golang notes

November 2, 2014

Contents

Chapter 1

Getting Started

1.1 Go Tools

Go is a **compiled programming language**, which means source code (the code you write) is translated into a language that your computer can understand.
To find out the go version which is running:

```
1 luca@scippane:~$ go version
2 go version go1.3.1 linux/amd64
```

The **Go tool suite** is made up of several different commands and sub-commands. A list of those commands is available by typing:

```
1 luca@scippane:~$ go help
```

To run a program, given a file with the code called *main.go*, then use the command:

```
1 go run main.go
```

from the folder where main.go is saved.
The *go run* command takes the subsequent files (separated by spaces), compiles them into an executable saved in a temporary directory and then runs the program.

1.2 A Go Program

1.2.1 Package Declaration

```
1 package main
```

This is know as a **package declaration**. Every Go program **must start with a package declaration**. Packages are Go's way of organizing and reusing code.

There are two types of Go programs: **executables** and **libraries**. Executable applications are the kinds of programs that we can run directly from the terminal. Libraries are collections of code that we package together so that we can use them in other programs.

1.2.2 Import

```
1 import "fmt"
```

The **import** keyword is how we **include code from other packages** to use with our program. The *fmt* package (shorthand for format) implements formatting for input and output.

1.2.3 Comments

Go supports two different styles of comments:

- // comments in which all the text between the // and the end of the line is part of the comment
- /* */ comments where everything between the * s is part of the comment. (And may include multiple lines).

1.2.4 Function Declaration

```
1  func main() {
2 fmt.Println("Hello World")
3 }
```

All functions start with the **keyword *func*** followed by the **name of the function** (*main* in this case), a list of zero or more **parameters** surrounded by parentheses, an optional **return type** and a **body** which is surrounded by curly braces.
The name *main* is special because it's the function that gets called when you execute the program.

1.2.5 Documentation

You can find out more about a function by using the command *godoc*; for example:

```
1 godoc fmt Println
```

Chapter 2

Types

Go is a **statically typed programming language**. This means that variables always have a specific type and that type cannot change.

2.1 Numbers

Go has several different types to represent numbers. Generally we split numbers into two different kinds: **integers** and **floating-point** numbers.

2.1.1 Integers

Integers are numbers without a decimal component.
Go's integer types are: *uint8 , uint16 , uint32 , uint64 , int8 , int16 , int32 and int64*. 8, 16, 32 and 64 tell us how many bits each of the types use.
uint means unsigned integer while *int* means signed integer.
There are also **3 machine dependent integer types**: *uint , int and uintptr*.
They are machine dependent because their size depends on the type of architecture you are using.
Generally if you are working with integers you should just use the *int* type.

2.1.2 Floating Point

Floating point numbers are numbers that contain a decimal component. Go has **two floating point types**: *float32* and *float64*.
As well as **two additional types for representing complex numbers**: *complex64* and *complex128*.
Generally **we should stick with float64 when working with floating point numbers**.

2.2 Strings

A string is a sequence of characters with a definite length used to represent text.
String literals can be created using **double quotes** "Hello World" or **back ticks** `Hello World`. The difference between these is that double quoted strings can not contain newlines and they allow special escape sequences. For example \n gets replaced with a new line and \t gets replaced with a tab character.
Several common operations on strings include:

- finding the length of a string: *len("Hello World")*
- accessing an individual character in the string: *"Hello World"(1)*
- and concatenating two strings together: *"Hello " + "World"*

Things to notice

- A space is also considered a character
- Strings are indexed starting at 0 not 1
- The function which returns a character in a position returns an int not a string
- Concatenation uses the same symbol as addition

2.3 Boolean

A boolean value is a special 1 bit integer type used to represent true and false.
Three logical operators are used with boolean values:

- && AND
- | | OR
- ! NOT

Chapter 3

Variables

A variable is a storage location, with a specific type and an associated name.
Variables in Go are created by first using the ***var* keyword**, then specifying the **variable name** (e.g. x), the **type** (e.g. string)and finally assigning a **value** to the variable (e.g. "Hello World"). The last step is optional.

```
1 var x string = "Hello World"
```

To declare a variable Go also supports a shorter statement:

```
1 x := "Hello World"
2 x := 5
```

In this shorter statement the type and the keyword *var* can be omitted.

3.1 Equality

The symbol for equalityis == .
== is an operator like + and it returns a boolean.

3.2 Scope

Go is lexically scoped using blocks.
Basically this means that the variable exists within the nearest curly braces (a block) including any nested curly braces (blocks), but not outside of them.
Example 1 - This is ok

```
1 var x string = "Hello World"
2 func main() {
3   fmt.Println(x)
4 }
```

```
5  func f() {
6    fmt.Println(x)
7  }
```

Example 2 - This is NOT ok (variable x is undefined for func f)

```
1  func main() {
2    var x string = "Hello World"
3    fmt.Println(x)
4  }
5  func f() {
6    fmt.Println(x)
7  }
```

3.3 Constants

Go also has support for **constants**. Constants are basically variables whose values cannot be changed later. They are created in the same way you create variables but instead of using the *var* keyword we use the *const* keyword.

3.4 Defining Multiple Variables

Go also has another shorthand when you need to define multiple variables. Use the keyword *var* (or *const*) followed by parentheses with each variable on its own line.

```
1  var (
2    a = 5
3    b = 10
4    c = 15
5  )
```

Chapter 4

Control Structures

4.1 for

The *for* statement allows us to repeat a list of statements (a block) multiple times.
Other programming languages have a lot of different types of loops (while, do, until, foreach, ...) but Go only has one that can be used in a variety of different ways.
Syntax

```
for i := 1; i <= 10; i++ {
  fmt.Println(i)
}
```

4.2 if

It looks like the compiler really requires else to be after the } like below and the curly brackets are mandatory.

```
if i % 2 == 0 {
// even
} else {
// odd
}
```

If statements also have an **optional *else* part**. If the condition evaluates to true then the block after the condition is run, otherwise either the block is skipped or if the else block is present that block is run.
If statements can also have ***else if* parts**:

```
if i % 2 == 0 {
  // divisible by 2
} else if i % 3 == 0 {
  // divisible by 3
} else if i% 4 == 0 {
```

```
6     // divisible by 4
7 }
```

The conditions are checked top down and **the first one to result in true will have its associated block executed**. None of the other blocks will execute, even if their conditions also pass. (So for example the number 8 is divisible by both 4 and 2, but the // divisible by 4 block will never execute because the // divisible by 2 block is done first).

4.3 switch

A *switch* statement starts with the keyword *switch* followed by an **expression** and then a series of *case*(s).
The value of the expression is compared to the expression following each *case* keyword. If they are equivalent then the statement(s) following the *:* is executed.
Like an if statement each case is checked top down and **the first one to succeed is chosen**. A switch also supports **a default case** which will happen if none of the cases matches the value.

```
1 switch i {
2 case 0: fmt.Println("Zero")
3 case 1: fmt.Println("One")
4 case 2: fmt.Println("Two")
5 default: fmt.Println("Unknown Number")
6 }
```

Chapter 5

Arrays, Slices and Maps

5.1 Arrays

An array is a numbered sequence of elements of a single type with a fixed length. In Go they look like this:

```
var x [5]int
```

Like strings, arrays are indexed starting from 0.
Note how it is possible to print all elements in an array:

```
package main
import "fmt"
func main() {
  var x [5]int
  x[4] = 100
  fmt.Println(x) //Print all elements
}
```

The result is:

```
[0 0 0 0 100]
```

Consider this simple program:

```
func main() {
  var x [5]float64
  x[0] = 98
  x[1] = 93
  x[2] = 77
  x[3] = 82
  x[4] = 83
  var total float64 = 0
  for _, value := range x {
    total += value
  }
  fmt.Println(total / float64(len(x)))
}
```

There are few things to note:

- *float64(len(x))* is an example of **type conversion**. In general to convert between types you use the type name like a function.
- *for _, value := range x* is another way of using the for loop. *value* is the same as *x(i)* . We use the keyword *range* followed by the name of the variable we want to loop over
- A single _ (underscore) is used to tell the compiler that **we don't need this**. (In this case we don't need the iterator variable)

Go also provides **a shorter syntax for creating arrays**:

```
x := [5]float64{ 98, 93, 77, 82, 83 }
```

Go allows you to break it up like this:

```
x := [5]float64{
  98,
  93,
  77,
  82,
  83,
}
```

Notice the extra trailing , after 83 .
If we want to remove an element from the array we can do something like this:

```
x := [4]float64{
  98,
  93,
  77,
  82,
  // 83,
}
```

This example illustrates **a major issue with arrays**: their **length is fixed** and **part of the array's type name**. In order to remove the last item, we actually had to **change the type as well**. Go's solution to this problem is to use a different type: **slices**.

5.1.1 Creating Arrays

Arrays are created using the syntaxes:

```
[length]Type
[N]Type{value1, value2, ... , valueN}
[ ... ]Type{value1, value2, ... , valueN}
```

If the ... (ellipsis) operator is used in this context, Go will **calculate the array's length for us**. In all cases an **array's length is fixed and unchangeable**.
Go guarantees that **all array items are initialized to their zero value** if they are not explicitly initialized or are only partly initialized when they are created. Example:

```
grid1[1][0], grid1[1][1], grid1[1][2] = 8, 6, 2
fmt.Printf("%-8T %2d %v\n", grid1, len(grid1), grid1)
```

would result in:

```
[[0 0 0] [8 6 2] [0 0 0]]
```

The length of an array is given by the *len()* function.
Arrays can be iterated using a *for ... range* loop

5.1.2 Comments About Arrays

In general, **Go's slices are more flexible, powerful, and convenient than arrays**.

- Arrays are passed by value (i.e., copied), whereas slices are cheap to pass, regardless of their length or capacity, since they are references.
- Arrays are of fixed size whereas slices can be resized.

It is recommended to slices unless there is a very specific need to use an array in a particular case.
Both arrays and slices can be sliced using the syntaxes shown in Section 5.2.5

5.2 Slices

A slice is a segment of an array. Like arrays slices are indexable and have a length. Unlike arrays **this length is allowed to change**. Here's an example of a slice:

```
var x []float64
```

A Go slice is a **variable-length fixed-capacity sequence of items of the same type**. Despite their fixed capacity, slices can be shrunk by slicing them and can be grown using the efficient built-in *append()* function.
Multidimensional slices can be created quite naturally by using items that are themselves slices and the lengths of the inner slices in multidimensional slices may vary.
Although arrays and slices store items of the same type there is no limitation in practice. This is because the type used could be an interface. So we could store items of any types provided that they all met the specified interface.

5.2.1 Creating Slices

Slices are created using the syntaxes:

```
make([]Type, length, capacity)
make([]Type, length)
[]Type{}
[]Type{value1, value2, ... , valueN}
```

The built-in *make()* function is used to create slices, maps, and channels. When used to create a slice it creates a **hidden zero-value initialized array and returns a slice reference** that refers to the hidden array.
The hidden array, like all arrays in Go, is of fixed length, with the length being the slice's **capacity** if the first syntax is used, or the slice's length if the second syntax is used, or the number of items in braces if the composite literal (third and fourth) syntax is used.
The composite literal (fourth) syntax is very convenient, since it allows us to create a slice with some **initial values**. The syntax *()Type* is equivalent to *make()Type, 0)* ; both create an **empty slice**. This isn't useless since we can use the built-in *append()* function to effectively increase a slice's capacity.
Valid index positions for a slice range from 0 to *len(slice)-1* . A slice can be resliced to reduce its length, and if a slice's capacity is greater than its length the slice can be resliced to increase its length up to its capacity. We can also increase a slice's capacity using the built-in *append()* function.

5.2.2 Examples

If you want to create a slice you should use the built-in *make* function:

```
x := make([]float64, 5)
```

This creates a slice that is associated with an underlying *float64* array of length 5.
The make function also allows a **3rd parameter**:

```
x := make([]float64, 5, 10)
```

10 represents the **capacity** of the underlying array which the slice points to.
Another way to create slices is to use the *(low : high)* expression:

```
arr := []float64{1,2,3,4,5}
x := arr[0:5]
```

low is the index of where to start the slice and *high* is the index where to end it (but **not including the index itself**). For example while arr(0:5) returns (1,2,3,4,5) , arr(1:4) returns (2,3,4).
For convenience we are also allowed to **omit low , high or even both low and high** . arr(0:) is the same as arr(0:len(arr)) , arr(:5) is the same as

arr(0:5) and arr(:) is the same as arr(0:len(arr)) .

5.2.3 Slices and Their Hidden Arrays

In this example:

```
s := []string{"A", "B", "C", "D", "E", "F", "G"}
t := s[:5]
// [A B C D E]
u := s[3 : len(s)-1] // [D E F]
fmt.Println(s, t, u)
u[1] = "x"
fmt.Println(s, t, u)
```

The result is:

```
[A B C D E F G] [A B C D E] [D E F]
[A B C D x F G] [A B C D x] [D x F]
```

Since the slices s , t , and u all refer to the same underlying data, a change to one will affect any of the others that refer to the same data.

5.2.4 Slices Functions

Go includes two built-in functions to assist with slices: *append* and *copy*. Here is an example of *append* :

```
func main() {
  slice1 := []int{1,2,3}
  slice2 := append(slice1, 4, 5)
  fmt.Println(slice1, slice2)
}
```

After running this program *slice1* has *(1,2,3)* and *slice2* has *(1,2,3,4,5)*. *append* creates a new slice by taking an existing slice (the first argument) and appending all the following arguments to it.
Example of *copy*:

```
func main() {
  slice1 := []int{1,2,3}
  slice2 := make([]int, 2)
  copy(slice2, slice1)
  fmt.Println(slice1, slice2)
}
```

After running this program slice1 has (1,2,3) and slice2 has (1,2)
The contents of slice1 are copied into slice2 , but since slice2 has room for only two elements only the first two elements of slice1 are copied.

5.2.5 Slice Operations

Syntax	Description
s(n)	The item at index position n in slice s
s(n:m)	A slice taken from slice s from index positions n to m-1
s(n:)	A slice taken from slice s from index positions n to len(s)-1
s(:m)	A slice taken from slice s from index positions 0 to m-1
s(:)	A slice taken from slice s from index positions 0 to len(s)-1
cap(s)	The capacity of slice s; always $\geq$ len(s)
len(s)	The number of items in slice s; always $\leq$ cap(s)
s = s(:cap(s))	Increase slice s's length to its capacity if they are different

5.2.6 Indexing and Slicing Slices

A slice is a reference to a hidden array and slices of slices are also references to the same hidden array. Here is an example to illustrate what this means.

```
s := []string{"A", "B", "C", "D", "E", "F", "G"}
t := s[2:6]
fmt.Println(t, s, "=", s[:4], "+", s[4:])
s[3] = "x"
t[len(t)-1] = "y"
fmt.Println(t, s, "=", s[:4], "+", s[4:])
```

Result:

```
[C D E F] [A B C D E F G] = [A B C D] + [E F G]
[C x E y] [A B C x E y G] = [A B C x] + [E y G]
```

When we change the data, whether via the original *s* slice or from the *t* slice of the *s* slice, the same underlying data is changed, so both slices are affected.

5.2.7 Iterating Slices

If we want to access the items **without modifying** them we can use a *for ... range* loop; and if we need to **modify** items we can use a *for* loop with a loop counter.

```
amounts := []float64{237.81, 261.87, 273.93, 279.99, 281.07,
    303.17,
231.47, 227.33, 209.23, 197.09}
sum := 0.0
for _, amount := range amounts {
sum += amount
}
fmt.Printf("Sum %.1f --> %.1f\n", amounts, sum)
```

Result:

```
Sum [237.8 261.9 273.9 280.0 281.1 303.2 231.5 227.3 209.2 197.1]
    --> 2503.0
```

Note The *for ... range* loop assigns a 0-based loop counter, which in this case we have discarded using the blank identifier (_) and a **copy** of the corresponding item from the slice. This means that any changes that are applied to the item **affect only the copy**, not the item in the slice. **If we want to modify** the items in the slice we must use a *for* loop that just provides valid slice indexes and not copies of the slice's items.

```
for i := range amounts {
amounts[i] *= 1.05
sum += amounts[i]
}
fmt.Printf("Sum %.1f --> %.1f\n", amounts, sum)
```

Result:

```
Sum [249.7 275.0 287.6 294.0 295.1 318.3 243.0 238.7 219.7 206.9]
    --> 2628.1
```

Here we have increased each item in the slice by 5% and accumulated their sum.

5.2.8 Modifying Slices

If we need to append to a slice we can use the built-in *append()* function. **This function takes the slice to be appended to and one or more individual items to append**. If we want to append a slice to a slice we must use the ... (ellipsis) operator to tell Go to pass the slice to be added as individual values. The values to append **must be of the same type as the slice's value type**. In the case of a string we can append its individual bytes to a byte slice by using the ellipsis syntax.

```
s := []string{"A", "B", "C", "D", "E", "F", "G"}
t := []string{"K", "L", "M", "N"}
u := []string{"m", "n", "o","p", "q", "r"}
s = append(s, "h", "i", "j") //Append individual values
s = append(s, t...)     //Append all of a slice values
s = append(s, u[2:5]...)    //Append a sub-slice
b := []byte{'U', 'V'}
letters := "wxy"
b = append(b, letters...) //Append a string's bytes to a byte slice
fmt.Printf("%v\n%s\n", s, b)
```

Result:

```
[A B C D E F G h i j K L M N o p q]
UVwxy
```

The built-in *append()* function takes a slice and one or more values and returns a (possibly new) slice which has the original slice's contents, plus the given value or values as its last item or items. If the original slice's capacity is sufficient for the new items *append()* puts the new value or values in the empty position or positions at the end and returns the original slice with its length increased by the number of items added. If the original slice doesn't have sufficient capacity, the *append()* function

creates a new slice under the hood and copies the original slice's items into it, plus the new value or values at the end, and returns the new slice.

5.2.9 Sorting Slices

The standard library's *sort* package provides functions for sorting slices of *int*, *float64*, and *string*, for checking if such a slice is sorted, and for searching for an item in a sorted slice using the fast binary search algorithm.
There are also generic *sort.Sort()* and *sort.Search()* functions that can easily be used with custom data.

Syntax	Description
sort.Float64s(fs)	Sorts fs of type []float64 into ascending order
sort.Float64sAreSorted(fs)	Returns true if fs of type []float64 is sorted
sort.Ints(is)	Sorts is of type []int into ascending order
sort.IntsAreSorted(is)	Returns true if is of type []int is sorted
sort.IsSorted(d)	Returns true if d of type sort.Interface is sorted
sort.Search(size, fn)	Returns the index position in a sorted slice in scope of length size where function fn with the signature func(int) bool returns true (see text)
sort.SearchFloat64s(fs, f)	Returns the index position of f of type float64 in sorted fs of type []float64
sort.SearchInts(is, i)	Returns the index position of i of type int in sorted is of type []int
sort.SearchStrings(ss, s)	Returns the index position of s of type string in sorted ss of type []string
sort.Sort(d)	Sorts d of type sort.Interface (see text)
sort.Strings(ss)	Sorts ss of type []string into ascending order
sort.StringsAreSorted(ss)	Returns true if ss of type []string is sorted

The standard library's *sort.Strings()* function takes a *[]string* and sorts the strings in-place in ascending order in terms of their underlying bytes. The *sort.Sort()* function can sort items of any type that provide the methods in the *sort.Interface* , that is, items of a type that provide the *Len()* , *Less()* , and *Swap()* methods, each with the required signatures.
Example:

```
1 files := []string{"Test.conf", "util.go", "Makefile", "misc.go", "
      main.go"}
2 fmt.Printf("Unsorted:%q\n", files)
3 sort.Strings(files) // Standard library sort function
4 fmt.Printf("Underlying bytes: %q\n", files)
```

Result:

```
1 Unsorted:["Test.conf" "util.go" "Makefile" "misc.go" "main.go"]
2 Underlying bytes: ["Makefile" "Test.conf" "main.go" "misc.go" "util
      .go"]
```

5.2.10 Searching Slices

Go provides a *sort.Search()* method which uses the **binary search algorithm**. This requires the comparison of only $log_2(n)$ items (where n is the number of items) each time.

```
files := []string{"Test.conf", "util.go", "Makefile", "misc.go", "
    main.go"}
sort.Strings(files)
fmt.Printf("%q\n", files)
i := sort.Search(len(files),
  func(i int) bool { return files[i] >= target })
if i < len(files) && files[i] == target {
  fmt.Printf("found \"%s\" at files[%d]\n", files[i], i)
}
```

Result:

```
["Makefile" "Test.conf" "main.go" "misc.go" "util.go"]
found "Makefile" at files[0]
```

The *sort.Search()* function takes two arguments:

- the **length** of the slice to work
- a **function** that compares an item in a **sorted slice** with a target item using the >= operator for slices that are sorted in ascending order or the <= operator for slices sorted in descending order. The function must be a closure, that is, it must be created in the scope of the slice it is to work on since it must capture the slice as part of its state.

The *sort.Search()* function returns an *int* ; only if this is less than the length of the slice and the item at that index position matches the target, can we be sure that we have found the item we are looking for.

5.3 Maps

Although slices can account for most data structure use cases, in some situations we need to be able to store key-value pairs with fast lookup by key. This functionality is provided by Go's **map type**. A map is **an unordered collection of key-value pairs**.
Also known as an associative array, a hash table or a dictionary, maps are used to look up a value by its associated key.
A map's keys must all be of the same type, and so must its values- although the key and value types can (and often do) differ.
A Go map is an unordered collection of key-value pairs whose capacity is limited only by machine memory.
Keys are unique and may only be of a type that sensibly supports the == and != operators; so most of the built-in types can be used as keys (e.g.,

int , *float64* , *rune* , *string* , comparable arrays and struct s, and custom types based on these, as well as pointers).
With respect to a map's values, just as with the items in a slice, **there is no limitation in practice**. This is because the value type used could be an interface. So we could store values of any types provided that they all met the specified interface (i.e., had the method or methods that the interface requires).

5.3.1 Maps Operations

Go's map operations are listed below:

Syntax	Description
m(k) = v	Assigns value *v* to map *m* under key *k* ; if *k* is already in the map its previous value is discarded
delete(m, k)	Deletes key *k* and its associated value from map *m* , or safely does nothing
v := m(k)	Retrieves the value that corresponds to map *m* 's key *k* and assigns it to *v* ; or assigns the zero value for the value's type to *v* , if *k* isn't in the map
v, found := m(k)	Retrieves the value that corresponds to map *m* 's key *k* and assigns it to *v* and true to found ; or assigns the zero value for the value's type to *v* and false to found , if *k* isn't in the map
len(m)	The number of items (key-value pairs) in map *m*

Maps are reference types that are cheap to pass no matter how much data they hold and Map lookups are fast.

5.3.2 Creating Maps

Maps are created using the syntaxes:

```
1 make(map[KeyType]ValueType, initialCapacity)
2 make(map[KeyType]ValueType)
3 map[KeyType]ValueType{}
4 map[KeyType]ValueType{key1: value1, key2: value2, ..., keyN: valueN
      }
```

The built-in *make()* function is used to create slices, maps, and channels. When used to create a map it creates an **empty map**, and if the optional *initialCapacity* is specified, the map is initialized to have enough space for that number of items. If more items are added to the map than the **initial capacity** allows for, the map **will automatically grow** to accommodate the new items.
The last two syntaxes show how to create a map using the composite literal syntax; this is very convenient in practice, either to create a new empty map, or to create a map with some initial values. Examples:

```
massForPlanet := make(map[string]float64) // Same as: map[string]
    float64{}
massForPlanet["Mercury"] = 0.06
massForPlanet["Venus"] = 0.82
massForPlanet["Earth"] = 1.00
massForPlanet["Mars"] = 0.11
fmt.Println(massForPlanet)
```

Result:

```
map[Venus:0.82 Mars:0.11 Earth:1 Mercury:0.06]
```

For small maps it doesn't really matter whether we specify their initial capacity, but for large maps doing so can improve performance. In general **it is best to specify the initial capacity if it is known** (even if only approximately).
Maps use the *()* index operator just like arrays and slices, only for maps the index inside the square brackets is of the map's key type which might not be an *int*
Remember pointers and struct can be used as map keys too.
When a ***for ... range* loop is applied to a map** and there are two variables present, the loop returns a key and a value on each iteration until every key-value item has been returned or the loop is broken out of. If just one variable is present only the key is returned on each iteration. Since maps are unordered we cannot know what particular sequence the items will come in. Example:

```
populationForCity := map[string]int{"Istanbul": 12610000,
"Karachi": 10620000, "Mumbai": 12690000, "Shanghai": 13680000}
//Note the for range loop with two variables
for city, population := range populationForCity {
  fmt.Printf("%-10s %8d\n", city, population)
}
```

5.3.3 Map Lookups

Go provides **two very similar syntaxes** for map lookups, both of which use the *()* index operator. Example of the simplest syntax:

```
population := populationForCity["Mumbai"]
fmt.Println("Mumbai's population is", population)
population = populationForCity["Emerald City"]
fmt.Println("Emerald City's population is", population)
```

If we look up a key that is present in the map the corresponding value is returned. But if the key is not present then the map's value type's zero value is returned. So, in this example, we cannot tell whether the 0 returned for the "Emerald City" key means that the population of Emerald City really is zero, or that the city isn't in the map. Go's second map lookup syntax provides the solution to this problem.

```
city := "Istanbul"
```

```
if population, found := populationForCity[city]; found {
fmt.Printf("%s's population is %d\n", city, population)
} else {
fmt.Printf("%s's population data is unavailable\n", city)
}
city = "Emerald City"
_, present := populationForCity[city]
fmt.Printf("%q is in the map == %t\n", city, present)
```

If we provide two variables for the map's *()* index operator to return to, the first will get the value that corresponds to the key (or the map's value type's zero value if the key isn't present), and the second will get true (or false if the key isn't present). This allows us **to check for a key's existence in the map**.

5.3.4 Modifying Maps

Items, that is, key-value pairs, can be inserted into maps and deleted from maps. And any given key's value can be changed. Example:

```
fmt.Println(len(populationForCity), populationForCity)
delete(populationForCity, "Shanghai") // Delete
fmt.Println(len(populationForCity), populationForCity)
populationForCity["Karachi"] = 11620000 // Update
fmt.Println(len(populationForCity), populationForCity)
populationForCity["Beijing"] = 11290000 // Insert
fmt.Println(len(populationForCity), populationForCity)
```

Result:

```
4 map[Shanghai:13680000 Mumbai:12690000 Istanbul:12610000 Karachi
    :10620000]
3 map[Mumbai:12690000 Istanbul:12610000 Karachi:10620000]
3 map[Mumbai:12690000 Istanbul:12610000 Karachi:11620000]
4 map[Mumbai:12690000 Istanbul:12610000 Karachi:11620000 Beijing
    :11290000]
```

The syntax for inserting and updating map items is identical: If an item with the given key isn't present, a new item with the given key and value will be inserted; and if an item with the given key is present, its value will be set to the given value, and the original value will be discarded. And if we try to delete an item which isn't in the map, Go will safely do nothing.

5.3.5 Key-Ordered Map Iteration and Map Inversion

Here is an example that shows how to output the *populationForCity* map in **alphabetical order**:

```
cities := make([]string, 0, len(populationForCity))
for city := range populationForCity {
cities = append(cities, city)
}
sort.Strings(cities)
for _, city := range cities {
```

```
fmt.Printf("%-10s %8d\n", city, populationForCity[city])
}
```

Result:

```
Beijing    11290000
Istanbul     12610000
Karachi    11620000
Mumbai     12690000
```

We begin by creating a slice of type *()string* with zero length (i.e., empty), but with enough capacity to hold all of the map's keys. Then we iterate over the map retrieving only the keys (since we have used just one variable, city , rather than the two needed to retrieve each key-value pair), and appending each city in turn to the cities slice. Next, we sort the slice, and then we iterate over the slice (ignoring the int index by using the blank identifier), looking up the corresponding city's population at each iteration.
We can easily **invert a map** whose values are unique and whose type is acceptable for use as map keys. Example:

```
cityForPopulation := make(map[int]string, len(populationForCity))
for city, population := range populationForCity {
cityForPopulation[population] = city
}
fmt.Println(cityForPopulation)
```

Result:

```
map[12610000:Istanbul 11290000:Beijing 12690000:Mumbai 11620000:
    Karachi]
```

5.3.6 Other Examples with Maps

Example of a map in Go:

```
  var x map[string]int
```

The map type is represented by the keyword *map* , followed by the key type in brackets and finally the value type. Like arrays and slices maps can be **accessed using brackets**.
Maps have to be initialized before they can be used.
We can also create maps with a **key type of *int*** :

```
  x := make(map[int]int)
  x[1] = 10
  fmt.Println(x[1])
```

This looks very much like an array but there are a few **differences**. First the **length** of a map (found by doing len(x)) **can change** as we add new items to it. Second **maps are not sequential**.
We can also **delete items from a map** using the built-in *delete* function:

```
delete(x, 1)
```

Accessing an element of a map can return **two values instead of just one**. **The first value is the result of the lookup, the second tells us whether or not the lookup was successful**.
This can be used to check if a key does not exist:

```
name, ok := elements["Un"]
fmt.Println(name, ok)
```

If the key "Un" does not exist in the map *name* is going to be an empty string and *ok* is going to be *false*.
Maps are also often used to store general information. For example instead of just storing the name of the element we store its standard state (state at room temperature) as well:

```
func main() {
  elements := map[string]map[string]string{
    "H": map[string]string{
      "name":"Hydrogen",
      "state":"gas",
    },
    "He": map[string]string{
      "name":"Helium",
      "state":"gas",
    },
    "Li": map[string]string{
      "name":"Lithium",
      "state":"solid",
    },
    "Be": map[string]string{
      "name":"Beryllium",
      "state":"solid",
    },
    "B": map[string]string{
      "name":"Boron",
      "state":"solid",
    },
    "C": map[string]string{
      "name":"Carbon",
      "state":"solid",
    },
    "N": map[string]string{
      "name":"Nitrogen",
      "state":"gas",
    },
    "O": map[string]string{
      "name":"Oxygen",
      "state":"gas",
    },
    "F": map[string]string{
      "name":"Fluorine",
      "state":"gas",
    },
    "Ne": map[string]string{
      "name":"Neon",
      "state":"gas",
    },
  }
```

```
44    if el, ok := elements["Li"]; ok {
45      fmt.Println(el["name"], el["state"])
46    }
47  }
```

We now have a map of strings to maps of strings to strings.
Although maps are often used like this, there is a better way to store structured information.

Chapter 6

Functions

6.1 Basic Concepts

A function is an independent section of code that maps zero or more **input parameters** to zero or more **output parameters**.
Functions start with the **keyword** *func* , followed by the **function's name**.
The parameters (inputs) of the function are defined like this: name type, name type, ...
Collectively the parameters and the return type are known as the **function's signature**.
Finally we have the **function body** which is a series of statements between curly braces.
Example of a function:

```
1 func average(xs []float64) float64 {
2   total := 0.0
3   for _, v := range xs {
4     total += v
5   }
6   return total / float64(len(xs))
7 }
```

The *return* statement causes the function to immediately stop and return the value after it to the function that called this one.
Some things to remember:

- Functions don't have access to anything in the calling function.
- We can also name the return type

```
1 func f2() (r int) {
2   r = 1
3   return
4 }
```

6.2 Returning Multiple Values

Go is also capable of **returning multiple values from a function**:

```
1  func f() (int, int) {
2    return 5, 6
3  }
4  func main() {
5    x, y := f()
6  }
```

Three changes are necessary:

- change the return type to contain multiple types separated by , ,
- change the expression after the return so that it contains multiple expressions separated by ,
- and finally change the assignment statement so that multiple values are on the left side of the := or = .

Multiple values are often used to return an error value along with the result (*x, err := f()*), or a boolean to indicate success (*x, ok := f()*).

6.3 Variadic Functions

There is a special form available for the last parameter in a Go function:

```
1  func add(args ...int) int {
2    total := 0
3    for _, v := range args {
4      total += v
5    }
6    return total
7  }
8  func main() {
9    fmt.Println(add(1,2,3))
10 }
```

By using *...* before the type name of the last parameter you can indicate that **it takes zero or more of those parameters**. In this case we take zero or more *int*s.
We can also **pass a slice** of int s by following the slice with *...* :

```
1  func main() {
2    xs := []int{1,2,3}
3    fmt.Println(add(xs...))
4  }
```

6.4 Closure

It is possible to create **functions inside of functions**:

```
func main() {
  add := func(x, y int) int {
  return x + y
}
fmt.Println(add(1,1))
}
```

add is a local variable that has the type *func(int, int) int* (a function that takes two int s and returns an int).
When you create a local function like this it also has access to other local variables

```
func main() {
  x := 0
  increment := func() int {
    x++
    return x
  }
  fmt.Println(increment())
  fmt.Println(increment())
}
```

increment adds 1 to the variable *x* which is defined in the main function's scope. This *x* variable can be accessed and modified by the increment function. This is why the first time we call increment we see 1 displayed, but the second time we call it we see 2 displayed.
A function like this together with the non-local variables it references **is known as a closure**. In this case increment and the variable x form the closure.
Another example:

```
func makeEvenGenerator() func() uint {
  i := uint(0)
  return func() (ret uint) {
    ret = i
    i += 2
    return
    }
}
func main() {
  nextEven := makeEvenGenerator()
  fmt.Println(nextEven()) // 0
  fmt.Println(nextEven()) // 2
  fmt.Println(nextEven()) // 4
}
```

makeEvenGenerator returns a function which generates even numbers. Each time it's called it adds 2 to the local i variable which , unlike normal local variables, **persists between calls**.

6.5 Recursion

Finally a function is able to **call itself**.
Here is one way to compute the factorial of a number:

```
func factorial(x uint) uint {
  if x == 0 {
  return 1
  }
  return x * factorial(x-1)
}
```

6.6 Defer, Panic and Recover

Go has a special statement called *defer* which schedules a **function call to be run after the function completes**. Consider the following example:

```
package main
import "fmt"
func first() {
  fmt.Println("1st")
}
func second() {
  fmt.Println("2nd")
}
func main() {
  defer second()
  first()
}
```

This program prints 1st followed by 2nd .
defer is often used when resources need to be freed in some way.
We can **handle a run-time *panic*** with the built-in *recover* function. recover stops the *panic* and returns the value that was passed to the call to *panic* .
This is how to use it:

```
package main
import "fmt"
func main() {
  defer func() {
  str := recover()
  fmt.Println(str)
  }()
  panic("PANIC")
}
```

Chapter 7

Pointers

In this program the zero function will not modify the original *x* variable in the main function.

```
1 func zero(x int) {
2   x = 0
3 }
4 func main() {
5   x := 5
6   zero(x)
7   fmt.Println(x) // x is still 5
8 }
```

Pointers **reference a location in memory** where a value is stored rather than the value itself.

```
1 func zero(xPtr *int) {
2   *xPtr = 0
3 }
4 func main() {
5   x := 5
6   zero(&x)
7   fmt.Println(x) // x is 0
8 }
```

7.1 The * and the & operators

In Go a pointer is represented using the * (asterisk) character followed by the type of the stored value. For example in the zero function *xPtr* is a pointer to an *int* .
* is also used to "**dereference**" **pointer variables**. Dereferencing a pointer gives us access to the value the pointer points to (so **xPtr = 0*, means write 0 in the value pointed by xPtr).
If we try *xPtr = 0* instead we will get a compiler error because *xPtr* is not an *int* it's a **int* , which can only be given another **int* .

Finally we use the *&* operator **to find the address of a variable**. *&x* returns a **int* (pointer to an int) because x is an int .

7.2 More on Pointers

A pointer is a **variable that holds another variable's memory address**. Pointers are created to point to variables of a particular type; this ensures that Go knows how large (i.e., how many bytes) the pointed-to value occupies.
Some things to remember about pointers:

- A variable pointed to by a pointer can be modified through the pointer.
- Pointers are cheap to pass (8 bytes on 64-bit machines, 4 bytes on 32-bit machines), regardless of the size of the value they point to.
- Pointed to variables persist in memory for as long as there is at least one pointer pointing to them, so their lifetime is independent of the scope in which they were created.

7.2.1 & Operator, the Address of Operator

In Go the & operator is overloaded. When used as a binary operator it performs a bitwise AND. When used as a unary operator **it returns the memory address of its operand**. The unary & is sometimes called the address of operator.

7.2.2 * Operator, the Contents of Operator

The * operator is also overloaded. It multiplies its operands when used as a binary operator. And when used as a unary operator it **provides access to the value pointed to by the variable it is applied to**.
The unary * is sometimes called the **contents of** operator or the **indirection** operator or the **dereference** operator.
In addition to being the multiplication and dereferencing operator, the * operator is also overloaded for a third purpose: as a **type modifier**. When an * is placed on the left of a type name it changes the meaning of the name from specifying a value of the given type to specifying a pointer to a value of the given type.

7.3 new

Another way to get a pointer is to **use the built-in *new* function**:

```
func one(xPtr *int) {
  *xPtr = 1
}
func main() {
  xPtr := new(int)
  one(xPtr)
  fmt.Println(*xPtr) // x is 1
}
```

new takes **a type as an argument**, allocates enough memory to fit a value of that type and **returns a pointer to it**.
Go is a **garbage collected programming language** which means memory is cleaned up automatically when nothing refers to it anymore.

7.4 More about new()

Go provides **two syntaxes** for creating variables and at the same time acquiring pointers to them:

- one using the built-in *new()* function
- the other using the *address of operator*

Example:

```
type composer struct {
  name string
  birthYear int
}
antonio := composer{"Antonio Teixeira", 1707} // composer value
agnes := new(composer) // pointer to composer
agnes.name, agnes.birthYear = "Agnes Zimmermann", 1845
julia := &composer{} // pointer to composer, alternative syntax
julia.name, julia.birthYear = "Julia Ward Howe", 1819
augusta := &composer{"Augusta HolmÃ¨s", 1847} // pointer to
    composer
fmt.Println(antonio)
fmt.Println(agnes, augusta, julia)

{Antonio Teixeira 1707}
&{Agnes Zimmermann 1845} &{Augusta HolmÃ¨s 1847} &{Julia Ward Howe
    1819}
```

When Go prints pointers to struct *s* it prints the dereferenced struct but prefixed with the *&* address of operator to indicate that it is a pointer.
Note the equivalence when the type is one that can be initialized using braces:

new(Type) is equivalent to *&Type{}*

Both these syntaxes allocate a new zeroed value of the given *Type* and return a pointer to the value. We don't have to worry about the value's lifetime or ever delete it, since Go's **memory management system takes care** of all that for us.

7.5 Reference Types

Go has reference types.
A variable of a reference type refers to a **hidden value in memory that stores the actual data**. Variables holding reference types are cheap to pass and are used with the same syntax as a value (i.e., we don't need to take a reference type's address or dereference it to access the value it refers to).
Once we reach the stage where we need to return more than four or five values from a function or method, it is best to pass a slice if the values are homogeneous, or to use a pointer to a struct if they are heterogeneous.
Maps and slices are reference types, and any changes made to a map or to a slice's items are visible to all the variables that refer to them. Example:

```
1  func inflate(numbers []int, factor int) {
2    for i := range numbers {
3    numbers[i] *= factor
4    }
5  }
6  //The function inflate changes the values of the slice passed as
       argument
7  grades := []int{87, 55, 43, 71, 60, 43, 32, 19, 63}
8  inflate(grades, 3)
9  fmt.Println(grades)
10
11 [261 165 129 213 180 129 96 57 189]
```

The grades slice is passed in as the parameter numbers, but unlike when we pass values, any changes applied to numbers are reflected in grades since they both refer to the same underlying slice.
Another example:

```
1  type rectangle struct {
2    x0, y0, x1, y1 int
3    fill color.RGBA
4  }
5  func resizeRect(rect *rectangle, width, height int) {
6    (*rect).x1 += width // Ugly explicit dereference
7    rect.y1 += height // . automatically dereferences structs
8  }
9
10 rect := rectangle{4, 8, 20, 10, color.RGBA{0xFF, 0, 0, 0xFF}}
11 fmt.Println(rect)
12 resizeRect(&rect, 5, 5)
13 fmt.Println(rect)
14
15 {4 8 20 10 {255 0 0 255}}
16 {4 8 25 15 {255 0 0 255}}
```

In the *resizeRect* function there are two ways to dereference a struct. In the second case relying on Go to do the dereferencing for us. This

works because Go's . (dot) selector operator **automatically dereferences pointers to structs**.
Certain types in Go are reference types: maps, slices, channels, functions, and methods. Unlike with pointers, there is no special syntax for reference types since they are used just like values.

7.6 Variables Holding Functions

If we declare **a variable to hold a function**, the variable actually gets **a reference to the function**. Function references **know the signature** of the function they refer to, so it is not possible to pass a reference to a function that doesn't have the right signature

Chapter 8

Structs and Interfaces

8.1 Structs

A struct is a type which contains named fields.
Example:

```
type Circle struct {
  x float64
  y float64
  r float64
}
```

The *type* keyword introduces a new type. It's followed by the name of the type (*Circle*), the keyword *struct* to indicate that we are defining a *struct* type and a list of fields inside of curly braces. Like with functions we can collapse fields that have the same type:

```
type Circle struct {
  x, y, r float64
}
```

8.1.1 Initialization

We can **create an instance** of our new Circle type in a variety of ways:

```
var c Circle
```

Like with other data types, this will create a local *Circle* variable that is by default set to zero. For a struct **zero means** each of the fields is set to their corresponding zero value (0 for *int* s, 0.0 for *float* s, "" for *string* s, nil for pointers, ...).
We can also use the *new* function:

```
c:= new(Circle)
```

This allocates memory for all the fields, sets each of them to their zero value and returns a pointer.(*Circle)
More often we want to **give each of the fields a value**. We can do this in two ways. Like this:

```
c := Circle{x: 0, y: 0, r: 5}
//or
c := Circle{0, 0, 5}
```

8.1.2 Fields

We can access fields using the . operator:

```
fmt.Println(c.x, c.y, c.r)
c.x = 10
c.y = 5
```

One thing to remember is that **arguments are always copied in Go**. It means if a *struct* is passed to a function the function will not modify the value of the fields of the *struct* passed. If we want to do that, then we have to pass a pointer to the *struct*.

8.2 Methods

Methods can be considered a **special type of function**. Example:

```
func (c *Circle) area() float64 {
  return math.Pi * c.r*c.r
}
```

In between the keyword *func* and the name of the function we've added a "**receiver**". The receiver is like a parameter, it has a name and a type, but by creating the function in this way **it allows us to call the function using the . operator**:

```
fmt.Println(c.area())
```

8.2.1 Embedded Types

A struct's fields usually represent **the has-a relationship**. For example a Circle has a radius.
In other instances we may want to model a is-a relationship instead. Go supports the **is-a relationships by using an embedded types**. Example:

```
type Person struct {
  Name string
}
func (p *Person) Talk() {
  fmt.Println("Hi, my name is", p.Name)
}
```

```
type Android struct {
  Person
  Model string
}
```

We use the type (*Person*) and **don't give it a name**. When defined this way the *Person* struct can be accessed using the type name:

```
a := new(Android)
a.Person.Talk()
```

But we can also call any *Person* methods directly on the *Android* :

```
a := new(Android)
a.Talk()
```

8.3 Interfaces

Interfaces are another type available in Go. For Example

```
type Shape interface {
  area() float64
  }
```

Like a struct an interface is created using the *type* keyword, followed by a name and the keyword *interface* . But instead of defining fields, we define a "**method set**". A method set is a list of methods that a type must have in order to **implement the interface**.
The interesting thing is we can use interface types as **arguments to functions**:

```
func totalArea(shapes ...Shape) float64 {
  var area float64
  for _, s := range shapes {
    area += s.area()
  }
  return area
}
```

We would call this function like this:

```
fmt.Println(totalArea(&c, &r))
```

where *c* and *r* are two struct representing a circle and a rectangle.
Interfaces can also be used as **fields**:

```
type MultiShape struct {
  shapes []Shape
}
```

Chapter 9

Concurrency

Go has rich support for concurrency using **goroutines** and **channels**.

9.1 Goroutines

A **goroutine** is a function that is **capable of running concurrently with other functions**. To create a goroutine we use the keyword *go* followed by a function invocation:

```
1  package main
2  import "fmt"
3  func f(n int) {
4    for i := 0; i < 10; i++ {
5    fmt.Println(n, ":", i)
6    }
7  }
8
9  func main() {
10   go f(0)
11   var input string
12   fmt.Scanln(&input)
13 }
```

This program consists of two goroutines. The first goroutine is implicit and is the main function itself. The second goroutine is created when we call *go f(0)* . Normally when we invoke a function our program will execute all the statements in a function and then return to the next line following the invocation. With a goroutine **we return immediately to the next line and don't wait for the function to complete**.
Goroutines are lightweight and we can easily create thousands of them. We can modify our program to run 10 goroutines by doing this:

```
1  func main() {
2    for i := 0; i < 10; i++ {
3      go f(i)
4    }
```

```
  var input string
  fmt.Scanln(&input)
}
```

9.2 Channels

Channels **provide a way for two goroutines to communicate with one another and synchronize their execution**. Example:

```
package main
import (
  "fmt"
  "time"
)

func pinger(c chan string) {
  for i := 0; ; i++ {
    c <- "ping"
  }
}

func printer(c chan string) {
  for {
    msg := <- c
    fmt.Println(msg)
    time.Sleep(time.Second * 1)
  }
}

func main() {

var c chan string = make(chan string)
go pinger(c)
go printer(c)
var input string
fmt.Scanln(&input)
}
```

This program will print "ping" forever (hit enter to stop it). A **channel type** is represented with the keyword *chan* followed by the type of the things that are passed on the channel (in this case we are passing strings). The <- (left arrow) operator is used to **send and receive messages** on the channel. *c <- "ping"* means send "ping" . *msg := <- c* means receive a message and store it in *msg* .
Using a channel like this **synchronizes the two goroutines**. When *pinger* attempts to send a message on the channel it will wait until *printer* is ready to receive the message. (this is known as blocking).

9.2.1 Channel Direction

We can **specify a direction on a channel type** thus restricting it to either sending or receiving. For example pinger's function signature can be changed to this:

```
func pinger(c chan<- string)
```

Now c can only be sent to. Similarly we can change printer to this:

```
func printer(c <-chan string)
```

9.2.2 Select

Go has a special statement called *select* which works like a *switch* but for channels:

```
func main() {
  c1 := make(chan string)
  c2 := make(chan string)
  go func() {
    for {
      c1 <- "from 1"
      time.Sleep(time.Second * 2)
    }
  }()

  go func() {
    for {
      c2 <- "from 2"
      time.Sleep(time.Second * 3)
    }
  }()
  go func() {
    for {
      select {
        case msg1 := <- c1:
          fmt.  Println(msg1)
        case msg2 := <- c2:
          fmt.Println(msg2)
      }
    }
  }()

var input string
fmt.Scanln(&input)
}
```

This program prints "from 1" every 2 seconds and "from 2" every 3 seconds. *select* **picks the first channel that is ready** and receives from it (or sends to it).
If more than one of the channels are ready then it randomly picks which one to receive from. If none of the channels are ready, the statement blocks until one becomes available.
The select statement is often used to **implement a timeout**:

```
select {
case msg1 := <- c1:
  fmt.Println("Message 1", msg1)
case msg2 := <- c2:
  fmt.Println("Message 2", msg2)
case <- time.After(time.Second):
  fmt.Println("timeout")
}
```

time.After creates a channel and after the given duration will send the current time on it.
We can also specify a *default* case:

```
select {
case msg1 := <- c1:
  fmt.Println("Message 1", msg1)
case msg2 := <- c2:
  fmt.Println("Message 2", msg2)
case <- time.After(time.Second):
  fmt.Println("timeout")
default:
  fmt.Println("nothing ready")
}
```

The *default* case **happens immediately** if none of the channels are ready.

Chapter 10

Packages

Go also provides another mechanism for code reuse: **packages**.

10.1 Creating Packages

Packages only really make sense in the context of a separate program which uses them.
See example in the book.
You may have noticed that every function in the packages we've seen start with a capital letter. In Go if something **starts with a capital letter** that means other packages (and programs) are able to see it. If we had named the function average instead of Average our main program would not have been able to see it.

10.2 Documentation

Go has the ability to automatically generate documentation for packages we write in a similar way to the standard package documentation. In a terminal run this command:

```
1 godoc golang-book/chapter11/math Average
```

This documentation is also available in **web form** by running this command:

```
1 godoc -http=":6060"
```

and entering this URL into your browser:

```
1 http://localhost:6060/pkg/
```

Chapter 11

Testing

Go includes a **special program** that makes writing tests easier. Example:

```
1 package math
2 import "testing"
3   func TestAverage(t *testing.T) {
4     var v float64
5     v = Average([]float64{1,2})
6     if v != 1.5 {
7     t.Error("Expected 1.5, got ", v)
8   }
9 }
```

Now running this command

```
1 go test
```

we get something like this:

```
1 $ go test
2 PASS
3 ok golang-book/chapter11/math 0.032s
```

The *go test* command will look for any tests in any of the files in the current folder and run them. Tests are identified by starting a function with the word *Test* and taking one argument of type **testing.T* . In our case since we're testing the *Average* function we name the test function *TestAverage* .
Once we have the testing function setup we write tests that use the code we're testing.

```
1 package math
2
3 import "testing"
4
5 type testpair struct {
6   values []float64
7   average float64
8 }
9
```

```
10  var tests = []testpair{
11    { []float64{1,2}, 1.5 },
12    { []float64{1,1,1,1,1,1}, 1 },
13    { []float64{-1,1}, 0 },
14  }
15
16  func TestAverage(t *testing.T) {
17    for _, pair := range tests {
18    v := Average(pair.values)
19    if v != pair.average {
20    t.Error(
21      "For", pair.values,
22      "expected", pair.average,
23    "got", v,
24    )
25  }
26  }
27  }
```

This is a **very common way** to setup tests; we create a struct to represent the inputs and outputs for the function. Then we create a list of these struct s (pairs). Then we loop through each one and run the function.

Chapter 12

Core Packages

12.1 Strings

Go includes a large number of functions to work with strings in the *strings* package:

```
package main
import (
"fmt"
"strings"
)
func main() {
  fmt.Println(
  // true
  strings.Contains("test", "es"),
  // 2
  strings.Count("test", "t"),
  // true
  strings.HasPrefix("test", "te"),
  // true
  strings.HasSuffix("test", "st"),
  // 1
  strings.Index("test", "e"),
  // "a-b"
  strings.Join([]string{"a","b"}, "-"),
  // == "aaaaa"
  strings.Repeat("a", 5),
  // "bbaa"
  strings.Replace("aaaa", "a", "b", 2),
  // []string{"a","b","c","d","e"}
  strings.Split("a-b-c-d-e", "-"),
  // "test"
  strings.ToLower("TEST"),
  // "TEST"
  strings.ToUpper("test"),
  )
}
```

12.2 Files and Folders

To open a file in Go use the Open function from the *os* package. Example, read a file:

```
package main

import (
"fmt"
"io/ioutil"
)

func main() {
  bs, err := ioutil.ReadFile("test.txt")
    if err != nil {
    return
  }
  str := string(bs)
  fmt.Println(str)
}
```

Example, write a file:

```
package main

import (
"os"
)

func main() {
  file, err := os.Create("test.txt")
  if err != nil {
    // handle the error here
  retu  rn
  }
  defer file.Close()
  file.WriteString("test")
}
```

Example, read a directory:

```
package main

import (
"fmt"
"os"
)

func main() {
  dir, err := os.Open(".")
  if err != nil {
    return
  }
  defer dir.Close()
  fileInfos, err := dir.Readdir(-1)
  if err != nil {
    return
  }
```

```
18    for _, fi := range fileInfos {
19    fmt.Println(fi.Name())
20    }
21  }
```

12.3 Containers and Sort

In addition to lists and maps Go has **several more collections** available underneath the container package. Lists are one example.

12.3.1 List

The *container/list* package **implements a doubly-linked list**.
Each node of the list contains a **value** and a **pointer** to the next node. Since this is a doubly-linked list each node will also have pointers to the previous node. Example:

```
1   package main
2
3   import ("fmt" ; "container/list")
4
5   func main() {
6     var x list.List
7     x.PushBack(1)
8     x.PushBack(2)
9     x.PushBack(3)
10    for e := x.Front(); e != nil; e=e.Next() {
11      fmt.Println(e.Value.(int))
12    }
13  }
```

The zero value for a List is an empty list (a **List* can also be created using *list.New*). Values are appended to the list using *PushBack* .

12.3.2 Sort

The sort package contains functions for sorting arbitrary data. There are several **predefined sorting functions** (for slices of *ints* and *floats*).

Chapter 13

Commentary

Go provides C-style /* */ block comments and C++ style // line comments. Line comments are the norm; block comments appear mostly as package comments, but are useful within an expression or to disable large swaths of code.
The program - and web server - *godoc* processes Go source files to extract documentation about the contents of the package. Comments that appear before top-level declarations, with no intervening newlines, are extracted along with the declaration to serve as explanatory text for the item. The nature and style of these comments determines the quality of the documentation godoc produces.