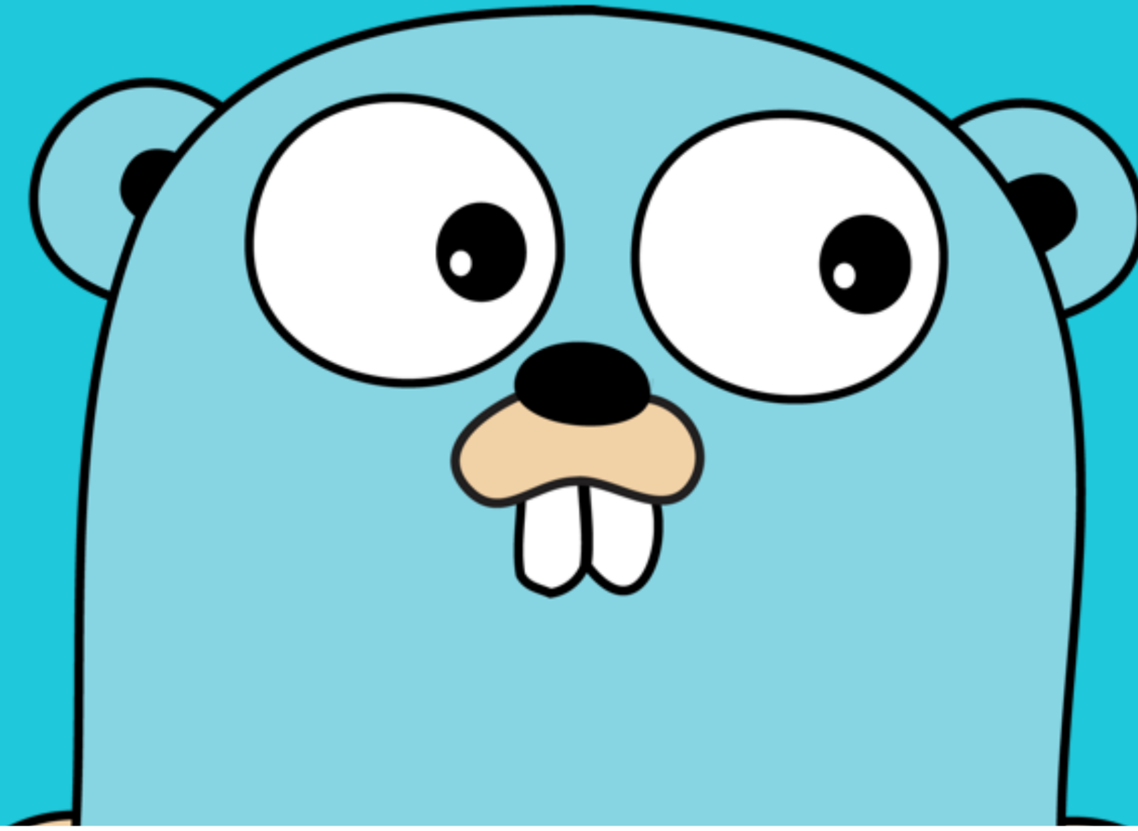
Production Go

Build modern, production-ready systems in Go



Herman Schaaf • Shawn Smith

Production Go

Build modern, production-ready systems in Go

Herman Schaaf and Shawn Smith

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

This version was published on 2018-11-03

Contents

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 think they will be positive gains. 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 Go is a statically typed language, it compiles quickly and can often be used as a replacement for scripting languages like Python. And many users report that when writing Go, once a program works, it continues to "just work". We suspect that this is due to the 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 reduced the number of production-breaking bugs.

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 all important aspects of running a production system, including advanced topics like profiling the memory usage of a Go program, deploying and monitoring apps written in Go, and writing idiomatic tests.

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.

Getting Started

“Clear is better than clever” - Rob Pike

Installing Go

Installation

The Go Downloads page[1] contains binary distributions of Go for Windows, Apple macOS, or Linux.

You can also find instructions for installing Go from source on their Installing Go from source[2] page. We recommend installing a binary release first, as there are some extra steps necessary to install from source, and you really only need to install from source if you're planning to contribute changes upstream to the Go language itself.

Also, if you use Homebrew on macOS, you should be able to install go with a simple `brew install go`.

Once installed, test the command by running:

```
$ go version
```

and you should see something like:

```
go version go1.9 darwin/amd64
```

GOPATH

Once you've installed Go, you need to set up a workspace. Traditionally this is called the GOPATH, but since Go 1.8 there is a default workspace at `$HOME/go` on Unix and `%USERPROFILE%/go` on Windows (for the sake of brevity, we're going to assume you're using UNIX and only refer to `$HOME/go` from now on). You'll have to create this directory if it doesn't already exist. This is where you'll be writing most if not all of your Go code.

Typically you'll be working with Go packages that are hosted on GitHub. When you retrieve a package using the `go get` command, the source will be automatically downloaded into your workspace. For example, if you want to download the popular URL router library `gorilla/mux`[3], you would run:

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

and the source of that library will be downloaded into `$HOME/go/src/github.com/gorilla/mux`. You can now also import that library in your own code. Here's an example based on some sample code from the `gorilla/mux` README:

[1] https://golang.org/dl/
[2] https://golang.org/doc/install/source
[3] https://github.com/gorilla/mux

```go
package main

import (
    "fmt"
    "net/http"

    "github.com/gorilla/mux"
)

func HomeHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello, world")
}

func main() {
    r := mux.NewRouter()
    r.HandleFunc("/", HomeHandler)
    http.Handle("/", r)
    http.ListenAndServe(":8080", r)
}
```

Note the import of `"github.com/gorilla/mux"` at the top.

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. Similar to the gofmt command, goimports formats your Go code, but it also automatically adds or removes imports according to what is referenced in the code. 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 times when you want to call `fmt.Println` but don't feel like importing `fmt` manually (or when you remove a `fmt.Println` and don't feel like deleting the import). It is worthwhile exploring the plugins available for your editor to make sure this is working: it will greatly speed up your Go development process.

Goimports can be installed with go get:

```
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[4]. This is done via Sublime Text package control. Package control can be installed with the commands provided at https://packagecontrol.io/installation. Once installed, open package control by pressing 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:

```
{
    // you may set specific environment variables here
    // e.g "env": { "PATH": "$HOME/go/bin:$PATH" }
    // in values, $PATH and ${PATH} are replaced with
    // the corresponding environment(PATH) variable, if it exists.
    "env": {"GOPATH": "$HOME/Code/go", "PATH": "$GOPATH/bin:$PATH" },

    "fmt_cmd": ["goimports"]
}
```

vim

For vim, you will want to install vim-go[5]. Instructions can be found on that page for the various vim package managers.

[4]https://github.com/DisposaBoy/GoSublime
[5]https://github.com/fatih/vim-go

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"
```

Linters and Correctness Tooling

If you want to install a suite of linters and tools to run on your code, we recommend gometalinter[6]. To install:

```
$ go get -u github.com/alecthomas/gometalinter
$ gometalinter --install
```

Gometalinter 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`, and `misspell` (disabled by default, enable with `--enable=misspell`) to be particularly useful. If you don't want to install all of the linters available, you can install them individually instead. For `misspell` for example, install with:

```
$ go get -u github.com/client9/misspell/cmd/misspell
```

Gometalinter should automatically pick up that you have misspell installed and you'll be able to do:

```
$ gometalinter --deadline=180s --exclude=vendor --disable-all --enable=misspell ./...
```

While you may not want to require some of these linters to pass in your build, one tool we recommend requiring is go vet[7]. 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`:

```go
package main

import (
    "fmt"
)

func main() {
    fmt.Printf("%d", "test")
}
```

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

[6]https://github.com/alecthomas/gometalinter
[7]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, 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`:

```
package main

import (
        "fmt"
)

func main() {
        fmt.Println("%d", "test")
}
```

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 of the 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

```go
package main

import (
	"fmt"
	"math/rand"
	"time"
)

func main() {
	dinners := []string{
		"tacos",
		"pizza",
                "ramen",
        }
        rand.Seed(time.Now().Unix())
        fmt.Printf("We'll have %s for dinner tonight!\n", dinners[rand.Intn(len(dinn\
ers))])
}
```

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 just 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

```go
package dinner

import (
        "math/rand"
        "time"
)

func Choose() string {
        dinners := []string{
                "tacos",
                "pizza",
			"ramen",
	}
	rand.Seed(time.Now().Unix())

	return dinners[rand.Intn(len(dinners))]
}
```

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

Using our random dinner selection package

```go
package main

import (
	"fmt"

	"github.com/prodgopher/dinner"
)

func main() {
	fmt.Println(dinner.Choose())
}
```

```
$ 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:

```go
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:

```
var found bool
for _, x := range entries {
    if x.Name == "Gopher" {
        found = true
    }
}
```

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

```
package main

import "fmt"

var a = 1

func main() {
        fmt.Println(a)
}
```

but this will not compile:

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

```go
package main

import "fmt"

a := 1 // this is invalid, use var instead

func main() {
	fmt.Println(a)
}
```

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,

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

but this is not valid:

```go
var a, b = 0, 1 // declare some variables
var b, c = 1, 2 // this is not okay, because b already exists
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

```go
package main

import "fmt"

func main() {
	a, b := true, false
	c := a && b
	d := a || b

	fmt.Println("a:", a)
	fmt.Println("b:", b)
	fmt.Println("c:", c)
	fmt.Println("d:", d)

	// Output: a: true
	// b: false
	// c: false
	// d: true
}
```

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

```go
package main

import "fmt"

func main() {
        sound := "meow"
        sentence := "The cat says " + sound + "."
        fmt.Println(sentence)
}
```

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 just 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

```go
package main

import (
	"fmt"
)

func main() {
	var i32 int32 = 100
	var i64 int64 = 100

	// This will result in a compile-time error:
	fmt.Println(i32 + i64)
}
```

This results in a compile-time error:

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

```go
package main

import (
	"fmt"
)

func main() {
	var i32 int32 = 100
	var i64 int64 = 100

	fmt.Println(int64(i32) + i64)
	// Output: 200
}
```

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

```go
package main

import (
	"fmt"
)

func main() {
	var i int8 = 128
	fmt.Println(i)
}
```

```
$ 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.

`uint`s 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

```go
package main

import (
	"fmt"
)

func main() {
	var p, s uint32 = 0, 1
	fmt.Printf("p - s = %d - %d = %d", p, s, p-s)
	// Output: p - s = 0 - 1 = 4294967295
}
```

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:

```go
x := 15
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:

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

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

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

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

```
1
2
```

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

```
fmt.Println(1.3i)
```

```
(0+1.3i)
```

Structs

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

Struct example

```
package main

import "fmt"

type Person struct {
        Name  string
        Email string
}

func main() {
        p := Person{"Robert", "robert@example.com"}

        fmt.Println(p.Name)
}
```

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

```go
package main

import (
	"encoding/json"
	"fmt"
	"log"
)

type Person struct {
	Name  string `json:"name"`
	Email string `json:"email"`
}

func main() {
	p := Person{"Robert", "robert@example.com"}
	b, err := json.Marshal(p)
	if err != nil {
		log.Fatal(err)
	}

	fmt.Println(string(b))
}
```

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 just to show how it works:

Anonymous struct example

```go
package main

import (
	"encoding/json"
	"fmt"
	"log"
)

func main() {
	p := struct {
		Name  string `json:"name"`
		Email string `json:"email"`
	}{
		Name:  "Robert",
		Email: "robert@example.com",
	}
	b, err := json.Marshal(p)
	if err != nil {
		log.Fatal(err)
	}

	fmt.Println(string(b))
}
```

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.

```
var x int
var y *int = &x

// Print the memory address of x,
// e.g. 0x10410020
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.

```go
var x *int = nil
*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`):

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

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`):

```go
if hour >= 12 {
    b = append(b, "PM"...)
} else {
    b = append(b, "AM"...)
}
```

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:

```
func ishex(c byte) bool {
    switch {
    case '0' <= c && c <= '9':
        return true
    case 'a' <= c && c <= 'f':
        return true
    case 'A' <= c && c <= 'F':
        return true
    }
    return false
}
```

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`):

```
func (p *pp) fmtBool(v bool, verb rune) {
    switch verb {
    case 't', 'v':
        p.fmt.fmt_boolean(v)
    default:
        p.badVerb(verb)
    }
}
```

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

```
switch err := err.(type) {
    case NotFoundError:
        ...
}
```

Arrays

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

```
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:

```
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:

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

will give us:

```
[1 2 3]
```

We can also leave out the `0`:

```
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:

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

and we get:

```
[4 5]
```

A slice's zero value is `nil`:

```
var x []int
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:

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

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

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

We can sort a slice with the `sort.Slice`[8] 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

```
package main

import (
	"fmt"
	"sort"
)

type Country struct {
	Name       string
	Population int
}

func main() {
	c := []Country{
		{"South Africa", 55910000},
		{"United States", 323100000},
		{"Japan", 127000000},
		{"England", 53010000},
	}

	sort.Slice(c,
		func(i, j int) bool { return c[i].Name < c[j].Name })
	fmt.Println("Countries by name:", c)

```

[8]https://golang.org/pkg/sort/#Slice

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

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:

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

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

```
package main

var m = map[string]string{}

func main() {
}
```

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

```go
package main

import "fmt"

func main() {
	m := make(map[string]string)
	m["en"] = "Hello"
	m["ja"] = "こんにちは"
	// loop over keys and values of map
	for k, v := range m {
		fmt.Printf("%q => %q\n", k, v)
	}

	// nonexistent key returns zero value
	fmt.Printf("zh: %q\n", m["zh"])

	// check for existence with _, ok := m[Key]
	if _, ok := m["ja"]; ok {
		fmt.Printf("key %q exists in map\n", "ja")
	}

	delete(m, "en")
	fmt.Printf("m length: %d\n", len(m))
	fmt.Println(m)
}
```

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`[9].

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[10] 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

```go
package main

import (
	"fmt"
)

func main() {
	x := 1
	for x < 4 {
		fmt.Println(x)
		x++
	}
}
```

The above code outputs:

[9] https://golang.org/pkg/sync/#Map
[10] 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

```go
package main

import (
	"fmt"
)

func main() {
	for i := 0; i < 3; i++ {
		fmt.Println(i)
	}
}
```

This will output:

```
0
1
2
```

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

Infinite for loop

```go
package main

func main() {
	for {
	}
}
```

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

```go
package main

import "fmt"

func main() {
	nums := []int{1, 2, 3}
	for i, x := range nums {
		fmt.Printf("nums[%d] => %d\n", i, x)
	}
}
```

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

```go
package main

import "fmt"

func main() {
	nums := []int{1, 2, 3}
	for _, x := range nums {
		fmt.Println(x)
	}
}
```

and this will output:

```
1
2
3
```

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

Range over map

```go
package main

import "fmt"

func main() {
	m := map[int]string{
		1: "一",
		2: "二",
		3: "三",
	}
	for k, v := range m {
		fmt.Printf("%d => %q\n", k, v)
	}
}
```

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

```go
package countries

import (
	"math/rand"
	"time"
)

type Country struct {
	Name       string
	Population int
}

var data = []Country{
	{"South Africa", 55910000},
	{"United States", 323100000},
	{"Japan", 127000000},
	{"England", 53010000},
}

// Random returns a random country
func Random() Country {
	rand.Seed(time.Now().Unix())

	return data[rand.Intn(len(data))]
}
```

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:

```go
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.[11]

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

```go
package main

import (
	"fmt"
	"time"
)

func hello() {
	fmt.Println("hello from a goroutine!")
}

func main() {
	go hello()
	time.Sleep(1 * time.Second)
}
```

(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 just to show a small example of a goroutine; it should not be considered a valid way of managing goroutines.)

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

[11] https://golang.org/doc/faq#no_pointer_arithmetic

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

One way to synchronize goroutines is to use `sync.WaitGroup`[12]:

Using sync.WaitGroup

```go
package main

import (
	"fmt"
	"sync"
)

func main() {
	var wg sync.WaitGroup
	wg.Add(3)
	go func() {
		defer wg.Done()
		fmt.Println("goroutine 1")
	}()
	go func() {
		defer wg.Done()
		fmt.Println("goroutine 2")
	}()
	go func() {
		defer wg.Done()
		fmt.Println("goroutine 3")
	}()

	wg.Wait()
}
```

[12]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 were finished. On the other hand, 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:

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

We can create a buffered channel like so:

```
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:

```
ch <- 5
```

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

```
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

```go
package main

func main() {
	ch := make(chan int, 5)

	ch <- 1
	ch <- 2
	ch <- 3
	ch <- 4
	ch <- 5
	ch <- 6
}
```

```
$ 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

```go
package main

import "fmt"

func main() {
	ch := make(chan int, 5)

	ch <- 1
	ch <- 2
	ch <- 3
	ch <- 4
	ch <- 5

	fmt.Println("channel length:", len(ch))
	switch {
	case len(ch) >= 5:
		<-ch
	case len(ch) < 5:
		ch <- 6
	}
}
```

```
$ 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

```go
package main

import "fmt"

func main() {
	ch := make(chan int, 5)

	ch <- 1
	ch <- 2
```

```
        ch <- 3
        ch <- 4
        ch <- 5

        select {
        case ch <- 6:
        default:
                fmt.Println("channel is full, ignoring send")
        }
}
```

```
$ 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:

```
type Entry interface {
    Title() string
}
```

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

Interfaces example

```
package main

import (
        "fmt"
        "time"
)

type Entry interface {
        Title() string
}
```

```go

type Book struct {
        Name      string
        Author    string
        Published time.Time
}

func (b Book) Title() string {
        return fmt.Sprintf("%s by %s (%s)", b.Name, b.Author, b.Published.Format("Ja\
n 2006"))
}

type Movie struct {
        Name     string
        Director string
        Year     int
}

func (m Movie) Title() string {
        return fmt.Sprintf("%s (%d)", m.Name, m.Year)
}

func Display(e Entry) string {
        return e.Title()
}

func main() {
        b := Book{Name: "John Adams", Author: "David McCullough", Published: time.Da\
te(2001, time.May, 22, 0, 0, 0, 0, time.UTC)}
        m := Movie{Name: "The Godfather", Director: "Francis Ford Coppola", Year: 19\
72}
        fmt.Println(Display(b))
        fmt.Println(Display(m))
}
```

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:

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

```
こんにちは
```

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

```go
var i interface{}
i = "こんにちは"
s, ok := i.(string)
if !ok {
    fmt.Println("s is not type string")
}
fmt.Println(s)
```

```
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

```go
package main

import (
	"fmt"
)

func main() {
	var i interface{}
	fmt.Println(i == nil)
	fmt.Printf("%T, %v\n", i, i)

	var s *string
	fmt.Println("s == nil:", s == nil)

	i = s
	fmt.Println("i == nil:", i == nil)
	fmt.Printf("%T, %v\n", i, i)

}
```

```
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

```go
package strconv

import "fmt"

func ParseBool(str string) (bool, error) {
	switch str {
	case "1", "t", "T", "true", "TRUE", "True":
		return true, nil
	case "0", "f", "F", "false", "FALSE", "False":
		return false, nil
	}
	return false, fmt.Errorf("invalid input %q", str)
}
```

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

```go
package main

import (
	"bufio"
	"log"
	"os"
)

func main() {
	m := map[string]int{}
	scanner := bufio.NewScanner(os.Stdin)
	for scanner.Scan() {
		m[scanner.Text()]++
	}
	if err := scanner.Err(); err != nil {
		log.Printf("ERROR: could not read stdin: %s", err)
	}
	for k, v := range m {
		log.Printf("%s => %d", k, v)
	}
}
```

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

```go
package main

import (
	"log"
	"os"
)

func main() {
	langs := []string{"python", "php", "go"}
	f, err := os.Create("langs.txt")
	if err != nil {
		log.Fatal(err)
	}
	defer f.Close()
	for _, lang := range langs {
		f.WriteString(lang + "\n")
	}
}
```

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`[13] which will open the file and write our data in one function call.

[13] 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 automatically, and the tool is so prominent in the Go community that 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 feel that such strict formatting (maybe you hate tabs) sounds backwards and annoying, but we would argue that gofmt actually played a big role in Go's success. Since everyone's code looks the same, it takes an element of surprise out of looking at someone else's code when helping them debug it, or simply trying to understand it. We believe 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:

```go
package main

import "fmt"

func main(){
a       := "foo"
someVar := "bar"

fmt.Println(a, someVar)
}
```

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

```go
package main

import "fmt"

func main() {
    a := "foo"
    someVar := "bar"

    fmt.Println(a, someVar)
}
```

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 will 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."[14]

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

```go
package main

type Animal struct {
    Name    string
    Species string
}

func main() {
    animals := []Animal{
        Animal{"Henry", "cat"},
        Animal{"Charles", "dog"},
    }
}
```

After `gofmt -s`, this becomes:

[14]https://godoc.org/golang.org/x/tools/cmd/goimports

```go
package main

type Animal struct {
	Name    string
	Species string
}

func main() {
	animals := []Animal{
		{"Henry", "cat"},
		{"Charles", "dog"},
	}
}
```

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:

```go
func Unmarshal(data []byte, v interface{}) error {
    // Check for well-formedness.
    // Avoids filling out half a data structure
    // before discovering a JSON syntax error.
    var d decodeState
    err := checkValid(data, &d.scan)
    if err != nil {
        return err
    }

    d.init(data)
    return d.unmarshal(v)
}
```

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[15]:

"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 the functions being called. 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:

[15] https://github.com/golang/go/wiki/CodeReviewComments#variable-names

```go
func truncate(s, suf string, l int) string {
    if len(s) < l {
        return s
    }
    return s[:l] + suf
}
```

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:

```go
// lineCount returns the number of lines in a given file
func lineCount(filepath string) (int, error) {
        out, err := exec.Command("wc", "-l", filepath).Output()
        if err != nil {
                return 0, err
        }
        // wc output is like: 999 filename.go
        count, err := strconv.Atoi(strings.Fields(string(out))[0])
        if err != nil {
                return 0, err
        }

        return count, nil
}
```

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 eventually warmed up to the error checking. When you're that strict about returning and checking errors, it's very hard to miss where and why an error is happening.

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

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

	return count, nil
}
```

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:

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

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[16] itself. One way to do this is to browse Go standard library's godoc documentation[17]. 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.

[16]https://github.com/golang/go
[17]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

```go
package main

import "fmt"

func ExampleString() {
	s := "I am a string - 你好"
	fmt.Println(s)
	// Output: I am a string - 你好
}
```

As the example shows, Go string literals and code may also contain non-English characters, like the Chinese 你好 [18].

Appending to Strings

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

Appending to a string

```go
package main

import "fmt"

func ExampleAppend() {
	greeting := "Hello, my name is "
	greeting += "Inigo Montoya"
	greeting += "."
	fmt.Println(greeting)
	// Output: Hello, my name is Inigo Montoya.
}
```

[18]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`[19] 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

```go
package main

import "fmt"

func ExampleFmtString() {
	name := "Inigo Montoya"
	sentence := fmt.Sprintf("Hello, my name is %s.", name)
	fmt.Println(sentence)
	// Output: Hello, my name is Inigo Montoya.
}
```

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

```go
package main

import "fmt"

func ExampleFmtComplexString() {
	name := "Inigo Montoya"
	age := 32
	weight := 76.598
	t := "Hello, my name is %s, age %d, weight %.2fkg"
	fmt.Printf(t, name, age, weight)
	// Output: Hello, my name is Inigo Montoya, age 32, weight 76.60kg
}
```

[19]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` package[20]. 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

```go
package main

func main() {
	s := "I am" + 32 + "years old"
}
```

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

```
$ go run bad_append.go
# command-line-arguments
./bad_append.go:4: cannot convert "I am" to type int
./bad_append.go:4: invalid operation: "I am" + 32 (mismatched types string an\
d 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:

[20] https://golang.org/pkg/fmt/

Splitting a string

```go
package main

import "fmt"
import "strings"

func ExampleSplit() {
	l := strings.Split("a,b,c", ",")
	fmt.Printf("%q", l)
	// Output: ["a" "b" "c"]
}
```

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 string[21]. 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

```go
package main

import (
	"fmt"
	"strings"
)

func ExampleCount() {
	s := "banana"
	c1 := strings.Count(s, "a")
	c2 := strings.Count(s, "ana")
```

[21]Remember, 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.

```go
	fmt.Println(c1, c2)
	// Output: 3 1
}
```

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

```go
package main

import (
	"fmt"
	"strings"
)

func ExampleContains() {
	str := "two gophers on honeymoon"
	if strings.Contains(str, "moon") {
		fmt.Println("Contains moon")
	}
	if strings.HasPrefix(str, "moon") {
		fmt.Println("Starts with moon")
	}
	if strings.HasSuffix(str, "moon") {
		fmt.Println("Ends with moon")
	}
	// Output: Contains moon
	// Ends with moon
}
```

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

```go
package main

import "fmt"
import "strings"

func ExampleIndex() {
	an := strings.Index("banana", "an")
	am := strings.Index("banana", "am")
	fmt.Println(an, am)
	// Output: 1 -1
}
```

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`[22]. 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 or not 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.

[22]https://golang.org/pkg/strings/

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

```go
package main

import (
	"bufio"
	"fmt"
	"os"
	"strings"
	"unicode"
)

// getInput prompts the user for some text, and then
// reads a line of input from standard input. This line
// of text is then returned.
func getInput() string {
	fmt.Print("Enter a sentence: ")
	scanner := bufio.NewScanner(os.Stdin)
	scanner.Scan()
	return scanner.Text()
}

func isNotLetter(c rune) bool {
	return !unicode.IsLetter(c)
}

// isPalindromicSentence returns whether or not the given sentence
// is palindromic. To calculate this, it splits the string into words,
// then creates a reversed copy of the word slice. It then checks
// whether the reverse is equal (ignoring case) to the original.
// It also ignores any non-alphabetic characters.
func isPalindromicSentence(s string) bool {
	// split into words and remove non-alphabetic characters
	// in one operation by using FieldsFunc and passing in
	// isNotLetter as the function to split on.
	w := strings.FieldsFunc(s, isNotLetter)

	// iterate over the words from front and back
	// simultaneously. If we find a word that is not the same
	// as the word at its matching from the back, the sentence
```

```
        // is not palindromic.
        l := len(w)
        for i := 0; i < l/2; i++ {
                fw := w[i]     // front word
                bw := w[l-i-1] // back word
                if !strings.EqualFold(fw, bw) {
                        return false
                }
        }

        // all the words matched, so the sentence must be
        // palindromic.
        return true
}

func main() {
        // Go doesn't have while loops, but we can use for loop
        // syntax to read into a new variable, check that it's not
        // empty, and read new lines on subsequent iterations.
        for l := getInput(); l != ""; l = getInput() {
                if isPalindromicSentence(l) {
                        fmt.Println("... is palindromic!")
                } else {
                        fmt.Println("... is not palindromic.")
                }
        }
}
```

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

An example run of the palindrome program

```
$ go run palindromes.go
Enter a sentence: This is magnificent!
... is not palindromic.
Enter a sentence: This is magnificent, is this!
... is palindromic!
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` package[23] 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 or not 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, functions can be passed around just 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 `l - 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:

[23] https://golang.org/pkg/bufio/

- the init statement: executed before the first iteration
- 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

```
package main

import "fmt"

func ExampleIteration() {
        s := "ABC你好"
        for i, r := range s {
                fmt.Printf("%q(%d) ", r, i)
        }
        // Output: 'A'(0) 'B'(1) 'C'(2) '你'(3) '好'(6)
}
```

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!)[24].

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:

```go
package main

import (
    "fmt"
)

func main() {
    b := []byte{65, 66, 67}
    s := string(b)
```

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

```go
    fmt.Println(s)
    // Output: ABC
}
```

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.

```go
package main

import (
    "fmt"
)

func showVerb(v, s string) {
    n := fmt.Sprintf(v, s)
    fmt.Println(v, "\t", n)
}

func main() {
    s := "ABC 你好"
    showVerb("%s", s)
    showVerb("%q", s)
    showVerb("%+q", s)
    showVerb("%x", s)
    showVerb("% x", s)
    showVerb("%# x", s)
}
```

Running this produces the following output:

```
%s	ABC 你好
%q	"ABC 你好"
%+q		"ABC \u4f60\u597d"
%x	41424320e4bda0e5a5bd
% x		41 42 43 20 e4 bd a0 e5 a5 bd
%# 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 [^verb_fmt_source]:

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

and these verbs are only for strings and slices of bytes (treated equivalently by the:

```
%s	the uninterpreted bytes of the string or slice
%q	a double-quoted string safely escaped with Go syntax
%x	base 16, lower-case, two characters per byte
%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:

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

```go
package main

import "fmt"

func ExampleIteration() {
	s := "ABC你好"
	for i, r := range s {
		fmt.Printf("%q(%d) ", r, i)
	}
	// Output: 'A'(0) 'B'(1) 'C'(2) '你'(3) '好'(6)
}
```

Running this, we get the following output:

```
'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. Below the hood, instead of going byte by byte, the `range` keyword is fetching the next *rune* in the string. We can achieve the same effect without `range` by using the `NextRune`

Handling right-to-left languages

[^verb_fmt_source](https://golang.org/pkg/fmt/#pkg-overview)

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:

```
package main

import (
    "fmt"
    "time"
)

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

    time.Sleep(1 * time.Second)
}
```

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

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

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    go func() {
        fmt.Println("hello, world 1")
    }()

    go func() {
        fmt.Println("hello, world 2")
    }()

    go func() {
        fmt.Println("hello, world 3")
    }()

    time.Sleep(1 * time.Second)
}
```

```
$ 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

```go
package main

import (
	"fmt"
	"sync"
)

func main() {
	var wg sync.WaitGroup
	wg.Add(1)
	go func() {
		defer wg.Done()
		fmt.Println("hello, world 1")
	}()

	wg.Add(1)
	go func() {
		defer wg.Done()
		fmt.Println("hello, world 2")
	}()

	wg.Add(1)
	go func() {
		defer wg.Done()
		fmt.Println("hello, world 3")
	}()

	wg.Wait()
}
```

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.

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

```
package main

import "fmt"

func main() {
	ch := make(chan string)

	go func(ch chan string) {
		ch <- "hello, world 1"
	}(ch)

	go func(ch chan string) {
		ch <- "hello, world 2"
	}(ch)

	go func(ch chan string) {
		ch <- "hello, world 3"
	}(ch)

	a, b, c := <-ch, <-ch, <-ch

	fmt.Println(a)
	fmt.Println(b)
	fmt.Println(c)
}
```

This code is functionally equivalent to our sync.WaitGroup example in the previous section. Note that the channel must be declared 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

```go
package main

import "fmt"

func main() {
	ch := make(chan string)

	go func(ch chan string) {
		ch <- "hello, world 1"
	}(ch)

	go func(ch chan string) {
		ch <- "hello, world 2"
	}(ch)

	// go func(ch chan string) {
	// 	ch <- "hello, world 3"
	// }(ch)

	a, b, c := <-ch, <-ch, <-ch

	fmt.Println(a)
	fmt.Println(b)
	fmt.Println(c)
}
```

```
$ 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

```go
package main

import (
	"log"
	"net/http"
	"time"
)

func main() {
	http.HandleFunc("/", func(w http.ResponseWriter, req *http.Request) {
		go func() {
			time.Sleep(3 * time.Second)
			log.Println("hello, world")
		}()

		return
	})

	log.Println("Running on :8080...")
	log.Fatal(http.ListenAndServe("127.0.0.1:8080", nil))
}
```

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

```go
package main

import (
	"encoding/json"
	"io/ioutil"
	"log"
	"net/http"
	"time"
)

const bsaEndpoint = "http://api.bart.gov/api/bsa.aspx?cmd=bsa&key=MW9S-E7SL-2\
6DU-VV8V&json=y"

type bsaResponse struct {
	Root struct {
		Advisories []BSA `json:"bsa"`
	}
}

// BSA is a BART service advisory
type BSA struct {
	Station     string
	Description struct {
		Text string `json:"#cdata-section"`
	}
```

```
}

func poll() {
	ticker := time.NewTicker(5 * time.Second)
	defer ticker.Stop()
	for {
		select {
		case <-ticker.C:
			resp, err := http.Get(bsaEndpoint)
			if err != nil {
				log.Println("ERROR: could not GET bsaEndpoint:", err)
			}
			defer resp.Body.Close()

			b, err := ioutil.ReadAll(resp.Body)
			if err != nil {
				log.Println("ERROR: could not parse response body:", err)
			}

			var br bsaResponse
			err = json.Unmarshal(b, &br)
			if err != nil {
				log.Println("ERROR: json.Unmarshal:", err)
			}

			if len(br.Root.Advisories) > 0 {
				for _, adv := range br.Root.Advisories {
					log.Println(adv.Station, adv.Description.Text)
				}
			}
		}
	}
}

func main() {
	http.HandleFunc("/", func(w http.ResponseWriter, req *http.Request) {
		go func() {
			log.Println("hello, world 1")
		}()
```

```

                return
        })

        go poll()
        log.Println("Running on :8080...")
        log.Fatal(http.ListenAndServe("127.0.0.1:8080", nil))
}
```

Let's go over what this code does. We have an HTTP handler similar to 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.[25]

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:

[25]https://golang.org/doc/articles/race_detector.html

Goroutines updating same map

```go
package main

import (
	"log"
	"time"
)

func main() {
	m := map[string]string{}

	go func() {
		m["test"] = "hello, world 1"
	}()

	go func() {
		m["test"] = "hello, world 2"
	}()

	time.Sleep(time.Second)
	log.Println(m["test"])
}
```

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

```go
package main

import (
	"log"
	"sync"
	"time"
)

func main() {
	var m = &sync.Map{}

	go func() {
		m.Store("test", "hello, world 1")
	}()

	go func() {
		m.Store("test", "hello, world 2")
	}()

	time.Sleep(time.Second)

	val, _ := m.Load("test")
	log.Println(val)
}
```

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.[26]

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

[26]https://golang.org/pkg/sync/#Map

sync.Map

```go
package main

import (
	"log"
	"sync"
	"time"
)

type safeMap struct {
	sync.Mutex
	m map[string]string
}

func main() {
	var sm = safeMap{
		m: make(map[string]string),
	}

	go func() {
		sm.Lock()
		defer sm.Unlock()
		sm.m["test"] = "hello, world 1"
	}()

	go func() {
		sm.Lock()
		defer sm.Unlock()
		sm.m["test"] = "hello, world 2"
	}()

	time.Sleep(time.Second)

	sm.Lock()
	log.Println(sm.m["test"])
	sm.Unlock()
}
```

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

sync.Map

```go
package main

import (
	"log"
	"sync"
	"time"
)

type safeMap struct {
	sync.Mutex
	m map[string]string
}

func (sm *safeMap) Store(key, val string) {
	sm.Lock()
	defer sm.Unlock()
	sm.m[key] = val
}

func (sm *safeMap) Load(key string) (val string, exists bool) {
	sm.Lock()
	defer sm.Unlock()
	val, ok := sm.m[key]
	return val, ok
}

func (sm *safeMap) Delete(key, val string) {
	sm.Lock()
	defer sm.Unlock()
	delete(sm.m, key)
}

func main() {
	var sm = safeMap{
		m: make(map[string]string),
	}

	go func() {
```

```
                sm.Store("test", "hello, world 1")
        }()

        go func() {
                sm.Store("test", "hello, world 2")
        }()

        time.Sleep(time.Second)

        val, _ := sm.Load("test")
        log.Println(val)
}
```

```
$ 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[27], 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

```go
package main

import "testing"

// TestIsPrime tests that the IsPrime function
// returns true when the input is prime, and false
// otherwise.
func TestIsPrime(t *testing.T) {
	// check a prime number
	got := IsPrime(19)
	if got != true {
		t.Errorf("IsPrime(%d) = %t, want %t", 19, got, true)
	}

	// check a non-prime number
```

[27] https://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

```go
package main

import "testing"

// TestIsPrime tests that the IsPrime function
// returns true when the input is prime, and false
// otherwise.
func TestIsPrimeTD(t *testing.T) {
	cases := []struct {
		give int
		want bool
	}{
		{19, true},
		{21, false},
		{10007, true},
		{1, false},
		{0, false},
		{-1, false},
	}

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

In the refactored test, we use a slice of an anomymous 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 [28]. 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

```go
package main

import (
	"fmt"
	"net/http"
)

// helloHandler writes a friendly "Hello, friend :)" response.
func helloHandler(w http.ResponseWriter, r *http.Request) {
	fmt.Fprintln(w, "Hello, friend :)")
}

func main() {
	http.HandleFunc("/hello", helloHandler)
	http.ListenAndServe(":8080", nil)
}
```

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.

[28] https://github.com/golang/go/wiki/CodeReviewComments#useful-test-failures

Testing an HTTP handler

```go
package main

import (
	"net/http"
	"net/http/httptest"
	"testing"
)

func TestHTTPHandler(t *testing.T) {
	// Create a request to pass to our handler.
	req, err := http.NewRequest("GET", "/hello", nil)
	if err != nil {
		t.Fatal(err)
	}

	// We create a ResponseRecorder, which satisfies
	// the http.ResponseWriter interface, to record
	// the response.
	r := httptest.NewRecorder()
	handler := http.HandlerFunc(helloHandler)

	// Our handler satisfies http.Handler, so we can call
	// the ServeHTTP method directly and pass in our
	// Request and ResponseRecorder.
	handler.ServeHTTP(r, req)

	// Check that the status code is what we expect.
	if r.Code != http.StatusOK {
		t.Errorf("helloHandler returned status code %v, want %v",
			r.Code, http.StatusOK)
	}

	// Check that the response body is what we expect.
	want := "Hello, friend :)\n"
	got := r.Body.String()
	if got != want {
		t.Errorf("helloHandler returned body %q want %q",
			got, want)
```

```
        }
}
```

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. Anagolous 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`[29]:

```
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.

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.

[29]https://golang.org/pkg/net/http/#HandlerFunc

Using an interface to abstract away API calls

```go
package eightball

import (
	"math/rand"
	"time"
)

// randIntGenerator is an interface that includes Intn, a
// method in the built-in math/rand package. This allows us
// to mock out the math/rand package in the tests.
type randIntGenerator interface {
	Intn(int) int
}

// EightBall simulates a very simple magic 8-ball,
// a magical object that predicts the future by
// answering yes/no questions.
type EightBall struct {
	rand randIntGenerator
}

// NewEightBall returns a new EightBall.
func New() *EightBall {
	return &EightBall{
		rand: rand.New(rand.NewSource(time.Now().UnixNano())),
	}
}

// Answer returns a magic eightball answer
// based on a random result provided by
// randomGenerator. It supports only four
// possible answers.
func (e EightBall) Answer(s string) string {
	n := e.rand.Intn(3)
	switch n {
	case 0:
		return "Definitely not"
	case 1:
```

```
                return "Maybe"
        case 2:
                return "Yes"
        default:
                return "Absolutely"
        }
}
```

We define a simple `eightball` package that implements a simple Magic 8-Ball[30]. 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`[31] 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

```
package eightball

import (
        "testing"
)

type fixedRandIntGenerator struct {
        // the number that should be "randomly" generated
        randomNum int

        // record the paramater that Intn gets called with
        calledWithN int
}

func (g *fixedRandIntGenerator) Intn(n int) int {
        g.calledWithN = n
```

[30]https://en.wikipedia.org/wiki/Magic_8-Ball
[31]https://golang.org/pkg/math/rand/#Rand.Intn

```go
        return g.randomNum
}

func TestEightBall(t *testing.T) {
        cases := []struct {
                randomNum int
                want      string
        }{
                {0, "Definitely not"},
                {1, "Maybe"},
                {2, "Yes"},
                {3, "Absolutely"},
                {-1, "Absolutely"}, // default case
        }

        for _, tt := range cases {
                g := &fixedRandIntGenerator{randomNum: tt.randomNum}
                eb := EightBall{
                        rand: g,
                }

                got := eb.Answer("Does this really work?")
                if got != tt.want {
                        t.Errorf("EightBall.Answer() is %q for num %d, want %q",
                                got, tt.randomNum, tt.want)
                }

                if g.calledWithN != 3 {
                        t.Errorf("EightBall.Answer() did not call Intn(3) as expected")
                }
        }
}
```

Sometimes, when the existing code uses a specific library implementation, it takes refactoring to use interfaces to mock out impementation 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.

Generating Coverage Reports

To generate test coverage percentages for your code, simply 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

```go
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 contains invalid character")
	}

	return true, nil
}
```

Now let's write a test for it:

Test for username validation function

```go
package validate

import "testing"

var usernameTests = []struct {
	in        string
	wantValid bool
}{
	{"gopher", true},
}

func TestUsername(t *testing.T) {
	for _, tt := range usernameTests {
		valid, err := Username(tt.in)
		if err != nil && tt.wantValid {
			t.Fatal(err)
		}

		if valid != tt.wantValid {
			t.Errorf("Username(%q) = %t, want %t", tt.in, valid, tt.wantValid)
		}
	}
}
```

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 probably get 100% coverage. We'd like to know, however, 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%
```

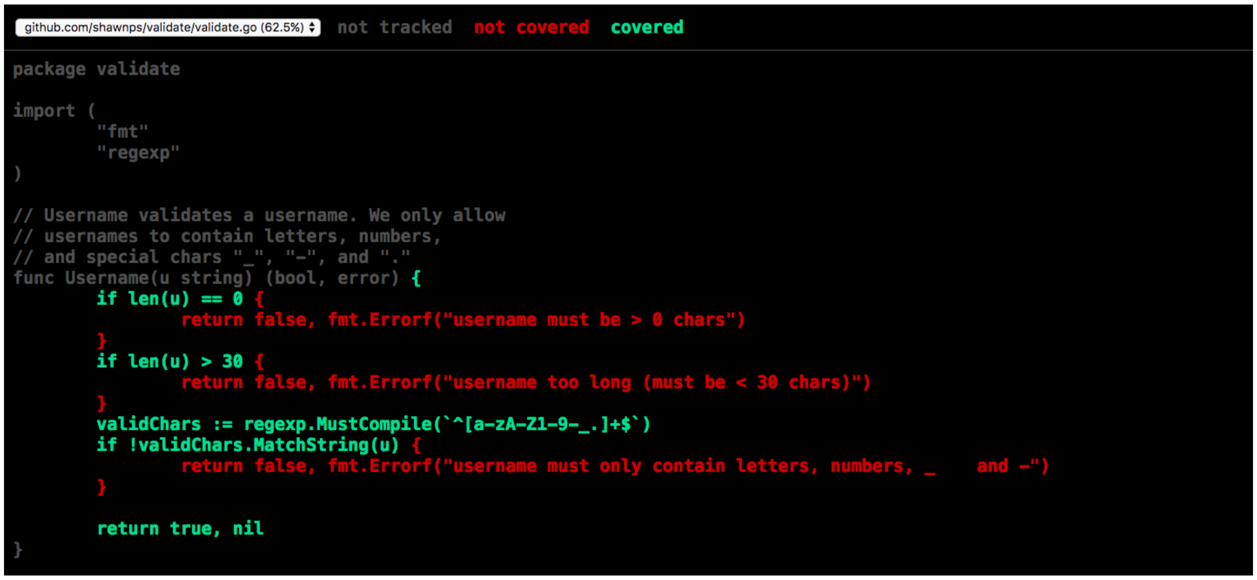
What we really want to see is a line-by-line breakdown. We can get this with the HTML report, which we'll cover in the next section.

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:



```
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
}
```

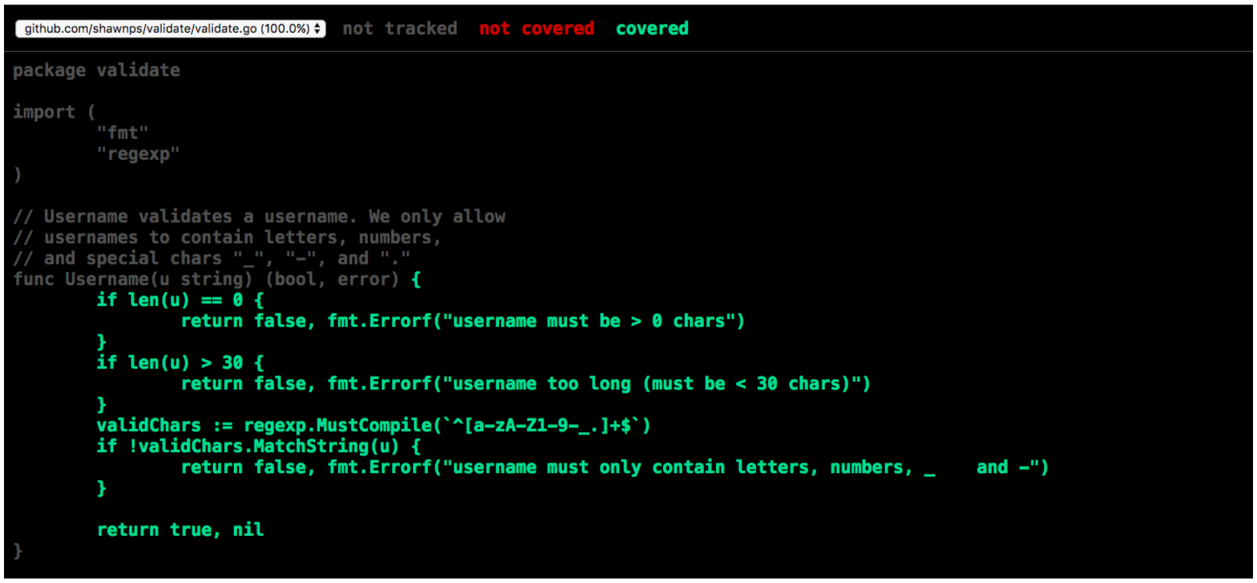
Username test coverage

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

```go
package validate

import "testing"

var usernameTests = []struct {
	in        string
	wantValid bool
}{
	{"", false},
	{"gopher", true},
	{"gopher$", false},
	{"abcdefghijklmnopqrstuvwxyzabcde", false},
}

func TestUsername(t *testing.T) {
	for _, tt := range usernameTests {
		valid, err := Username(tt.in)
		if err != nil && tt.wantValid {
			t.Fatal(err)
		}

		if valid != tt.wantValid {
			t.Errorf("Username(%q) = %t, want %t", tt.in, valid, tt.wantValid)
		}
	}
}
```

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. Examples are rendered in godoc underneath the function's documentation.

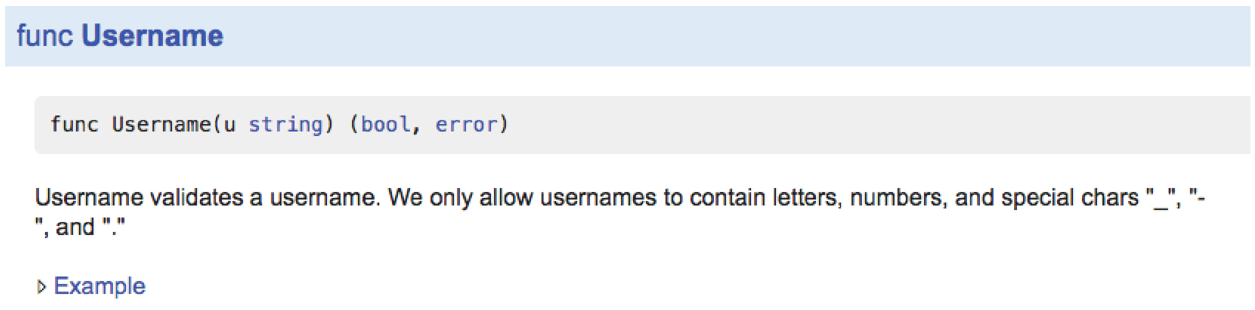
Let's write an example for our username validation function:

Username validation function example test

```
package validate

import (
        "fmt"
        "log"
)

func ExampleUsername() {
        usernames := []struct {
                in    string
                valid bool
        }{
                {"", false},
                {"gopher", true},
                {"gopher$", false},
                {"abcdefghijklmnopqrstuvwxyzabcde", 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
	// "abcdefghijklmnopqrstuvwxyzabcde": false
}
```

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 the tests are run.

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

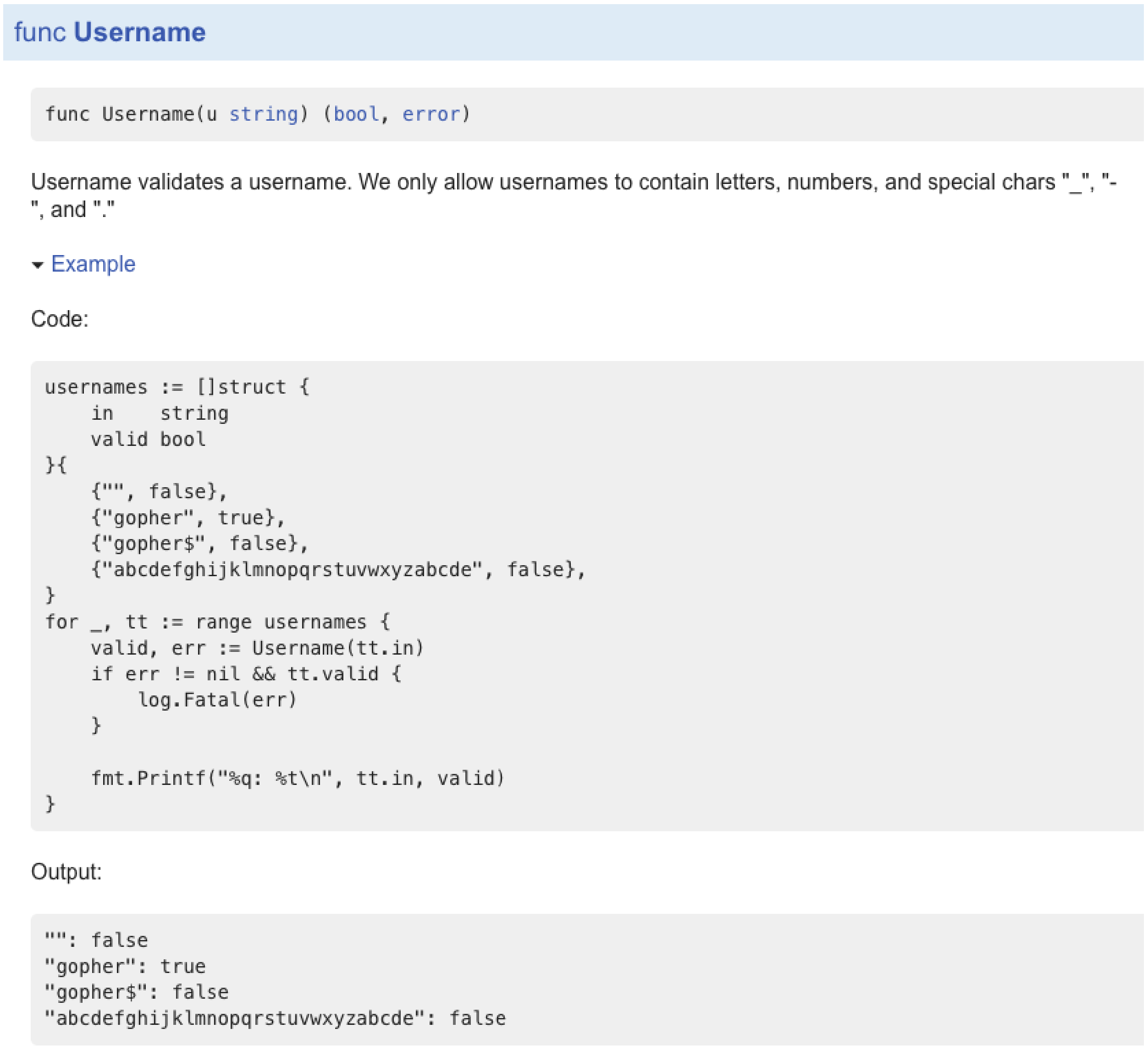
```
func Username(u string) (bool, error)
```

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 "."

Code:

```
usernames := []struct {
    in    string
    valid bool
}{
    {"", false},
    {"gopher", true},
    {"gopher$", false},
    {"abcdefghijklmnopqrstuvwxyzabcde", 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
"abcdefghijklmnopqrstuvwxyzabcde": false
```

Godoc example full

Another note about examples is that they have a specific naming convention. Our example above is named `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`:

```go
type User struct {
    Name string
}

func (u *User) ValidateName() (bool, error) {
        ...
}
```

Then our example code would look like this:

```go
func ExampleUser_ValidateName() {
    ...
}
```

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

If we need multiple examples for a single function, that can be done by appending 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 nth 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 nth 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 nth Fibonacci number.

A function that recursively obtains the nth Fibonacci number

```go
package fibonacci

// F returns the nth Fibonnaci number.
func F(n int) int {
	if n <= 0 {
		return 0
	} else if n == 1 {
		return 1
	}
	return F(n-1) + F(n-2)
}
```

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 nth Fibonacci number

```go
// fibonacci_test.go
package fibonacci

import "testing"

func TestF(t *testing.T) {
	cases := []struct {
		n    int
		want int
	}{
		{-1, 0},
		{0, 0},
		{1, 1},
		{2, 1},
		{3, 2},
		{8, 21},
	}

	for _, tt := range cases {
		got := FastF(tt.n)
		if got != tt.want {
			t.Errorf("F(%d) = %d, want %d", tt.n, got, tt.want)
		}
	}
}
```

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. Anagalous to `TestX(t *testing.T)`, benchmarks are created 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 explicitly 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 can simply 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 were run 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 (~1.2ms) 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

```
package fibonacci

// FastF returns the nth Fibonnaci number,
// but does not use recursion.
func FastF(n int) int {
	var a, b int = 0, 1
	for i := 0; i < n; i++ {
		a, b = b, a+b
	}
	return a
}
```

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

```go
package fibbonaci

// F returns the nth Fibonnaci number.
func F(n int) int {
	var a, b int = 0, 1
	for i := 0; i < n; i++ {
		a, b = b, a+b
	}
	return a
}
```

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    50000000            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 execution time of our setup code to be included in the overall benchmark timing.

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

crypto/aes BenchmarkEncrypt

```go
package aes

import "testing"

func BenchmarkEncrypt(b *testing.B) {
	tt := encryptTests[0]
	c, err := NewCipher(tt.key)
	if err != nil {
		b.Fatal("NewCipher:", err)
	}
	out := make([]byte, len(tt.in))
	b.SetBytes(int64(len(out)))
	b.ResetTimer()
	for i := 0; i < b.N; i++ {
		c.Encrypt(out, tt.in)
	}
}
```

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 Fibonnaci 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 Fibonnaci 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[32]. 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:

```go
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`:

[32]https://en.wikipedia.org/wiki/Escape_analysis

A high-memory implementation of F

```go
package fibonacci

// FastHighMemF returns the nth Fibonacci number, but
// stores the full slice of intermediate
// results, consuming more memory than necessary.
func FastHighMemF(n int) int {
	if n <= 0 {
		return 0
	}

	r := make([]int, n+1)
	r[0] = 0
	r[1] = 1
	for i := 2; i <= n; i++ {
		r[i] = r[i-1] + r[i-2]
	}
	return r[n]
}
```

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:

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

which we then loop over in the BenchmarkF function:

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

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 `%`:

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

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          100000000           16.7 ns/op
BenchmarkFastFBitwiseAnd-4      200000000            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'll 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"[33], 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 normally take the form of a comment directly above the package declaration, starting with "Package [name] ...", like so:

[33]https://blog.golang.org/godoc-documenting-go-code

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

but if your package comment is long and you don't want it to clutter your code, you can split it out into a separate `doc.go` file, which only contains the comment and a package clause. See net/http/doc.go[34] for an example.

Go Guru

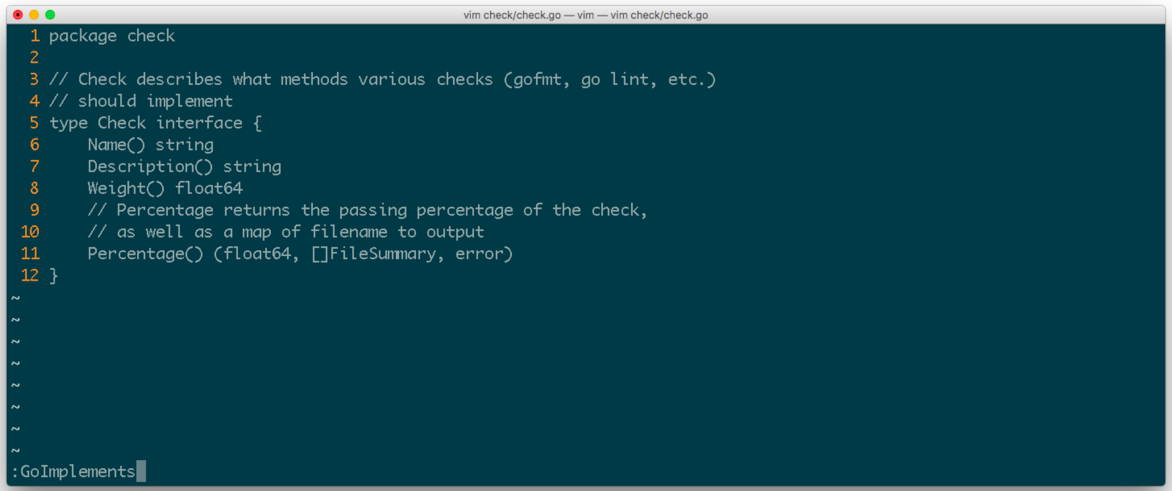
Go Guru is "a tool for answering questions about Go source code."[35] It ships with a command-line tool, but really it's meant to be integrated into an editor. You can install it with:

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

A list of supported editor integrations can be found on the Using Go Guru[36] document linked in the godoc for guru. In this section we'll be showing screenshots of guru being used in vim.

Let's take a look at one of the questions that guru answers for us, which is, "what concrete types implement this interface?" We're going to 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.

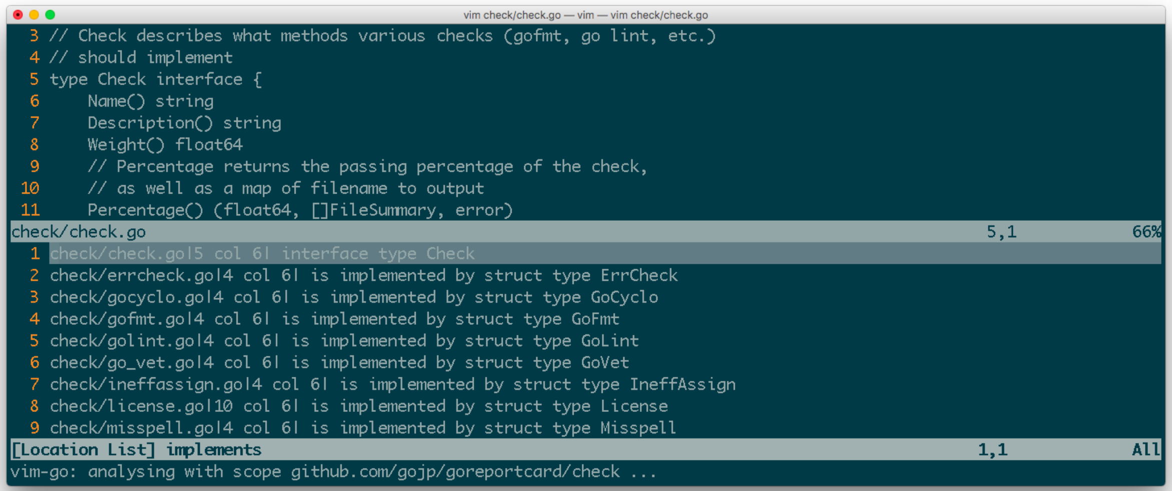
Navigate to the line that contains an interface, and type `:GoImplements`:



```
package check

// Check describes what methods various checks (gofmt, go lint, etc.)
// should implement
type Check interface {
    Name() string
    Description() string
    Weight() float64
    // Percentage returns the passing percentage of the check,
    // as well as a map of filename to output
    Percentage() (float64, []FileSummary, error)
}
~
~
~
~
~
~
~
~
: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:

[34]https://github.com/golang/go/blob/master/src/net/http/doc.go
[35]https://godoc.org/golang.org/x/tools/cmd/guru
[36]https://docs.google.com/document/d/1_Y9xCEMj5S-7rv2ooHpZNH15JgRT5iM742gJkw5LtmQ/edit#heading=h.ojv16z1d1gas



```
vim check/check.go — vim — vim check/check.go
// Check describes what methods various checks (gofmt, go lint, etc.)
// should implement
type Check interface {
    Name() string
    Description() string
    Weight() float64
    // Percentage returns the passing percentage of the check,
    // as well as a map of filename to output
    Percentage() (float64, []FileSummary, error)
check/check.go                                                        5,1           66%
check/check.go|5 col 6| interface type Check
check/errcheck.go|4 col 6| is implemented by struct type ErrCheck
check/gocyclo.go|4 col 6| is implemented by struct type GoCyclo
check/gofmt.go|4 col 6| is implemented by struct type GoFmt
check/golint.go|4 col 6| is implemented by struct type GoLint
check/go_vet.go|4 col 6| is implemented by struct type GoVet
check/ineffassign.go|4 col 6| is implemented by struct type IneffAssign
check/license.go|10 col 6| is implemented by struct type License
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`. More help on guru for vim-go can be found at the vim-go-tutorial[37], and a list of guru queries as well as other help output can be seen when running `guru -help`.

Race Detector

Go comes with a builtin mechanism for detecting race conditions[38]. 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 that we'll run with the race detector enabled. A data race can occur in Go 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.

[37]https://github.com/fatih/vim-go-tutorial#guru
[38]https://golang.org/doc/articles/race_detector.html

A simple program with an asynchronous update method

```go
package cat

import (
	"log"
)

// Cat is a small, carnivorous mammal with excellent night vision
type Cat struct {
	noise string
}

// SetNoise sets the noise that our cat makes
func (c *Cat) SetNoise(n string) {
	c.noise = n
}

// Noise makes the cat make a noise
func (c *Cat) Noise() string {
	return c.noise
}

func updateCat(c *Cat) {
	go c.SetNoise("にゃん")

	log.Println(c.Noise())
}
```

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 end up writing 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, the expected functionality of the `updateCat` function is that the `Noise` attribute for the passed-in`"cat"` will be set 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

```go
package cat

import (
	"testing"
)

func TestUpdateCat(t *testing.T) {
	c := Cat{noise: "meow"}
	c = updateCat(c)
	if got := c.Noise(); got != "にゃん" {
		t.Fatalf("c.Noise() = %q, want %q", got, "にゃん")
	}
}
```

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`[39] to lock our `Cat` structure whenever we read or write to it:

Cat program with sync.Mutex

```go
package cat

import (
	"log"
	"sync"
)

// Cat is a small, carnivorous mammal with excellent night vision
type Cat struct {
	mu    sync.Mutex
```

[39]https://golang.org/pkg/sync/#Mutex

```
        noise string
}

// SetNoise sets the noise that our cat makes
func (c *Cat) SetNoise(n string) {
        c.mu.Lock()
        defer c.mu.Unlock()
        c.noise = n
}

// Noise makes the cat make a noise
func (c *Cat) Noise() string {
        c.mu.Lock()
        defer c.mu.Unlock()

        return c.noise
}

func updateCat(c *Cat) *Cat {
        go c.SetNoise("にゃん")

        log.Println(c.Noise())

        return c
}
```

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. To show a somewhat trivial 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

```go
package main

import (
	"encoding/json"
	"flag"
	"fmt"
	"log"
	"net/http"
	"strings"
	"unicode"
)

var (
	addr      = flag.String("http", "127.0.0.1:8000", "HTTP listen address")
	countries = map[string]string{
		"england":      "England",
		"japan":        "Japan",
		"southafrica":  "South Africa",
		"unitedstates": "United States",
	}
)

// Entry is a user-submitted entry summarizing their trip
// to a given country
type Entry struct {
	Country string
	Text    string
}

// removeSpaces is a function for strings.Map that removes
// all spaces from a string
func removeSpaces(r rune) rune {
	if unicode.IsSpace(r) {
		return -1
	}
	return r
}

```

```
// normalizeCountry takes an Entry and normalizes its country name
func normalizeCountry(e *Entry) error {
        co := strings.Map(removeSpaces, strings.ToLower(e.Country))
        if _, ok := countries[co]; !ok {
                return fmt.Errorf("invalid country %q", e.Country)
        }

        e.Country = countries[co]

        return nil
}

// EntryPostHandler is the POST endpoint for entries
func EntryPostHandler(w http.ResponseWriter, req *http.Request) {
        decoder := json.NewDecoder(req.Body)
        defer req.Body.Close()
        var entry Entry
        err := decoder.Decode(&entry)
        if err != nil {
                log.Printf("ERROR: could not decode Entry: %s", err)
                w.WriteHeader(http.StatusBadRequest)
                w.Write([]byte(fmt.Sprintf("Could not decode Entry: %s\n", err)))
                return
        }

        go func(e *Entry) {
                normalizeCountry(e)
        }(&entry)

        log.Printf("INFO: received Entry %v", entry)
}

func main() {
        http.HandleFunc("/entries", EntryPostHandler)

        log.Printf("Running on %s ...", *addr)
        log.Fatal(http.ListenAndServe(*addr, nil))
}
```

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`[40]:

Country journal API with sync.WaitGroup

```go
package main

import (
	"encoding/json"
	"flag"
	"fmt"
	"log"
	"net/http"
	"strings"
	"sync"
	"unicode"
)

var (
	addr      = flag.String("http", "127.0.0.1:8000", "HTTP listen address")
	countries = map[string]string{
		"england":      "England",
		"japan":        "Japan",
		"southafrica":  "South Africa",
		"unitedstates": "United States",
	}
)

// Entry is a user-submitted entry summarizing their trip
// to a given country
type Entry struct {
	Country string
	Text    string
}

// removeSpaces is a function for strings.Map that removes
// all spaces from a string
func removeSpaces(r rune) rune {
	if unicode.IsSpace(r) {
```

[40]https://golang.org/pkg/sync/#WaitGroup

```
		return -1
	}
	return r
}

// normalizeCountry takes an Entry and normalizes its country name
func normalizeCountry(e *Entry) error {
	co := strings.Map(removeSpaces, strings.ToLower(e.Country))
	if _, ok := countries[co]; !ok {
		return fmt.Errorf("invalid country %q", e.Country)
	}

	e.Country = countries[co]

	return nil
}

// EntryPostHandler is the POST endpoint for entries
func EntryPostHandler(w http.ResponseWriter, req *http.Request) {
	decoder := json.NewDecoder(req.Body)
	defer req.Body.Close()
	var entry Entry
	err := decoder.Decode(&entry)
	if err != nil {
		log.Printf("ERROR: could not decode Entry: %s", err)
		w.WriteHeader(http.StatusBadRequest)
		w.Write([]byte(fmt.Sprintf("Could not decode Entry: %s\n", err)))
		return
	}

	var wg sync.WaitGroup
	wg.Add(1)
	go func(e *Entry) {
		defer wg.Done()
		normalizeCountry(e)
	}(&entry)
	wg.Wait()

	log.Printf("INFO: received Entry %v", entry)
```

```
}

func main() {
        http.HandleFunc("/entries", EntryPostHandler)

        log.Printf("Running on %s ...", *addr)
        log.Fatal(http.ListenAndServe(*addr, nil))
}
```

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[41] is a web application that takes a `go get` path to a package and gives the package a grade based on how well it scores with various linters and tools. It is a popular application in the open source community, with thousands of projects using the generated badge to indicate the code quality and link to known quality issues from the project's GitHub repository. You can try it on goreportcard.com if your source code is open source, or run the server locally to make use of the tool on your internal network or private repositories.

[41]https://goreportcard.com/

Security

CSRF

Content Security Policy (CSP)

Setting a `Content-Security-Policy` header can help us prevent attacks such as cross-site scripting (XSS) on our web application. The CSP header value allows us to specify the origins of our content.

HTTP Strict Transport Security (HSTS)

bluemonday (https://github.com/microcosm-cc/bluemonday)

Continuous Integration

The build

Now that we have tests and linters/tooling, we need a build.

We recommend setting up a Makefile with targets for installing, building, and testing the application. Here is a slightly modified example from Go Report Card[42]:

Go Report Card Makefile

```
all: lint build test

build:
        go build ./...

install:
        ./scripts/make-install.sh

lint:
        gometalinter --exclude=vendor --exclude=repos --disable-all --enable=golint \
--enable=vet --enable=gofmt ./...
        find . -name '*.go' | xargs gofmt -w -s

test:
         go test -cover ./check ./handlers

start:
         go run main.go

misspell:
        find . -name '*.go' -not -path './vendor/*' -not -path './_repos/*' | xargs \
misspell -error
```

For an open source project, we can use Travis CI[43] for free. Even if the project is not open source, Travis CI is a good choice. It's easy to set up - just sign in to their webapp, add the repository, and add a `.travis.yml` file to the repository. Here is an example `.travis.yml` from Go Report Card:

[42]http://github.com/gojp/goreportcard
[43]https://travis-ci.org/

Go Report Card .travis.yml

```yaml
language: go

go:
  - 1.8.x
  - 1.9.x
  - 1.10.x
  - 1.11.x
  - tip

install:
  - make install

script:
  - make lint
  - make test
```

This runs our `make lint` and `make test` targets on multiple Go versions.

Deployment

Deployment of a Go application will be highly dependent on existing infrastructure; if there is no infrastructure in place already,

Monitoring

Prometheus

Prometheus is an open source system (written in Go) which, at the time of writing this book, is very common and popular.

Grafana

Grafana is a platform for analytics and monitoring. We can use Grafana to graph various data exposed by Prometheus on our Go server.

Alerts

Optimization

Common Gotchas

Nil interface

Further Reading

The Go Programming Language Specification[44]

Effective Go[45] is a good overview document for how to write idiomatic Go code.

Golang Weekly[46] is a weekly newsletter about Go.

The Go Blog[47] is the official blog for Go.

Gophers Slack[48] is a Slack community of Go developers. There is a #golang-newbies channel, #performance, #golang-jobs, and many more.

[44] https://golang.org/ref/spec
[45] https://golang.org/doc/effective_go.html
[46] https://golangweekly.com/
[47] https://blog.golang.org/
[48] https://invite.slack.golangbridge.org/

Acknowledgements

The Go gopher was designed by Renée French.

Licenses

The Go source code license

github.com/gorilla/mux

TODO: Add licenses for any open source code we use as examples