Introduction to Functions

This lesson discusses the introductory concepts of functions in Go before going into detail.

WE'LL COVER THE FOLLOWING

- Basics of a function in Go
  - Purpose of a function
  - Execution of a function
  - Function call within a function call
  - Function overloading
  - Declaring function type

Functions are the basic building blocks of the Go code. They are very versatile, so Go can be said to have a lot of characteristics of a *functional language*. Functions are a kind of data because they are themselves values and have types. Let's start this chapter by elaborating on the elementary function-description, which we discussed briefly in Chapter 2.

Basics of a function in Go

Every program consists of several functions. It is the basic code block. The order in which the functions are written in the program does not matter. However, for readability, it is better to start with **main()** and write the functions in a *logical* order (for example, the calling order).

Purpose of a function

The main purpose of functions is to break a large problem, which requires breaking many code lines into a number of smaller tasks. Also, the same task can be invoked several times, so a function promotes code reuse. In fact, a good program honors the Don't Repeat Yourself principle, meaning that the code which performs a certain task may only appear once in the program.

Execution of a function

A function ends when it has executed its last statement before **}**, or when it executes a *return* statement, which can be with or without argument(s). These arguments are the values that a function returns from their computation. There are **3** types of functions in Go:

- Normal functions with an identifier
- Anonymous or lambda functions
- Methods

Any of these can have parameters and return values.

The definition of all the function parameters and return values together with their types is called the **function signature**. As a reminder, a syntax prerequisite like:

```
func g()

{ // INVALID
  ...
}
```

is invalid Go-code. It must be:

```
func g() { // VALID
  ...
}
```

A function is *called* or *invoked* in code in a general format like:

```
pack1.Function(arg1,arg2,...,argn)
```

`Function` is a function in package `pack1`, and `arg1`, and so on are the arguments. When a function is invoked, copies of the arguments are made, and these are then passed to the called function.

The invocation happens in the code of another function called the **calling function**. A function can call other functions as much as needed, and these functions, in turn, can call other functions. This can go on with theoretically no limit (unless the stack upon which these function calls are placed is no limit (unless the stack upon which these function calls are placed is exhausted).

Here is the simplest example of a function calling another function without needing arguments:

```
package main
import "fmt"

func main() { // main function started
    fmt.Println("In main before calling greeting")

    greeting() // greeting function invoked

    fmt.Println("In main after calling greeting") // executed after greeting function
} // main function ended

// greeting function declared
func greeting() {
    fmt.Println("In greeting: Hi!!!!!")
}
```

Calling Functions with no Parameters

As seen in the above code, the main function started on **line 4**. **Line 5** will be executed, and the message **In main before calling greeting** will be printed on the screen. At **line** 7, a function `greeting()` is invoked. Now, control will transfer to **line 13**, where the function `greeting` is created. Now, **line 14** will be executed, and the message **In greeting: Hi!!!** will be printed on the screen. From **line 15**, control will move to **line 9** right after the statement, where we call the function `greeting`. Then, the message **In main after calling greeting** will be printed on the screen.

Here is a slight variant of this example:

```
package main
import "fmt"

func main() {
    lastName := "John"
    fmt.Println("In main before calling greeting")
    greeting(lastName) // greeting function invoked
    fmt.Println("In main after calling greeting")
    fmt.Println("variable lastName is still: ", lastName)
}

// greeting function declared
```

```
func greeting(name string) {
    fmt.Println("In greeting: Hi!!!!!", name)
    name = "Johnny"

    fmt.Println("In greeting: Hi!!!!!", name)
}
```

Calling Functions with Parameters

As seen in the above code, the main function started on **line 4**. At **line 5**, variable `lastName` is declared and initialized with a value **John**. **Line 6** will be executed, and the message **In main before calling greeting** will be printed on the screen. At **line** 7, function `greeting(lastName)` is invoked.

Now control will transfer to **line 13**, where the function `greeting` is created. Now, **line 14** will be executed and message **In greeting: Hi!!! John** will be printed on the screen. At **line 15**, the `name` is reassigned a value of **Johnny**. Now, **line 16** will be executed, and the message **In greeting: Hi!!! Johnny** will be printed on the screen.

After **line 16**, control will move to **line 8**, right after the statement, where we call the function `greeting`. The message **In main after calling greeting** will be printed on the screen. Now, **line 9** is executed, and the message **variable lastName is still: John** will be printed because the value **Johnny** was assigned to the copy of `lastName` in `greeting` scope. After exiting from `greeting`, the variable `lastName` still has the value **John**.

Function call within a function call

A function call can have another function call as its argument, provided that the latter has the *same* number and types of arguments in the correct order that the first function needs. For example, suppose `f1` needs 3 `int` arguments:

```
f1(a, b, c int)
```

and `f2` returns 3 arguments:

```
f2(a, b int) (int, int, int)
```

then, this can be a call to `f1`:

then, this can be a call to `f1`:

```
f1(f2(a, b))
```

Let’s implement this concept in the code below:

```
package main
import "fmt"

func f1(a,b,c int)int{  // taking three parameters and returning their sum
  return a+b+c
}

func f2(a,b int)(int, int , int){ // taking two parameters and returning their sum, differenc
  n1 := a+b
  n2 := a-b
  n3 := a*b
  return n1,n2,n3
}

func main(){
  fmt.Print(f1(f2(20,10)))  // function call within a function call
}
```

Function Call within a Function Call

In the code above, in the `main` function at **line 16**, due to the function call, control will first transfer to the function `f2()` with `a` as **20** and `b` as **10**. The function `f2` will return three arguments, which are sum, difference, and the product of `a` and `b`. Now, control will transfer back to **line 16**. The values returned by `f2()` are now the arguments of `f1()`. The function `f1()` will gain control and return the sum of the values returned from `f2()`.

Function overloading

Coding two or more functions in a program with the same function name but a different parameter list and/or a different return-type(s) is called **function overloading**. It is not allowed in Go. It gives the compiler error: `<funcName>redeclared in this block, previous declaration at <lineno>`. The main reason for this is that overloading functions force the runtime to do additional type matching, which reduces performance. No overloading means only a simple function dispatch is needed. Therefore, you need to give your functions appropriate unique names, probably according to their signature.

functions appropriate unique names, probably according to their signature.

To declare a function implemented outside Go, such as an assembly routine, you simply give the name and signature with no body:

```go
func flushICache(begin, end uintptr) // implemented externally
```

Functions can also be used in the form of a declaration, as a function type like :

```go
type binOp func(int, int) int
```

Declaring function type

In that case, the body **{ }** is also omitted. Functions are first-class values. They can be assigned to a variable, like in:

```go
add := binOp
```

The variable `add` gets a reference (points) to the function, and it knows the signature of the function it refers to. It is not possible to assign a function to a variable with a different signature. Like variables, functions have a zero value, which is *nil*. Function values can be compared. They are equal if they refer to the same function or if both are nil. A function cannot be declared inside another function (no nesting), but this can be mimicked by using anonymous functions, which we will study later.

That's it about the introduction to functions. Now in the next lesson, we'll see how Golang, being a functional language, handles parameters and return values.