because c is new
3 fmt.Println(a, b, c) // Outputs: 0, 1, 2
```

but this is not valid:

```
1 var a, b = 0, 1 // declare some variables
2 var b, c = 1, 2 // this is not okay, because b already exists
3 fmt.Println(a, b, c)
```

The second example, when placed into a program, fails to compile:

```
./var_shorthand_diff2.go:7: b redeclared in this block
    previous declaration at ./var_shorthand_diff2.go:6
```

This is because the shorthand `:=` may redeclare a variable if at least *one* of the variables to its left is new. The `var` declaration may not redeclare an existing variable.

Basic Data Types

Basic Types

Go supports the following basic types:

- bool
- string
- int8, int16, int32, int64, int
- uint8, uint16, uint32, uint64, uint
- float32, float64
- complex64, complex128
- byte (alias for uint8)
- rune (alias for int32)

Booleans

The bool type represents a boolean and is either `true` or `false`.

Usage of booleans in Go

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     a, b := true, false
7     c := a && b
8     d := a || b
9
10    fmt.Println("a:", a)
11    fmt.Println("b:", b)
12    fmt.Println("c:", c)
13    fmt.Println("d:", d)
14
15    // Output: a: true
16    // b: false
17    // c: false
18    // d: true
19 }
```

In the above example we first create `a` and `b`, and assign them the values `true` and `false`, respectively. `c` is assigned the value of the expression `a && b`. The `&&` operator returns `true` when both `a` and `b` are `true`, so in this case `c` is `false`. The `||` operator returns `true` when either `a` or `b` are `true`, or both. We assign `d` the value of `a || b`, which evaluates to `true`.

Note that unlike some other languages, Go does not define `true` or `false` values for data types other than `bool`.

Strings

The `string` type represents a collection of characters. When defined in code, a string is a piece of text surrounded by double quotes (`"`). Let's write a simple program using strings.

Usage of booleans in Go

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     sound := "meow"
7     sentence := "The cat says " + sound + "."
8     fmt.Println(sentence)
9 }
```

The example demonstrates that strings support the `+` operator for concatenation. The variable `sentence` contains a concatenation of the strings `"The cat says "`, `"meow"`, and `"."`. When we print it to the screen, we get `The cat says meow.`

This only scratches the surface of strings in Go. We will discuss strings in more depth in the chapter on [Strings](#).

Integers

The integer types can be divided into two classes, signed and unsigned.

Signed integers

The signed integer types are `int8`, `int16`, `int32`, `int64`, and `int`. Being signed, these types store both negative and positive values, but up to a maximum half the value of its `uint` counterpart. `int8` uses 8 bits to store values between -128 and 127 (inclusive). `int16` stores values in the range -32,768 to 32,767. `int32` stores values in the range -2,147,483,648 to 2,147,483,647. `int64` stores values in the range -2^{63} to $2^{63}-1$, which is to say, between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807.

Unlike the other signed integer types, the `int` type does not explicitly state its width. This is because it acts as an alias for either `int32` or `int64`, depending on the architecture being compiled to. This means that it will perform optimally on either architecture, and it is the most commonly used integer type in Go code.

Go does not allow implicit type conversions. When converting between integer types, an explicit type cast is required. For example, see the following code:

Type mixing that will result in a compile-time error

```
1 package main
2
3 import (
4     "fmt"
5 )
6
7 func main() {
8     var i32 int32 = 100
9     var i64 int64 = 100
10
11     // This will result in a compile-time error:
12     fmt.Println(i32 + i64)
13 }
```

This results in a compile-time error:

```
1 $ go run type_mix.go
2 type_mix.go:12:18: invalid operation: i32 + i64 (mismatched types int32 and i\
3 nt64)
```

To fix the error, we can either use the same types from the start, or do a type cast. We will discuss type casts again later in this chapter, but here is how we might use a type cast to solve the problem:

Using an integer type cast

```
1 package main
2
3 import (
4     "fmt"
5 )
6
7 func main() {
8     var i32 int32 = 100
```

```
9      var i64 int64 = 100
10
11      fmt.Println(int64(i32) + i64)
12      // Output: 200
13 }
```

A nice feature of the Go compiler is that it warns us if a constant value overflows the integer type it is being assigned to:

Putting an overflowing constant into an integer type

```
1 package main
2
3 import (
4     "fmt"
5 )
6
7 func main() {
8     var i int8 = 128
9     fmt.Println(i)
10 }
```

```
$ go run int.go
int.go:8:15: constant 128 overflows int8
```

Here we try to assign a number that is one bigger than the maximum value of `int8`, and the compiler prevents us from doing so. This is neat, but as we will see in the next section on [Unsigned integers](#), the compiler will not save us from calculations which result in over- or underflow.

Unsigned integers

Under unsigned integers, we have `uint8`, `uint16`, `uint32`, `uint64`, and `uint`. `uint8` uses 8 bits to store a non-negative integer in the inclusive range 0 to 255. That is, between 0 and 2^8-1 . Similarly, `uint16` uses 16 bits and stores values in the inclusive range 0 to 65,535, `uint32` uses 32 bits to store values from 0 to 4,294,967,295, and `uint64` uses 64 bits to store values from 0 to $2^{64}-1$. The `uint` type is an alias for either `uint32` or `uint64`, depending on whether the code is being compiled for a 32-bit architecture or a 64-bit architecture.

`uints` are useful when the values to be stored are always positive. However, take special care before deciding to use `uint`. Go strictly enforces types, so `uint` requires a cast to be used with `int`. Built-in slice functions, like `len`, `cap`, and almost all library functions, return `int`. So using those functions with `uint` will require explicit type casts, which can be both inefficient and hard to read. Furthermore, underflow is a common enough problem with the `uint` type that it's worth showing an example of how it can happen:

An example of uint underflow

```
1 package main
2
3 import (
4     "fmt"
5 )
6
7 func main() {
8     var p, s uint32 = 0, 1
9     fmt.Printf("p - s = %d - %d = %d", p, s, p-s)
10    // Output: p - s = 0 - 1 = 4294967295
11 }
```

Running this, we get:

```
$ go run uint.go
p - s = 0 - 1 = 4294967295
```

As this example illustrates, if we are not careful when subtracting from the `uint` type, we can run into underflow and get a large positive value instead of `-1`. Be aware of the limitations before choosing unsigned integer types.

Floating point numbers

For floating point numbers we have two types, `float32` and `float64`. A `float32` represents a 32-bit floating-point number as described in the IEEE-754 standard, and `float64` represents a 64-bit floating-point number.

An integer can be converted to a floating-point number with a type conversion:

```
1 x := 15
2 y := float64(x)
```

This will be especially useful when using the `math` package, as the functions of that package typically work with `float64` (for example, `math.Mod` and `math.Abs` both take `float64`).

Complex numbers

Complex numbers are expressed by the types `complex64` and `complex128`. A `complex64` number is a `float32` with real and imaginary parts, and a `complex128` is a `float64` with real and imaginary parts. Creating a complex number is done like so:

```
1 x := complex(1.0, 2.0)
2 fmt.Println(x)
```

```
1 (1+2i)
```

There are built-in functions to extract the real and imaginary parts of a complex number:

```
1 x := complex(1.0, 2.0)
2 fmt.Println(real(x))
3 fmt.Println(imag(x))
```

```
1 1
2 2
```

You can express an imaginary literal by appending `i` to a decimal of float literal:

```
1 fmt.Println(1.3i)

1 (0+1.3i)
```

Structs

A struct is a collection of fields, which can be declared with the `type` keyword:

Struct example

```
1 package main
2
3 import "fmt"
4
5 type Person struct {
6     Name string
7     Email string
8 }
9
10 func main() {
11     p := Person{"Robert", "robert@example.com"}
12
13     fmt.Println(p.Name)
14 }
```

Struct fields are accessed with a dot, so our above example should print:

Robert

Since structs are commonly used for reading and writing data formats such as JSON, there are struct tags which define how you would like your fields to be decoded or encoded in that data format. Here is an example of JSON struct tags:

Struct tags example

```
1 package main
2
3 import (
4     "encoding/json"
5     "fmt"
6     "log"
7 )
8
9 type Person struct {
10     Name string `json:"name"`
11     Email string `json:"email"`
12 }
13
14 func main() {
15     p := Person{"Robert", "robert@example.com"}
16     b, err := json.Marshal(p)
17     if err != nil {
18         log.Fatal(err)
19     }
20
21     fmt.Println(string(b))
22 }
```

Running this code will output:

```
{"name": "Robert", "email": "robert@example.com"}
```

If we didn't have the struct tags, then we would have:

```
{"Name": "Robert", "Email": "robert@example.com"}
```

since it isn't very common to see the first letter of a field capitalized like that in JSON, we use the struct tags to define how we want our struct fields to be encoded.

One important note about struct field names and JSON: field names must be exported (first letter of field name must be capitalized) in order for encoding to JSON to work. If our struct field names were `name` and `email`, we would get an empty JSON object when marshalling.

Golang also supports anonymous structs, which can be commonly found in table-driven tests for example. You can see some examples in our Testing chapter, but here is a quick (not testing-related) example to show how it works:

Anonymous struct example

```
1 package main
2
3 import (
4     "encoding/json"
5     "fmt"
6     "log"
7 )
8
9 func main() {
10     p := struct {
11         Name string `json:"name"`
12         Email string `json:"email"`
13     }{
14         Name: "Robert",
15         Email: "robert@example.com",
16     }
17     b, err := json.Marshal(p)
18     if err != nil {
19         log.Fatal(err)
20     }
21
22     fmt.Println(string(b))
23 }
```

This will print the same as our “Struct tags example” example above.

Operators

Operators in Go are very similar to other languages in the C-family. They can be divided into five broad categories: arithmetic operators, comparison operators, logical operators, address operators and the receive operator.

Arithmetic Operators

Arithmetic operators apply to numeric values. From the Go specification:

Arithmetic operators apply to numeric values and yield a result of the same type as the first operand. The four standard arithmetic operators (+, -, *, /) apply to integer, floating-point, and complex types; + also applies to strings. The bitwise logical and shift operators apply to integers only.

The following table summarizes the different arithmetic operators and when they may be applied:

+	sum	ints, floats, complex values, strings
-	difference	ints, floats, complex values
*	product	ints, floats, complex values
/	quotient	ints, floats, complex values
%	remainder	ints
&	bitwise AND	ints
	bitwise OR	ints
^	bitwise XOR	ints
&^	bit clear (AND NOT)	ints
<<	left shift	int << uint
>>	right shift	int >> uint

Comparison Operators

Comparison operators compare two operands and yield a boolean value. The comparison operators are:

==	equal
!=	not equal
<	less
<=	less or equal
>	greater
>=	greater or equal

In any comparison, the first operand must be assignable to the type of the second operand, or vice versa. Go is strict about types, and it is invalid to use a comparison operator on two types that are not comparable. For example, this is valid:

```
var x int = 1
var y int = 2

// eq will be false
var eq bool = x == y
```

but this is not valid, and will result in a compile-time type error:

```
var x int = 1
var y int32 = 2

// error: mismatched types int and int32
var eq bool = x == y
```

Logical Operators

Logical operators apply to boolean values and yield a boolean result.

&&	conditional AND	p && q	is	"if p then q else false"
	conditional OR	p q	is	"if p then true else q"
!	NOT	!p	is	"not p"

As in C, Java, and JavaScript, the right operand of && and || is evaluated conditionally.

Address Operators

Go has two address operators: the address operation, &, and the pointer indirection, *.

&x returns a pointer to x. The pointer will be a pointer of the same type as x. A run-time panic will occur if the address operation is applied to an x that is not addressable.

```
1 var x int
2 var y *int = &x
3
4 // Print the memory address of x,
5 // e.g. 0x10410020
6 fmt.Println(y)
```

When x is a pointer, *x denotes the variable pointed to by x. If x is nil, an attempt to evaluate *x will cause a run-time panic.

```
1 var x *int = nil
2 *x    // causes a run-time panic
```

The Receive Operator

The receive operator, `<-` is a special operator used to receive data from a channel. For more details on this, see [channels](#).

Conditional Statements

We've seen some simple `if` statements in previous sections' code snippets. Here we'll cover some other common uses of conditional statements in Go.

An `if` can contain a variable declaration before moving on to the condition. This can often be seen in tests, like in this example from the Go source code (`bytes/reader_test.go`):

```
1 if got := string(buf); got != want {
2     t.Errorf("ReadAll: got %q, want %q", got, want)
3 }
```

Variables declared in the condition are restricted to the scope of the `if` statement - meaning that in the example above, we cannot access the `got` variable outside of the `if` statement's scope.

An `else` statement is done as follows (this example is taken from Go's `time` package, in `time/format.go`):

```
1 if hour >= 12 {
2     b = append(b, "PM"... )
3 } else {
4     b = append(b, "AM"... )
5 }
```

It is also quite common to see `switch` statements used in lieu of `if/else` statements. Here is an example from Go's source code (`net/url/url.go`), of a `switch` statement:

```
1 func ishex(c byte) bool {
2     switch {
3     case '0' <= c && c <= '9':
4         return true
5     case 'a' <= c && c <= 'f':
6         return true
7     case 'A' <= c && c <= 'F':
8         return true
9     }
10    return false
11 }
```

This `switch` statement has no condition, meaning it is functionally equivalent to `switch true`.

A `switch` with a condition looks like this (taken from Go's `fmt/print.go`):

```
1 func (p *pp) fmtBool(v bool, verb rune) {
2     switch verb {
3     case 't', 'v':
4         p.fmt.fmt_boolean(v)
5     default:
6         p.badVerb(verb)
7     }
8 }
```

and we can also declare a variable in the condition and switch on that:

```
1 switch err := err.(type) {
2     case NotFoundError:
3         ...
4 }
```

Arrays

An array of a specific length can be declared like so:

```
1 var x [3]int
```

In Go, however, arrays are rarely used unless you have a specific need for them. Slices are more common, which we'll cover in the next section.

Slices

While arrays have a fixed size, slices are dynamic. To create a slice of integers for example, we can do:

```
1 x := []int{1, 2, 3, 4, 5}
```

Slices are abstractions on top of arrays. A slice contains a pointer to an array, its length, and its capacity. We can get the length and capacity of a slice with the built-in `len` and `cap` functions, respectively. We'll call "slicing" the act of creating a new slice which points to a range of contiguous elements in the original array. We can "slice" arrays as well as slices - in which case the new slice will point to the underlying array. For example:

```
1 x := []int{1, 2, 3, 4, 5}
2 y := x[0:3]
3 fmt.Println(y)
```

will give us:

```
[1 2 3]
```

We can also leave out the 0:

```
1 y := x[:3]
```

and the result will be the same. Likewise for the high bound, we can leave that out and it will default to the length of the slice:

```
1 x := []int{1, 2, 3, 4, 5}
2 y := x[3:]
3 fmt.Println(y)
```

and we get:

```
[4 5]
```

A slice's zero value is `nil`:

```
1 var x []int
2 fmt.Println(x == nil)
```

true

To append to a slice, we use the builtin `append` function:

```
1 x := []int{}
2 x = append(x, 1)
3 x = append(x, 2, 3)
4 y := []int{4, 5}
5 x = append(x, y...)
6 fmt.Println(x)
```

Notice the `y...` on line 5: `append` is a variadic function. The first argument is a slice, and the second is 0 or more arguments of the same type as the slice's values. Running the above code will give us the following output:

```
[1 2 3 4 5]
```

Use `copy` to copy the contents of a slice:

```
1 x := []int{1, 2, 3}
2 y := make([]int, 3)
3 copy(y, x)
4 fmt.Println(x, y)
```

```
[1 2 3] [1 2 3]
```

Note that we're using `make` to create the slice `y`, with a size argument of 3. This is to ensure that `y` has enough capacity to copy `x` into it. If we had used an empty `y` with 0 capacity, for example, our `y` would have remained empty:

```
1 x := []int{1, 2, 3}
2 y := []int{}
3 fmt.Println("y capacity:", cap(y))
4 copy(y, x)
5 fmt.Println(x, y)
```

```
y capacity: 0
[1 2 3] []
```

We can sort a slice with the `sort.Slice`¹⁰ function. All we have to do is provide the slice and a `less` argument which serves as a comparator function in which we define how one element in the slice is considered “less” than another when sorting:

Sorting slices

```
1 package main
2
3 import (
4     "fmt"
5     "sort"
6 )
7
8 type Country struct {
9     Name      string
10    Population int
11 }
12
13 func main() {
14     c := []Country{
15         {"South Africa", 55910000},
16         {"United States", 323100000},
17         {"Japan", 127000000},
18         {"England", 53010000},
19     }
20
21     sort.Slice(c,
22         func(i, j int) bool { return c[i].Name < c[j].Name })
23     fmt.Println("Countries by name:", c)
24 }
```

¹⁰<https://golang.org/pkg/sort/#Slice>


```
25     sort.Slice(c,  
26         func(i, j int) bool { return c[i].Population < c[j].Population })  
27     fmt.Println("Countries by population:", c)  
28 }
```

Running this, our output should be:

```
Countries by name: [{England 53010000} {Japan 127000000} {South Africa 559100\  
00} {United States 323100000}]  
Countries by population: [{England 53010000} {South Africa 55910000} {Japan 1\  
27000000} {United States 323100000}]
```

Maps

Maps are a necessary and versatile data type in any programming language, including Go. Here we'll go over some ways to use maps, and cover some idiosyncrasies in their usage.

First, as we know from earlier, there are a couple of ways to instantiate variables in Go, and this goes for maps as well. Let's look at some of them:

```
1  var m = map[string]string{}  
2  m := make(map[string]string)  
3  m := map[string]string{}
```

The `var` declaration could be used in the top-level (or “package block”) scope:

```
1  package main  
2  
3  var m = map[string]string{}  
4  
5  func main() {  
6  }
```

But otherwise these are all basically functionally equivalent.

If you're familiar with maps in other programming languages, you can probably pick up on using maps in Go pretty quickly. Here is an example that is self-explanatory:

Using maps in Go

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     m := make(map[string]string)
7     m["en"] = "Hello"
8     m["ja"] = "こんにちは"
9     // loop over keys and values of map
10    for k, v := range m {
11        fmt.Printf("%q => %q\n", k, v)
12    }
13
14    // nonexistent key returns zero value
15    fmt.Printf("zh: %q\n", m["zh"])
16
17    // check for existence with _, ok := m[Key]
18    if _, ok := m["ja"]; ok {
19        fmt.Printf("key %q exists in map\n", "ja")
20    }
21
22    delete(m, "en")
23    fmt.Printf("m length: %d\n", len(m))
24    fmt.Println(m)
25 }
```

Running this code, we should get this output:

```
"en" => "Hello"
"ja" => "こんにちは"
zh: ""
key "ja" exists in map
m length: 1
map[ja:こんにちは]
```

The first output is from the loop, then we check a nonexistent key “zh”, then check for the existence of “ja”, print the length of the map, then print the map itself.

Another important note is that the `map` type is not safe for concurrent use. If you need to use a map in a concurrent way, take a look at [sync.Map](#)¹¹.

Also, the iteration order of maps is not guaranteed, so you can't rely on any specific order when looping over your map.

Lastly, the following will make a nil map, which will panic when writing to it:

```
var m map[string]string
```

So avoid using this declaration style when making maps.

For further reading, although it is slightly outdated as it doesn't mention `sync.Map`, check [Go maps in action](#)¹² on the Go blog.

Loops

Loops in Go are done with the `for` construct. There is no `while` in Go, but you can achieve the same effect with `for`:

For loop

```
1 package main
2
3 import (
4     "fmt"
5 )
6
7 func main() {
8     x := 1
9     for x < 4 {
10         fmt.Println(x)
11         x++
12     }
13 }
```

The above code outputs:

¹¹<https://golang.org/pkg/sync/#Map>

¹²<https://blog.golang.org/go-maps-in-action>

```
1  
2  
3
```

A more traditional version that you may be familiar with is also available:

For loop with three components

```
1 package main  
2  
3 import (  
4     "fmt"  
5 )  
6  
7 func main() {  
8     for i := 0; i < 3; i++ {  
9         fmt.Println(i)  
10    }  
11 }
```

This will output:

```
0  
1  
2
```

An infinite loop can be expressed with an empty `for`:

Infinite for loop

```
1 package main  
2  
3 func main() {  
4     for {  
5     }  
6 }
```

To loop over a slice, we can do the following, where `i` is the index and `x` is the value at that index:

For loop over slice with index

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     nums := []int{1, 2, 3}
7     for i, x := range nums {
8         fmt.Printf("nums[%d] => %d\n", i, x)
9     }
10 }
```

The above code will output:

```
nums[0] => 1
nums[1] => 2
nums[2] => 3
```

If we don't need the index, we can leave it out with `_`:

For loop over slice without index

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     nums := []int{1, 2, 3}
7     for _, x := range nums {
8         fmt.Println(x)
9     }
10 }
```

and this will output:

```
1
2
3
```

We can also range over the keys and values of a map like so:

Range over map

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     m := map[int]string{
7         1: "一",
8         2: "二",
9         3: "三",
10    }
11    for k, v := range m {
12        fmt.Printf("%d => %q\n", k, v)
13    }
14 }
```

This will output:

```
1 => "一"
2 => "二"
3 => "三"
```

Functions

Functions are declared with the `func` keyword. They can take zero or more arguments, and can return multiple results.

Functions are first-class citizens, and can be stored in variables or used as values to other functions.

Exported Names

In a Go package, a name is exported if it starts with an uppercase letter. Take the following code for example:

Exported names

```
1 package countries
2
3 import (
4     "math/rand"
5     "time"
6 )
7
8 type Country struct {
9     Name      string
10    Population int
11 }
12
13 var data = []Country{
14     {"South Africa", 55910000},
15     {"United States", 323100000},
16     {"Japan", 127000000},
17     {"England", 53010000},
18 }
19
20 // Random returns a random country
21 func Random() Country {
22     rand.Seed(time.Now().Unix())
23
24     return data[rand.Intn(len(data))]
25 }
```

When importing this package, you would be able to access the `countries.Country` type, as well as `countries.Random()` function, but you cannot access the `countries.data` variable because it begins with a lowercase letter.

Pointers

Declaring a variable with `*` before the type indicates a pointer:

```
var p *int
```

The zero-value of this is `nil`. To generate a pointer to an existing variable, use `&`:

```
x := 100
p = &x
```

Now we can dereference the pointer with `*`:

```
fmt.Println(*p)
```

and this results in:

```
100
```

Lastly, there is no pointer arithmetic in Go. The authors decided to leave it out for reasons such as safety, and simplifying the implementation of the garbage collector.¹³

Goroutines

Goroutines are functions that run concurrently in a Go program. They are lightweight, and cheap to use. Prefixing a function with the `go` keyword will run the function in a new goroutine:

Using a goroutine

```
1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 func hello() {
9     fmt.Println("hello from a goroutine!")
10 }
11
12 func main() {
13     go hello()
14     time.Sleep(1 * time.Second)
15 }
```

(Note that we have a call to `time.Sleep` in the `main` function. This is to prevent the `main` from returning before our goroutine completes. We're using a sleep to show a small example of a goroutine; it is not a valid way of managing goroutines.)

It's also common to see a goroutine used with an anonymous function:

¹³https://golang.org/doc/faq#no_pointer_arithmetic


```
1 go func() {
2     fmt.Println("hello from a goroutine!")
3 }()
```

One way to synchronize goroutines is to use `sync.WaitGroup`¹⁴:

Using `sync.WaitGroup`

```
1 package main
2
3 import (
4     "fmt"
5     "sync"
6 )
7
8 func main() {
9     var wg sync.WaitGroup
10    wg.Add(3)
11    go func() {
12        defer wg.Done()
13        fmt.Println("goroutine 1")
14    }()
15    go func() {
16        defer wg.Done()
17        fmt.Println("goroutine 2")
18    }()
19    go func() {
20        defer wg.Done()
21        fmt.Println("goroutine 3")
22    }()
23
24    wg.Wait()
25 }
```

¹⁴<https://golang.org/pkg/sync/#WaitGroup>

```
$ go run waitgroup_example.go
goroutine 3
goroutine 1
goroutine 2
```

In this example, we instantiate a `sync.WaitGroup` and add 1 to it for each goroutine. The goroutines then call `defer wg.Done()` to signify that they've finished. We then wait for the goroutines with `wg.Wait()`.

When using `sync.WaitGroup`, we must know the exact number of goroutines ahead of time. If we had called `wg.Add(2)` instead of `wg.Add(3)`, then we would risk returning before all of the goroutines finished. If we had called `wg.Add(4)`, the code would have panicked with the following error:

```
fatal error: all goroutines are asleep - deadlock!
```

Another way to manage goroutines is with channels, which we'll discuss in the next section.

Channels

Channels are used for sending and receiving values of a specific type. It is common to see them used inside of goroutines. Channels must be created with their specific type before use:

```
1 ch := make(chan int)
```

We can create a buffered channel like so:

```
1 ch := make(chan int, 5)
```

This means that sending to the channel will block when the buffer is full. When the buffer is empty, receives will block.

To send to a channel, we use the `<-` operator:

```
1 ch <- 5
```

And to receive a value from a channel, we do:

```
1 v := <-ch
```

Let's see what happens when we send too many integers to a buffered channel of ints:

Sending too many ints to buffered channel

```
1 package main
2
3 func main() {
4     ch := make(chan int, 5)
5
6     ch <- 1
7     ch <- 2
8     ch <- 3
9     ch <- 4
10    ch <- 5
11    ch <- 6
12 }
```

```
$ go run channels_sending_too_many.go
fatal error: all goroutines are asleep - deadlock!
```

```
goroutine 1 [chan send]:
main.main()
    /Users/gopher/mygo/src/github.com/gopher/basics/channels_sending_too_many.go\
:11 +0xdb
exit status 2
```

We get an error - all goroutines are asleep - deadlock!. What if we were to read one `int` off of the channel before sending the final 6?

```
$ go run channels_read_one.go
1
```

We're no longer overfilling the buffered channel, because we read one `int` off of it before sending a sixth item.

What if we want to know whether a buffered channel is full before sending to it? There are a couple of ways we can do this.

One way would be to check the `len` of the channel before sending to it again:

Checking length of channel

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     ch := make(chan int, 5)
7
8     ch <- 1
9     ch <- 2
10    ch <- 3
11    ch <- 4
12    ch <- 5
13
14    fmt.Println("channel length:", len(ch))
15    switch {
16    case len(ch) >= 5:
17        <-ch
18    case len(ch) < 5:
19        ch <- 6
20    }
21 }
```

```
$ go run channels_check_length.go
channel length: 5
```

We could also use a `select` statement with a `default` that does nothing when the channel is full:

Channel select statement

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     ch := make(chan int, 5)
7
8     ch <- 1
9     ch <- 2
```

```
10     ch <- 3
11     ch <- 4
12     ch <- 5
13
14     select {
15     case ch <- 6:
16     default:
17         fmt.Println("channel is full, ignoring send")
18     }
19 }
```

```
$ go run channels_select.go
channel is full, ignoring send
```

Interfaces

An interface in Go is a set of methods that another type can define in order to implement that interface. We define an interface type with the `interface` keyword:

```
1 type Entry interface {
2     Title() string
3 }
```

Now any concrete type we define that implements a `Title() string` method will implement the `Entry` interface:

Interfaces example

```
1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 type Entry interface {
9     Title() string
10 }
```

```
11
12 type Book struct {
13     Name      string
14     Author    string
15     Published time.Time
16 }
17
18 func (b Book) Title() string {
19     return fmt.Sprintf("%s by %s (%s)", b.Name, b.Author, b.Published.Format("Ja\
20 n 2006"))
21 }
22
23 type Movie struct {
24     Name      string
25     Director  string
26     Year      int
27 }
28
29 func (m Movie) Title() string {
30     return fmt.Sprintf("%s (%d)", m.Name, m.Year)
31 }
32
33 func Display(e Entry) string {
34     return e.Title()
35 }
36
37 func main() {
38     b := Book{Name: "John Adams", Author: "David McCullough", Published: time.Da\
39 te(2001, time.May, 22, 0, 0, 0, 0, time.UTC)}
40     m := Movie{Name: "The Godfather", Director: "Francis Ford Coppola", Year: 19\
41 72}
42     fmt.Println(Display(b))
43     fmt.Println(Display(m))
44 }
```

Note that the `Display` function takes `e Entry`, not a concrete type like `Book` or `Movie`. Our concrete types implement the `Entry` interface, so we're now allowed to pass implementations of those types into any function that takes an `Entry`.

The empty interface

We define an empty interface as `interface{}`, and it can hold a value of any type:

```
1 var i interface{}
2 i = "こんにちは"
3 fmt.Println(i)
```

```
1 こんにちは
```

If we want to test whether an interface is a certain type, we use a type assertion:

```
1 var i interface{}
2 i = "こんにちは"
3 s, ok := i.(string)
4 if !ok {
5     fmt.Println("s is not type string")
6 }
7 fmt.Println(s)
```

```
1 こんにちは
```

In our example above, `i` is a type `string`, so the second return value from our type assertion is `true`, and `s` contains the underlying value. If `i` had been another type, such as an `int`, then our `ok` would have been `false` and our `s` would have been the zero value of the type we were trying to assert, or in other words `0`.

Nil interfaces

An interface in Go is essentially a tuple containing the underlying type and value. For an interface to be considered `nil`, both the type and value must be `nil`. Here is an example:

Nil interfaces example

```

1 package main
2
3 import (
4     "fmt"
5 )
6
7 func main() {
8     var i interface{}
9     fmt.Println(i == nil)
10    fmt.Printf("%T, %v\n", i, i)
11
12    var s *string
13    fmt.Println("s == nil:", s == nil)
14
15    i = s
16    fmt.Println("i == nil:", i == nil)
17    fmt.Printf("%T, %v\n", i, i)
18
19 }
```

```

true
<nil>, <nil>
s == nil: true
i == nil: false
*string, <nil>

```

Note how our `s` variable is `nil`, but when we set our interface `i` to `s` then check if `i` is `nil`, `i` is not considered `nil`. This is because our interface has an underlying concrete type, and interfaces are only `nil` when both the concrete type and the value are `nil`.

Error Handling

Functions in Go often return an error value as the final return argument. When the function does not encounter any error conditions, we return `nil`. The `error` type is a builtin interface type that we can create with functions like `errors.New` and `fmt.Errorf`. As an example, let's make a function that parses a string and returns the boolean value that string represents. This function is inspired by the `ParseBool` function in the Go standard library's `strconv` package:

Error handling example

```
1 package strconv
2
3 import "fmt"
4
5 func ParseBool(str string) (bool, error) {
6     switch str {
7     case "1", "t", "T", "true", "TRUE", "True":
8         return true, nil
9     case "0", "f", "F", "false", "FALSE", "False":
10        return false, nil
11    }
12    return false, fmt.Errorf("invalid input %q", str)
13 }
```

Here we are returning a `nil` error when we're able to parse the input properly, and using `fmt.Errorf` to return an error in the case that we cannot parse the input.

We'll cover more about error handling in the “Style and Error Handling” chapter that follows this one.

Reading Input

You can use a `bufio.Scanner` to read input from `stdin`, which by default will split the input line by line. Here is an example of a Go program that reads a file containing one word per line, and keeps a count of every occurrence of each word:

Reading standard input

```
1 package main
2
3 import (
4     "bufio"
5     "log"
6     "os"
7 )
8
9 func main() {
10     m := map[string]int{}
11     scanner := bufio.NewScanner(os.Stdin)
```

```
12     for scanner.Scan() {
13         m[scanner.Text()]++
14     }
15     if err := scanner.Err(); err != nil {
16         log.Printf("ERROR: could not read stdin: %s", err)
17     }
18     for k, v := range m {
19         log.Printf("%s => %d", k, v)
20     }
21 }
```

If we had a file that looked like this:

```
go
python
ruby
php
go
go
go
```

and we piped it into our program like so:

```
$ go run main.go < langs.txt
```

we would see output like the following (order is not guaranteed when iterating over maps, so the order of our output might change when running more than once):

```
2017/10/13 13:12:21 go => 4
2017/10/13 13:12:21 python => 1
2017/10/13 13:12:21 ruby => 1
2017/10/13 13:12:21 php => 1
```

Writing Output

One way of writing output to a file in Go is to use the `*File` returned from `os.Create`. `os.Create` will create a file for reading and writing. If the file already exists, it will be truncated:

Writing output

```
1 package main
2
3 import (
4     "log"
5     "os"
6 )
7
8 func main() {
9     langs := []string{"python", "php", "go"}
10    f, err := os.Create("langs.txt")
11    if err != nil {
12        log.Fatal(err)
13    }
14    defer f.Close()
15    for _, lang := range langs {
16        f.WriteString(lang + "\n")
17    }
18 }
```

Running this code will give us the following content in a file called `langs.txt`:

```
python
php
go
```

Another utility function we could use is `ioutil.WriteFile`¹⁵ which will open the file and write our data in one function call.

¹⁵<https://golang.org/pkg/io/ioutil/#WriteFile>

Style and Error Handling

There are two quite major points about Go that take some getting used to when ramping up on learning the language: style and error handling. We'll first talk about what is considered "idiomatic" style in the Go community.

Style

Gofmt

Go comes with a tool that formats Go programs called "gofmt". Gofmt formats your program, and is prominent in the Go community. It would not be a stretch to say that every popular (let's say > 500 stars on GitHub) open source library uses it. When using gofmt, you are not allowed to add exceptions like you can with tools such as PEP8 for Python. Your lines can be longer than 80 chars without warning. You can of course split your long lines up, and gofmt will format them accordingly. You cannot tell gofmt to use spaces instead of tabs.

You might find such strict formatting (maybe you hate tabs) backwards and annoying, but gofmt likely played a big role in Go's success. Since everyone's code looks similar, it takes an element of surprise out of looking at others' code when debugging or trying to understand it. This made it easier to contribute to the standard library and open source libraries, in turn speeding up the growth of the Go community.

To show a very simple example of gofmt in action, here is some code that hasn't been run through gofmt:

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     a      := "foo"
7     someVar := "bar"
8
9     fmt.Println(a, someVar)
10 }
```

This is a valid program and it will run, but it should be run through gofmt to look like this:

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     a := "foo"
7     someVar := "bar"
8
9     fmt.Println(a, someVar)
10 }
```

While you may not agree with all of the rules of `gofmt`, it is so widely used within the community that it has become a requirement. You should always `gofmt` your Go code.

Many editors have integrations that allow you to run `gofmt` on save. We recommend running `goimports` on save. `Goimports`, according to its `godoc`, “updates your Go import lines, adding missing ones and removing unreferenced ones. In addition to fixing imports, `goimports` also formats your code in the same style as `gofmt` so it can be used as a replacement for your editor’s `gofmt-on-save` hook.”¹⁶

Also, as a bonus, use `gofmt -s` to automatically simplify some of your code:

```
1 package main
2
3 type Animal struct {
4     Name    string
5     Species string
6 }
7
8 func main() {
9     animals := []Animal{
10         Animal{"Henry", "cat"},
11         Animal{"Charles", "dog"},
12     }
13 }
```

After `gofmt -s`, this becomes:

¹⁶<https://godoc.org/golang.org/x/tools/cmd/goimports>

```
1 package main
2
3 type Animal struct {
4     Name    string
5     Species string
6 }
7
8 func main() {
9     animals := []Animal{
10         {"Henry", "cat"},
11         {"Charles", "dog"},
12     }
13 }
```

Note how the extra `Animal` struct names are unnecessary in the slice `[]Animal`, and are therefore removed. `gofmt -s` makes stylistic changes and does not affect the logic of the code at all, meaning it is safe to run `gofmt -s` all the time.

Short Variable Names

Another contentious topic in the earlier days of Go was the use of short variable names. If you look at the Go source code, you'll see a lot of code that looks like this:

```
1 func Unmarshal(data []byte, v interface{}) error {
2     // Check for well-formedness.
3     // Avoids filling out half a data structure
4     // before discovering a JSON syntax error.
5     var d decodeState
6     err := checkValid(data, &d.scan)
7     if err != nil {
8         return err
9     }
10
11     d.init(data)
12     return d.unmarshal(v)
13 }
```

You might be thinking, “why use such a short and useless variable name like `d`? It doesn't tell me anything about what the variable is holding.” It's a fair point, especially considering that for years we've been told that having descriptive variable names is very important. But the authors of Go had

something else in mind, and many people have come to embrace short variable names. From a [page containing advice on reviewing Go code](#)¹⁷:

“Variable names in Go should be short rather than long. This is especially true for local variables with limited scope. Prefer `c` to `lineCount`. Prefer `i` to `sliceIndex`.”

Shorter variable names make control flow easier to follow, and allow the reader of the code to focus on the important logic, such as function calls. A general rule of thumb is, if a variable spans less than 10 lines, use a single character. If it spans more, use a descriptive name. At the same time, try to minimize variable span, and functions shorter than 15 lines. Most of the time, this produces readable, idiomatic Go code.

Golint

`golint` is a linter for Go, and it differs from `gofmt` in that it prints style mistakes, whereas `gofmt` reformats your code. To install `golint`, run:

```
go get -u github.com/golang/lint/golint
```

As with `gofmt`, you can't tell `golint` to ignore certain errors. However, `golint` is not meant to be used as a standard, and will sometimes have false positives and false negatives. On Go Report Card we've noticed a lot of repositories with `golint` errors like the following:

```
Line 29: warning: exported type Entry should have comment or be unexported (golint)
```

This is just suggesting that an exported type should have a comment, otherwise it should be unexported. This is nice for `godoc`, which displays the type's comment right below it. You might also see a warning like this:

```
Line 5: warning: if block ends with a return statement, so drop this else and outdent its block (golint)
```

Here's a piece of code where that warning would show up when running `golint`:

```
1 func truncate(s, suf string, l int) string {
2     if len(s) < l {
3         return s
4     } else {
5         return s[:l] + suf
6     }
7 }
```

What `golint` is saying here is that because we return on line 3, there's no need for the `else` on the following line. Thus our code can become:

¹⁷<https://github.com/golang/go/wiki/CodeReviewComments#variable-names>

```
1 func truncate(s, suf string, l int) string {
2     if len(s) < l {
3         return s
4     }
5     return s[:l] + suf
6 }
```

Which is a bit smaller and easier to read.

That's all we're going to cover on golint - we do suggest using it because it can show you ways to make your code simpler as well as more suitable for godoc. There's no need to fix all of its warnings though, if you think it's too noisy.

Error Handling

Error handling may take some getting used to when learning Go. In Go, your functions will normally return whatever values you want to return, as well as an optional error value. To give a simple example:

```
1 // lineCount returns the number of lines in a given file
2 func lineCount(filepath string) (int, error) {
3     out, err := exec.Command("wc", "-l", filepath).Output()
4     if err != nil {
5         return 0, err
6     }
7     // wc output is like: 999 filename.go
8     count, err := strconv.Atoi(strings.Fields(string(out))[0])
9     if err != nil {
10        return 0, err
11    }
12
13    return count, nil
14 }
```

Note the multiple `if err != nil` checks. These are very common in idiomatic Go code, and sometimes people who are new to Go have trouble adjusting to having to write them all the time. You may see it as unnecessary code duplication. Why not have try/except like other languages?

We had similar thoughts when first starting out with Go, but warmed up to the error checking. When you're that strict about returning and checking errors, it's hard to miss where and why an error is happening.

We could even go ahead and make those errors more specific:


```
1 // lineCount returns the number of lines in a given file
2 func lineCount(filepath string) (int, error) {
3     out, err := exec.Command("wc", "-l", filepath).Output()
4     if err != nil {
5         return 0, fmt.Errorf("could not run wc -l: %s", err)
6     }
7     // wc output is like: 999 filename.go
8     count, err := strconv.Atoi(strings.Fields(string(out))[0])
9     if err != nil {
10        return 0, fmt.Errorf("could not convert wc -l output to integer: %s", err)
11    }
12
13    return count, nil
14 }
```

Just make sure not to capitalize the error string unless beginning with proper nouns or acronyms, because the error will be logged in the caller with something like this:

```
1 filepath := "/home/gopher/somefile.txt"
2 lines, err := lineCount(filepath)
3 if err != nil {
4     log.Printf("ERROR: lineCount(%q): %s", filepath, err)
5 }
```

and the error line will flow better without a capital letter appearing in the middle of the log line:

2017/09/21 03:57:55 ERROR: lineCount("/home/gopher/somefile.txt"): Could not run wc -l

vs.

2017/09/21 03:57:55 ERROR: lineCount("/home/gopher/somefile.txt"): could not run wc -l

Wrapping Up

Compared to most languages, Go is very opinionated about proper style. It can take getting used to, but the advantage is that Go projects all follow the same style. This reduces mental overhead and lets you focus on the logic of the program. For more examples of idiomatic Go code, we recommend reading the [Go source code](https://github.com/golang/go)¹⁸ itself. One way to do this is to browse Go standard library's [godoc documentation](https://godoc.org/)¹⁹. Clicking on a function name on godoc.org will take you to a page which displays the source containing that function. Don't be afraid to read the source - it is very approachable and, partly due to it being run through `gofmt`, very readable.

¹⁸<https://github.com/golang/go>

¹⁹<https://godoc.org/-/go>

Strings

In Go, string literals are defined using double quotes, similar to other popular programming languages in the C family:

An example of a string literal

```
1 package main
2
3 import "fmt"
4
5 func ExampleString() {
6     s := "I am a string - 你好"
7     fmt.Println(s)
8     // Output: I am a string - 你好
9 }
```

As the example shows, Go string literals and code may also contain non-English characters, like the Chinese 你好²⁰.

Appending to Strings

Strings can be appended to with the addition (+) operator:

Appending to a string

```
1 package main
2
3 import "fmt"
4
5 func ExampleAppend() {
6     greeting := "Hello, my name is "
7     greeting += "Inigo Montoya"
8     greeting += "."
9     fmt.Println(greeting)
10    // Output: Hello, my name is Inigo Montoya.
11 }
```

²⁰你好, pronounced *nǐ hǎo*, is Hello in Chinese.

This method of string concatenation is easy to read, and great for simple cases. But while Go does allow us to concatenate strings with the `+` (or `+=`) operator, it is not the most efficient method. It is best used only when very few strings are being added, and not in a hot code path. For a discussion on the most efficient way to do string concatenation, see the later [chapter on optimization](#).

In most cases, the built-in `fmt.Sprintf`²¹ function available in the standard library is a better choice for building a string. We can rewrite the previous example like this:

Using `fmt.Sprintf` to build a string

```
1 package main
2
3 import "fmt"
4
5 func ExampleFmtString() {
6     name := "Inigo Montoya"
7     sentence := fmt.Sprintf("Hello, my name is %s.", name)
8     fmt.Println(sentence)
9     // Output: Hello, my name is Inigo Montoya.
10 }
```

The `%s` sequence is a special placeholder that tells the `Sprintf` function to insert a string in that position. There are also other sequences for things that are not strings, like `%d` for integers, `%f` for floating point numbers, or `%v` to leave it to Go to figure out the type. These sequences allow us to add numbers and other types to a string without casting, something the `+` operator would not allow due to type conflicts. For example:

Using `fmt.Printf` to combine different types of variables in a string

```
1 package main
2
3 import "fmt"
4
5 func ExampleFmtComplexString() {
6     name := "Inigo Montoya"
7     age := 32
8     weight := 76.598
9     t := "Hello, my name is %s, age %d, weight %.2fkg"
10    fmt.Printf(t, name, age, weight)
11    // Output: Hello, my name is Inigo Montoya, age 32, weight 76.60kg
12 }
```

²¹<https://golang.org/pkg/fmt/#Sprintf>

Note that here we used `fmt.Printf` to print the new string directly. In previous examples, we used `fmt.Sprintf` to first create a string variable, then `fmt.Println` to print it to the screen (notice the `S` in `Sprintf`, short for string). In the above example, `%d` is a placeholder for an integer, `%.2f` a for a floating point number that should be rounded to the second decimal, and `%s` a placeholder for a string, as before. These codes are analogous to ones in the `printf` and `scanf` functions in C, and old-style string formatting in Python. If you are not familiar with this syntax, have a look at the documentation for the `fmt` package²². It is both expressive and efficient, and used liberally in Go code.

What would happen if we tried to append an integer to a string using the plus operator?

Breaking code that tries to append an integer to a string

```
1 package main
2
3 func main() {
4     s := "I am" + 32 + "years old"
5 }
```

Running this with `go run`, Go returns an error message during the build phase:

```
1 $ go run bad_append.go
2 # command-line-arguments
3 ./bad_append.go:4: cannot convert "I am" to type int
4 ./bad_append.go:4: invalid operation: "I am" + 32 (mismatched types string and
5 int)
```

As expected, Go's type system catches our transgression, and complains that it cannot append an integer to a string. We should rather use `fmt.Sprintf` for building strings that mix different types.

Next we will have a look at a very useful standard library package that allows us to perform many common string manipulation tasks: the built-in `strings` package.

Splitting strings

The `strings` package is imported by simply adding `import "strings"`, and provides us with many string manipulation functions. One of these is a function that split a string by separators, and obtain a slice of strings:

²²<https://golang.org/pkg/fmt/>

Splitting a string

```
1 package main
2
3 import "fmt"
4 import "strings"
5
6 func ExampleSplit() {
7     l := strings.Split("a,b,c", ",")
8     fmt.Printf("%q", l)
9     // Output: ["a" "b" "c"]
10 }
```

The `strings.Split` function takes a string and a separator as arguments. In this case, we passed in "a,b,c" and the separator "," and received a string slice containing the separate letters a, b, and c as strings.

Counting and finding substrings

Using the `strings` package, we can also count the number of non-overlapping instances of a substring in a string with the aptly-named `strings.Count`. The following example uses `strings.Count` to count occurrences of both the single letter a, and the substring ana. In both cases we pass in a string²³. Notice that we get only one occurrence of ana, even though one may have expected it to count ana both at positions 1 and 3. This is because `strings.Count` returns the count of *non-overlapping* occurrences.

Count occurrences in a string

```
1 package main
2
3 import (
4     "fmt"
5     "strings"
6 )
7
8 func ExampleCount() {
9     s := "banana"
10    c1 := strings.Count(s, "a")
11    c2 := strings.Count(s, "ana")
```

²³Remember, Go does not support function overloading, so a single character should be passed as a string if the function expects a string, like most functions in the `strings` standard library.

```
12     fmt.Println(c1, c2)
13     // Output: 3 1
14 }
```

If we want to know whether a string contains, starts with, or ends with some substring, we can use the `strings.Contains`, `strings.HasPrefix`, and `strings.HasSuffix` functions, respectively. All of these functions return a boolean:

Count occurrences in a string

```
1  package main
2
3  import (
4      "fmt"
5      "strings"
6  )
7
8  func ExampleContains() {
9      str := "two gophers on honeymoon"
10     if strings.Contains(str, "moon") {
11         fmt.Println("Contains moon")
12     }
13     if strings.HasPrefix(str, "moon") {
14         fmt.Println("Starts with moon")
15     }
16     if strings.HasSuffix(str, "moon") {
17         fmt.Println("Ends with moon")
18     }
19     // Output: Contains moon
20     // Ends with moon
21 }
```

For finding the index of a substring in a string, we can use `strings.Index`. `Index` returns the index of the first instance of `substr` in `s`, or -1 if `substr` is not present in `s`:

Using strings.Index to find substrings in a string

```
1 package main
2
3 import "fmt"
4 import "strings"
5
6 func ExampleIndex() {
7     an := strings.Index("banana", "an")
8     am := strings.Index("banana", "am")
9     fmt.Println(an, am)
10    // Output: 1 -1
11 }
```

The `strings` package also contains a corresponding `LastIndex` function, which returns the index of the *last* (ie. right-most) instance of a matching substring, or -1 if it is not found.

The `strings` package contains many more useful functions. To name a few: `ToLower`, `ToUpper`, `Trim`, `Equals` and `Join`, all performing actions that match their names. For more information on these and other functions, refer to the [strings package docs](https://golang.org/pkg/strings/)²⁴. As a final example, let's see how we might combine some of the functions in the `strings` package in a real program, and discover some of its more surprising functions.

Advanced string functions

The program below repeatedly takes input from the user, and declares whether the typed sentence is palindromic. For a sentence to be palindromic, we mean that the words should be the same when read forwards and backwards. We wish to ignore punctuation, and assume the sentence is in English, so there are spaces between words. Take a look and notice how we use two new functions from the `strings` package, `FieldsFunc` and `EqualFold`, to keep the code clear and concise.

²⁴<https://golang.org/pkg/strings/>

A program that declares whether a sentence reads the same backward and forward, word for word

```
1 package main
2
3 import (
4     "bufio"
5     "fmt"
6     "os"
7     "strings"
8     "unicode"
9 )
10
11 // getInput prompts the user for some text, and then
12 // reads a line of input from standard input. This line
13 // of text is then returned.
14 func getInput() string {
15     fmt.Print("Enter a sentence: ")
16     scanner := bufio.NewScanner(os.Stdin)
17     scanner.Scan()
18     return scanner.Text()
19 }
20
21 func isNotLetter(c rune) bool {
22     return !unicode.IsLetter(c)
23 }
24
25 // isPalindromicSentence returns whether or not the given sentence
26 // is palindromic. To calculate this, it splits the string into words,
27 // then creates a reversed copy of the word slice. It then checks
28 // whether the reverse is equal (ignoring case) to the original.
29 // It also ignores any non-alphabetic characters.
30 func isPalindromicSentence(s string) bool {
31     // split into words and remove non-alphabetic characters
32     // in one operation by using FieldsFunc and passing in
33     // isNotLetter as the function to split on.
34     w := strings.FieldsFunc(s, isNotLetter)
35
36     // iterate over the words from front and back
37     // simultaneously. If we find a word that is not the same
38     // as the word at its matching from the back, the sentence
```



```

39     // is not palindromic.
40     l := len(w)
41     for i := 0; i < l/2; i++ {
42         fw := w[i]      // front word
43         bw := w[l-i-1]  // back word
44         if !strings.EqualFold(fw, bw) {
45             return false
46         }
47     }
48
49     // all the words matched, so the sentence must be
50     // palindromic.
51     return true
52 }
53
54 func main() {
55     // Go doesn't have while loops, but we can use for loop
56     // syntax to read into a new variable, check that it's not
57     // empty, and read new lines on subsequent iterations.
58     for l := getInput(); l != ""; l = getInput() {
59         if isPalindromicSentence(l) {
60             fmt.Println("... is palindromic!")
61         } else {
62             fmt.Println("... is not palindromic.")
63         }
64     }
65 }

```

Save this code to `palindromes.go`, and we can then run it with `go run palindromes.go`.

An example run of the palindrome program

```

1 $ go run palindromes.go
2 Enter a sentence: This is magnificent!
3 ... is not palindromic.
4 Enter a sentence: This is magnificent, is this!
5 ... is palindromic!
6 Enter a sentence:

```

As expected, when we enter a sentence that reads the same backwards and forwards, ignoring

punctuation and case, we get the output ... is palindromic!. Now, let's break down what this code is doing.

The `getInput` function uses a `bufio.Scanner` from the `bufio` package²⁵ to read one line from standard input. `scanner.Scan()` scans until the end of the line, and `scanner.Text()` returns a string containing the input line.

The meat of this program is in the `isPalindromicSentence` function. This function takes a string as input, and returns a boolean indicating whether the sentence is palindromic, word-for-word. We also want to ignore punctuation and case in the comparison. First, on line 34, we use `strings.FieldsFunc` to split the string at each Unicode code point for which the `isNotLetter` function returns true. In Go, you can pass around functions like any other value. A function's type signature describes the types of its arguments and return values. Our `isNotLetter` function satisfies the function signature specified by `FieldsFunc`, which is to take a rune as input, and return a boolean. Runes are a special character type in the Go language - for now, just think of them as more or less equivalent to a single character, like `char` in Java.

In `isNotLetter`, we return `false` if the passed in rune is a letter as defined by the Unicode standard, and `true` otherwise. We can achieve this in a single line by using `unicode.IsLetter`, another built-in function provided by the standard `unicode` library.

Putting it all together, `strings.FieldsFunc(s, isNotLetter)` will return a slice of strings, split by sequences of non-letters. In other words, it will return a slice of words.

Next, on line 40, we iterate over the slice of words. We keep an index `i`, which we use to create both `fw`, the word at index `i`, and `bw`, the matching word at index `1 - i - 1`. If we can walk all the way through the slice without finding two words that are not equal, we have a palindromic sentence. And we can stop halfway through, because then we have already done all the necessary comparisons. The next table shows how this process works for an example sentence as `i` increases. As we walk through the slice, words match, and so we continue walking until we reach the middle. If we were to find a non-matching pair, we can immediately return `false`, because the sentence is not palindromic.

The palindromic sentence algorithm by example

	"Fall"	"leaves"	"as"	"soon"	"as"	"leaves"	"fall"	EqualFold
i=0	fw						bw	true
i=1		fw				bw		true
i=2			fw		bw			true

The equality check of strings is performed on line 44 using `strings.EqualFold` - this function compares two strings for equality, ignoring case.

Finally, on line 58, we make use of the semantics of the Go for loop definition. The basic for loop has three components separated by semicolons:

- the init statement: executed before the first iteration

²⁵<https://golang.org/pkg/bufio/>

- the condition expression: evaluated before every iteration
- the post statement: executed at the end of every iteration

We use these definition to instantiate a variable `l` and read into it from standard input, conditionally break from the loop if it is empty, and set up reading for each subsequent iteration in the post statement.

Ranging over a string

When the functions in the `strings` package don't suffice, it is also possible to range over each character in a string:

Iterating over the characters in a string

```
1 package main
2
3 import "fmt"
4
5 func ExampleIteration() {
6     s := "ABC你好"
7     for i, r := range s {
8         fmt.Printf("%q(%d) ", r, i)
9     }
10    // Output: 'A' (0) 'B' (1) 'C' (2) '你' (3) '好' (6)
11 }
```

You might be wondering about something peculiar about the output above. The printed indexes start from 0, 1, 2, 3 and then jump to 6. Why is that? This is the topic in the next chapter, [Supporting Unicode](#).

Supporting Unicode

Part of preparing production-ready code is making sure that it behaves as expected for supported languages and inputs. In the previous chapter, we showed how easy Go makes many common string manipulation tasks. Many of these examples were implicitly English-centric, with only some hints that there may be more brewing below the surface when it comes to handling international character sets. In this chapter we will examine strings in greater depth, and learn how to write bug-free, production-ready code that handles strings in any language supported by the Unicode standard.

We will start by taking a detour through the history of string encodings. This will then inform the rest of our discussion on handling different character sets in Go.

A very brief history of string encodings

What are string encodings, and why do we need them? You can skip this section if you already know the difference between Unicode and UTF-8, and between a character and a Unicode code point.

Consider how a computer might represent a string of text. Because computers operate on binary, human-readable text needs to be represented as binary numbers in some way. Early computer pioneers came up with one such scheme, which they called ASCII (pronounced *ASS-kee*). ASCII is one way of mapping characters to numbers. For example, A is 65 (binary 0100 0001, or hexadecimal 0x41), B is 66, C is 67, and so on. We could represent the ASCII-encoded string “ABC” in hexadecimal notation, like so:

```
0x41 0x42 0x43
```

ASCII defines a mapping for 127 different characters, using exactly 7 bits. For the old 8-bit systems, this was perfect. The only problem is ASCII only covers unaccented English letters.

As computers became more widespread, other countries also needed to represent their text in binary format, and unaccented English letters were not enough. So a plethora of new encodings were invented. Now when code encountered a string, it also needed to know which *encoding* the string is using in order to map the bytes to the correct human-readable characters.

Identifying this as a problem, a group called the Unicode consortium undertook the herculean task of assigning a number to every letter used in any language. Such a magic number is called a Unicode code point, and is represented by a U+ followed by a hexadecimal number. For example, the string “ABC” corresponds to these three Unicode code points:

U+0041 U+0042 U+0043

Notice how for the string “ABC”, the hexadecimal numbers are the same as for ASCII.

So Unicode assigns each character with a number, but it does not specify how this number should be represented in binary. This is left to the *encoding*. The most popular encoding of the Unicode standard is called UTF-8. UTF-8 is popular because it has some nice properties.

One nice property of UTF-8 is that every code point between 0-127 is stored in a single byte. Only code points 128 and above are stored using 2, 3, or up to 6 bytes. Because the first 128 Unicode code points were chosen to match ASCII, this has the side effect that English text looks exactly the same in UTF-8 as it did in ASCII. (Notice how the hexadecimally-encoded ASCII of “ABC” from earlier is the same as the Unicode code points for the same letters.)

In this chapter we will keep things simple by focusing on only these two encodings: ASCII and UTF-8. UTF-8 has become the universal standard, and supports every language your application might need, from Chinese to Klingon. But the same principles apply for any encoding, and should your application need to handle the conversion from other encodings, most common encodings are available in the golang.org/x/text/encoding package.

For a more complete history of string encoding, we recommend Joel Spolsky’s excellent blog post from 2003, titled [The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets \(No Excuses!\)](https://www.joelonsoftware.com/2003/10/08/the-absolute-minimum-every-software-developer-absolutely-positively-must-know-about-unicode-and-character-sets-no-excuses/)²⁶.

With an understanding of what string encodings are and why they exist, let’s now turn to how they are handled in Go.

Strings are byte slices

In Go, strings are read-only (or *immutable*) byte slices. The byte slice representing the string is not required to hold Unicode text, UTF-8 text, or any other specific encoding. In other words, strings can hold arbitrary bytes. For example, we can take a slice of bytes, and convert it to a string:

```
1 package main
2
3 import (
4     "fmt"
5 )
6
7 func main() {
8     b := []byte{65, 66, 67}
9     s := string(b)
```

²⁶<https://www.joelonsoftware.com/2003/10/08/the-absolute-minimum-every-software-developer-absolutely-positively-must-know-about-unicode-and-character-sets-no-excuses/>

```
10     fmt.Println(s)
11     // Output: ABC
12 }
```

In this byte slice to string conversion, Go makes no assumptions about the encoding. To Go, this is just another byte slice, now with the type of string. When the string gets printed by `fmt.Println`, again, Go is just sending some bytes to standard output. The terminal that outputs the bytes to the screen needs to use the appropriate character encoding for the bytes to render correctly as human-readable text. In this case, either ASCII or UTF-8 encodings would do.

Printing strings

For debugging strings it is often useful to see the raw bytes in different forms. Strings starting with a `%` symbol, like `%s` or `%q`, are placeholders for parameters when passed into certain functions in the `fmt` package, like `fmt.Printf`, `fmt.Sprintf`, which returns a formatted string, and `fmt.Scanf`, which reads a variable from input. Package `fmt` implements formatted I/O with functions similar to C's `printf` and `scanf`. The format 'verbs' are derived from C's but are simpler.

The next program showcases some different ways to print strings in Go using the `fmt` package.

```
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func showVerb(v, s string) {
8      n := fmt.Sprintf(v, s)
9      fmt.Println(v, "\t", n)
10 }
11
12 func main() {
13     s := "ABC 你好"
14     showVerb("%s", s)
15     showVerb("%q", s)
16     showVerb("%+q", s)
17     showVerb("%x", s)
18     showVerb("% x", s)
19     showVerb("%# x", s)
20 }
```

Running this produces the following output:

```

1 %s    ABC 你好
2 %q    "ABC 你好"
3 %+q    "ABC \u4f60\u597d"
4 %x    41424320e4bda0e5a5bd
5 % x    41 42 43 20 e4 bd a0 e5 a5 bd
6 %# x    0x41 0x42 0x43 0x20 0xe4 0xbd 0xa0 0xe5 0xa5 0xbd

```

The `showVerb` function is a single line that prints the given verb `v`, plus the string `s` formatted using that verb. We use the `fmt.Sprintf` function to create a new string formatted with the passed in verb `v`. On the next line we print `v` and the new string `n`, to see what it looks like.

The verb following the `%` character specifies how Go should format the given parameter. There are many verbs to choose from, including some verbs for specific data types. The following verbs can be used for any data type ²⁷:

```

1 %v    the value in a default format
2        when printing structs, the plus flag (%+v) adds field names
3 %#v    a Go-syntax representation of the value
4 %T    a Go-syntax representation of the type of the value
5 %%    a literal percent sign; consumes no value

```

and these verbs are only for strings and slices of bytes:

```

1 %s    the uninterpreted bytes of the string or slice
2 %q    a double-quoted string safely escaped with Go syntax
3 %x    base 16, lower-case, two characters per byte
4 %X    base 16, upper-case, two characters per byte

```

Other common verbs include those for integers (e.g. `%d` for numbers in base 10) and booleans (`%b`). You can refer to the `fmt` package documentation for the full list.

As our example demonstrated, some verbs allow special flags between the `%` symbol and the verb. From the `fmt` package documentation:

²⁷<https://golang.org/pkg/fmt/#pkg-overview>

```

1  +   always print a sign for numeric values;
2      guarantee ASCII-only output for %q (%+q)
3  -   pad with spaces on the right rather than the left (left-justify the field)
4  #   alternate format: add leading 0 for octal (%#o), 0x for hex (%#x);
5      0X for hex (%#X); suppress 0x for %p (%#p);
6      for %q, print a raw (backquoted) string if strconv.CanBackquote
7      returns true;
8      always print a decimal point for %e, %E, %f, %F, %g and %G;
9      do not remove trailing zeros for %g and %G;
10     write e.g. U+0078 'x' if the character is printable for %U (%#U).
11 ' ' (space) leave a space for elided sign in numbers (% d);
12     put spaces between bytes printing strings or slices in hex (% x, % X)
13 0    pad with leading zeros rather than spaces;
14     for numbers, this moves the padding after the sign

```

Flags are ignored by verbs that do not expect them.

There are plenty more options for formatting with the `fmt` package, and we recommend reading the documentation for a full breakdown of all the available options. One aspect not covered so far is that it is possible to specify a width by an optional decimal number immediately preceding the verb. By default the width is whatever is necessary to represent the value, but when provided, width will pad with spaces until there are least that number of *runes*. This is not the same as in C, which uses bytes to count the width. So what exactly are *runes*?

Runes and safely ranging over strings

Previously we stated that Go makes no assumptions about the string encoding when it converts bytes to the string type. This is true. The string type carries with it no information about its encoding. But in certain instances, Go *does* need to make assumptions about the underlying encoding. One such case is when we range over a string.

Let's return to the last example from the previous chapter, and range over a string that contains some non-ASCII characters:

Iterating over the characters in a string

```
1 package main
2
3 import "fmt"
4
5 func ExampleIteration() {
6     s := "ABC你好"
7     for i, r := range s {
8         fmt.Printf("%q(%d) ", r, i)
9     }
10    // Output: 'A' (0) 'B' (1) 'C' (2) '你' (3) '好' (6)
11 }
```

Running this, we get the following output:

```
1 'A' (0) 'B' (1) 'C' (2) '你' (3) '好' (6)
```

The `range` keyword returns two values on each iteration: an integer indicating the current position in the string, and the current *rune*. `rune` is a built-in type in Go, and it is meant to contain a single Unicode character. As such, it is an alias of `int32`, and contains 4 bytes. So on every iteration, we get the current position in the string, in this case called `i`, and a rune called `r`. We use the `%q` and `%d` verbs to print out the current rune and position in the string.

By now it might be clear why the position variable `i` jumps from 3 to 6 between `你` and `好`. Under the hood, instead of going byte by byte, the `range` keyword is fetching the next *rune* in the string.

Text Normalization

In the Unicode standard, there are often several ways to represent the same string. For example, the acute e in the word `café` can be represented in a string as a single rune ("`\u00e9`") or as an e followed by an acute accent ("`e\u0301`"). According to the standard, these two should be treated as equivalent. Is this what happens in Go?

Basic text normalization in Go

```

1 package main
2
3 import (
4     "fmt"
5
6     "golang.org/x/text/unicode/norm"
7 )
8
9 func main() {
10     a := string([]rune{0x00E9})
11     b := string([]rune{0x0065, 0x0301})
12     fmt.Println(a, b)
13     fmt.Println("same:", a == b)
14
15     normA := norm.NFC.String(a)
16     normB := norm.NFC.String(b)
17     fmt.Println(normA, normB)
18     fmt.Println("same:", normA == normB)
19
20     // Output: é é
21     //same: false
22     //é é
23     //same: true
24 }
```

As the example demonstrates, the answer is “no”. String comparison, by default, does not consider the two versions of é equivalent. However, the example also shows how we can use the `golang.org/x/text/unicode/norm` package to write the strings into NFC (“Normalization Form C”) before doing the comparison, in which case they *do* come back as equivalent. Unicode provides four normalization forms: NFD, NFC, NFKD and NFKC. These are summarized in the table below, taken from the [Unicode standard documentation](https://unicode.org/reports/tr15/)²⁸:

Form	Description
Normalization Form D (NFD)	Canonical Decomposition
Normalization Form C (NFC)	Canonical Decomposition, followed by Canonical Composition
Normalization Form KD (NFKD)	Compatibility Decomposition
Normalization Form KC (NFKC)	Compatibility Decomposition, followed by Canonical Composition

²⁸<https://unicode.org/reports/tr15/>

Here “decomposition” means writing the “e” and accent separately, as in “e\u0301”, and “composition” means writing them as a single character, i.e. “\u00e9”. The details of these are far beyond the scope of this book, but the important thing for our purposes is that there are several instances where normalizing text can be a good idea. We will discuss a few of these next.

Handling look-alikes

Some Unicode characters look exactly alike, yet use different underlying codes. This can pose a security threat. For example, can you tell the difference between users `Kevin` and `Kevin`? One Kevin is using `Ɔ` (unicode `\u004B`) and the other is using `Ɔ` (Kelvin sign, unicode `\u212A`). The second Kevin is likely doing this to circumvent detection mechanisms that check whether usernames are taken, and might intend on impersonating the first Kevin.

Fortunately, the `golang.org/x/text/unicode/norm` package comes to the rescue. We can use this to mitigate (but not completely eliminate) the risk. In the following example, we again `norm.NFC.String` to convert the strings to canonical form before doing comparisons:

Comparing user names in canonical form

```

1 package main
2
3 import (
4     "fmt"
5
6     "golang.org/x/text/unicode/norm"
7 )
8
9 func equalNorm(a, b string) bool {
10     return norm.NFC.String(a) == norm.NFC.String(b)
11 }
12
13 func main() {
14     user1 := `Kevin` // uses normal K "\u004B"
15     user2 := `Kevin` // uses Kelvin K "\u212A"
16     fmt.Println("naive equal:", user1 == user2)
17     fmt.Println("norm equal:", equalNorm(user1, user2))
18
19     // Output: naive equal: false
20     //norm equal: true
21 }

```

This trick can help us detect similar usernames that may be an attempt to impersonate someone else and treat visually similar names as equivalent. It's not perfect, though. Some characters that look the same are still treated as unequal. For example, the Latin `o`, Greek `ο`, and Cyrillic `о` are still different characters because they come from different alphabets, even though they look the same.

A note on efficiency: in the toy examples so far we made use of `norm.NFC.String`, a convenience function for converting a string to NFC form. This is great for short strings and simple cases. It is, however, not as efficient as the longer form:

```
1 wc := norm.NFC.Writer(w)
2 defer wc.Close()
```

Here `w` needs to be an `io.Writer`, such as `bytes.Buffer`. Use this form in applications where the efficiency gains outweigh the brevity of the convenience functions.

Saving bytes on the wire

Generally speaking, the NFC form will not save all that much in terms of bytes stored in your database or sent over the wire. But in some languages, like Korean, the savings can be significant. You should do benchmarking for your own application, but it is well worth considering converting text to NFC form before storing it in a database or sending it to other applications.

Correctly modifying Unicode text

The `norm` package can also be of use when replacing text. Consider an example, adapted from the official [Go blog post](https://go.dev/blog/normalization)²⁹, where we wish to replace the word `cafe` in some text. Without taking multi-rune character boundaries into account, an instance of `cafe` followed by an accented "e" might become `cafeś` (with the accent now incorrectly on the `s`) after replacement. We can handle this if we transform the text to canonical form first before doing replacement.

²⁹<https://go.dev/blog/normalization>

Replacing text while respecting boundaries between multi-rune characters

```
1 package main
2
3 import (
4     "fmt"
5     "strings"
6
7     "golang.org/x/text/unicode/norm"
8 )
9
10 func replaceNormalized(text string, old, new []string) string {
11     for i := 0; i < len(old) && i < len(new); i++ {
12         text = strings.ReplaceAll(norm.NFC.String(text), old[i], new[i])
13     }
14     return text
15 }
16
17 func main() {
18     text := "We went to eat at multiple cafe" + string(rune(0x0301)) // accented\
19     e in decomposed form
20     incorrect := strings.ReplaceAll(text, "cafe", "cafes")
21     fmt.Println("Incorrect:", incorrect)
22
23     correct := replaceNormalized(text, []string{"cafe", "café"}, []string{"cafes\
24 ", "cafés"})
25     fmt.Println("Correct:", correct)
26
27     // Output: Incorrect: We went to eat at multiple cafés
28     //Correct: We went to eat at multiple cafés
29 }
```

Summary

In this chapter we covered:

- how Go treats unicode characters in strings differently from other languages. Be sure you test with unicode characters when ranging over strings or counting characters, unless your input guarantees to have only characters from the ASCII subset.

- Text normalization and handling look-alike characters, as well as some other edge cases related to Unicode to look out for when writing production-ready Go code.

If you would like to learn more about this topic, also see the Go blog posts [Strings, bytes, runes and characters in Go](https://go.dev/blog/strings)³⁰ and [Text Normalization in Go](https://go.dev/blog/normalization)³¹.

³⁰<https://go.dev/blog/strings>

³¹<https://go.dev/blog/normalization>

Concurrency

Go's concurrency model is based on what are called "goroutines" - essentially lightweight threads. To invoke a goroutine, we use the `go` keyword:

```
go func() {  
    fmt.Println("hello, world")  
}()
```

If we were to add the above to a `main()` function and run it, we would probably see no output. This is because the `main` exited before the goroutine had time to finish. To see this in action, let's add a sleep to give the goroutine some time to run:

```
1 package main  
2  
3 import (  
4     "fmt"  
5     "time"  
6 )  
7  
8 func main() {  
9     go func() {  
10         fmt.Println("hello, world")  
11     }()  
12  
13     time.Sleep(1 * time.Second)  
14 }
```

```
$ go run main.go  
hello, world
```

What would happen if we had multiple goroutines? Let's try:

```
1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 func main() {
9     go func() {
10         fmt.Println("hello, world 1")
11     }()
12
13     go func() {
14         fmt.Println("hello, world 2")
15     }()
16
17     go func() {
18         fmt.Println("hello, world 3")
19     }()
20
21     time.Sleep(1 * time.Second)
22 }
```

```
$ go run main.go
```

```
hello, world 2
```

```
hello, world 1
```

```
hello, world 3
```

We notice the goroutines all ran, and not in the order in which they were written. This is the expected behavior.

But we can't use `time.Sleep` everywhere in our code to wait for our goroutines to finish, right? In the next sections we'll discuss how to organize our goroutines.

sync.WaitGroup

With `sync.WaitGroup` we can avoid using `time.Sleep` to wait for our goroutines to finish. Instead, we create a `sync.WaitGroup` and add 1 to its counter for every goroutine we expect to launch. Then, inside each goroutine we decrement the counter. Finally we call the `Wait()` method on the `WaitGroup` to wait for all of our goroutines to finish. Let's modify our previous example to use a `sync.WaitGroup`:

sync.WaitGroup

```
1 package main
2
3 import (
4     "fmt"
5     "sync"
6 )
7
8 func main() {
9     var wg sync.WaitGroup
10    wg.Add(1)
11    go func() {
12        defer wg.Done()
13        fmt.Println("hello, world 1")
14    }()
15
16    wg.Add(1)
17    go func() {
18        defer wg.Done()
19        fmt.Println("hello, world 2")
20    }()
21
22    wg.Add(1)
23    go func() {
24        defer wg.Done()
25        fmt.Println("hello, world 3")
26    }()
27
28    wg.Wait()
29 }
```

When we run this code, we get the same or similar output to when we were using `time.Sleep`:

```
$ go run main.go
hello, world 3
hello, world 1
hello, world 2
```

All of our goroutines finished, and we didn't need to use a `time.Sleep` to wait for them.

errgroup

`errgroup`³² is similar to `sync.WaitGroup` but provides convenient error handling functionality.

`errgroup.Group`

```
1 package main
2
3 import (
4     "fmt"
5     "log"
6     "os/exec"
7     "strconv"
8     "strings"
9     "sync/atomic"
10
11     "golang.org/x/sync/errgroup"
12 )
13
14 // lineCount returns the number of lines in a given file
15 func lineCount(filepath string) (int, error) {
16     out, err := exec.Command("wc", "-l", filepath).Output()
17     if err != nil {
18         return 0, fmt.Errorf("could not run wc -l: %s", err)
19     }
20
21     // wc output is like: 999 filename.go
22     count, err := strconv.Atoi(strings.Fields(string(out))[0])
23     if err != nil {
24         return 0, fmt.Errorf("could not convert wc -l output to integer: %s", err)
25     }
26
27     return count, nil
28 }
29
30 func main() {
31     var totalLineCount int64
32     var g errgroup.Group
33
```

³²<https://pkg.go.dev/golang.org/x/sync/errgroup>

```

34     files := []string{
35         "foo.txt",
36         "bar.txt",
37         "baz.txt",
38     }
39
40     for _, file := range files {
41         file := file
42         g.Go(func() error {
43             lc, err := lineCount(file)
44             if err != nil {
45                 return err
46             }
47
48             atomic.AddInt64(&totalLineCount, int64(lc))
49
50             return nil
51         })
52     }
53
54     if err := g.Wait(); err != nil {
55         log.Fatal(err)
56     }
57
58     log.Printf("total line count: %d", totalLineCount)
59 }

```

In this example, we have a list of files for which we want to get the word count concurrently. We loop over each filename and launch a goroutine with the errgroup's `Go` function. The `lineCount` function calls out to the `wc` command. Note that due to how closures work with goroutines, we must set `file := file` before launching the goroutine (https://golang.org/doc/faq#closures_and_goroutines).

Inside the goroutine, we get the line count and error return values from `lineCount`. If we encounter an error, we return it. After launching the goroutines, we call `g.Wait()` to wait for them to finish. We also get an error back if any errors were returned in the process:

```

$ go run main.go
2020/05/21 13:51:59 total line count: 17

```

To force an error, let's remove one of the files we're checking:

```
$ rm foo.txt
$ go run main.go
2020/05/21 13:52:35 could not run wc -l: exit status 1
exit status 1
```

Channels

Channels can be used to send and receive values. They're often used in goroutines; a goroutine will do some work and then send the result to the channel that was passed in as an argument:

Channel usage

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     ch := make(chan string)
7
8     go func(ch chan string) {
9         ch <- "hello, world 1"
10    }(ch)
11
12    go func(ch chan string) {
13        ch <- "hello, world 2"
14    }(ch)
15
16    go func(ch chan string) {
17        ch <- "hello, world 3"
18    }(ch)
19
20    a, b, c := <-ch, <-ch, <-ch
21
22    fmt.Println(a)
23    fmt.Println(b)
24    fmt.Println(c)
25 }
```

This code is functionally equivalent to our `sync.WaitGroup` example in the previous section. Note that we must declare the channel before it's used, as we see with `ch := make(chan string)`. We create

the channel, then each goroutine sends a string onto the channel. Finally we select 3 values from the channel and print them out at the end of the program.

What would happen if we only sent 2 strings but tried to select 3?

Channel usage

```

1 package main
2
3 import "fmt"
4
5 func main() {
6     ch := make(chan string)
7
8     go func(ch chan string) {
9         ch <- "hello, world 1"
10    }(ch)
11
12    go func(ch chan string) {
13        ch <- "hello, world 2"
14    }(ch)
15
16    // go func(ch chan string) {
17    //     ch <- "hello, world 3"
18    // }(ch)
19
20    a, b, c := <-ch, <-ch, <-ch
21
22    fmt.Println(a)
23    fmt.Println(b)
24    fmt.Println(c)
25 }
```

```
$ go run main.go
```

```
fatal error: all goroutines are asleep - deadlock!
```

```
goroutine 1 [chan receive]:
```

```
main.main()
```

```
    /Users/gopher/foo.go:20 +0xf2
```

```
exit status 2
```

Goroutines in web handlers

We mentioned that goroutines run until the `main` function exits. This means that a web handler can create goroutines to do background processing and can return before the goroutines finish. Let's see an example:

Goroutine in web handler

```
1 package main
2
3 import (
4     "log"
5     "net/http"
6     "time"
7 )
8
9 func main() {
10     http.HandleFunc("/", func(w http.ResponseWriter, req *http.Request) {
11         go func() {
12             time.Sleep(3 * time.Second)
13             log.Println("hello, world")
14         }()
15
16         return
17     })
18
19     log.Println("Running on :8080...")
20     log.Fatal(http.ListenAndServe("127.0.0.1:8080", nil))
21 }
```

If we run this code in one terminal, we should see:

```
$ go run main.go
2018/11/02 13:21:23 Running on :8080...
```

Now in a second terminal, let's do a `curl`:

```
$ curl localhost:8080
$
```

The `curl` returns almost immediately. Now if we wait a couple of seconds, we should see our print statement in the first terminal:

```
$ go run main.go
2018/11/02 13:21:23 Running on :8080...
2018/11/02 13:21:26 hello, world
```

Meaning the goroutine continued running in the background after our HTTP handler returned.

Pollers

Goroutines are useful when writing pollers as well. Let's say we have a Go program that functions as a web server, but we also want to poll for some data in the background. We'll keep the web server functionality small for the sake of simplicity in our example. Let's poll for advisory information from BART, the public transportation system serving the San Francisco Bay Area:

BART advisory poller

```
1 package main
2
3 import (
4     "encoding/json"
5     "io/ioutil"
6     "log"
7     "net/http"
8     "time"
9 )
10
11 const bsaEndpoint = "http://api.bart.gov/api/bsa.aspx?cmd=bsa&key=MW9S-E7SL-2\
12 6DU-VV8V&json=y"
13
14 type bsaResponse struct {
15     Root struct {
16         Advisories []BSA `json:"bsa"`
17     }
18 }
19
20 // BSA is a BART service advisory
21 type BSA struct {
22     Station      string
23     Description struct {
24         Text string `json:"#cdata-section"`
25     }
26 }
```

```
26 }
27
28 func poll() {
29     ticker := time.NewTicker(5 * time.Second)
30     defer ticker.Stop()
31     for {
32         select {
33         case <-ticker.C:
34             resp, err := http.Get(bsaEndpoint)
35             if err != nil {
36                 log.Println("ERROR: could not GET bsaEndpoint:", err)
37             }
38             defer resp.Body.Close()
39
40             b, err := ioutil.ReadAll(resp.Body)
41             if err != nil {
42                 log.Println("ERROR: could not parse response body:", err)
43             }
44
45             var br bsaResponse
46             err = json.Unmarshal(b, &br)
47             if err != nil {
48                 log.Println("ERROR: json.Unmarshal:", err)
49             }
50
51             if len(br.Root.Advisories) > 0 {
52                 for _, adv := range br.Root.Advisories {
53                     log.Println(adv.Station, adv.Description.Text)
54                 }
55             }
56         }
57     }
58 }
59
60 func main() {
61     http.HandleFunc("/", func(w http.ResponseWriter, req *http.Request) {
62         go func() {
63             log.Println("hello, world 1")
64         }()
```



```

65
66         return
67     })
68
69     go poll()
70     log.Println("Running on :8080...")
71     log.Fatal(http.ListenAndServe("127.0.0.1:8080", nil))
72 }

```

Let's go over what this code does. We have an HTTP handler like our previous example but without the `time.Sleep`. If we do `curl localhost:8080`, we should not only see an immediate response, but also a `hello, world` log statement in the console running our program. After we set up our handler, we run our `poll` function in a goroutine.

The `poll` function is making a GET request every 5 seconds to a public BART API endpoint which contains advisory information, such as delays at certain stations. After making the request and unmarshalling it, we print the results.

Why does `poll` need to be run in a goroutine? Since it contains an infinite loop (`for{ ... }`), if we were to run it on its own we would never get to the code following it.

Race conditions

When writing concurrent code, we must be careful not to create any race conditions. A race condition occurs in Go when two goroutines access the same variable and at least one access is a write.³³

We discuss race conditions and the race detector in detail in the [Tooling]{#racedetector} section. Here we'll go over a simple example and fix it, but please see the Tooling section for advice on preventing data races in your build.

Consider the following example of two goroutines updating a map concurrently:

Goroutines updating same map

```

1 package main
2
3 import (
4     "log"
5     "time"
6 )
7
8 func main() {

```

³³https://golang.org/doc/articles/race_detector.html

```
9      m := map[string]string{}
10
11      go func() {
12          m["test"] = "hello, world 1"
13      }()
14
15      go func() {
16          m["test"] = "hello, world 2"
17      }()
18
19      time.Sleep(time.Second)
20      log.Println(m["test"])
21 }
```

Running this code gives us the following output:

```
$ go run main.go
2018/11/02 17:29:27 map[test:hello, world 2]
```

Since these two goroutines are updating the same map concurrently, however, there is a data race. We can prevent this by using `sync.Map`:

`sync.Map`

```
1 package main
2
3 import (
4     "log"
5     "sync"
6     "time"
7 )
8
9 func main() {
10     var m = &sync.Map{}
11
12     go func() {
13         m.Store("test", "hello, world 1")
14     }()
15
16     go func() {
```

```

17         m.Store("test", "hello, world 2")
18     }()
19
20     time.Sleep(time.Second)
21
22     val, _ := m.Load("test")
23     log.Println(val)
24 }

```

This works fine, but the documentation warns us that `sync.Map` should only be used in the following situations:

The Map type is optimized for two common use cases: (1) when the entry for a given key is only ever written once but read many times, as in caches that only grow, or (2) when multiple goroutines read, write, and overwrite entries for disjoint sets of keys. In these two cases, use of a Map may significantly reduce lock contention compared to a Go map paired with a separate Mutex or RWMutex.³⁴

So let's take their advice and use map paired with a Mutex:

`sync.Map`

```

1  package main
2
3  import (
4      "log"
5      "sync"
6      "time"
7  )
8
9  type safeMap struct {
10     sync.Mutex
11     m map[string]string
12 }
13
14 func main() {
15     var sm = safeMap{
16         m: make(map[string]string),
17     }
18

```

³⁴<https://golang.org/pkg/sync/#Map>

```
19     go func() {
20         sm.Lock()
21         defer sm.Unlock()
22         sm.m["test"] = "hello, world 1"
23     }()
24
25     go func() {
26         sm.Lock()
27         defer sm.Unlock()
28         sm.m["test"] = "hello, world 2"
29     }()
30
31     time.Sleep(time.Second)
32
33     sm.Lock()
34     log.Println(sm.m["test"])
35     sm.Unlock()
36 }
```

We can add Store, Load, Delete etc. methods to our `safeMap` type as well:

sync.Map

```
1  package main
2
3  import (
4      "log"
5      "sync"
6      "time"
7  )
8
9  type safeMap struct {
10     sync.Mutex
11     m map[string]string
12 }
13
14 func (sm *safeMap) Store(key, val string) {
15     sm.Lock()
16     defer sm.Unlock()
17     sm.m[key] = val
18 }
```

```
18 }
19
20 func (sm *safeMap) Load(key string) (val string, exists bool) {
21     sm.Lock()
22     defer sm.Unlock()
23     val, ok := sm.m[key]
24     return val, ok
25 }
26
27 func (sm *safeMap) Delete(key, val string) {
28     sm.Lock()
29     defer sm.Unlock()
30     delete(sm.m, key)
31 }
32
33 func main() {
34     var sm = safeMap{
35         m: make(map[string]string),
36     }
37
38     go func() {
39         sm.Store("test", "hello, world 1")
40     }()
41
42     go func() {
43         sm.Store("test", "hello, world 2")
44     }()
45
46     time.Sleep(time.Second)
47
48     val, _ := sm.Load("test")
49     log.Println(val)
50 }
```

```
$ go run main.go
```

```
2018/11/02 18:45:57 hello, world 2
```

It also runs with the race detector enabled, no data races found:

```
$ go run -race main.go
```

```
2018/11/02 18:46:22 hello, world 2
```

Testing

A critical part of any production-ready system is a complete test suite. If you have not written tests before, and wonder why they are important, this introduction is for you. If you already understand the importance of proper testing, you can skip to the particulars of writing tests in Go, in [writing tests](#).

Why do we need tests?

A line of reasoning we sometimes hear, is that “my code clearly works, why do I need to write tests for it?” This is a natural enough question, but make no mistake, a modern production-ready system absolutely must have automated tests. Let’s use an analogy from the business world to understand why this is so: double entry bookkeeping.

Double entry is the idea that every financial transaction has equal and opposite effects in at least two different accounts. For example, if you spend \$10 on groceries, your bank account goes down by \$10, and your groceries account goes up by \$10. This trick allows you to see, at a glance, simultaneously how much money is in your bank account, and how much you spent on groceries. It also allows you to spot mistakes. Suppose a smudge in your books made the \$10 entry look like \$18. The total balance of your assets would no longer match your liabilities plus equity - there would be an \$8 difference. We can compare entries in the bank account with entries in the groceries account to discover which amount is incorrect. Before double-entry bookkeeping, it was much harder to prove mistakes, and impossible to see different account balances at a glance. The idea revolutionized bookkeeping, and underpins accounting to this day.

Back to tests. For every piece of functionality we write, we also write a test. The test should prove that the code works in all reasonable scenarios. Like double-entry bookkeeping, tests are our way to ensure our system is correct, and remains correct. Your system might work now - you might even prove it to yourself by trying out some cases manually. But systems, especially production systems, require changes over time. Requirements change, environments change, bugs emerge, new features become needed, inefficiencies are discovered. All these things will require changes to be made to the code. After making these changes, will you still be sure that the system is correct? Will you run through manual test cases after every change? What if someone else is maintaining the code? Will they know how to test changes? How much time will it take you to manually perform these test cases?

Automated tests cost up-front investment, but they uncover bugs early, improve maintainability, and save time in the long run. Tests are the checks and balances to your production system.

Many books and blog posts have been written about good testing practice. There are even movements that promote [writing tests first](#)³⁵, before writing the code. We don't think it's necessary to be quite that extreme, but if it helps you write good tests, then more power to you. No production system is complete without a test suite that makes sensible assertions on the code to prove it correct.

Now that we have discussed the importance of testing in general, let's see how tests are written in Go. As we'll see, testing was designed with simplicity in mind.

Writing Tests

Test files in Go are located in the same package as the code being tested, and end with the suffix `_test.go`. Usually, this means having one `_test.go` to match each code file in the package. Below is the layout of a simple package for testing prime numbers.

- prime
 - prime.go
 - prime_test.go
 - sieve.go
 - sieve_test.go

This is a very typical Go package layout. Go packages contain all files in the same directory, including the tests. Now, let's look at what the test code might look like in `prime_test.go`.

A simple test in `prime_test.go` that tests a function called `IsPrime`

```
1 package main
2
3 import "testing"
4
5 // TestIsPrime tests that the IsPrime function
6 // returns true when the input is prime, and false
7 // otherwise.
8 func TestIsPrime(t *testing.T) {
9     // check a prime number
10    got := IsPrime(19)
11    if got != true {
12        t.Errorf("IsPrime(%d) = %t, want %t", 19, got, true)
13    }
14
15    // check a non-prime number
```

³⁵https://en.wikipedia.org/wiki/Test-driven_development


```
16     got = IsPrime(21)
17     if got != false {
18         t.Errorf("IsPrime(%d) = %t, want %t", 21, got, false)
19     }
20 }
```

We start by importing the `testing` package. Then, on line 8, we define a test as a normal Go function taking a single argument: `t *testing.T`. All tests must start with the word `Test` and have a single argument, a pointer to `testing.T`. In the function body, we call the function under test, `IsPrime`. First we pass in the integer 19, which we expect should return `true`, because 19 is prime. We check this assertion with a simple `if` statement on line 11, `if got != true`. If the statement evaluates to `false`, `t.Errorf` is called. `Errorf` formats its arguments in a way analogous to `Printf`, and records the text in the error log. We repeat a similar check for the number 21, this time asserting that the `IsPrime` function returns `true`, because 21 is not prime.

We can run the tests in this package using `go test`. Let's see what happens:

```
$ go test
PASS
ok      _/Users/productiongo/code/prime  0.018s
```

It passed! But did it actually run our `TestIsPrime` function? Let's check by adding the `-v` (verbose) flag to the command:

```
$ go test -v
=== RUN   TestIsPrime
--- PASS: TestIsPrime (0.00s)
PASS
ok      _/Users/productiongo/code/prime  0.019s
```

Our test is indeed being executed. The `-v` flag is a useful trick to remember, and we recommend running tests with it turned on most of the time.

All tests in Go follow essentially the same format as the `TestIsPrime`. The Go authors made a conscious decision not to add specific assertion functions, advising instead to use the existing control flow tools that come with the language. The result is that tests look very similar to normal Go code, and the learning curve is minimal.

Table-driven tests

Our initial `TestIsPrime` test is a good start, but it only tests two numbers. The code is also repetitive. We can do better by using what is called a *table-driven* test. The idea is to define all the inputs and expected outputs first, and then loop through each case with a `for` loop.

A table-driven test in `prime_test.go` that tests a function called `IsPrime`

```
1 package main
2
3 import "testing"
4
5 // TestIsPrime tests that the IsPrime function
6 // returns true when the input is prime, and false
7 // otherwise.
8 func TestIsPrimeTD(t *testing.T) {
9     cases := []struct {
10         give int
11         want bool
12     }{
13         {19, true},
14         {21, false},
15         {10007, true},
16         {1, false},
17         {0, false},
18         {-1, false},
19     }
20
21     for _, c := range cases {
22         got := IsPrime(c.give)
23         if got != c.want {
24             t.Errorf("IsPrime(%d) = %t, want %t", c.give, got, c.want)
25         }
26     }
27 }
```

In the refactored test, we use a slice of an anonymous struct to define all the inputs we want to test. We then loop over each test case, and check that the output matches what we want. This is much cleaner than before, and it only took a few keystrokes to add more test cases into the mix. We now also check some edge cases: inputs of 0, 1, 10007, and negative inputs. Let's run the test again and check that it still passes:

```
$ go test -v
=== RUN   TestIsPrimeTD
--- PASS: TestIsPrimeTD (0.00s)
PASS
ok      _/Users/productiongo/code/prime 0.019s
```

It looks like the `IsPrime` function works as advertised! To be sure, let's add a test case that we expect to fail:

```
...
    {-1, false},

    // 17 is prime, so this test should fail:
    {17, false},
}
...
```

We run `go test -v` again to see the results:

```
$ go test -v
=== RUN   TestIsPrimeTD
--- FAIL: TestIsPrimeTD (0.00s)
    prime_test.go:25: IsPrime(17) = true, want false
FAIL
exit status 1
FAIL    _/Users/productiongo/code/prime 0.628s
```

This time `go test` reports that the test failed, and we see the error message we provided to `t.Errorf`.

Writing error messages

In the tests above, we had the following code:

```
if got != c.want {
    t.Errorf("IsPrime(%d) = %t, want %t", c.give, got, c.want)
}
```

The ordering of the if statement is not accidental: by convention, it should be `actual != expected`, and the error message uses that order too. This is the recommended way to format test failure messages

in Go ³⁶. In the error message, first state the function called and the parameters it was called with, then the actual result, and finally, the result that was expected. We saw before that this results in a message like

```
prime_test.go:25: IsPrime(17) = true, want false
```

This makes it clear to the reader of the error message what function was called, what happened, and what should have happened. The onus is on you, the test author, to leave a helpful message for the person debugging the code in the future. It is a good idea to assume that the person debugging your failing test is not you, and is not your team. Make both the name of the test and the error message relevant.

Testing HTTP Handlers

Let's look at an example of testing that comes up often when developing web applications: testing an HTTP handler. First, let's define a comically simple HTTP handler that writes a friendly response:

A very simple HTTP handler

```
1 package main
2
3 import (
4     "fmt"
5     "net/http"
6 )
7
8 // helloHandler writes a friendly "Hello, friend :)" response.
9 func helloHandler(w http.ResponseWriter, r *http.Request) {
10     fmt.Fprintln(w, "Hello, friend :)")
11 }
12
13 func main() {
14     http.HandleFunc("/hello", helloHandler)
15     http.ListenAndServe(":8080", nil)
16 }
```

The `httptest` package provides us with the tools we need to test this handler as if it were running in a real web server. The `TestHTTPHandler` function in the following example illustrates how to use `httptest.NewRecorder()` to send a real request to our friendly `helloHandler`, and read the resulting response.

³⁶<https://github.com/golang/go/wiki/CodeReviewComments#useful-test-failures>

Testing an HTTP handler

```
1 package main
2
3 import (
4     "net/http"
5     "net/http/httptest"
6     "testing"
7 )
8
9 func TestHTTPHandler(t *testing.T) {
10     // Create a request to pass to our handler.
11     req, err := http.NewRequest("GET", "/hello", nil)
12     if err != nil {
13         t.Fatal(err)
14     }
15
16     // We create a ResponseRecorder, which satisfies
17     // the http.ResponseWriter interface, to record
18     // the response.
19     r := httptest.NewRecorder()
20     handler := http.HandlerFunc(helloHandler)
21
22     // Our handler satisfies http.Handler, so we can call
23     // the ServeHTTP method directly and pass in our
24     // Request and ResponseRecorder.
25     handler.ServeHTTP(r, req)
26
27     // Check that the status code is what we expect.
28     if r.Code != http.StatusOK {
29         t.Errorf("helloHandler returned status code %v, want %v",
30             r.Code, http.StatusOK)
31     }
32
33     // Check that the response body is what we expect.
34     want := "Hello, friend :)\n"
35     got := r.Body.String()
36     if got != want {
37         t.Errorf("helloHandler returned body %q want %q",
38             got, want)
```

```
39         }  
40     }
```

In this example we see the `t.Fatal` method used for the first time. This method is similar to `t.Error`, but unlike `t.Error`, if `t.Fatal` is called, the test will not execute any further. This is useful when a condition happens that will cause the rest of the test to be unnecessary. In our case, if our call to create a request on line 11 were to fail for some reason, the call to `t.Fatal` ensures that we log the error and abandon execution immediately. Analogous to `t.Errorf`, there is also a `t.Fatalf` method, which takes arguments the same way as `fmt.Printf`.

On line 19 we create a new `httptest.Recorder` with which to record the response. We also create `handler`, which is `helloHandler`, but now of type `http.HandlerFunc`. We can do this, because `helloHandler` uses the appropriate signature defined by [http.HandlerFunc](https://golang.org/pkg/net/http/#HandlerFunc)³⁷:

```
type HandlerFunc func(ResponseWriter, *Request)
```

`http.HandlerFunc` is an adapter to allow the use of ordinary functions as HTTP handlers. As the final step of the setup, we pass the recorder and the request we created earlier in to `handler.ServeHTTP(r, req)`. Now we can use the fields provided by `httptest.Recorder`, like `Code` and `Body`, to make assertions against our HTTP handler, as shown in the final lines of the test function.

Mocking

Imagine you need to test code that uses a third party library. Perhaps this library is a client library to an external API, or perhaps it performs database operations. In your unit tests, it is best to assume that the library does its job, and only test your functions and their interactions. This allows your test case failures to accurately reflect where the problem is, rather than leave the question of whether it's your function, or the library, that's at fault. There is a place for tests that include third party libraries, and that place is in integration tests, not unit tests.

Interfaces

How do we go about testing our functions, but not the libraries they use? The answer: interfaces. Interfaces are an incredibly powerful tool in Go.

In Java, interfaces need to be explicitly implemented. You rely on your third party vendor to provide an interface that you can use to stub methods for tests. In Go, we don't need to rely on the third party author; we can define our own interface. As long as our interface defines a subset of the methods implemented by the library, the library will automatically implement our interface.

The next example illustrates one particular case where mocking is very useful: testing code that relies on random number generation.

³⁷<https://golang.org/pkg/net/http/#HandlerFunc>

Using an interface to abstract away API calls

```
1 package eightball
2
3 import (
4     "math/rand"
5     "time"
6 )
7
8 // randIntGenerator is an interface that includes Intn, a
9 // method in the built-in math/rand package. This allows us
10 // to mock out the math/rand package in the tests.
11 type randIntGenerator interface {
12     Intn(int) int
13 }
14
15 // EightBall simulates a very simple magic 8-ball,
16 // a magical object that predicts the future by
17 // answering yes/no questions.
18 type EightBall struct {
19     rand randIntGenerator
20 }
21
22 // NewEightBall returns a new EightBall.
23 func New() *EightBall {
24     return &EightBall{
25         rand: rand.New(rand.NewSource(time.Now().UnixNano())),
26     }
27 }
28
29 // Answer returns a magic eightball answer
30 // based on a random result provided by
31 // randomGenerator. It supports only four
32 // possible answers.
33 func (e EightBall) Answer(s string) string {
34     n := e.rand.Intn(3)
35     switch n {
36     case 0:
37         return "Definitely not"
38     case 1:
```

```

39         return "Maybe"
40     case 2:
41         return "Yes"
42     default:
43         return "Absolutely"
44 }
45 }

```

We define a simple `eightball` package that implements a simple [Magic 8-Ball](https://en.wikipedia.org/wiki/Magic_8-Ball)³⁸. We ask it a yes/no question, and it will return its prediction of the future. As you might expect, it completely ignores the question, and just makes use of a random number generator. But random numbers are hard to test, because they change all the time. One option would be to set the random seed in our code, or in our tests. This is indeed an option, but it doesn't allow us to specifically test the different outcomes without some trial and error. Instead, we create an `randIntGenerator` interface, which has only one method, `Intn(int) int`. This method signature is the same as the `Intn`³⁹ method implemented by Go's built-in `math/rand` package. Instead of using the `math/rand` package directly in `Answer`, we decouple our code by referencing the `Intn` method on the `EightBall`'s `rand` interface. Since `EightBall.rand` is not exported, users of this package will not be aware of this interface at all. To create the struct, they will need to call the `New` method, which assigns the built-in struct from `math/rand` struct to satisfy our interface. So to package users the code looks the same, but under the hood, we can now mock out the call to `Intn` in our tests:

Testing using our interface

```

1  package eightball
2
3  import (
4      "testing"
5  )
6
7  type fixedRandIntGenerator struct {
8      // the number that should be "randomly" generated
9      randomNum int
10
11      // record the paramater that Intn gets called with
12      calledWithN int
13  }
14
15  func (g *fixedRandIntGenerator) Intn(n int) int {
16      g.calledWithN = n

```

³⁸https://en.wikipedia.org/wiki/Magic_8-Ball

³⁹<https://golang.org/pkg/math/rand/#Rand.Intn>


```

17         return g.randomNum
18     }
19
20 func TestEightBall(t *testing.T) {
21     cases := []struct {
22         randomNum int
23         want      string
24     }{
25         {0, "Definitely not"},
26         {1, "Maybe"},
27         {2, "Yes"},
28         {3, "Absolutely"},
29         {-1, "Absolutely"}, // default case
30     }
31
32     for _, tt := range cases {
33         g := &fixedRandIntGenerator{randomNum: tt.randomNum}
34         eb := EightBall{
35             rand: g,
36         }
37
38         got := eb.Answer("Does this really work?")
39         if got != tt.want {
40             t.Errorf("EightBall.Answer() is %q for num %d, want %q",
41                 got, tt.randomNum, tt.want)
42         }
43
44         if g.calledWithN != 3 {
45             t.Errorf("EightBall.Answer() did not call Intn(3) as expected")
46         }
47     }
48 }

```

Sometimes, when the existing code uses a specific library implementation, it takes refactoring to use interfaces to mock out implementation details. However, the resulting code is more decoupled. The tests run faster (e.g. when mocking out external network calls) and are more reliable. Don't be afraid to make liberal use of interfaces. This makes for more decoupled code and more focused tests.

GoMock

Another way to generate mocks for tests is [GoMock](#)⁴⁰. WIP

Generating Coverage Reports

To generate test coverage percentages for your code, run the `go test -cover` command. Let's make a quick example and a test to go with it.

We're going to write a simple username validation function. We want our usernames to only contain letters, numbers, and the special characters "-", "_", and ".". Usernames also cannot be empty, and they must be less than 30 characters long. Here's our username validation function:

Username validation function

```
1 package validate
2
3 import (
4     "fmt"
5     "regexp"
6 )
7
8 // Username validates a username. We only allow
9 // usernames to contain letters, numbers,
10 // and special chars "_", "-", and "."
11 func Username(u string) (bool, error) {
12     if len(u) == 0 {
13         return false, fmt.Errorf("username must be > 0 chars")
14     }
15     if len(u) > 30 {
16         return false, fmt.Errorf("username too long (must be < 30 chars)")
17     }
18     validChars := regexp.MustCompile(`^[a-zA-Z1-9-_.]+$`)
19     if !validChars.MatchString(u) {
20         return false, fmt.Errorf("username contains invalid character")
21     }
22
23     return true, nil
24 }
```

Now let's write a test for it:

⁴⁰<https://github.com/golang/mock>

Test for username validation function

```
1 package validate
2
3 import "testing"
4
5 var usernameTests = []struct {
6     in      string
7     wantValid bool
8 }{
9     {"gopher", true},
10 }
11
12 func TestUsername(t *testing.T) {
13     for _, tt := range usernameTests {
14         valid, err := Username(tt.in)
15         if err != nil && tt.wantValid {
16             t.Fatal(err)
17         }
18
19         if valid != tt.wantValid {
20             t.Errorf("Username(%q) = %t, want %t", tt.in, valid, tt.wantValid)
21         }
22     }
23 }
```

As you can see, we're not covering very many cases. Let's see what exactly our test coverage is for this function:

```
$ go test -cover
PASS
coverage: 62.5% of statements
ok      github.com/gopher/validate 0.008s
```

62.5% is a bit too low. This function is simple enough that we can get close to 100% coverage. We'd like to know exactly what parts of the function are not being covered. This is where the coverage profile and HTML report come in.

To generate a test coverage profile, we run `go test -coverprofile=coverage.out:`

```
$ go test -coverprofile=coverage.out
PASS
coverage: 62.5% of statements
ok      github.com/gopher/validate 0.008s
```

We can now get a breakdown of coverage percentages per function, although we only have one function so it's not very interesting:

```
go tool cover -func=coverage.out
github.com/gopher/validate/validate.go:11: Username      62.5%
total:                                (statements)    62.5%
```

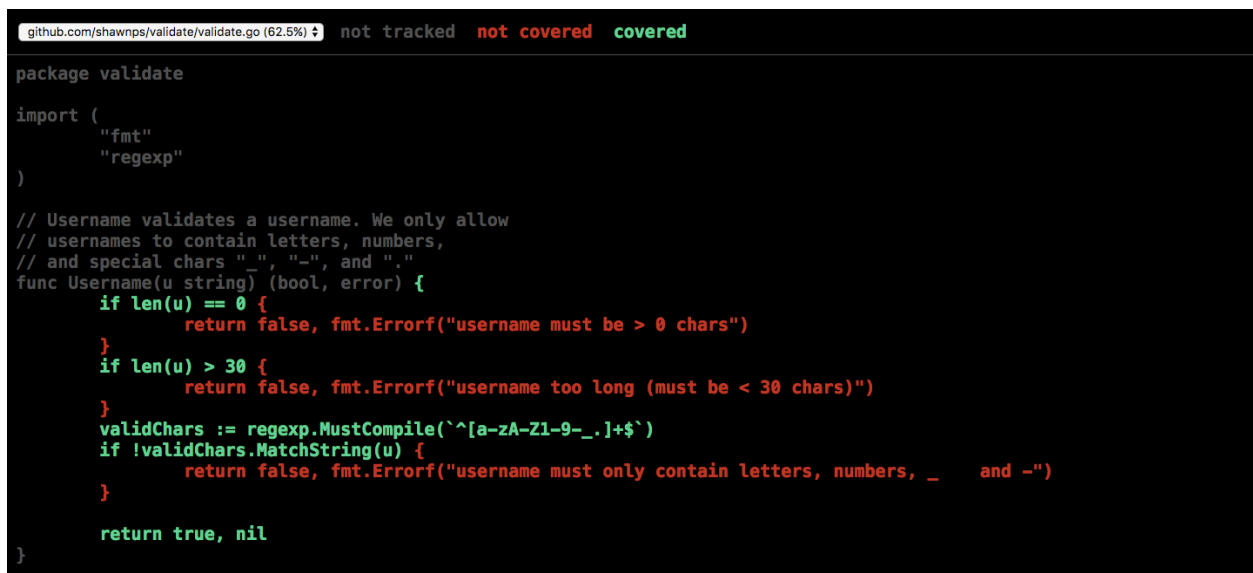
What we really want to see is a line-by-line breakdown. We can get this with the HTML report, which we'll cover in the next section.

HTML Coverage Reports

We can generate an HTML coverage report using the same `coverage.out` file from before, by running the following command:

```
go tool cover -html=coverage.out
```

This should open up a browser and show us an HTML page like the following:



Username test coverage

Now we can see exactly where we need to improve our coverage. We need to cover the cases where the username length is either 0 or > 30, as well as the case where the username contains an invalid character. Let's update our test for those cases:

Test for username validation function, 100% coverage

```
1 package validate
2
3 import "testing"
4
5 var usernameTests = []struct {
6     in      string
7     wantValid bool
8 }{
9     {"", false},
10    {"gopher", true},
11    {"gopher$", false},
12    {"abcdefghijklmnopqrstuvwxyabcde", false},
13 }
14
15 func TestUsername(t *testing.T) {
16     for _, tt := range usernameTests {
17         valid, err := Username(tt.in)
18         if err != nil && tt.wantValid {
19             t.Fatal(err)
20         }
21
22         if valid != tt.wantValid {
23             t.Errorf("Username(%q) = %t, want %t", tt.in, valid, tt.wantValid)
24         }
25     }
26 }
```

Now if we re-run `go test -coverprofile=coverage.out` to get a new coverage profile, and then `go tool cover -html=coverage.out` to view the HTML report again, we should see all green:

```

github.com/shawnps/validate/validate.go (100.0%) not tracked not covered covered
package validate

import (
    "fmt"
    "regexp"
)

// Username validates a username. We only allow
// usernames to contain letters, numbers,
// and special chars "_", "-", and "."
func Username(u string) (bool, error) {
    if len(u) == 0 {
        return false, fmt.Errorf("username must be > 0 chars")
    }
    if len(u) > 30 {
        return false, fmt.Errorf("username too long (must be < 30 chars)")
    }
    validChars := regexp.MustCompile(`^[a-zA-Z1-9-_.]+$`)
    if !validChars.MatchString(u) {
        return false, fmt.Errorf("username must only contain letters, numbers, _ and -")
    }
    return true, nil
}

```

Username test coverage 100%

Writing Examples

We can also write example code and the `go test` tool will run our examples and verify the output. `godoc` renders examples underneath the function's documentation.

Let's write an example for our username validation function:

Username validation function example test

```

1 package validate
2
3 import (
4     "fmt"
5     "log"
6 )
7
8 func ExampleUsername() {
9     usernames := []struct {
10         in    string
11         valid bool
12     }{
13         {"", false},
14         {"gopher", true},
15         {"gopher$", false},
16         {"abcdefghijklmnopqrstuvwxyabcde", false},

```

```

17     }
18     for _, tt := range usernames {
19         valid, err := Username(tt.in)
20         if err != nil && tt.valid {
21             log.Fatal(err)
22         }
23
24         fmt.Printf("%q: %t\n", tt.in, valid)
25     }
26     // Output:
27     // "": false
28     // "gopher": true
29     // "gopher$": false
30     // "abcdefghijklmnopqrstuvwxyzabcde": false
31 }

```

Note the `Output:` at the bottom. That’s a special construct that tells `go test` what the standard output of our example test should be. `go test` is actually going to validate that output when it runs the tests.

If we run a local `godoc` server with `godoc -http:6060`, and navigate to our `validate` package, we can also see that `godoc` renders the example, as expected:

func Username

```
func Username(u string) (bool, error)
```

Username validates a username. We only allow usernames to contain letters, numbers, and special chars `"_"`, `"-"`, and `"."`

► [Example](#)

Godoc example

If we click “Example” we’ll see our example code:

func Username

```
func Username(u string) (bool, error)
```

Username validates a username. We only allow usernames to contain letters, numbers, and special chars "_", "-", and ".".

▼ Example

Code:

```
usernames := []struct {
    in    string
    valid bool
}{
    {"", false},
    {"gopher", true},
    {"gopher$", false},
    {"abcdefghijklmnopqrstuvwxyabcde", false},
}
for _, tt := range usernames {
    valid, err := Username(tt.in)
    if err != nil && tt.valid {
        log.Fatal(err)
    }

    fmt.Printf("%q: %t\n", tt.in, valid)
}
```

Output:

```
"": false
"gopher": true
"gopher$": false
"abcdefghijklmnopqrstuvwxyabcde": false
```

Godoc example full

Another note about examples is that they have a specific naming convention. We named our example above `ExampleUsername` because we wrote an example for the `Username` function. But what if we want to write an example for a method on a type? Let's say we had a type `User` with a method `ValidateName`:


```
1  type User struct {
2      Name string
3  }
4
5  func (u *User) ValidateName() (bool, error) {
6      ...
7  }
```

Then our example code would look like this:

```
1  func ExampleUser_ValidateName() {
2      ...
3  }
```

where the convention for writing examples for methods on types is `ExampleT_M()`.

If we need multiple examples for a single function, we append an underscore and a lowercase letter. For example with our `Validate` function, we could have `ExampleValidate`, `ExampleValidate_second`, `ExampleValidate_third`, and so on.

In the next chapter, we will discuss one last important use of the Go testing package: [benchmarking](#).

Benchmarks

The Go testing package contains a benchmarking tool for examining the performance of our Go code. In this chapter, we will use the benchmark utility to progressively improve the performance of a piece of code. We will then discuss advanced benchmarking techniques to ensure that we are measuring the right thing.

A simple benchmark

Let's suppose we have a simple function that computes the n^{th} Fibonacci number. The sequence F_n of Fibonacci numbers is defined by the recurrence relation, $F_n = F_{n-1} + F_{n-2}$, with $F_0 = 0, F_1 = 1$. That is, every number after the first two is the sum of the two preceding ones:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

Because the sequence is recursively defined, a function that calculates the n^{th} Fibonacci number is often used to illustrate programming language recursion in computer science text books. Below is such a function that uses the definition to recursively calculate the n^{th} Fibonacci number.

A function that recursively obtains the n^{th} Fibonacci number

```
1 package fibonacci
2
3 // F returns the nth Fibonacci number.
4 func F(n int) int {
5     if n <= 0 {
6         return 0
7     } else if n == 1 {
8         return 1
9     }
10    return F(n-1) + F(n-2)
11 }
```

Let's make sure it works by writing a quick test, as we saw in the chapter on [Testing](#).

A test for the function that recursively obtains the n^{th} Fibonacci number

```

1  // fibonacci_test.go
2  package fibonacci
3
4  import "testing"
5
6  func TestF(t *testing.T) {
7      cases := []struct {
8          n      int
9          want int
10     }{
11         {-1, 0},
12         {0, 0},
13         {1, 1},
14         {2, 1},
15         {3, 2},
16         {8, 21},
17     }
18
19     for _, tt := range cases {
20         got := FastF(tt.n)
21         if got != tt.want {
22             t.Errorf("F(%d) = %d, want %d", tt.n, got, tt.want)
23         }
24     }
25 }

```

Running the test, we see that indeed, our function works as promised:

```

$ go test -v
=== RUN   TestF
--- PASS: TestF (0.00s)
PASS
ok      _/home/productiongo/benchmarks/fibonacci 0.001s

```

Now, this recursive Fibonacci function works, but we can do better. How much better? Before we rewrite this function, let's establish a baseline to which we can compare our future efficiency improvements. Go provides a benchmark tool as part of the testing package. Analogous to `TestX(t *testing.T)`, we create benchmarks with `BenchmarkX(b *testing.B)`:

A benchmark for the Fibonacci function

```

1 // fibonacci_bench_test.go
2 package fibonacci
3
4 import "testing"
5
6 var numbers = []int{
7     0, 10, 20, 30,
8 }
9
10 func BenchmarkF(b *testing.B) {
11     // run F(n) b.N times
12     m := len(numbers)
13     for n := 0; n < b.N; n++ {
14         F(numbers[n%m])
15     }
16 }

```

The BenchmarkF function can be saved in any file ending with `_test.go` to be included by the testing package. The only real surprise in the code is the for loop defined on line 13,

```

13 for n := 0; n < b.N; n++ {

```

The benchmark function must run the target code `b.N` times. During benchmark execution, `b.N` is adjusted until the benchmark function lasts long enough to be timed reliably.

To run the benchmark, we need to instruct `go test` to run benchmarks using the `-bench` flag. Similar to the `-run` command-line argument, `-bench` also accepts a regular expression to match the benchmark functions we want to run. To run all the benchmark functions, we provide `-bench=.` `go test` will first run all the tests (or those matched by `-run`, if provided), and then run the benchmarks. The output for our benchmark above looks is as follows:

```

$ go test -bench=.
goos: linux
goarch: amd64
BenchmarkF-4      1000      1255534 ns/op
PASS
ok      _/home/productiongo/benchmarks/fibonacci 1.387s

```

The output tells us that the benchmarks ran on a Linux x86-64 environment. Furthermore, the testing package executed our one benchmark, `BenchmarkF`. It ran the `b.N` loop 1000 times, and each iteration (i.e. each call to `F`) lasted 1,255,534ns ($\sim 1.2\text{ms}$) on average.

1.2ms per call seems a bit slow! Especially considering that the numbers we provided to the Fibonacci function were quite small. Let's improve our original function by not using recursion.

An improved Fibonacci function

```
1 package fibonacci
2
3 // FastF returns the nth Fibonacci number,
4 // but does not use recursion.
5 func FastF(n int) int {
6     var a, b int = 0, 1
7     for i := 0; i < n; i++ {
8         a, b = b, a+b
9     }
10    return a
11 }
```

This new function `FastF`, is equivalent to the original, but uses only two variables and no recursion to calculate the final answer. Neat! Let's check whether it's actually any faster. We can do this by adding a new benchmark function for `FastF`:

```
func BenchmarkFastF(b *testing.B) {
    // run FastF(n) b.N times
    m := len(numbers)
    for n := 0; n < b.N; n++ {
        FastF(numbers[n%m])
    }
}
```

Again we run `go test -bench=.`. This time we will see the output of both benchmarks:

```
$ go test -bench=.
goos: linux
goarch: amd64
BenchmarkF-4          1000      1245008 ns/op
BenchmarkFastF-4      50000000    20.3 ns/op
PASS
ok      _/home/productiongo/benchmarks/fibonacci 2.444s
```

The output is telling us that `F` still took around 1245008ns per execution, but `FastF` took only 20.3ns! The benchmark proves that our non-recursive `FastF` is indeed orders of magnitude faster than the textbook recursive version, at least for the provided inputs.

Comparing benchmarks

The `benchcmp` tool parses the output of two `go test -bench` runs and compares the results.

To install, run:

```
go get golang.org/x/tools/cmd/benchcmp
```

Let's output the benchmark for the original `F` function from earlier to a file, using `BenchmarkF`:

```
$ go test -bench . > old.txt
```

The file will look as follows:

```
goos: darwin
goarch: amd64
BenchmarkF-4          1000      1965113 ns/op
PASS
ok      _/Users/productiongo/benchmarks/benchcmp 2.173s
```

Now instead of implementing `FastF`, we copy the `FastF` logic into our original `F` function:

Fast F implementation

```

1 package fibbonaci
2
3 // F returns the nth Fibonacci number.
4 func F(n int) int {
5     var a, b int = 0, 1
6     for i := 0; i < n; i++ {
7         a, b = b, a+b
8     }
9     return a
10 }

```

and re-run the benchmark, outputting to a file called `new.txt`:

```
$ go test -bench . > new.txt
```

`new.txt` should look like this:

```

goos: darwin
goarch: amd64
BenchmarkF-4      500000000          25.0 ns/op
PASS
ok      _/Users/productiongo/benchmarks/benchcmp  1.289s

```

Now let's run `benchcmp` on the results:

```

benchcmp old.txt new.txt
benchmark      old ns/op    new ns/op    delta
BenchmarkF-4    1965113      25.0         -100.00%

```

We can see the old performance, new performance, and a delta. In this case, the new version of `F` performs so well that it reduced the runtime of the original by 99.9987%. Thus rounded to two decimals, we get a delta of -100.00%.

Resetting benchmark timers

We can reset the benchmark timer if we don't want the overall benchmark timing to include the execution time of our setup code.

A benchmark from the `crypto/aes` package in the Go source code provides an example of this:

crypto/aes BenchmarkEncrypt

```

1 package aes
2
3 import "testing"
4
5 func BenchmarkEncrypt(b *testing.B) {
6     tt := encryptTests[0]
7     c, err := NewCipher(tt.key)
8     if err != nil {
9         b.Fatalf("NewCipher:", err)
10    }
11    out := make([]byte, len(tt.in))
12    b.SetBytes(int64(len(out)))
13    b.ResetTimer()
14    for i := 0; i < b.N; i++ {
15        c.Encrypt(out, tt.in)
16    }
17 }

```

As we can see, there is some setup done in the benchmark, then a call to `b.ResetTimer()` to reset the benchmark time and memory allocation counters.

Benchmarking memory allocations

The Go benchmarking tools also allow us to output the number memory allocations by the benchmark, alongside the time taken by each iteration. We do this by adding the `-benchmem` flag. Let's see what happens if we do this on our Fibonacci benchmarks from before.

```

$ go test -bench=. -benchmem
goos: linux
goarch: amd64
BenchmarkF-4      1000      1241017 ns/op      0 B/op      0 \
allocs/op
BenchmarkFastF-4  100000000    20.6 ns/op      0 B/op      0 \
allocs/op
PASS
ok      _/Users/productiongo/benchmarks/fibonacci 3.453s

```


We now have two new columns on the right: the number of bytes per operation, and the number of heap allocations per operation. For our Fibonacci functions, both of these are zero. Why is this? Let's add the `-gcflags=-m` option to see the details. The output below is truncated to the first 10 lines:

```
$ go test -bench=. -benchmem -gcflags=-m
# _/home/herman/Dropbox/mastergo/manuscript/code/benchmarks/fibonacci
./fibonacci_bench_test.go:10:20: BenchmarkF b does not escape
./fibonacci_fast_bench_test.go:6:24: BenchmarkFastF b does not escape
./fibonacci_test.go:22:5: t.common escapes to heap
./fibonacci_test.go:6:15: leaking param: t
./fibonacci_test.go:22:38: tt.n escapes to heap
./fibonacci_test.go:22:38: got escapes to heap
./fibonacci_test.go:22:49: tt.want escapes to heap
./fibonacci_test.go:10:3: TestF []struct { n int; want int } literal does not\
escape
./fibonacci_test.go:22:12: TestF ... argument does not escape
...
```

The Go compiler performs [escape analysis](https://en.wikipedia.org/wiki/Escape_analysis)⁴¹. If an allocation does not escape the function, it can be stored on the stack. Variables placed on the stack avoid the costs involved with a heap allocation and the garbage collector. The omission of the `fibonacci.go` file from the output above implies that no variables from our `F` and `FastF` functions escaped to the heap. Let's take another look at the `FastF` function to see why this is:

```
func FastF(n int) int {
    var a, b int = 0, 1
    for i := 0; i < n; i++ {
        a, b = b, a+b
    }
    return a
}
```

In this function, the `a`, `b`, and `i` variables are declared locally and do not need to be put onto the heap, because they are not used again when the function exits. Consider what would happen if, instead of storing only the last two values, we naively stored all values calculated up to `n`:

⁴¹https://en.wikipedia.org/wiki/Escape_analysis

A high-memory implementation of F

```

1 package fibonacci
2
3 // FastHighMemF returns the nth Fibonacci number, but
4 // stores the full slice of intermediate
5 // results, consuming more memory than necessary.
6 func FastHighMemF(n int) int {
7     if n <= 0 {
8         return 0
9     }
10
11     r := make([]int, n+1)
12     r[0] = 0
13     r[1] = 1
14     for i := 2; i <= n; i++ {
15         r[i] = r[i-1] + r[i-2]
16     }
17     return r[n]
18 }

```

Running the same test and benchmark from before on this high-memory version of F, we get:

```

$ go test -bench=. -benchmem
goos: linux
goarch: amd64
BenchmarkFastHighMemF-4      20000000          72.0 ns/op      132 B/op      \
    0 allocs/op
PASS
ok      _/Users/productiongo/benchmarks/fibonacci_mem 1.518s

```

This time our function used 132 bytes per operation, due to our use of a slice in the function. If you are wondering why the number is 132 specifically: the exact number of bytes is sensitive to the numbers we use in the benchmark. The higher the input `n`, the more memory the function will allocate. The average of the values used in the benchmark (0, 10, 20, 30) is 15. Because this was compiled for a 64-bit machine, each `int` will use 8 bytes (8x8=64 bits). The slice headers also use some bytes. We still have zero heap allocations per operation, due to all variables being contained within the function. We will discuss advanced memory profiling and optimization techniques in [Optimization](#).

Modulo vs Bitwise-and

In our Fibonacci benchmarks so far, we have made use of a list of four integer test cases:

```
6  var nums = []int{
7      0, 10, 20, 30,
8  }
```

which we then loop over in the BenchmarkF function:

```
13  for n := 0; n < b.N; n++ {
14      FastF(nums[n%m])
15  }
```

But when it comes down to the nanoseconds, modulo is a relatively slow computation to do on every iteration. It can actually have an impact on the accuracy of our results! Let's peek at the Go assembler code. Go allows us to do this with the `go tool compile -S` command, which outputs a pseudo-assembly language called ASM. In the command below, we filter the instructions for the line we are interested in with `grep`:

```
$ go tool compile -S fibonacci.go fibonacci_bench_test.go | grep "fibonacci_b\
ench_test.go:14"
0x0036 00054 (fibonacci_bench_test.go:14)  MOVQ    (BX)(DX*8), AX
0x003a 00058 (fibonacci_bench_test.go:14)  MOVQ    AX, (SP)
0x003e 00062 (fibonacci_bench_test.go:14)  PCDATA  $0, $0
0x003e 00062 (fibonacci_bench_test.go:14)  CALL    ".F(SB)
0x0043 00067 (fibonacci_bench_test.go:14)  MOVQ    "..autotmp_5+16(SP),\
AX
0x0061 00097 (fibonacci_bench_test.go:14)  MOVQ    ".numbers(SB), BX
0x0068 00104 (fibonacci_bench_test.go:14)  MOVQ    ".numbers+8(SB), SI
0x006f 00111 (fibonacci_bench_test.go:14)  TESTQ   CX, CX
0x0072 00114 (fibonacci_bench_test.go:14)  JEQ     161
0x0074 00116 (fibonacci_bench_test.go:14)  MOVQ    AX, DI
0x0077 00119 (fibonacci_bench_test.go:14)  CMPQ    CX, $-1
0x007b 00123 (fibonacci_bench_test.go:14)  JEQ     132
0x007d 00125 (fibonacci_bench_test.go:14)  CQO
0x007f 00127 (fibonacci_bench_test.go:14)  IDIVQ   CX
0x0082 00130 (fibonacci_bench_test.go:14)  JMP     137
0x0084 00132 (fibonacci_bench_test.go:14)  NEGQ    AX
```

```

0x0087 00135 (fibonacci_bench_test.go:14) XORL    DX, DX
0x0089 00137 (fibonacci_bench_test.go:14) CMPQ    DX, SI
0x008c 00140 (fibonacci_bench_test.go:14) JCS     49
0x008e 00142 (fibonacci_bench_test.go:14) JMP     154
0x009a 00154 (fibonacci_bench_test.go:14) PCDATA  $0, $1
0x009a 00154 (fibonacci_bench_test.go:14) CALL    runtime.panicindex(SB)
0x009f 00159 (fibonacci_bench_test.go:14) UNDEF
0x00a1 00161 (fibonacci_bench_test.go:14) PCDATA  $0, $1
0x00a1 00161 (fibonacci_bench_test.go:14) CALL    runtime.panicdivide(S\
B)
0x00a6 00166 (fibonacci_bench_test.go:14) UNDEF
0x00a8 00168 (fibonacci_bench_test.go:14) NOP

```

The details of this output are not as important as it is to notice how many instructions there are. Now, let's rewrite the code to use bitwise-and (&) instead of modulo %:

```

12     m := len(nums)-1
13     for n := 0; n < b.N; n++ {
14         FastF(nums[n&m])
15     }

```

Now, the ASM code becomes:

```

$ go tool compile -S fibonacci.go fibonacci_bench_test.go | grep "fibonacci_b\
ench_test.go:14"
0x0032 00050 (fibonacci_bench_test.go:14) MOVQ    (BX)(DI*8), AX
0x0036 00054 (fibonacci_bench_test.go:14) MOVQ    AX, (SP)
0x003a 00058 (fibonacci_bench_test.go:14) PCDATA  $0, $0
0x003a 00058 (fibonacci_bench_test.go:14) CALL    "".F(SB)
0x003f 00063 (fibonacci_bench_test.go:14) MOVQ    "".n+16(SP), AX
0x005e 00094 (fibonacci_bench_test.go:14) MOVQ    "".numbers(SB), BX
0x0065 00101 (fibonacci_bench_test.go:14) MOVQ    "".numbers+8(SB), SI
0x006c 00108 (fibonacci_bench_test.go:14) LEAQ    -1(AX), DI
0x0070 00112 (fibonacci_bench_test.go:14) ANDQ    CX, DI
0x0073 00115 (fibonacci_bench_test.go:14) CMPQ    DI, SI
0x0076 00118 (fibonacci_bench_test.go:14) JCS     45
0x0078 00120 (fibonacci_bench_test.go:14) JMP     132
0x0084 00132 (fibonacci_bench_test.go:14) PCDATA  $0, $1
0x0084 00132 (fibonacci_bench_test.go:14) CALL    runtime.panicindex(SB)

```

```
0x0089 00137 (fibonacci_bench_test.go:14) UNDEF
0x008b 00139 (fibonacci_bench_test.go:14) NOP
```

This is considerably shorter than before. In other words, the Go runtime will need to perform fewer operations, but the results will be the same. We can use modulo instead of ampersand because we have exactly four items in our `nums` slice. In general, $n \% m == n \& (m - 1)$ if m is a power of two. For example,

```
0 % 4 = 0 & 3 = 0
1 % 4 = 1 & 3 = 1
2 % 4 = 2 & 3 = 2
3 % 4 = 3 & 3 = 3
4 % 4 = 4 & 3 = 0
5 % 4 = 5 & 3 = 1
6 % 4 = 6 & 3 = 2
...
```

If you are not yet convinced, expand the binary version of the bitwise-and operations to show that this is true:

```
0 & 3 = 00b & 11b = 0
1 & 3 = 01b & 11b = 1
2 & 3 = 10b & 11b = 2
3 & 3 = 11b & 11b = 3
...
```

To evaluate the impact of changing from modulo to ampersand on the benchmark results, let us create two benchmarks for `FastF`, one with modulo and the other with bitwise-and:

```
$ go test -bench=BenchmarkFastF
goos: linux
goarch: amd64
BenchmarkFastFModulo-4          1000000000      16.7 ns/op
BenchmarkFastFBitwiseAnd-4      2000000000      7.40 ns/op
PASS
ok      _/Users/productiongo/benchmarks/bench_bitwise_and 4.058s
```

The version using bitwise-and runs twice as fast. Our original benchmark was spending half the time recalculating modulo operations! This is unlikely to have a big impact on benchmarks of bigger functions, but when benchmarking small pieces of code, using bitwise-and instead of modulo will make the benchmark results more accurate.

Tooling

In this chapter we will discuss some tooling you may find useful for writing and running production Go applications.

Godoc

Godoc uses the comments in your code to generate documentation. You can use it via the command line, or as an HTTP server where it will generate HTML pages.

To install, run:

```
go get golang.org/x/tools/cmd/godoc
```

You can run `godoc` on the standard library, or packages in your own `GOPATH`. To see the documentation for `encoding/json` for example, run:

```
$ godoc encoding/json
```

To see the documentation for a specific function, such as `Marshal`:

```
$ godoc encoding/json Marshal
```

To run the HTTP server locally:

```
$ godoc -http:6060
```

This will run a `godoc` HTTP server locally, where you can see the generated HTML documentation for packages in your `GOPATH` as well as the standard library.

There is also a hosted version of `godoc` at <https://godoc.org>. If you host your code on GitHub for example, `godoc.org` can generate the documentation for it. It's a good idea to keep your comments clean and up to date in case others are checking your package's page on `godoc.org`. `Golint` is useful for surfacing parts of your code that need comments.

More can be found at the official Go blog post "[Godoc: documenting Go code](https://blog.golang.org/godoc-documenting-go-code)"⁴², but to summarize the main points:

"The convention is simple: to document a type, variable, constant, function, or even a package, write a regular comment directly preceding its declaration, with no intervening blank line."

Package-level comments take the form of a comment directly above the package declaration, starting with "Package [name] ...", like so:

⁴²<https://blog.golang.org/godoc-documenting-go-code>

```
1 // Package sort provides primitives for sorting slices and user-defined
2 // collections.
3 package sort
```

but if your package comment is long, you can split it out into a separate `doc.go` file, which only contains the comment and a package clause. See <http://doc.go>⁴³ for an example.

Go Guru

Go Guru is “a tool for answering questions about Go source code.”⁴⁴ It ships with a command-line tool, but you can also integrate it into an editor. You can install it with:

```
go get -u golang.org/x/tools/cmd/guru
```

You can find a list of supported editor integrations on the [Using Go Guru](#)⁴⁵ document linked in the godoc for guru. In this section we'll show screenshots of guru in vim.

We'll assume you have `vim-go` installed; if not please see the “Editor Integration” section of the “Installing Go” chapter in the beginning of this book.

Let's take a look at one of the questions that guru answers for us: "what concrete types implement this interface?"

Navigate to the line that contains an interface, and type `:GoImplements:`

```
vim check/check.go — vim — vim check/check.go
1 package check
2
3 // Check describes what methods various checks (gofmt, go lint, etc.)
4 // should implement
5 type Check interface {
6     Name() string
7     Description() string
8     Weight() float64
9     // Percentage returns the passing percentage of the check,
10    // as well as a map of filename to output
11    Percentage() (float64, []FileSummary, error)
12 }
~
~
~
~
~
~
~
~
:GoImplements
```

Then hit Enter. Your vim window should split and your cursor should be in the quickfix list on the bottom half, with a list of files containing concrete structs that implement the interface:

⁴³<https://github.com/golang/go/blob/master/src/net/http/doc.go>

⁴⁴<https://godoc.org/golang.org/x/tools/cmd/guru>

⁴⁵<http://golang.org/s/using-guru>

```

vim check/check.go -- vim -- vim check/check.go
3 // Check describes what methods various checks (gofmt, go lint, etc.)
4 // should implement
5 type Check interface {
6     Name() string
7     Description() string
8     Weight() float64
9     // Percentage returns the passing percentage of the check,
10    // as well as a map of filename to output
11    Percentage() (float64, []FileSummary, error)
}
check/check.go 5,1 66%
1 check/check.go:5 col 6: interface type Check
2 check/errcheck.go:4 col 6: is implemented by struct type ErrCheck
3 check/gocyclo.go:4 col 6: is implemented by struct type GoCyclo
4 check/gofmt.go:4 col 6: is implemented by struct type GoFmt
5 check/golint.go:4 col 6: is implemented by struct type GoLint
6 check/go_vet.go:4 col 6: is implemented by struct type GoVet
7 check/ineffassign.go:4 col 6: is implemented by struct type IneffAssign
8 check/license.go:10 col 6: is implemented by struct type License
9 check/misspell.go:4 col 6: is implemented by struct type Misspell
[Location List] implements 1,1 All
vim-go: analysing with scope github.com/gojp/goreportcard/check ...

```

Hit Enter on any one of those and you'll jump to the struct definition in the listed file. To get back to the list, do `<Ctrl-W> j`. Then you can scroll through it as before, or quit out of it as usual with `:q`.

Go ahead and try some of the other guru commands like `:GoReferrers`, `:GoCallees`, and `:GoCallers`. You can find more help on guru for `vim-go` at the [vim-go-tutorial](https://github.com/fatih/vim-go-tutorial#guru)⁴⁶, and a list of guru queries and other help output with `guru -help`.

Race Detector

Go comes with a builtin [mechanism for detecting race conditions](https://golang.org/doc/articles/race_detector.html)⁴⁷. There are multiple ways to invoke it:

```

$ go test -race mypkg    // to test the package
$ go run -race mysrc.go  // to run the source file
$ go build -race mycmd   // to build the command
$ go install -race mypkg // to install the package

```

In this section we'll write some code that contains a data race, and catch it with a test with the race detector enabled. A data race occurs when two goroutines try to access the same object in memory concurrently, one of which is trying to write to the object, and there is no lock in place to control access to the object.

⁴⁶<https://github.com/fatih/vim-go-tutorial#guru>

⁴⁷https://golang.org/doc/articles/race_detector.html

A simple program with an asynchronous update method

```
1 package cat
2
3 import (
4     "log"
5 )
6
7 // Cat is a small, carnivorous mammal with excellent night vision
8 type Cat struct {
9     noise string
10 }
11
12 // SetNoise sets the noise that our cat makes
13 func (c *Cat) SetNoise(n string) {
14     c.noise = n
15 }
16
17 // Noise makes the cat make a noise
18 func (c *Cat) Noise() string {
19     return c.noise
20 }
21
22 func updateCat(c *Cat) {
23     go c.SetNoise("にゃん")
24
25     log.Println(c.Noise())
26 }
```

In the above code, we execute a goroutine that sets the `Noise` attribute of the `Cat` argument to `"にゃん"`. That goroutine goes off and runs in the background and the flow of execution continues to where we attempt to `log c.Noise`. This causes a race condition as we might write the value in the goroutine at the same time as reading it in the `log.Println` call.

Without considering the race condition and reading the code from top to bottom, we expect `updateCat` to set the `noise` attribute for the passed-in `"cat"` to `"にゃん"`. So let's write a test that makes that assertion for us:

A test for the simple program with an update method

```

1 package cat
2
3 import (
4     "testing"
5 )
6
7 func TestUpdateCat(t *testing.T) {
8     c := Cat{noise: "meow"}
9     c = updateCat(c)
10    if got := c.Noise(); got != "にゃん" {
11        t.Fatalf("c.Noise() = %q, want %q", got, "にゃん")
12    }
13 }

```

When running this test normally with `go test`, we might get lucky and it will pass:

```

go test
2017/09/24 16:14:52 meow
PASS
ok      github.com/gopher/cat      0.007

```

When we run the test with the race detector enabled, however, we'll see something different:

```

$ go test -race
=====
WARNING: DATA RACE
Write at 0x00c4200783e0 by goroutine 7:
    github.com/gopher/cat.(*Cat).SetNoise()
        /Users/gopher/mygo/src/github.com/gopher/cat/cat.go:14 +0x3b

Previous read at 0x00c4200783e0 by goroutine 6:
    github.com/gopher/cat.updateCat()
        /Users/gopher/mygo/src/github.com/gopher/cat/cat.go:19 +0x76
    github.com/gopher/cat.TestUpdateCat()
        /Users/gopher/mygo/src/github.com/gopher/cat/cat_test.go:9 +0x88
    testing.tRunner()
        /Users/gopher/go/src/testing/testing.go:746 +0x16c

```

```

Goroutine 7 (running) created at:
github.com/gopher/cat.updateCat()
    /Users/gopher/mygo/src/github.com/gopher/cat/cat.go:23 +0x65
github.com/gopher/cat.TestUpdateCat()
    /Users/gopher/mygo/src/github.com/gopher/cat/cat_test.go:9 +0x88
testing.tRunner()
    /Users/gopher/go/src/testing/testing.go:746 +0x16c

Goroutine 6 (running) created at:
testing.(*T).Run()
    /Users/gopher/go/src/testing/testing.go:789 +0x568
testing.runTests.func1()
    /Users/gopher/go/src/testing/testing.go:1004 +0xa7
testing.tRunner()
    /Users/gopher/go/src/testing/testing.go:746 +0x16c
testing.runTests()
    /Users/gopher/go/src/testing/testing.go:1002 +0x521
testing.(*M).Run()
    /Users/gopher/go/src/testing/testing.go:921 +0x206
main.main()
    github.com/gopher/cat/_test/_testmain.go:44 +0x1d3
=====
2017/09/24 18:12:58 meow
--- FAIL: TestUpdateCat (0.00s)
    testing.go:699: race detected during execution of test
FAIL
exit status 1
FAIL    github.com/gopher/cat    0.013s

```

That's a lot of output, but let's take a look at the first two blocks of text:

WARNING: DATA RACE

Write at 0x00c4200783e0 by goroutine 7:

```
github.com/gopher/cat.(*Cat).SetNoise()
/Users/gopher/mygo/src/github.com/gopher/cat/cat.go:14 +0x3b
```

Previous read at 0x00c4200783e0 by goroutine 6:

```
github.com/gopher/cat.updateCat()
/Users/gopher/mygo/src/github.com/gopher/cat/cat.go:19 +0x76
github.com/gopher/cat.TestUpdateCat()
/Users/gopher/mygo/src/github.com/gopher/cat/cat_test.go:9 +0x88
testing.tRunner()
/Users/gopher/go/src/testing/testing.go:746 +0x16c
```

This tells us exactly where our data race is. Our package's filename is `cat.go`, so if we narrow it down to the lines containing our file, we can see the write occurred here:

```
/Users/gopher/mygo/src/github.com/gopher/cat/cat.go:14
```

and the read here:

```
/Users/gopher/mygo/src/github.com/gopher/cat/cat.go:19
```

And indeed if we check our code, those are the lines where we attempt to set `c.noise`, as well as read `c.noise`.

So how do we fix this? We're going to need a lock around our data structure. We'll use a `sync.Mutex`⁴⁸ to lock our `Cat` structure whenever we read or write to it:

Cat program with `sync.Mutex`

```
1 package cat
2
3 import (
4     "log"
5     "sync"
6 )
7
8 // Cat is a small, carnivorous mammal with excellent night vision
9 type Cat struct {
10     mu    sync.Mutex
```

⁴⁸<https://golang.org/pkg/sync/#Mutex>

```
11         noise string
12     }
13
14     // SetNoise sets the noise that our cat makes
15     func (c *Cat) SetNoise(n string) {
16         c.mu.Lock()
17         defer c.mu.Unlock()
18         c.noise = n
19     }
20
21     // Noise makes the cat make a noise
22     func (c *Cat) Noise() string {
23         c.mu.Lock()
24         defer c.mu.Unlock()
25
26         return c.noise
27     }
28
29     func updateCat(c *Cat) *Cat {
30         go c.SetNoise("にゃん")
31
32         log.Println(c.Noise())
33
34         return c
35     }
```

Now we run the test again:

```
go test -race
2017/09/24 18:09:38 meow
PASS
ok          github.com/gopher/cat    1.018s
```

and the race condition is fixed.

You can also build your application with the race detector enabled, and see potential data races while running the application, or during an integration test. As an example, let's make an API that accepts user-contributed entries about the countries the user has visited, and a description of their trip:

A simple API with an asynchronous update method

```
1 package main
2
3 import (
4     "encoding/json"
5     "flag"
6     "fmt"
7     "log"
8     "net/http"
9     "strings"
10    "unicode"
11 )
12
13 var (
14     addr      = flag.String("http", "127.0.0.1:8000", "HTTP listen address")
15     countries = map[string]string{
16         "england":      "England",
17         "japan":         "Japan",
18         "southafrica":  "South Africa",
19         "unitedstates": "United States",
20     }
21 )
22
23 // Entry is a user-submitted entry summarizing their trip
24 // to a given country
25 type Entry struct {
26     Country string
27     Text    string
28 }
29
30 // removeSpaces is a function for strings.Map that removes
31 // all spaces from a string
32 func removeSpaces(r rune) rune {
33     if unicode.IsSpace(r) {
34         return -1
35     }
36     return r
37 }
38
```

```
39 // normalizeCountry takes an Entry and normalizes its country name
40 func normalizeCountry(e *Entry) error {
41     co := strings.Map(removeSpaces, strings.ToLower(e.Country))
42     if _, ok := countries[co]; !ok {
43         return fmt.Errorf("invalid country %q", e.Country)
44     }
45
46     e.Country = countries[co]
47
48     return nil
49 }
50
51 // EntryPostHandler is the POST endpoint for entries
52 func EntryPostHandler(w http.ResponseWriter, req *http.Request) {
53     decoder := json.NewDecoder(req.Body)
54     defer req.Body.Close()
55     var entry Entry
56     err := decoder.Decode(&entry)
57     if err != nil {
58         log.Printf("ERROR: could not decode Entry: %s", err)
59         w.WriteHeader(http.StatusBadRequest)
60         w.Write([]byte(fmt.Sprintf("Could not decode Entry: %s\n", err)))
61         return
62     }
63
64     go func(e *Entry) {
65         normalizeCountry(e)
66     }(&entry)
67
68     log.Printf("INFO: received Entry %v", entry)
69 }
70
71 func main() {
72     http.HandleFunc("/entries", EntryPostHandler)
73
74     log.Printf("Running on %s ...", *addr)
75     log.Fatal(http.ListenAndServe(*addr, nil))
76 }
```

You can probably spot the race condition already - we're trying to normalize the country name in a goroutine in the background, then immediately trying to log the entry. Running the server with `go run` normally won't give us any errors:

```
$ go run server.go
2017/09/24 19:35:50 Running on 127.0.0.1:8000 ...
```

and we can even POST an Entry:

```
$ curl --data '{"country": "enGLaND", "text": "A country that is part of the \
United Kingdom"}' localhost:8000/entries
```

we then see this on the server:

```
2017/09/24 19:52:56 INFO: received Entry {enGLaND A country that is part of t\
he United Kingdom}
```

which is wrong - we're supposed to be normalizing the country name to "England". Let's see what happens when we run the server with `-race` enabled:

```
$ go run -race server.go
2017/09/24 20:06:17 Running on 127.0.0.1:8000 ...
```

That looks fine, but now let's try to POST an Entry again:

```
$ curl --data '{"country": "enGLaND", "text": "A country that is part of the \
United Kingdom"}' localhost:8000/entries
```

```
2017/09/24 20:06:19 INFO: received Entry {enGLaND A country that is part of t\
he United Kingdom}
```

```
=====
```

```
WARNING: DATA RACE
```

```
Write at 0x00c4200f2340 by goroutine 8:
```

```
    main.normalizeCountry()
        /Users/gopher/mygo/src/github.com/gopher/country_journal/server.go:46 +\
0x16d
```

```
    main.EntryPostHandler.func1()
        /Users/gopher/mygo/src/github.com/gopher/country_journal/server.go:65 +\
0x38
```


Previous read at 0x00c4200f2340 by goroutine 6:

```
main.EntryPostHandler()
    /Users/gopher/mygo/src/github.com/gopher/country_journal/server.go:68 +\
0x439
net/http.HandlerFunc.ServeHTTP()
    /Users/gopher/go/src/net/http/server.go:1918 +0x51
net/http.(*ServeMux).ServeHTTP()
    /Users/gopher/go/src/net/http/server.go:2254 +0xa2
net/http.serverHandler.ServeHTTP()
    /Users/gopher/go/src/net/http/server.go:2619 +0xbc
net/http.(*conn).serve()
    /Users/gopher/go/src/net/http/server.go:1801 +0x83b
```

Goroutine 8 (running) created at:

```
main.EntryPostHandler()
    /Users/gopher/mygo/src/github.com/gopher/country_journal/server.go:64 +\
0x41c
net/http.HandlerFunc.ServeHTTP()
    /Users/gopher/go/src/net/http/server.go:1918 +0x51
net/http.(*ServeMux).ServeHTTP()
    /Users/gopher/go/src/net/http/server.go:2254 +0xa2
net/http.serverHandler.ServeHTTP()
    /Users/gopher/go/src/net/http/server.go:2619 +0xbc
net/http.(*conn).serve()
    /Users/gopher/go/src/net/http/server.go:1801 +0x83b
```

Goroutine 6 (running) created at:

```
net/http.(*Server).Serve()
    /Users/gopher/go/src/net/http/server.go:2720 +0x37c
net/http.(*Server).ListenAndServe()
    /Users/gopher/go/src/net/http/server.go:2636 +0xc7
net/http.ListenAndServe()
    /Users/gopher/go/src/net/http/server.go:2882 +0xfe
main.main()
    /Users/gopher/mygo/src/github.com/gopher/country_journal/server.go:75 +\
0x14f
```

=====

and there is our data race. We could fix this race in a similar manner to the way we fixed the cat race earlier, but instead let's try using a `sync.WaitGroup`⁴⁹:

Country journal API with `sync.WaitGroup`

```
1 package main
2
3 import (
4     "encoding/json"
5     "flag"
6     "fmt"
7     "log"
8     "net/http"
9     "strings"
10    "sync"
11    "unicode"
12 )
13
14 var (
15     addr      = flag.String("http", "127.0.0.1:8000", "HTTP listen address")
16     countries = map[string]string{
17         "england":      "England",
18         "japan":         "Japan",
19         "southafrica":  "South Africa",
20         "unitedstates": "United States",
21     }
22 )
23
24 // Entry is a user-submitted entry summarizing their trip
25 // to a given country
26 type Entry struct {
27     Country string
28     Text    string
29 }
30
31 // removeSpaces is a function for strings.Map that removes
32 // all spaces from a string
33 func removeSpaces(r rune) rune {
34     if unicode.IsSpace(r) {
```

⁴⁹<https://golang.org/pkg/sync/#WaitGroup>

```
35         return -1
36     }
37     return r
38 }
39
40 // normalizeCountry takes an Entry and normalizes its country name
41 func normalizeCountry(e *Entry) error {
42     co := strings.Map(removeSpaces, strings.ToLower(e.Country))
43     if _, ok := countries[co]; !ok {
44         return fmt.Errorf("invalid country %q", e.Country)
45     }
46
47     e.Country = countries[co]
48
49     return nil
50 }
51
52 // EntryPostHandler is the POST endpoint for entries
53 func EntryPostHandler(w http.ResponseWriter, req *http.Request) {
54     decoder := json.NewDecoder(req.Body)
55     defer req.Body.Close()
56     var entry Entry
57     err := decoder.Decode(&entry)
58     if err != nil {
59         log.Printf("ERROR: could not decode Entry: %s", err)
60         w.WriteHeader(http.StatusBadRequest)
61         w.Write([]byte(fmt.Sprintf("Could not decode Entry: %s\n", err)))
62         return
63     }
64
65     var wg sync.WaitGroup
66     wg.Add(1)
67     go func(e *Entry) {
68         defer wg.Done()
69         normalizeCountry(e)
70     }(&entry)
71     wg.Wait()
72
73     log.Printf("INFO: received Entry %v", entry)
```

```
74 }
75
76 func main() {
77     http.HandleFunc("/entries", EntryPostHandler)
78
79     log.Printf("Running on %s ...", *addr)
80     log.Fatal(http.ListenAndServe(*addr, nil))
81 }
```

You can see that we now have a `sync.WaitGroup`, onto which we add a delta of 1 to the counter. Inside the goroutine we decrement the counter with `defer wg.Done()`, then we block until the counter is zero with `wg.Wait()`. Since we're blocking until the goroutine finishes, there is no longer a data race:

```
$ go run -race server.go
2017/09/24 20:08:36 Running on 127.0.0.1:8000 ...
```

```
$ curl --data '{"country": "enGLaND", "text": "A country that is part of the \
United Kingdom"}' localhost:8000/entries
```

```
2017/09/24 20:08:44 INFO: received Entry {England A country that is part of t\
he United Kingdom}
```

and we see that our country is normalized now to “England”.

Go Report Card

Full disclosure: we are the authors of this free and open source tool.

[Go Report Card](https://goreportcard.com/)⁵⁰ is a web application that gives packages a grade based on how well they pass various linters and tools. It is a popular application in the open source community, with thousands of projects using the badge to indicate the code quality. You can try it on goreportcard.com if your source code is open source, or run the server locally to use the tool on your internal network or private repositories.

⁵⁰<https://goreportcard.com/>

Staticcheck

[Staticcheck](http://staticcheck.io/)⁵¹ is a static analysis tool for Go programs. It helps for simplifying code and catches some issues such as unused error values. It also shows some style suggestions such as suggesting that the first character of an error string should not be capitalized.⁵² Here is an example of running staticcheck on the Go Report Card source:

```
❯ goreportcard git:(master) staticcheck ./...
check/grade.go:10:2: only the first constant in this group has an explicit type (SA9004)
check/utils.go:374:25: HasSuffix is a pure function but its return value is ignored (SA4017)
handlers/check.go:79:6: should use strings.EqualFold(a, b) instead of strings.ToLower(a) == strings.ToLower(b) (SA6005)
handlers/checks.go:40:21: error strings should not be capitalized (ST1005)
tools/db/manage_db.go:41:7: should use strings.EqualFold(a, b) instead of strings.ToLower(a) == strings.ToLower(b) (SA6005)
```

And here is another example of staticcheck being run on code that doesn't set an error value properly:

Improper error variable assignment

```
1 package main
2
3 import (
4     "errors"
5     "fmt"
6 )
7
8 func badFunc() error {
9     return errors.New("an error has occurred")
10 }
11
12 func foo() error {
13     return nil
14 }
15
16 func main() {
```

⁵¹<http://staticcheck.io/>

⁵²<https://github.com/golang/go/wiki/CodeReviewComments#error-strings>

```
17     var err error
18
19     if err := foo(); err == nil {
20         err = badFunc()
21     }
22
23     fmt.Println(err)
24 }
```

We might think that the `err = badFunc()` line would set the outside `var err error` to `errors.New("an error has occurred")`, but actually when we print the `err` in the main scope it prints `<nil>`:

```
$ go run err_val_staticcheck.go
<nil>
```

Staticcheck will alert us that the `err` value inside the `if` statement is never used:

```
$ staticcheck err_val_staticcheck.go
err_val_staticcheck.go:20:3: this value of err is never used (SA4006)
```

Go-fuzz

[Go-fuzz](https://github.com/dvyukov/go-fuzz)⁵³ is a randomized testing tool for Go packages. [Fuzzing](https://en.wikipedia.org/wiki/Fuzzing)⁵⁴ is a method of testing code by providing various types of input to a program and monitoring that program for crashes or memory leaks. Having an automated fuzzing process is a great way to get ahead of potential input bugs and allows us to write more robust programs.

To download `go-fuzz`, run:

```
go get -u github.com/dvyukov/go-fuzz/go-fuzz github.com/dvyukov/go-fuzz/go-fuzz-build
```

Now we must add a `Fuzz` function in the package we want to fuzz. Let's use [Nihongo.io](http://github.com/gojp/nihongo)⁵⁵, a Japanese dictionary written in Go, as an example:

⁵³<https://github.com/dvyukov/go-fuzz>

⁵⁴<https://en.wikipedia.org/wiki/Fuzzing>

⁵⁵<http://github.com/gojp/nihongo>

Nihongo.io dictionary fuzzing

```
1 package dictionary
2
3 import (
4     "compress/gzip"
5     "log"
6     "os"
7 )
8
9 func Fuzz(data []byte) int {
10     file, err := os.Open("../data/edict2.json.gz")
11     if err != nil {
12         log.Fatal("Could not load edict2.json.gz: ", err)
13     }
14     defer file.Close()
15
16     reader, err := gzip.NewReader(file)
17     if err != nil {
18         log.Fatal("Could not create reader: ", err)
19     }
20
21     dict, err := Load(reader)
22     if err != nil {
23         log.Fatal("Could not load dictionary: ", err)
24     }
25
26     dict.Search(string(data), 10)
27
28     return 1
29 }
```

This file exists in the github.com/gojp/nihongo/lib/dictionary package. We then run `go-fuzz-build` and then `go-fuzz` (in our case with a high `-timeout` flag since it takes some time to load the dictionary) to start the fuzzing:

```
$ go-fuzz-build
```

```
$ go-fuzz -timeout=999
```

```
2019/08/19 14:58:09 workers: 4, corpus: 1 (3s ago), crashers: 0, restarts: 1/\
0, execs: 0 (0/sec), cover: 0, uptime: 3s
2019/08/19 14:58:12 workers: 4, corpus: 1 (6s ago), crashers: 0, restarts: 1/\
0, execs: 0 (0/sec), cover: 0, uptime: 6s
2019/08/19 14:58:15 workers: 4, corpus: 1 (9s ago), crashers: 0, restarts: 1/\
0, execs: 0 (0/sec), cover: 0, uptime: 9s
2019/08/19 14:58:18 workers: 4, corpus: 1 (12s ago), crashers: 0, restarts: 1/\
0, execs: 0 (0/sec), cover: 0, uptime: 12s
```

```
...
```

and we see that fuzzer is running successfully.

According to the [go-fuzz README](#)⁵⁶:

The function must return 1 if the fuzzer should increase priority of the given input during subsequent fuzzing (for example, the input is lexically correct and was parsed successfully); -1 if the input must not be added to corpus even if gives new coverage; and 0 otherwise; other values are reserved for future use.

govalidate

[govalidate](#)⁵⁷ is a useful command for validating your Go environment. It runs such checks as whether you have the latest Go version installed, whether your `$PATH` environment variable contains `$GOPATH/bin`, and others.

Pinning tools with Go modules

It's good to ensure that new contributors can clone a repo and run all the necessary tools. If a project enforces in the build that `staticcheck` must pass, contributors should be able to run it locally. One way to do this is with a `tools.go` file.

Let's assume we want contributors and our build to run `staticcheck`. We create a `tools/` directory with a `tools.go` file:

⁵⁶<https://github.com/dvyukov/go-fuzz>

⁵⁷<https://github.com/rakyll/govalidate>

tools.go

```
1 // +build tools
2
3 package tools
4
5 import (
6     _ "honnef.co/go/tools/cmd/staticcheck"
7 )
```

Our Makefile can have:

```
staticcheck:
    @[ -x "$(shell which staticcheck)" ] || go install honnef.co/go/tools/cmd\
/staticcheck
    staticcheck ./...
```

and when we run `make staticcheck`, if we don't already have `staticcheck` installed, `go` will install it based on the generated entry in `go.mod`.

Sometimes we want our projects to use specific versions of these tools. To do this, modify the tool's version in `go.mod`.

Security

In this chapter, we'll discuss a few ways to improve the security of our Go server. Most are quick and easy to implement. In reality a large production environment will require a robust security setup with a Web Application Firewall (WAF) and other tools, but here we will only discuss improvements for our Go code.

Keep Go Up to Date

One of the most important things we can do to ensure the security of our Go environment is to keep our Go version up to date.

The [golang-announce](https://groups.google.com/forum/#!forum/golang-announce)⁵⁸ mailing list will announce (in a message prefixed with [security]) when a minor revision will be released with security fixes. For example, Go version 1.11.3 was released in December 2018 with a few security fixes, including a fix in `go get -u` for a remote execution bug.

See more about Go's security policy [here](https://golang.org/security)⁵⁹.

CSRF

Cross-Site Request Forgery, or CSRF, is “an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated.”⁶⁰

Imagine a user is logged into a bank's website and receives a link from someone with malicious intent. The HTML of the malicious page redirects the user to the bank's website and the money transfer form is pre-filled with the malicious user's account and routing information. The malicious HTML might look like this:

⁵⁸<https://groups.google.com/forum/#!forum/golang-announce>

⁵⁹<https://golang.org/security>

⁶⁰[https://www.owasp.org/index.php/Cross-Site_Request_Forgery_\(CSRF\)](https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF))

CSRF example

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="UTF-8" />
5     <title>CSRF</title>
6   </head>
7   <body onload="document.forms[0].submit()">
8     <form action="http://insecurebank.com/transfer" onsubmit="" method="POST">
9       <input type="hidden" name="account" value="5555555555" />
10      <input type="hidden" name="routing" value="5555555555" />
11      <input type="hidden" name="action" value="save" />
12    </form>
13  </body>
14 </html>
```

This attack might work if the bank is not protected against CSRF attacks.

To prevent CSRF attacks in our application, we must generate a unique per-user CSRF token and embed it in our HTML forms as a hidden field. The token is generated per-session, and must be sufficiently unique such that an attacker cannot guess it. If our web server doesn't receive a CSRF token in a POST, or the token received doesn't match the token from the user's session, the request should be denied.

Luckily there are packages such as [nosurf](https://github.com/justinas/nosurf)⁶¹ and [gorilla/csrf](https://github.com/gorilla/csrf)⁶² which make it easy to protect our web application against CSRF attacks.

HTTP Strict Transport Security (HSTS)

HTTP Strict Transport Security helps protect websites against protocol downgrade attacks and cookie hijacking.⁶³

Setting the `Strict-Transport-Security` on our web server will allow us to tell clients (web browsers) to only access our site over HTTPS, and never HTTP. A `max-age` expiration in seconds must be set with the header. In an example Go web server it could look like this, with a wrapper handler:

⁶¹<https://github.com/justinas/nosurf>

⁶²<https://github.com/gorilla/csrf>

⁶³https://en.wikipedia.org/wiki/HTTP_Strict_Transport_Security

HSTS example

```
1 package main
2
3 import (
4     "flag"
5     "fmt"
6     "log"
7     "net/http"
8 )
9
10 var (
11     addr = flag.String("http", ":8000", "HTTP listen address")
12 )
13
14 func headerWrap(h http.HandlerFunc) http.HandlerFunc {
15     return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
16         w.Header().Add("Strict-Transport-Security", "max-age=31536000; includeSubDo\
17 mains")
18         h.ServeHTTP(w, r)
19     })
20 }
21
22 func homeHandler(w http.ResponseWriter, r *http.Request) {
23     fmt.Fprint(w, "Hello, world")
24 }
25
26 func main() {
27     flag.Parse()
28     http.HandleFunc("/", headerWrap(homeHandler))
29
30     log.Printf("Running on %s ...", *addr)
31     log.Fatal(http.ListenAndServe(*addr, nil))
32 }
```

If we start the server and then run `curl -i localhost:8000` in another terminal, we can see that the header is set:

HSTS curl

```
$ curl -i localhost:8000
HTTP/1.1 200 OK
Strict-Transport-Security: max-age=31536000; includeSubDomains
Date: Thu, 20 Dec 2018 12:35:52 GMT
Content-Length: 12
Content-Type: text/plain; charset=utf-8
```

```
Hello, world%
```

We would wrap all of our handlers with `headerWrap` to ensure they all have the proper headers set.

Another option would be to set the header in the web server configuration, such as in `/etc/nginx/sites-available/[config]`.

Content Security Policy (CSP)

A `Content-Security-Policy` header helps prevent attacks such as cross-site scripting (XSS). The CSP header value allows us to specify the origins of our content.

For example, to ensure that all content comes from the site's own origin, the following header can be set in the response:

```
1 Content-Security-Policy: default-src 'self'
```

SQL Injection

SQL injection is a common website vulnerability in which an attacker enters a query into a text input box in order to either retrieve more data than is meant to be returned, or to run arbitrary commands on the database. Any backend code which creates SQL strings directly from HTML forms is vulnerable. For example, if we have code in our backend for a form on our website that looks like this:

```
1 username := r.FormValue("username")
2 query := "SELECT * from users WHERE username=' " + username + "';"
```

then an attacker could put the following malicious query into the HTML form that passes `username` to the backend:

```
1 ' OR 1=1; '
```

and we would end up with a query that looks like this:

```
1 SELECT * from users WHERE username='' OR 1=1; '';
```

This is problematic because now the query is selecting everything from the `users` table rather than scoping the query to a single `username`. This is a simplified example; there are many ways a query could be vulnerable, and many SQL injection queries besides our `' OR 1=1; '` example. And attackers will often use tools which try lots of different known SQL injections on webpages to see if the results they get back are anomalous and showing them that a page is vulnerable.

The way we can protect against SQL injection is by using what's called "prepared statements." Prepared statements are precompiled SQL statements which then insert values into the correct positions in the query. Go's `database/sql` package does this for us automatically:

```
1 query := "SELECT * from users WHERE username=?"
2 username := r.FormValue("username")
3 rows, err := db.Query(query, username)
4 if err != nil {
5     log.Fatal(err)
6 }
7 defer rows.Close()
8 ...
```

See Also

- <https://github.com/microcosm-cc/bluemonday>
- <https://github.com/unrolled/secure>
- https://infosec.mozilla.org/guidelines/web_security
- <https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP>
- https://en.wikipedia.org/wiki/SQL_injection
- https://www.owasp.org/index.php/SQL_Injection
- <https://golang.org/pkg/database/sql/#DB.Prepare>
- <https://golang.org/pkg/database/sql/#DB.Query>

Going into Production

In the preceding chapters we covered a lot of ground, from Go basics to advanced corner cases that production systems often need to concern themselves with. By now, your application should be running locally, well-tested, documented and ready for its first users. Good job! It's time to show it to the world. This is an exciting phase, but it comes with a whole new set of challenges. Your code is about to move from development to production, changing things forever. In this chapter, we present a high-level checklist of what it takes to get a modern Go application production-ready and running on the Internet, hopefully serving many satisfied users.

Continuous Integration & Continuous Deployment (CI/CD)

Continuous integration is a general term that describes the process of running tests and builds automatically whenever a team member adds new code to the codebase. Continuous integration in a production-ready Go system often also includes running miscellaneous checks and linters.

It is possible to write your own continuous integration system, but the effort is likely not worth it. There are many existing continuous integration systems out there. Some, like Travis CI, are hosted for you and are free for open source projects. Others, like Drone, are open source and free for self-hosting, but also comes with enterprise versions. We tabulated some popular options that currently support Go in the following table:

CI Service	Open Source	Configuration	Free version available
Travis CI	No	.travis.yml	Yes, for open source projects
Drone CI	Yes	.drone.yml that describes Docker build steps	Yes, as well as enterprise options. Always self-hosted
Jenkins	Yes	Jenkinsfile, either declarative or scripted	Yes, if self-hosted. Paid hosted versions also exist.
CircleCI	No	.circleci/config.yml	Yes, for open source projects hosted on GitHub or Bitbucket
GitHub Actions	No	.github/workflows/actions.yml	Yes, for open source projects hosted on GitHub

In larger organizations, the decision of which system to use will likely already have been made for you. But if you're doing this on your own or in a team without an existing CI/CD solution, you should bias towards simplicity so that your energy can be focused on improving the application itself. Github Actions and Travis CI are two good, fully managed solutions for cost-conscious small projects. Jenkins and Drone CI, in contrast, are open source options that can be run on your own infrastructure for free. These are good solutions if you are willing to put in more time to save a few dollars. The best choice will be project-dependent, and we encourage you to do your own research.

Continuous Deployment (CD) is similar to Continuous Integration in that it is an automated process that kicks off as soon as code is pushed to the central repository. It is usually, but not always, run on the same system as CI. After CI steps complete, a CD process will automatically deploy the latest version to Staging and/or Production environments, or make images available in a central repository for later deployment, ideally with a click of a button. Different engineers can have differing preferences when it comes to how often, and in how automated of a fashion, code should be deployed. This can even be a contentious topic, if not handled with care. You (and your team) will need to decide where you would like to fall on the scale between manual ad-hoc deployments over SSH and fully automated deployments as soon as code is pushed. That said, there is strong evidence that high-performing teams are consistently pushing themselves to get closer to the automated side of this scale ⁶⁴. Intuitively, fewer manual in a deployment lead to fewer mistakes, less time wasted and more frequent feedback from users.

Deployment

This section is a work in progress.

Logging

This section is a work in progress.

Monitoring

The key difference between running code in development and running code in production is that in production, you are no longer an active part of the loop. During development, you might notice an error or a bug and fix it. But in production, unless you have good observability over your system as it runs and others use it, you will not know when the system encounters problems. You may not be watching the logs, or you may be asleep! Besides, who has the time to stare at server logs all day long? Instead, you want to be sure that if something goes wrong, you will be notified and able to diagnose the issue quickly. This is a critical part of a production-ready web application, and is the focus of this chapter.

⁶⁴Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations

Metrics and Dashboards

For reasons of redundancy, latency or scale, a production web application is typically deployed to multiple servers. As soon as this happens, it becomes important to have high-level visibility over what is happening on each of the servers, without needing to log into them first. Centralized logging, discussed in [Logging](#), is one way to send logs from all your servers to a centralized location for ease of use. This can allow you to discover errors and even get the information you need to debug them. However, logs alone don't always tell the full story. If there are no error logs, does that mean users are not experiencing any issues? Or can it be that the server stopped running? Or stopped sending logs? How much disk and memory pressure do we have on each server, is there likely to be an outage soon? Are users experiencing acceptable latency?

A production application should be coupled with a dashboard that can answer these questions (and many others) from a high level. Dashboards like these *can* be powered by logs, but typically are not. Instead, engineers turn to specialized timeseries databases to collect high-level metrics across all servers. These are then displayed on a web interface where a developer acquainted with the system can tell at a glance whether anything is out of place.

Alerting

This section is a work in progress.

Further Reading

[The Go Programming Language Specification](https://golang.org/ref/spec)⁶⁵

[Effective Go](https://golang.org/doc/effective_go.html)⁶⁶ is a good overview document for how to write idiomatic Go code.

[Golang Weekly](https://golangweekly.com/)⁶⁷ is a weekly newsletter about Go.

[The Go Blog](https://blog.golang.org/)⁶⁸ is the official blog for Go.

[Gophers Slack](https://invite.slack.golangbridge.org/)⁶⁹ is a Slack community of Go developers. There is a #golang-newbies channel, #performance, #golang-jobs, and many more.

⁶⁵<https://golang.org/ref/spec>

⁶⁶https://golang.org/doc/effective_go.html

⁶⁷<https://golangweekly.com/>

⁶⁸<https://blog.golang.org/>

⁶⁹<https://invite.slack.golangbridge.org/>

Acknowledgements

The Go gopher was designed by Renée French.

Licenses

The Go source code license

Copyright (c) 2009 The Go Authors. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- * Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

- * Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

- * Neither the name of Google Inc. nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

github.com/gorilla/mux

Copyright (c) 2012 Rodrigo Moraes. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- * Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

- * Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

- * Neither the name of Google Inc. nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.