Go (Golang) Functions with Examples

author: Ramesh Fadatare

golang

In this tutorial, we will learn how to work functions in Golang with examples.

Go is an open-source programming language that makes it easy to build simple, reliable, and efficient software.

Go (Golang) Operators with Examples

Table of Contents

In this tutorial, we will learn the following topics with examples:

Let's get started with a defining simple function in Go.

Golang Function - Simple Example

In this example, we will show you how to create a simple function in **Go** with an example.

A function is a mapping of zero or more input parameters to zero or more output parameters. Functions in Go are created with the `func` keyword. We use the `return` keyword to return values from functions.

The following example creates a simple function in **Go**:

```
package main

import "fmt"

func main() {

    x := 4
    y := 5

    z := add(x, y)

    fmt.Printf("Output: %d\n", z)
}

func add(a int, b int) int {

    return a + b
}
```

Output:

```
Output: 9
```

In the above example, we define a function that adds two values.

Go - Function Multiple Return Values

Go functions allow to return of multiple values.

In the example, we have a `threerandom()` function, which returns three random values.

```
package main

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

func threerandom() (int, int, int) {

    rand.Seed(time.Now().UnixNano())
    x := rand.Intn(10)
    y := rand.Intn(10)
    z := rand.Intn(10)

    return x, y, z
}

func main() {

    r1, r2, r3 := threerandom()

    fmt.Println(r1, r2, r3)
}
```

Output:

```
0 8 7
```

Go - Anonymous Function

We create an anonymous function that adds three values. We pass three parameters to the function right after its definition.

```
package main

import "fmt"

func main() {

    sum := func(a, b, c int) int {
        return a + b + c
```

Output:

```
1+3+5 = 9
```

Go - variadic Function

A variadic function can accept a variable number of parameters. For instance, when we want to calculate the sum of values, we might have four, five, six, etc. values to pass to the function.

We use the **... (ellipses)** operator to define a variadic function.

In the example, we have a sum function that accepts a variable number of parameters.

```
package main

import "fmt"

func main() {

    s1 := sum(1, 2, 3)
    s2 := sum(1, 2, 3, 4)
    s3 := sum(1, 2, 3, 4, 5)

    fmt.Println(s1, s2, s3)
}

func sum(nums ...int) int {

    res := 0

    for _, n := range nums {
        res += n
    }

    return res
}
```

Output:

```
6 10 15
```

The `nums` variable is a slice, which contains all values passed to the sum function. We loop over the slice and calculate the sum of the parameters:

```
func sum(nums ...int) int {

    res := 0

    for _, n := range nums {
        res += n
    }

    return res
}
```

Go - Recursive Function

A recursive method calls itself to do its task. Recursion is a widely used approach to solve many programming tasks.

In this code example, we calculate the factorial of three numbers:

```
package main

import "fmt"

func fact(n int) int {

    if n == 0 || n == 1 {
        return 1
    }

    return n * fact(n-1)
}

func main() {

    fmt.Println(fact(7))
    fmt.Println(fact(10))
    fmt.Println(fact(15))
}
```

Output:

```
5040
3628800
1307674368000
```

Inside the body of the fact function, we call the fact function with a modified argument. The function calls itself:

```
func fact(n int) int {

    if n == 0 || n == 1 {
        return 1
    }

    return n * fact(n-1)
}
```

Go - defer Function Call

The defer statement defers the execution of a function until the surrounding function returns. The deferred call's arguments are evaluated immediately, but the function call is not executed until the surrounding function returns.

In the example, the sayHello function is called after the main function finishes.

```
package main

import "fmt"

func main() {

    fmt.Println("begin main")

    defer sayHello()

    fmt.Println("end main")
}

func sayHello() {

    fmt.Println("hello")
}
```

Output:

```
begin main
end main
hello
```

In the following example, an **integer** and a **User** structure are passed as parameters to functions.

```
package main

import "fmt"

type User struct {
    name       string
    occupation string
}

func main() {

    x := 10
    fmt.Printf("inside main %d\n", x)

    inc(x)

    fmt.Printf("inside main %d\n", x)

    fmt.Println("--------------------")

    u := User{"Raj", "Engineer"}
    fmt.Printf("inside main %v\n", u)

    change(u)
    fmt.Printf("inside main %v\n", u)
}

func inc(x int) {

    x++
    fmt.Printf("inside inc %d\n", x)
}

func change(u User) {

    u.occupation = "driver"
    fmt.Printf("inside change %v\n", u)
}
```

Output:

```
inside main 10
inside inc 11
inside main 10
--------------------
inside main {Raj Engineer}
inside change {Raj driver}
inside main {Raj Engineer}
```

In the above example, the original values of the `x` and `User` struct are not modified.

A copy of the integer value is created. Inside the function, we increment the value of this copy. So the original variable is intact.

```
func inc(x int) {

    x++
    fmt.Printf("inside inc %d\n", x)
}
```

Go - Function as a Parameter

A Go function can be passed to other functions as a parameter. Such a function is called a higher-order function.

In the example, the apply function takes the `inc()` and `dec()` functions as parameters.

```
package main

import "fmt"

func inc(x int) int {
    x++
    return x
}

func dec(x int) int {
    x--
    return x
}

func apply(x int, f func(int) int) int {

    r := f(x)
    return r
}

func main() {
    r1 := apply(5, inc)
    r2 := apply(4, dec)
    fmt.Println(r1)
    fmt.Println(r2)
}
```

Output:

```
6
3
```

Go - Custom Function Types

Go allows the creation of reusable function signatures with the `type` keyword. In simple words, Golang also supports defining our own function types.

In this example, we use the `type` keyword to create a function type that accepts one string parameter and returns a string.

```
package main

import "fmt"

type output func(string) string

func hello(name string) string {

    return fmt.Sprintf("hello %s", name)
}

func main() {
    var f output

    f = hello
    fmt.Println(f("Raj"))
}
```

Output:

```
hello Raj
```

Golang Related Tutorials

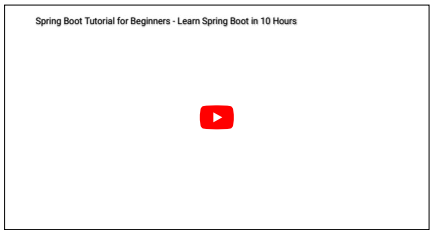
- **Go (Golang) Functions with Examples**
- **Go (Golang) Operators with Examples**
- **Go (Golang) - Read Input from User or Console**

- Go (Golang) Structs Tutorial with Examples

golang

Free Spring Boot Tutorial | Full In-depth Course | Learn Spring Boot in 10 Hours

Watch this course on YouTube at Spring Boot Tutorial | Fee 10 Hours Full Course



GOLANG