Arrays and slices

You can find all the code for this chapter here

Arrays allow you to store multiple elements of the same type in a variable in a particular order.

When you have an array, it is very common to have to iterate over them. So let's use our new-found knowledge of `for` to make a `Sum` function. `Sum` will take an array of numbers and return the total.

Let's use our TDD skills

Write the test first

Create a new folder to work in. Create a new file called `sum_test.go` and insert the following:

```go
package main

import "testing"

func TestSum(t *testing.T) {

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

	got := Sum(numbers)
	want := 15

	if got != want {
		t.Errorf("got %d want %d given, %v", got, want, numbers)
	}
}
```

Arrays have a *fixed capacity* which you define when you declare the variable. We can initialize an array in two ways:

- [N]type{value1, value2, ..., valueN} e.g. `numbers := [5]int{1, 2, 3, 4, 5}`
- [...]type{value1, value2, ..., valueN} e.g. `numbers := [...]int{1, 2, 3, 4, 5}`

It is sometimes useful to also print the inputs to the function in the error message. Here, we are using the `%v` placeholder to print the "default" format, which works well for arrays.

Read more about the format strings

Try to run the test

If you had initialized go mod with `go mod init main` you will be presented with an error `_testmain.go:13:2: cannot import "main"`. This is because according to common practice, package main will only contain integration of other packages and not unit-testable code and hence Go will not allow you to import a package with name `main`.

To fix this, you can rename the main module in `go.mod` to any other name.

Once the above error is fixed, if you run `go test` the compiler will fail with the familiar `./sum_test.go:10:15: undefined: Sum` error. Now we can proceed with writing the actual method to be tested.

Write the minimal amount of code for the test to run and check the failing test output

In `sum.go`

```go
package main

func Sum(numbers [5]int) int {
	return 0
}
```

Your test should now fail with *a clear error message*

`sum_test.go:13: got 0 want 15 given, [1 2 3 4 5]`

Write enough code to make it pass

```go
func Sum(numbers [5]int) int {
	sum := 0
	for i := 0; i < 5; i++ {
		sum += numbers[i]
	}
	return sum
}
```

To get the value out of an array at a particular index, just use `array[index]` syntax. In this case, we are using `for` to iterate 5 times to work through the array and add each item onto `sum`.

Refactor

Let's introduce `range` to help clean up our code

```go
func Sum(numbers [5]int) int {
	sum := 0
	for _, number := range numbers {
		sum += number
	}
	return sum
}
```

`range` lets you iterate over an array. On each iteration, `range` returns two values - the index and the value. We are choosing to ignore the index value by using `_` blank identifier.

Arrays and their type

An interesting property of arrays is that the size is encoded in its type. If you try to pass an `[4]int` into a function that expects `[5]int`, it won't compile. They are different types so it's just the same as trying to pass a `string` into a function that wants an `int`.

You may be thinking it's quite cumbersome that arrays have a fixed length, and most of the time you probably won't be using them!

Go has *slices* which do not encode the size of the collection and instead can have any size.

The next requirement will be to sum collections of varying sizes.

Write the test first

We will now use the slice type which allows us to have collections of any size. The syntax is very similar to arrays, you just omit the size when declaring them

`mySlice := []int{1,2,3}` rather than `myArray := [3]int{1,2,3}`

```go
func TestSum(t *testing.T) {

	t.Run("collection of 5 numbers", func(t *testing.T) {
		numbers := [5]int{1, 2, 3, 4, 5}

		got := Sum(numbers)
		want := 15

		if got != want {
			t.Errorf("got %d want %d given, %v", got, want, numbers)
		}
	})

	t.Run("collection of any size", func(t *testing.T) {
		numbers := []int{1, 2, 3}

		got := Sum(numbers)
		want := 6

		if got != want {
			t.Errorf("got %d want %d given, %v", got, want, numbers)
		}
	})

}
```

Try and run the test

This does not compile

`./sum_test.go:22:13: cannot use numbers (type []int) as type [5]int in argument to Sum`

Write the minimal amount of code for the test to run and check the failing test output

The problem here is we can either

- Break the existing API by changing the argument to `Sum` to be a slice rather than an array. When we do this, we will potentially ruin someone's day because our *other* test will no longer compile!
- Create a new function

In our case, no one else is using our function, so rather than having two functions to maintain, let's have just one.

```go
func Sum(numbers []int) int {
	sum := 0
	for _, number := range numbers {
		sum += number
	}
	return sum
}
```

If you try to run the tests they will still not compile, you will have to change the first test to pass in a slice rather than an array.

Write enough code to make it pass

It turns out that fixing the compiler problems were all we need to do here and the tests pass!

Refactor

We already refactored `Sum` - all we did was replace arrays with slices, so no extra changes are required. Remember that we must not neglect our test code in the refactoring stage - we can further improve our `Sum` tests.

```go
func TestSum(t *testing.T) {

	t.Run("collection of 5 numbers", func(t *testing.T) {
		numbers := []int{1, 2, 3, 4, 5}

		got := Sum(numbers)
		want := 15

		if got != want {
			t.Errorf("got %d want %d given, %v", got, want, numbers)
		}
	})

	t.Run("collection of any size", func(t *testing.T) {
		numbers := []int{1, 2, 3}

		got := Sum(numbers)
		want := 6

		if got != want {
			t.Errorf("got %d want %d given, %v", got, want, numbers)
		}
	})

}
```

It is important to question the value of your tests. It should not be a goal to have as many tests as possible, but rather to have as much *confidence* as possible in your code base. Having too many tests can turn in to a real problem and it just adds more overhead in maintenance. **Every test has a cost**.

In our case, you can see that having two tests for this function is redundant. If it works for a slice of one size it's very likely it'll work for a slice of any size (within reason).

Go's built-in testing toolkit features a coverage tool. Whilst striving for 100% coverage should not be your end goal, the coverage tool can help identify areas of your code not covered by tests. If you have been strict with TDD, it's quite likely you'll have close to 100% coverage anyway.

Try running

`go test -cover`

You should see

```
PASS
coverage: 100.0% of statements
```

Now delete one of the tests and check the coverage again.

Now that we are happy we have a well-tested function you should commit your great work before taking on the next challenge.

We need a new function called `SumAll` which will take a varying number of slices, returning a new slice containing the totals for each slice passed in.

For example

`SumAll([]int{1,2}, []int{0,9})` would return `[]int{3, 9}`

or

`SumAll([]int{1,1,1})` would return `[]int{3}`

Write the test first

```go
func TestSumAll(t *testing.T) {

	got := SumAll([]int{1, 2}, []int{0, 9})
	want := []int{3, 9}

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

Try and run the test

`./sum_test.go:23:9: undefined: SumAll`

Write the minimal amount of code for the test to run and check the failing test output

We need to define `SumAll` according to what our test wants.

Go can let you write *variadic functions* that can take a variable number of arguments.

```go
func SumAll(numbersToSum ...[]int) (sums []int) {
	return
}
```

This is valid, but our tests still won't compile!

`./sum_test.go:26:9: invalid operation: got != want (slice can only be compared to nil)`

Go does not let you use equality operators with slices. You *could* write a function to iterate over each `got` and `want` slice and check their values but for convenience sake, we can use `reflect.DeepEqual` which is useful for seeing if *any* two variables are the same.

```go
func TestSumAll(t *testing.T) {

	got := SumAll([]int{1, 2}, []int{0, 9})
	want := []int{3, 9}

	if !reflect.DeepEqual(got, want) {
		t.Errorf("got %v want %v", got, want)
	}
}
```

(make sure you `import reflect` in the top of your file to have access to `DeepEqual`)

It's important to note that `reflect.DeepEqual` is not "type safe" - the code will compile even if you did something a bit silly. To see this in action, temporarily change the test to:

```go
func TestSumAll(t *testing.T) {

	got := SumAll([]int{1, 2}, []int{0, 9})
	want := "bob"

	if !reflect.DeepEqual(got, want) {
		t.Errorf("got %v want %v", got, want)
	}
}
```

What we have done here is try to compare a `slice` with a `string`. This makes no sense, but the test compiles! So while using `reflect.DeepEqual` is a convenient way of comparing slices (and other things) you must be careful when using it.

Change the test back again and run it. You should have test output like the following

`sum_test.go:30: got [] want [3 9]`

Write enough code to make it pass

What we need to do is iterate over the varargs, calculate the sum using our existing `Sum` function, then add it to the slice we will return

```go
func SumAll(numbersToSum ...[]int) []int {
	lengthOfNumbers := len(numbersToSum)
	sums := make([]int, lengthOfNumbers)

	for i, numbers := range numbersToSum {
		sums[i] = Sum(numbers)
	}

	return sums
}
```

Lots of new things to learn!

There's a new way to create a slice. `make` allows you to create a slice with a starting capacity of the `len` of the `numbersToSum` we need to work through.

You can index slices like arrays with `mySlice[N]` to get the value out or assign it a new value with `=`

The tests should now pass.

Refactor

As mentioned, slices have a capacity. If you have a slice with a capacity of 2 and try to do `mySlice[10] = 1` you will get a *runtime* error.

However, you can use the `append` function which takes a slice and a new value, then returns a new slice with all the items in it.

```go
func SumAll(numbersToSum ...[]int) []int {
	var sums []int
	for _, numbers := range numbersToSum {
		sums = append(sums, Sum(numbers))
	}

	return sums
}
```

In this implementation, we are worrying less about capacity. We start with an empty slice `sums` and append to it the result of `Sum` as we work through the varargs.

Our next requirement is to change `SumAll` to `SumAllTails`, where it will calculate the totals of the "tails" of each slice. The tail of a collection is all items in the collection except the first one (the "head").

Write the test first

```go
func TestSumAllTails(t *testing.T) {
	got := SumAllTails([]int{1, 2}, []int{0, 9})
	want := []int{2, 9}

	if !reflect.DeepEqual(got, want) {
		t.Errorf("got %v want %v", got, want)
	}
}
```

Try and run the test

`./sum_test.go:26:9: undefined: SumAllTails`

Write the minimal amount of code for the test to run and check the failing test output

Rename the function to `SumAllTails` and re-run the test

`sum_test.go:30: got [3 9] want [2 9]`

Write enough code to make it pass

```go
func SumAllTails(numbersToSum ...[]int) []int {
	var sums []int
	for _, numbers := range numbersToSum {
		tail := numbers[1:]
		sums = append(sums, Sum(tail))
	}

	return sums
}
```

Slices can be sliced! The syntax is `slice[low:high]`. If you omit the value on one of the sides of the `:` it captures everything to that side of it. In our case, we are saying "take from 1 to the end" with `numbers[1:]`. You may wish to spend some time writing other tests around slices and experiment with the slice operator to get more familiar with it.

Refactor

Not a lot to refactor this time.

What do you think would happen if you passed in an empty slice into our function? What is the "tail" of an empty slice? What happens when you tell Go to capture all elements from `myEmptySlice[1:]`?

Write the test first

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

	t.Run("make the sums of some slices", func(t *testing.T) {
		got := SumAllTails([]int{1, 2}, []int{0, 9})
		want := []int{2, 9}

		if !reflect.DeepEqual(got, want) {
			t.Errorf("got %v want %v", got, want)
		}
	})

	t.Run("safely sum empty slices", func(t *testing.T) {
		got := SumAllTails([]int{}, []int{3, 4, 5})
		want := []int{0, 9}

		if !reflect.DeepEqual(got, want) {
			t.Errorf("got %v want %v", got, want)
		}
	})

}
```

Try and run the test

```
panic: runtime error: slice bounds out of range [recovered]
    panic: runtime error: slice bounds out of range
```

Oh no! It's important to note the test *has compiled*, it is a runtime error. Compile time errors are our friend because they help us write software that works, runtime errors are our enemies because they affect our users.

Write enough code to make it pass

```go
func SumAllTails(numbersToSum ...[]int) []int {
	var sums []int
	for _, numbers := range numbersToSum {
		if len(numbers) == 0 {
			sums = append(sums, 0)
		} else {
			tail := numbers[1:]
			sums = append(sums, Sum(tail))
		}
	}

	return sums
}
```

Refactor

Our tests have some repeated code around the assertions again, so let's extract those into a function

```go
func TestSumAllTails(t *testing.T) {

	checkSums := func(t testing.TB, got, want []int) {
		t.Helper()
		if !reflect.DeepEqual(got, want) {
			t.Errorf("got %v want %v", got, want)
		}
	}

	t.Run("make the sums of tails of", func(t *testing.T) {
		got := SumAllTails([]int{1, 2}, []int{0, 9})
		want := []int{2, 9}
		checkSums(t, got, want)
	})

	t.Run("safely sum empty slices", func(t *testing.T) {
		got := SumAllTails([]int{}, []int{3, 4, 5})
		want := []int{0, 9}
		checkSums(t, got, want)
	})

}
```

A handy side-effect of this is this adds a little type-safety to our code. If a developer mistakenly adds a new test with `checkSums(t, got, "dave")` the compiler will stop them in their tracks.

```
$ go test
./sum_test.go:52:21: cannot use "dave" (type string) as type []int in argument to che
```

Wrapping up

We have covered

- Arrays
- Slices
  - The various ways to make them
  - How they have a *fixed* capacity but you can create new slices from old ones using `append`
  - How to slice, slices!
- `len` to get the length of an array or slice
- Test coverage tool
- `reflect.DeepEqual` and why it's useful but can reduce the type-safety of your code

We've used slices and arrays with integers but they work with any other type too, including arrays/slices themselves. So you can declare a variable of `[][]string` if you need to.

Check out the Go blog post on slices for an in-depth look into slices. Try writing more tests to solidify what you learn from reading it.

Another handy way to experiment with Go other than writing tests is the Go playground. You can try most things out and you can easily share your code if you need to ask questions. I have made a go playground with a slice in it for you to experiment with.

Here is an example of slicing an array and how changing the slice affects the original array; but a "copy" of the slice will not affect the original array. Another example of why it's a good idea to make a copy of a slice after slicing a very large slice.