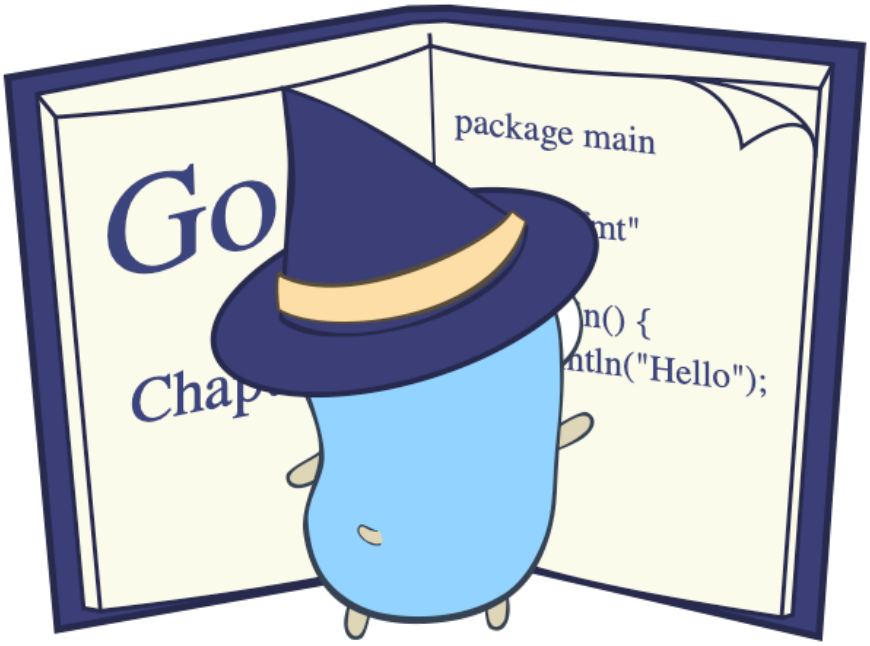
# For the Love of Go — Fundamentals

John Arundel

2020-10-30

## Contents

# Introduction




Hello, and welcome to learning Go! It's great to have you here.

## What's this?

This book is an introduction to the Go programming language, suitable for complete beginners. If you don't know anything about Go yet, or programming, but would like to learn, you're in the right place!

(If you do already know something about Go, you should find the material in this book relatively straightforward, but still worth your time.)

## What you'll need

You'll need to install Go on your computer, if you don't have it already. Follow the instructions here to download and install Go:

- https://golang.org/dl/

You'll also need at least Go version 1.14 to run the code examples in this book. Run the following command to see what version of Go you have:

```
go version
```

```
go version go1.15 darwin/amd64
```

While all you need to write and run Go programs is a terminal and a text editor, you'll find it very helpful to use an editor which has specific support for Go. For example, Visual Studio Code has excellent Go integration.

## What you'll learn

By working through the exercises in this book, you'll learn:

- How to run tests for a Go program
- How to automatically format your Go code correctly
- The basic pattern that all Go tests should follow
- How to declare and import Go *packages* (units of code)
- How to design multiple *test cases* and use them in your tests
- How to write functions that return error values
- How to test error handling in your programs
- A simple, reliable, test-first development workflow

## Test-driven development

There's a style of programming (which isn't unique to Go, but is very popular with Go developers) called *Test-Driven Development* (TDD).

What this means is that before you write a program to do something (multiply two numbers, let's say), you first of all write a *test*.

A test is also a program, but it's a program specifically designed to run *another* program with various different inputs, and check its result. The test verifies that the program actually behaves the way it's supposed to.

For example, if you were considering writing a function called `Multiply`, and you wanted to be able to check that it worked, you might write a program that calls `Multiply(2, 2)` and checks that the answer is `4`. That's a test!

Lots of programmers use tests, but without necessarily doing TDD. What makes TDD, in particular, so useful, is that you actually write the tests *first*. In this book, you'll start by writing a test for a particular function, then you'll write the function itself. That might seem backwards at first sight, but it actually makes a lot of sense!

By writing the test first, you are forced to think about what the program actually needs to do. You also have to design its interface (the way users interact with the program), since the test interacts with the program the same way a user would. And it also means you know when to stop coding, because when the test starts passing, you're done!

(You might ask why not write the code first, and then, once the code is working, write the test for it. Well, how would you know whether or not it's working without a test? You see, the whole scheme collapses. Write your tests first.)

## What makes this book different?

Most Go books will start by telling you about the basic syntax of the language, variables, data types, and so on. That's okay, but I want you to absorb the test-first style of programming described above, so we're going to *learn* Go test-first too. That means we'll need to use a few Go concepts in our tests that you won't be fully familiar with yet.

That won't stop you working your way through the book: at each stage, I'll give you the code you need to solve the current problem, and you should be able to modify and extend it to do what you need it to do, even if you don't have a complete understanding of how it works.

In fact, this is often the situation we find ourselves in as programmers: we *have* some existing code which we don't necessarily totally understand, but we will try to figure out just enough to solve the problem at hand. So this will be good practice for you!

If you like, you can read this book in conjunction with the next in the series, For the Love of Go: Data. That goes into much more detail about some of the concepts we'll use in this book, such as structs, slices, variables, and more. And in that book, you'll be solving more advanced problems using some of the Go testing skills you learned here! So the two books complement one another.

## How to use this book

Throughout this book we'll be working together to develop a project in Go. Each chapter introduces a new feature or concept, and sets you some goals to achieve. Goals are marked with **GOAL:** in bold. (Don't worry if you're not sure how to start. There'll usually be some helpful suggestions following the goal description.)

If you're finding the goals easy, or want a bit more of a challenge, there are some *stretch goals*, too. These are optional; if you're not interested or not ready to tackle them, just skip over them.

So let's get started!

# 1: Testing times

It's your first day as a Go developer at Texio Instronics, a major manufacturer of widgets and what-nots, and you've been assigned to the Future Projects division to work on a very exciting new product: a Go-powered electronic calculator. Welcome aboard.

In this exercise, all you need to do is make sure your Go environment is set up and everything is working right.

## Downloading the Git repository for this book

One of your new colleagues has already made a start on the calculator project. She's placed a Go package in the `calculator.go` file, and an accompanying test in `calculator_test.go`. You're going to run this test and make sure it passes.

**GOAL:** Get a copy of the code and successfully run the test.

First of all, you'll need a GitHub account; if you don't have one, you can find out how to create one here:

- https://github.com/join

Next, find the example repo for this book here:

- https://github.com/bitfield/ftl-fundamentals

Because you'll want to commit the code you write while working through this book, it's a good idea to *fork* this repo (create your own copy). To do that,

click the 'Fork' button at the top right. Follow the instructions here to fork and clone the repo so that you can work on it:

- https://docs.github.com/en/github/getting-started-with-github/fork-a-repo

## Solutions to exercises

The idea of this book is that you should try to solve the various challenges by yourself, but once you've done that, you may find it helpful to compare your solutions against mine. You can find a partial set of solutions in the `sample` branch of the GitHub repo:

- https://github.com/bitfield/ftl-fundamentals/tree/sample

Don't worry if your solutions are a bit different; there are usually several valid ways to solve any problem in Go. And if you got completely stuck on something, the `sample` branch may help. But try to resist the temptation to check the solution before you've made a serious attempt at it yourself—you'll learn more this way!

## Running the tests

Next, start a shell session using your terminal program (for example, the MacOS Terminal app) or your code editor. Set your working directory to the repo folder using the `cd` command (for example, `cd ftl-fundamentals`).

Now you're ready to run the tests. Here's how to do that:

In this directory, run the command:

```
go test
```

If everything is good, you will see this output:

```
PASS
ok      calculator      0.234s
```

This tells you that all the tests passed in the package called `calculator`, and that running them took 0.234 seconds. (It might take a longer or shorter time on your computer, but that's okay.)

If you see test failure messages, or any other error messages, there may be a problem with your Go setup. Don't worry, everybody runs into this kind of issue at first. Try Googling the error message to see if you can figure out what the problem is.

Once the test is passing, you can move on!

**STRETCH GOAL:** Use your editor's Go support features to run the test directly in the editor. For example, in Visual Studio Code, there's a little 'Run

test' link above each test function, and a 'Run package tests' link at the top of the file. Find out how to do this in the editor you're using.

## 2: Getting in shape

The next thing to do is ensure that your Go code is formatted correctly. All Go code is formatted in a standard way, using a tool called `gofmt` (pronounced, in case you were wondering, 'go-fumpt'). In this exercise you'll see how to use it to check and reformat your code (if necessary).

### Checking format with `gofmt -d`

The file `calculator.go` in this directory contains a deliberate formatting mistake, just to make sure you're on your toes!

**GOAL:** Run `gofmt` to check the formatting and find out what is wrong:

```
gofmt -d calculator.go
```

```
diff -u calculator.go.orig calculator.go
--- calculator.go.orig  2020-08-11 15:13:42.000000000 +0100
+++ calculator.go       2020-08-11 15:13:42.000000000 +0100
@@ -3,5 +3,5 @@
```

```
 // Add takes two numbers and returns the result of adding them together.
 func Add(a, b float64) float64 {
-return a + b
+       return a + b
 }
```

Look at the lines beginning with `-` and `+`:

```
-return a + b
+       return a + b
```

`gofmt` is telling you what it wants to do. It wants to remove (`-`) the line that has no indent before the `return` statement, and replace it (`+`) with a line where the statement is properly indented (by one tab character).

**NOTE:** If you're using Windows, you may see an error like this:

```
computing diff: exec: "diff": executable file not found in %PATH%
```

This is because the `gofmt` tool relies on there being a `diff` program in your `PATH` which it can use to generate a visual diff of the file. If you don't have `diff` installed, or if it's not in your default `PATH`, this won't work. Don't worry, though: this isn't a serious problem, and it won't stop you being able to develop with Go; it just means you won't be able to use the `gofmt -d` command to check your formatting.

You may find this Stack Overflow discussion helpful for resolving this issue:

- `gofmt -d` error on Windows

## Fixing format with `gofmt -w`

You could make this change using your editor, but there's no need: `gofmt` can do it for you.

**GOAL:** Run:

```
gofmt -w calculator.go
```

This will reformat the file in place. You can check that it's worked by running `gofmt -d` on the file again. Now there should be no output, because it's already formatted correctly.

If you're using an editor that has Go support, such as Vim, Visual Studio Code, or GoLand, you can set up the editor to automatically run your code through `gofmt` every time you save it. This will help you avoid forgetting to format your code. Check the documentation for your editor to see how to set this up.

**STRETCH GOAL:** Experiment with `gofmt`. Try formatting your code in different ways and see what changes `gofmt` makes to it. It's interesting to note what it does and doesn't change.

# 3: Fixing a hole

Now that your development environment is all set up, your colleague needs your help. She has been working on getting the calculator to subtract numbers, but there's a problem: the test is not passing. Can you help?

## A failing test

You'll find the test code (currently commented out) in the `calculator_test.go` file:

```
func TestSubtract(t *testing.T) {
    t.Parallel()
    var want float64 = 2
    got := calculator.Subtract(4, 2)
    if want != got {
        t.Errorf("want %f, got %f", want, got)
    }
}
```

**GOAL:** Get this test passing!

Uncomment the test function and run the tests:

```
go test
```

```
--- FAIL: TestSubtract (0.00s)
    calculator_test.go:22: want 2.000000, got -2.000000
FAIL
exit status 1
FAIL    calculator      0.178s
```

This test failure is telling you exactly where the problem occurred:

```
calculator_test.go:20
```

It also tells you what the problem was:

```
want 2.000000, got -2.000000
```

## The `testing` library

You might have noticed that in the `calculator_test.go` file, there's an `import` statement near the top of the file that looks like this:

```
import (
    ...
    "testing"
    ...
)
```

In Go, we have to explicitly *import* any packages that we want to use, including, in this case, the `calculator` package itself. Since we're testing it, we need to call its functions, and we can't do that unless we have imported it.

Similarly, we are using functions from the `testing` library, so we import that too. What is `testing`? It's a package in the Go standard library which relates to testing, as you might have guessed. Unlike many languages, Go actually has built-in support for writing unit tests and Test-Driven Development; you don't need any extra framework or third-party package in order to do this.

As you can imagine, you'll be using the `testing` package a lot in Go, so it's worth taking a little time to read the documentation and get familiar with it:

- The `testing` package

## The structure of tests

We can also see that `TestSubtract` is a Go function, with this signature:

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

The name of a test function must begin with `Test`, or Go won't recognise it as a test. It also has to accept a `*testing.T` as a parameter. What's that? It's the thing which allows us to fail the test. We'll see how in a moment.

So what exactly is the test doing? Let's take a closer look. Don't worry about the `t.Parallel()` part for now, as it merely relates to how Go runs the tests.

First, the test sets up a variable `want` to establish what it *wants* to receive from the function:

```
var want float64 = 2
```

Then it calls the function with the values `4, 2`, and stores the result into another variable `got`:

```
got := calculator.Subtract(4, 2)
```

The idea is to compare `want` with `got` and see if they are different. If they are, the `Subtract()` function is not working as expected, so the test fails:

```
if want != got {
    t.Errorf("want %f, got %f", want, got)
}
```

On the other hand, if `want` and `got` are equal, then everything's fine, so the test function ends here. (You don't need to explicitly make a test pass in Go; if it doesn't explicitly fail, that's considered a pass.)

## The function under test

So let's look at the code for the `Subtract()` function (in `calculator.go`). Here it is:

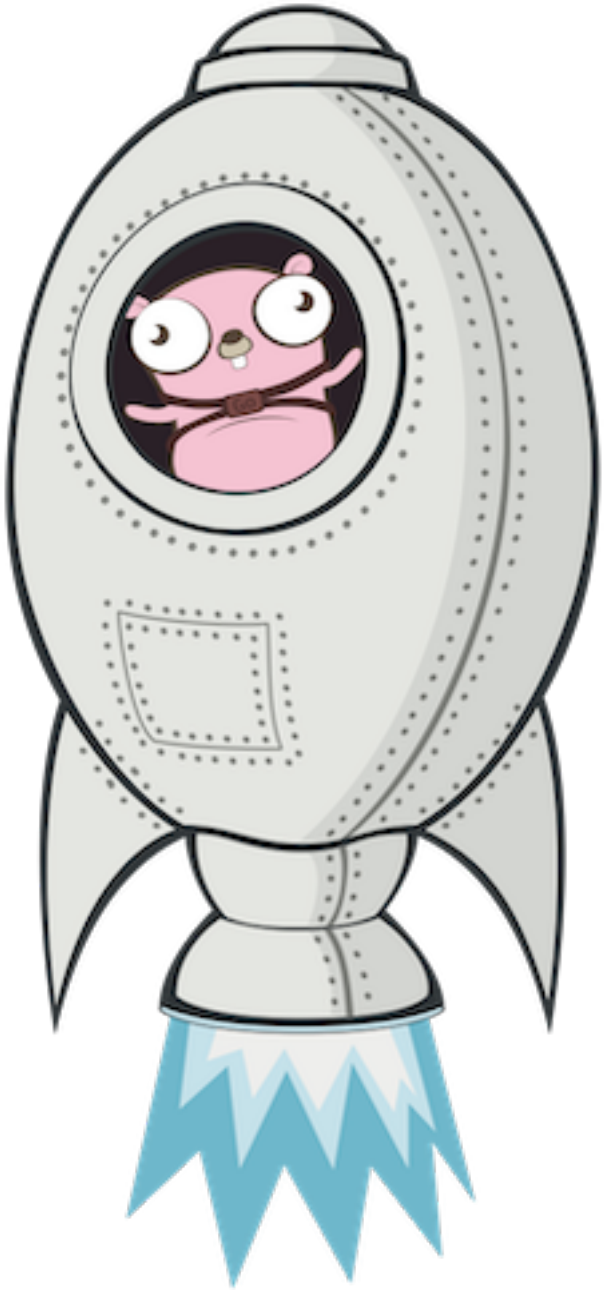
```
func Subtract(a, b float64) float64 {
    return b - a
}
```

If you spot a problem, try altering the code to fix it. Run `go test` again to check that you got it right.

When the test passes, you can move on!

**STRETCH GOAL:** Find out what other ways there are to fail a test using the `testing.T`. What other methods does it have? Do any of them look potentially useful?

## 4: Go forth and multiply



Excellent work. You now have a calculator which can add and (correctly) sub-

tract. That's a great start. Let's turn to multiplication now.

Up to now you've been running existing tests and modifying existing code. For the first time you're going to write the test, and the function it's testing!

**GOAL:** Write a test for a function `Multiply()` which, just like the `Add()` and `Subtract()` functions, takes two numbers as parameters, and returns a single number representing the result.

Where should you start? Well, this is a test, so start in the `calculator_test.go` file. Test functions in Go have to have a name that starts with `Test` (or Go won't call them when you run `go test`). So `TestMultiply` would be a good name, wouldn't it?

You'll see that `TestAdd` and `TestSubtract` look much the same, except for the specific inputs and the expected return value. So start by copying one of those functions, renaming it `TestMultiply`, and making the appropriate changes.

When you're done, running the tests should produce a compilation error:

```
undefined: calculator.Multiply
```

This makes sense; you haven't written that function yet!

**GOAL:** Write the minimum code necessary to get the test to compile and fail.

What is the minimum code necessary to compile? Well, you need to *define* the `Multiply` function, if not implement it. See if you can work out what its function signature should be, based on looking at `Add` and `Subtract`.

It will take two `float64` parameters which are its inputs, and since the function must return something if it's to be useful, it needs to be defined as returning a single `float64` value.

Given that function signature, the function will not compile unless it has a `return` statement, and the easiest way to write this is simply to have the function return zero. This should be enough to make the program compile, and if you run `go test` again, `TestMultiply` should fail.

## Why deliberately write incorrect code?

Why do this? Why write something in `Multiply` which is obviously incorrect; namely, returning zero? Why not go ahead and actually write the code to do the multiplication?

This is an important step in test-driven development. You start by writing a test for the function that doesn't yet exist. Before you start writing the function itself, you need to verify that the test is correct. (Otherwise, you might end up with *both* the test *and* the tested function being wrong.)

One way to do this is to ensure that, with a minimal and obviously incorrect implementation of `Multiply`, you see the test fail the expected way. If the test

passes, even though you know `Multiply` returns zero for all inputs, something is really wrong with the test!

Once you have `TestMultiply` failing correctly (that sounds wrong, but you know what I mean), you can move on to the next step.

**GOAL:** Write the minimum code necessary to get `TestMultiply` to pass.

Why did we say "the minimum code necessary"? Because, in a sense, the test itself defines what `Multiply` needs to do. Once it does that, and the test passes, you should *stop coding*.

## Why stop when the test passes?

Why? Because any further code you write after the test passes, is by definition untested. You have no way of knowing whether it's correct, because the test passes whether the extra code is there or not. So you don't do that. You write code until the test passes, and then you stop.

**GOAL:** Make `TestMultiply` fail deliberately.

There's one final step required for us to be really sure that both the test and the code are correct. You need to see the test fail again. To do this, change the expected return value from `Multiply` to something that you know is wrong, and run `go test` again.

## Why break a passing test?

Why do this? Well, during the implementation of `Multiply` you may have tweaked the test code a little. It's possible to have a passing test which passes incorrectly. For example, if you accidentally wrote `if want == got` instead of `if want != got`.

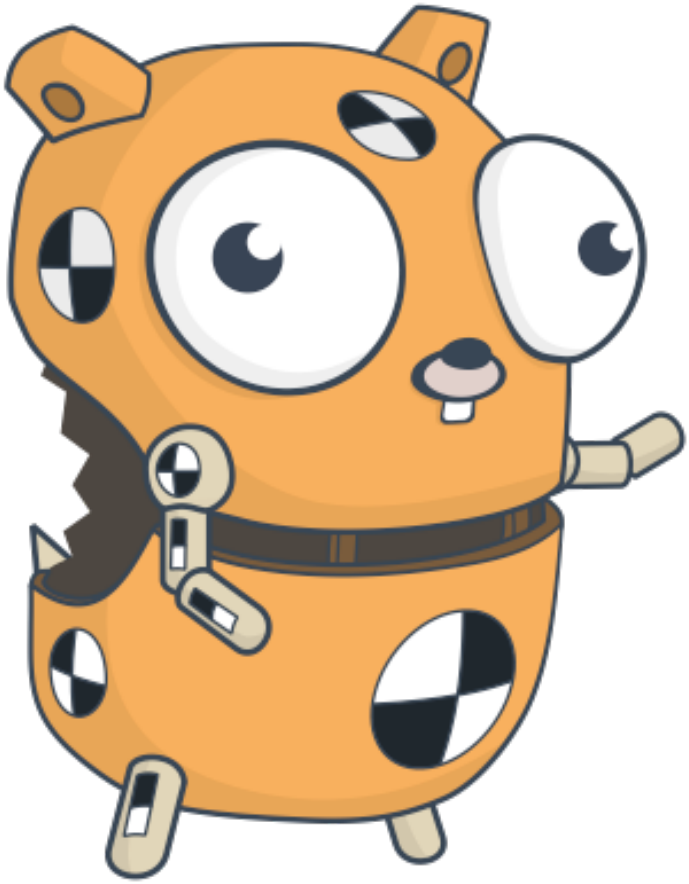
If you've already proved that `Multiply` is correct, and then you change the test to expect an *incorrect* result from `Multiply`, and it still passes, then, again, there's something really wrong with the test. (This step is also useful because you'll see the failure message that the test outputs. If it needs rewording or expanding with further information, this is a good time to do that.)

Once you're satisfied that the broken test fails correctly, restore it to the way it was, and check that it now *passes* again.

**STRETCH GOAL:** Think about other ways this test could be wrong. Try some of them out. What happens? What could you do to catch such potential problems?

Nice work! Go on to the next section.

## 5: Testing to destruction



Now that you've successfully designed and implemented a new calculator feature, test-first, it's time to think a little more about testing, and how you can extend it. The `Add`, `Subtract`, and `Multiply` functions are so simple that it's fairly difficult (though not at all impossible) to get them wrong. If `TestAdd` passes for one set of inputs, for example, it's hard to imagine other inputs that would make it fail.

But this doesn't apply to more complicated functions, where it's quite likely that they could work correctly for some inputs, but not for others. In this situation it will be a good idea to test multiple *cases* for each function: that is to say, different inputs with different expected results.

For example, the current version of `TestAdd` tests calling `Add(2, 2)` and checks that the result is 4. You could write a new function `TestAdd1and1`, for example, which calls `Add(1, 1)` and expects 2. But this would be tedious for many different pairs of inputs.

## Introducing test cases

Instead, the most elegant solution would be to define a number of different *test cases*—each representing a pair of inputs, together with the corresponding expected output—and then loop over them, calling `Add` with the inputs and checking the output against that expected. If *any* test case fails, the whole test fails. These are sometimes called *table tests*, because they represent the test data as a table of inputs and expected results.

Let's see what that might look like. First, it will be convenient to define a `struct` type to represent the test case. `struct` is Go's name for a structured data type: that is, something which contains multiple different bits of information, united into a single record. (Don't worry if this isn't familiar to you; we will cover these topics in more detail in subsequent books. What we're really concerned with here is the test logic, and this is just a way of setting up our test cases so that we can get to that.)

Let's do this inside the `TestAdd` function, since the test case will usually be specific to the function we're testing:

```
func TestAdd(t *testing.T) {
    t.Parallel()
    type testCase struct {
        a, b float64
        want float64
    }
```

## A slice of test cases

If you already have some experience of Go, you may be familiar with the notion of a *slice*: it's just Go's name for a *bunch* of things. In some languages this is called an *array*, but it's the same idea whatever you call it: a sequence of values of the same type. (You can learn more about structs, slices and other data types in the next book in this series, For the Love of Go: Data. But for now, you needn't worry about the details: just use the code suggested.)

This will be ideal for our test, where we would like to do the same operation on each of a bunch of test case: call the `Add` function with the test case inputs `a`

and `b`, and check its result against `want`.

In the `TestAdd` function, having already declared the `testCase` type, you could now declare a slice of these test cases with something like the following (to make these more concise, let's write each `testCase` literal on a single line):

```
testCases := []testCase{
    { a: 2, b: 2, want: 4 },
    { a: 1, b: 1, want: 2 },
    { a: 5, b: 0, want: 5 },
}
```

Make sure you're confident you understand what's happening here before moving on. You're setting up a slice called `testCases`, with three elements. Each element of this slice is an instance of the `testCase` struct type, which has the fields `a`, `b`, and `want`, representing the inputs and expected outputs of the test case.

## Looping over test cases

Now that you have the test cases, you can write the code to loop over them. That's straightforward; you can use the `range` operator to do this. We're still inside the `TestAdd` function, and having previously declared our `testCases` slice, we can now proceed to use it:

```
for _, tc := range testCases {
    ... // call Add() with the inputs
    ... // and check the result
}
```

You already know how to do the part inside the loop, because it's just the same as the existing `TestAdd` function, only with altered variable names:

```
got := calculator.Add(tc.a, tc.b)
if tc.want != got {
    t.Errorf("want %f, got %f", tc.want, got)
}
```

Each time through this code, `tc` will be the next `testCase` struct, so `tc.a` and `tc.b` will be the inputs to `Add`, and `tc.want` will be the expected result. Let's also extend the failure message to include the inputs, since these will be different for each test case:

```
t.Errorf("Add(%f, %f): want %f, got %f",
    tc.a, tc.b, tc.want, got)
```

What would that look like? Suppose the `1 + 1 == 2` case failed. You would see this message:

```
Add(1.000000, 1.000000): want 2.000000, got 3.000000
```

Very helpful!

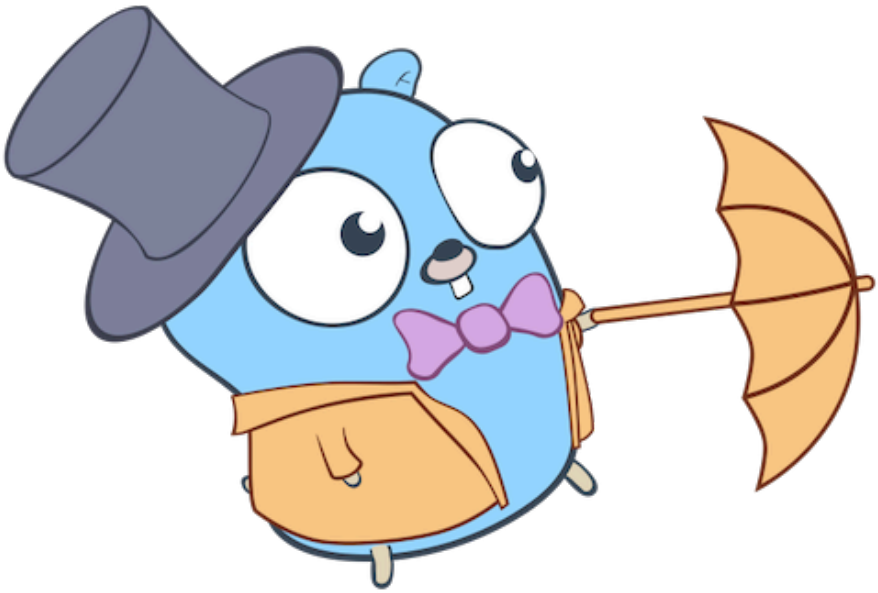
**GOAL:** Rewrite `TestAdd`, `TestSubtract`, and `TestMultiply` to test multiple cases for each function. Try to think of interesting and varied test cases: for example, negative numbers, zeroes, fractional numbers.

As before, once each test is passing, break it deliberately and make sure it fails in the expected way, with a helpful failure message.

**STRETCH GOAL:** Add a string field `name` to your test cases, and supply a human-readable name for each case. For example, a test case for `Add` might be named `"Two negative numbers which sum to a positive"`. Output the name of the test case along with the failure message.

When you're happy with the expanded tests, move on to the next section.

## 6: Trouble ahead



The board reviewed the calculator project yesterday, and they're really pleased with your progress. The only remaining major feature to be completed is division, and that needs a little more thought than the others.

Why? Because it's possible to write division expressions that have *undefined* results. For example:

```
6 / 0
```

This looks reasonable at first sight, but what you're really asking here is "What number, multiplied by zero, gives 6?" And clearly there is no such number. What should your calculator do in this case?

## Error values in Go

There are several options: it could give the answer `0`, which seems unsatisfying, or it could give an answer that represents infinity (perhaps the symbol $\infty$), or it could define a symbol that represents 'not a number' (the abbreviation `NaN` is sometimes used for this.)

But these are awkward attempts to work around a fundamental issue: dividing by zero is *not allowed* in the ordinary rules of arithmetic. So there's something about the `Divide` function which wasn't true of `Add`, `Subtract`, and `Multiply`: it's possible to give it *invalid input*.

A neat thing about functions in Go is that they're not limited to returning just a single value: they can return *multiple* values (and often do). So you could have `Divide` return two values: one can be the result of the calculation, as with the other functions, but the other can be an indicator saying whether or not the input was valid.

## Functions that return errors

The standard way to do this in Go is to write a function signature like:

```
func Divide(a, b float64) (float64, error)
```

It takes two inputs `a` and `b` of type `float64`, as with the other arithmetic functions, but unlike them, it returns two values: a `float64` (the answer) and a value of type `error` (the 'invalid input' indicator).

So how would this work? In the case of well-formed divisions (3 divided by 2, let's say), then `Divide` would return `1.5` as the answer (let's call this the 'data value') and `nil` as the error indicator (let's call this the 'error value'). `nil` means 'no error'—everything's fine. Here's what that would look like in the function:

```
return a / b, nil
```

Now let's take the case of dividing by zero. Instead of returning `nil` for the error value, the function would return something else. It doesn't really matter

what; literally anything other than `nil` sends the message that there was an error. But to be helpful, let's use `fmt.Errorf` to construct an error value which includes the invalid data and an explanation of what's wrong:

```
return 0, fmt.Errorf("bad input: %f, %f (division by zero is undefined)", a, b)
```

### What to return when there's an error

What would you return for the data value? Well, that doesn't matter either, and there is no right answer. Conventionally, you would just return zero in this case, but that's arbitrary: the presence of a non-nil error value is a warning that says "Ignore the data value. It's useless, because there was an error!"

**GOAL:** Define a test case struct type which allows you to test both valid and invalid inputs to `Divide`.

How would you need to modify the struct type we've been using up to now? Well, for functions with two inputs and one output, you had a type like this:

```
type testCase struct {
    a, b float64
    want float64
}
```

Two inputs, and one expected output. Now you have a function with two inputs and *two* outputs. Can you see what you need to add to the struct?

You need a way for the test case to specify whether or not the given inputs should produce an error. The simplest way to do that is to use a `bool` value, which you could call something like `errExpected`. If this value is `true`, it's saying the test case should cause `Divide` to return a non-nil error. For example, here's your division-by-zero test case expressed in this form:

```
{ a: 6, b: 0, want: 0, errExpected: true }
```

**GOAL:** Write a set of test cases for `Divide` using the new struct type. Make sure to include both valid and invalid inputs.

# 7: Managing expectations

Now that you have a set of suitable test cases for `Divide`, let's think about how to write the `TestDivide` function itself.

## Receiving multiple values from a function

First, since we've decided the `Divide` function will return *two* values, `float64` and `error`, we'll need to know how to receive those two results from the function call. You've already seen how to receive a single result from a function; for example, from `Add`:

```
got := calculator.Add(2, 2)
```

What does it look like when a function returns two values instead of one? Something like this:

```
got, err := calculator.Divide(4, 2)
```

Notice that there are two variables on the left hand side of the `:=`. That makes sense, because the function call on the right hand side returns two values. We have two values, so we need two variables to store them in. They are, in order,

`got` (which is the `float64` data value, as before), and `err`, which will hold the `error` value returned from the function.

## Testing functions which return multiple values

Previously, there were only two possible situations you had to deal with:

1. The data value didn't match the value you expected (fail)
2. The data value matched the expectation (pass)

Now there are *four* possible outcomes:

1. You expected an error, but you didn't get one (fail)
2. You *didn't* expect an error, but *did* get one (fail)
3. The error value was as you expected, but the data value didn't match the expectation (fail)
4. The error value was as you expected, and the data value also matched expectation (pass)

It might look a little complicated, but it's actually pretty straightforward, if you've followed everything so far. Make sure you're happy that you understand these four outcomes before moving on.

**GOAL:** Write the necessary logic for the `TestDivide` function, handling each of the four possible outcomes correctly.

## Checking for unexpected error status

This needs some care, because it's very easy to get the error-checking logic wrong. Consider this situation, for example (Outcome 1): you expected an error, but you got none (you got a `nil` error value). What should the failure message be for this outcome?

```
Divide(6, 0): unexpected error status: nil
```

This would be a reasonable output. What you *don't* want to see is something like this:

```
Divide(6, 0): want 0, got ...
```

Why not? Because the reason the test is failing is that the function is *not returning an error* when it's supposed to. In that situation, the data value is entirely irrelevant, and comparing it against any expectation would be a waste of time. Worse, it would give a misleading message, making it harder to debug the problem. (A good way to catch this kind of issue is to give *obviously crazy* `want` values for your error test cases, such as `999`. If you see these values in your test outputs, something's wrong with the test, because in the case of unexpected error status, it shouldn't even check the data value.)

The implication of this is that the test needs to check the error value against expectation *before even looking at the data value.* Here's one way to do that:

```
errReceived := err != nil
if tc.errExpected != errReceived {
    t.Fatalf("Divide(%f, %f): unexpected
      error status: %v", tc.a, tc.b,
      errReceived)
}
```

This neatly deals with both outcomes 1 and 2, which both involve "unexpected error status"; either because you expected one but didn't get one, or didn't expect one and did get one!

What is `errReceived`? It's a boolean value which tells you whether or not you got an error from `Divide`. If you think about it, the value of this variable needs to be the same as the test case's `errExpected` field. If not, it's a test failure.

A shorter way to write this logic is:

```
if tc.errExpected != (err != nil) {
    t.Fatalf(...)
}
```

and you'll sometimes see this in Go code. However, this isn't as clear as explicitly setting `errReceived`, and I recommend you be as explicit as possible, especially in tests.

## Failing and bailing

Why call `t.Fatalf()` in this case, rather than `t.Errorf()`? What's the difference?

Calling `t.Errorf()` outputs the failure message, but the test function will continue executing (it might go on to make other comparisons, for example). On the other hand, `t.Fatalf()` outputs its failure message and exits the test function immediately. It fails and bails, you might say.

Bailing is useful when you have established that things are so broken that there's no point continuing with the test. For example, if the error status from the function is unexpected, then comparing the data value is pointless (indeed, it would be misleading about the causes of the failure). So `t.Fatalf()` allows you to skip any further checks within this test (though it doesn't stop *other* test functions from being run. It bails out of *this* test, not all tests.)

Another common use of `t.Fatalf()` is when there's some error setting up the test itself. For example, suppose you want to compare the output of a function with the contents of some file (known as a *golden file*, and conventionally placed in a `testdata/` directory). You might start by reading this file from disk, in preparation for comparing it with the test output. But if reading the file fails for some reason (file missing, bad permissions, out of memory; life is full of unexpected snags) then clearly there's no point continuing with this test.

Bailing out with `t.Fatalf()` (and some diagnostic information) is appropriate here.

## Checking the data value

There's one thing left to do in this test, and it's something we've done before: checking the data value returned from the function against your expectation.

However, now there's an extra subtlety, because you're dealing with a function that can error. When you're testing a case which is *supposed* to error, you don't want to check the data value. No expectation for it would be meaningful (indeed, if you're wily enough, you have set an obviously wrong expectation of `999`, or something like that, to detect precisely this problem).

Therefore, once you have established that the error status is as expected (if not, we've already failed and bailed), you can write:

```
if !tc.errExpected && tc.want != got {
```

In other words, compare the data value *only if the test case does not expect an error*.

**GOAL:** Write the minimum code necessary to make `TestDivide` pass.

When your test cases all pass, and you've verified them all by deliberately breaking their expectations, and you've checked that the test outputs the correct failure message for unexpected error statuses, then you can move on to the next (and final) section.

**STRETCH GOAL:** Write a new test for one or more of your functions which generates *random* test inputs instead of using prepared cases (you can use the `math/rand` library for this). For example, you might write a `TestAddRandom` function which generates two random `float64` values, sums them to get the expected result (*not* using the `Add` function!), and calls `Add` to verify that it returns the same answer. Do this many times, say, a hundred times, to give the function a good workout.

# 8: Roots music

Well done! That was a good piece of work getting the test cases and error handling set up for the `Divide` function. In fact, you now know just about everything you need to write great tests in Go!

## The structure of tests

All tests follow more or less the same basic structure:

- Define the test cases with their inputs and expected outputs (including any error outputs)
- Iterate over the test cases, calling the function under test with each set of inputs

- Check that the outputs match expectations, and fail otherwise with some helpful output

And you've also practiced a specific workflow, or sequence of actions, that you can use when developing any piece of Go code:

## The development process

1. Start by writing a test for the function (even though it doesn't exist yet)
2. See the test fail with a compilation error because the function doesn't exist yet.
3. Write the minimum code necessary to make the test compile and fail.
4. Make sure that the test fails for the reason you expected.
5. Write the minimum code necessary to make the test pass.
6. Change the test expectations to deliberately make the test fail, and check that the failure message is accurate and informative.
7. Restore the original expectations and make sure the test still passes.
8. Commit!

In practice, you may go through this loop a few times with any particular test. You start with something very simple, so that you are never facing an insurmountable problem. If you are, then you try to break that problem down into smaller problems, and repeat this process with those.

## Comparing floating-point values

The way of representing decimal fractional numbers that's used in most programming languages is called *floating-point* (that's why Go's fractional data type is called `float64`: it's a 64-bit floating-point number).

And one thing about floating-point numbers is that they have limited *precision*. If you think about the decimal representation of a fraction like one-third, it's `0.33333333...`, but it never stops. It's 3s all the way down. Clearly, representing a number like this in a fixed number of bits is going to involve a loss of precision.

This is an issue for us when writing tests, because we want to compare the result from a function with an expected value, and if there's a tiny imprecision in the values, they may not compare exactly equal. So you may find puzzling test failures if you're trying to compare floating-point values with the `==` operator.

Instead, what we can do is write a function something like this:

```
func closeEnough(a, b, tolerance float64) bool {
    return math.Abs(a-b) <= tolerance
}
```

Then we can compare our `want` and `got` values using this function instead of `==`:

```
if !closeEnough(want, got, 0.0000001)
```

There are still more sophisticated ways of comparing floating-point values, but this is a good place to start.

## A `Sqrt` function

Our basic calculator is complete, but you've just received an urgent message from the VP of Sales. She wants an extra premium feature which can be used to upsell users to the 'Enterprise' calculator. The ordinary calculator has been re-branded as the 'Home' edition.

It seems that enterprise users would like the ability to take square roots, so the VP's asking you to develop a `Sqrt` function. You already know everything you need to know to design, test, and develop this feature, so let's get started!

**GOAL:** Develop a `Sqrt` function which takes a `float64` value and returns its square root as a `float64` value.

You can use any standard or third-party libraries you think are appropriate. The function should return an error for negative inputs (no real number squared equals a negative number, so taking the square root of a negative number is not a valid operation). Your test should check the error handling in the same way as the test for `Divide`.

When the feature is complete to your own satisfaction, you're done. Nice work. The company has awarded you a sizable bonus, and an all-expenses-paid vacation. Enjoy it!

## Going further

You've now learned not only the basics of using Go's `testing` library, but some fairly advanced techniques, including table tests, and accurately testing functions with error returns. You're ready to move on to book 2, For the Love of Go: Data, but before you do, you may like to try some of these challenges to make sure your testing skills are as sharp as possible.

### Variadic functions

So far we've dealt only with function which take a fixed number of parameters (for example, `Add` takes two parameters, `a` and `b`). But some Go functions can take a variable number of parameters; these are called *variadic functions*. How can we write these, and how can we test them?

A variadic function signature looks like this:

```
func AddMany(inputs ...float64) float64
```

The `...` indicates that there can be zero, one, two, or indeed any number of `float64` parameters to the function. Inside the function, these values will be available as a slice:

```
for _, input := range inputs {
    // 'input' is each successive parameter
}
```

Extend your existing tests and functions to allow *two or more* inputs to `Add`, `Subtract`, `Multiply`, and `Divide`. For example, calling `Divide(12, 4, 3)` should divide 12 by 4, divide the result of that by 3, then return *that* result. Calling `Multiply` with ten inputs should return the result of multiplying them all together.

#### Evaluating expressions

We've implemented most of the functionality of a traditional electronic calculator, but modern calculators (and computer programs such as spreadsheets) can understand *arithmetic expressions*. That is, strings like `12 * 3`, which would evaluate to 36, or `2 + 2`, which evaluates to 4.

Write a function which accepts strings of this form, and returns the result of evaluating that expression. You only need to handle expressions which have a single floating-point value, followed by a single operator, followed by a single floating-point value, ignoring any whitespace. For example, the following should all be valid inputs to your function:

```
2*2
1 + 1.5
18   /   6
100-0.1
```

# About this book

## Who wrote this?

John Arundel is a Go teacher and mentor of many years experience. He's helped literally thousands of people to learn Go, with friendly, supportive, professional mentoring, and he can help you too. Find out more:

- Learn Go with mentoring

## Feedback

If you enjoyed this book, let me know! Email go@bitfieldconsulting.com with your comments. If you didn't enjoy it, or found a problem, I'd like to hear that too. All your feedback will go to improving the book, and the others in this series.

Also, please tell your friends, or post about the book on social media. I'm not a global mega-corporation, and I don't have a publisher or a marketing budget: I write and produce these books myself, at home, in my spare time. They're priced very cheaply because I'm not doing this for the money: I'm doing it so that I can help bring the love of Go to as many people as possible.

That's where you can help, too. If you love Go, tell a friend about this book!

## Mailing list

If you'd like to hear about it first when I publish new books, or even join my exclusive group of beta readers to give feedback on drafts in progress, you can subscribe to my mailing list here:

- Subscribe to Bitfield updates

## Code and solutions

Throughout this book you've mostly been writing code yourself, with help from some basic starter files in the GitHub repo for the book:

- For the Love of Go: Fundamentals on GitHub

And you can also find some sample solutions in the same repo, on the `sample` branch:

- Sample `calculator.go`
- Sample `calculator_test.go`

You might like to compare your solutions with these. They don't represent the only possible way to solve the exercises, or even the best way. But they may be a useful reference point for you. (If you can improve on them, let me know!)

## What's next?

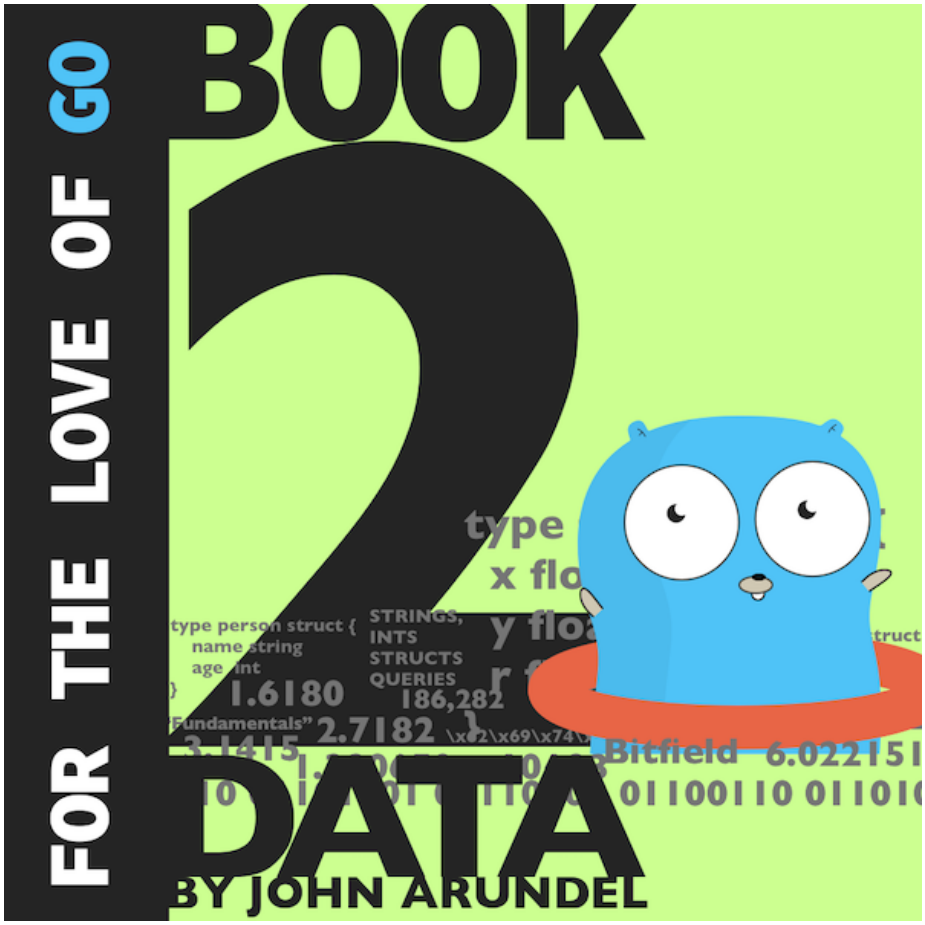
If you enjoyed the book, you'll be delighted to know that a sequel is now available! For the Love of Go: Data takes what we've learned here, and applies it to building the core functionality of a cool online bookstore. Along the way we'll find out about Go's basic data types, defining our own struct types, manipulating collections of data using slices and maps, and much more!

Specifically, you'll learn:

- What we mean by 'data' and why it's so important in programming
- All about Go's built-in data types
- How to define and use structured data types using the `struct` keyword
- How to work with collections of data using *slices*
- How to store and retrieve data by key using Go's *map* type

Because we'll be building a real software project, you'll also learn a few useful techniques that we can use for *all* programs, not just Go:

- How to write *user stories* to guide the design of a project
- How to prioritise your user stories using a *Minimum Viable Product* (MVP) strategy




- For the Love of Go: Data

You might like to try another set of exercises in this series:

- The G-machine guides you through the development of a simplified virtual CPU in Go, complete with its own machine language. Along the way you'll learn some useful computer science fundamentals.

## Further reading

This is the first in a series of ebooks on Go programming, called 'For the Love of Go'. You can find the others here:

- The 'For the Love of Go' series

You can find more Go tutorials and exercises here:

- Go tutorials from Bitfield

I have a YouTube channel where I post occasional videos on Go, and there are also some curated playlists of what I judge to be the very best Go talks and tutorials available, here:

- Bitfield Consulting on YouTube

Gopher images by the magnificent egonelbre