Learn Go: Introduction

Go Comments

Comments are useful for documentation in a Go file and are ignored by the compiler. There are two types of comments:

- a single-lined comment is preceded by a double forward slash, `//` , and ends at the end of the line.
- a multi-lined comment begins with `/*` followed by one or more lines of comments and ends with `*/`

```
// one line comment
/*
  this comment
  is on multiple lines
  and ends here
*/
```

Go Documentation

In Go, comments can be used as built-in documentation. To check the role of a function, in the command line, use the command go doc followed by a package or the function of a package. For example:

```
$ go doc fmt
```

To find more information about a package’s function:

```
$ go doc fmt.println
```

Import Multiple Packages

To import multiple packages in a Go file, use the `import` keyword followed by the package name enclosed in double-quotes and repeat this statement for every imported package on its own line, or write a single `import` keyword to import multiple packages, one per line, in enclosed parentheses, (...).

```
import "fmt"
import "math"
import "time"
```

or

```
import (
  "fmt"
  "math"
  "time"
)
```

Go Compiler

As a compiled language, Go does not run until its source file is processed through a separate software called a compiler to produce a final executable program. The Go compiler can be accessed on the command line via a generic command such as:

```
go <command> [arguments]
```

Packages in Go

A Go package is a directory made up of a collection of Go source files that are compiled together. This collection of reusable code typically contains functions related to a specific topic or concept. To use code from a particular package, we simply import it into our Go source file.

For example, to import the `fmt` package which contains functions for formatting input and output strings, we type the keyword `import` followed by the package name.

```
import "fmt"
```

Running Files in Go

The Go compiler can execute Go code from the source file without producing an executable file. Instead of `build` , use `run` . To do this, type the following in the command line:

```
$ go run exampleFile.go
```

Compile Go

The Go compiler takes a Go source file with a `.go` extension, processes it and produces an executable file without any extension. To compile a Go source file, `test.go` , type at the command line:

```
$ go build test.go
```

This will produce an executable file, `test` . To run `test` , type in the command line:

```
$ ./test
```

Go Import Package

To import a single package in a Go file, use the keyword `import` followed by the package name in double-quotes.

```go
import "time"
```

Learn Go: Variables and Formatting

Go Values

In Go, values can be unnamed or named. Unnamed values are literals such as `3.14` , `true` , and `"Codecademy"` . Named values have a name attached to the value and they can either be unchangeable as constants or changeable as variables once defined.

```
// literal unnamed value
fmt.Println("PI = ", 3.14159)

// constant named value
const pi = 3.14159

// variable named value
var radius = 6
```

Go Data Types

In Go, values have a data type. The data type determines what type of information is being stored and how much space is needed to store it. Go has basic data types such as:

- `string`
- `bool`
- numeric types:
    - `int8, uint8, int16, uint16, int32` , `uint32, int64, uint64, int, uint, uintptr`
    - `float32, float64`
    - `complex64, complex128`

Go Variables

A Go variable has a name attached to a value but unlike a Go constant, a variable's value can be changed after it has been defined. There are four ways to declare and assign a Go variable:

- use the `var` keyword followed by a name and its data type. This variable can be assigned later in the program. For example:

  ```
  var fruit string
  string = "apple"
  ```

- use the `var` keyword followed by a name, data type, `=` and value.

  ```
  var fruit string = "apple"
  ```

- use the `var` keyword, followed by a name, `=` and value. Ignore the data type and let the compiler infer its type.

  ```
  var fruit = "apple"
  ```

- skip the `var` keyword, define a name followed by `:=` and value and let the compiler infer its type.

  ```
  fruit := "apple"
  ```

Go Errors

In Go, errors are raised when the compiler doesn’t recognize the code as valid. The error message is printed to the terminal and contains the following information:

- The filename
- The line that raises the error
- The number of characters from the left side that raises the error
- The type of error and reason for raising the error

For example:

```
./Main.go:11:3: undefined: dinner
```

This particular error occurs in the file **main.go** at line `11` , `3` characters into the line, and its error type and reason is `"undefined: dinner"` .

Go Strings

A Go `string` is a data type that stores text or a sequence of characters in any length in double-quoted form. To concatenate two strings, use the `+` operator.

```go
var firstName string = "Abe"
var lastName string = "Lincoln"

// prints "Abe Lincoln"
fmt.Println(firstName + " " + lastName)
```

Go Zero Values

In Go, when a variable is declared without initializing a value, it has a default value. The default value is known as the zero value.
Different zero values exist for different data types:

```
Type     Zero Value
ints      0
floats    0
string    "" (empty string)
boolean   false
```

Go Inferred Int Type

When we declare a Go variable without specifying its data type and assign the variable (using `:=` or `var =`) to a whole number, the Go compiler automatically infers the variable data type as an `int`. For example:

```
score := 85
var temperature = 60
```

Go Updating Variables

Unlike constants, Go variables can change their values if we reassign new values to them. For example:

```
var zipcode = "02134"
zipcode = "03035"
```

Go supports additional assignment operators that updates a variable by performing an operation such as addition, subtraction, multiplication or division to iself.

```
// sum = sum + value
sum += value
// total = total - value
total -= value
// average = average / quantity
average /= quantity
// price = price * quantity
price *= quantity
```

Go Multiple Variable Declaration

Multiple Go variables can be declared and initialized on the same line delimited with a comma. If they are of the same type, the type can be optionally declared after the variable names before the assignment operator. For example:

```go
var x, y int = -1, 5
a, b := 7, 2
fmt.Println(x, y, a, b)
// -1, 5, 7, 2
```

If the variables are of different types, they can also be declared on the same line without the type designation.

```go
found, answer := true, "yes"
var name, age = "Steve", 35
fmt.Println(found, answer, name, age)
// true, "yes", "Steve", 35
```

Go Fmt .Print() and .Println()

The Go `fmt` package supports two closely-related functions for formatting a string to be displayed on the terminal. `.Print()` accepts strings as arguments and concatenates them without any spacing. `.Println()` , on the other hand, adds a space between strings and appends a new line to the concatenated output string.

```go
fmt.Print("I", "am", "cool")
// Iamcool
fmt.Println("I", "am", "cool")
// I am cool
```

Go Fmt .Printf() Function

The Go `.Printf()` function in `fmt` provides custom formatting of a string using one or more verbs. A verb is a placeholder for a named value (constant or variable) to be formatted according to these conventions:

- `%v` represents the named value in its default format
- `%d` expects the named value to be an integer type
- `%f` expects the named value to be a float type
- `%T` represents the type for the named value

The first argument for `.Printf()` is the string with verb(s) followed by one or more named values corresponding to the verb(s). Unlike `.Println()` , `.Printf()` does not append a newline to the formatted string.

```
name := "Leslie"
fmt.Printf("My name is %v", name)
// My name is Leslie

age := 34
fmt.Printf("I am %d years old", age)
// I am 34 years old

fmt.Printf("%v is of type %T", name, name)
// Leslie is of type string
```

Go Fmt .Scan() Function

The Go `fmt` `.Scan()` function scans user input from the terminal and extracts text delimited by spaces into successive arguments. A newline is considered a space. This function expects an address of each argument to be passed.

```
package main
import "fmt"

func main() {
  var name string
  var age int
  fmt.Println("What's your name and age?")
  fmt.Scan(&name, &age)
  fmt.Printf("You entered %v and %d.\n", name, age)
}
```

A session on the terminal may look like this:

```
$ What's your name and age?
$ Marcia 32
$ You entered Marcia and 32.
```

Learn Go: Conditionals

Go If Statement

A Go `if` statement evaluates a condition and executes a statement block enclosed in curly braces `{..}` if the evaluation returns `true`. The condition may be optionally enclosed inside a pair of parentheses `(...)`.

```
if (healthy) {
  fmt.Println("Work.")
}
if sick {
  fmt.Println("Stay home.")
}
```

Go else Statement

A Go `else` statement can succeed an `if` or `if else-if` statement block and its code executed if the conditions in the preceding `if` or `if else-if` statements evaluate to `false`.

```
sick := false
if sick {
  fmt.Println("Call the doctor.")
} else {
  fmt.Println("Enjoy your day.")
}
```

Go Comparison Operators

Go supports the standard comparison operators that compare two values and return a boolean. These include:

- `==` equivalence operator
- `!=` not equal
- `<` less than
- `>` greater than
- `<=` less than or equal
- `>=` greater than or equal

```go
same := 3 == 3
// evaluates to true
notsame := "ABC" != "abc"
// evaluates to true
lessthan := 5 <= -5
// evaluates to false
```

Go Logical Operators

In addition to comparison operators, Go also supports logical operators which evaluate boolean values and return a boolean value. For example:

- `&&` is the AND operator that returns `true` if all the booleans are `true`
- `||` is the OR operator that returns `true` if one of the booleans is `true`
- `!` is the NOT operator that returns the opposite of a boolean value

```
answer := true && false
// returns false
answer = true || false
// returns true
answer = !false
// returns true
```

Go Else If Statement

The Go `else if` statement provides an additional condition to evaluate besides the first `if` conditional. It can only appear after the `if` statement and before an `else` statement if it exists. For example:

```
if (temperature < 60) {
  fmt.Println("Put on a jacket.")
} else if (temperature >= 60 && temperature < 75) {
  fmt.Println("Put on a light sweater.")
} else {
  fmt.Println("Wear summer clothes.")
}
```

Multiple `else if` statements can exist alongside the `if` statement. The `if else if` statements are scanned from top to bottom and only the code block associated with a true condition is executed. If none of the conditions are satisfied, the `else` code block is executed if it exists.

Go Short Variable Declaration

A short variable declaration can be made within the scope of an `if` or `switch` statement before the condition is specified but after the `if` or `switch` keyword. A semicolon, `;` , is appended to the declaration to separate it from the condition.

```go
if age := 55; age >= 55 {
  fmt.Println("You are retiring!")
}
switch season := "spring"; season {
  case "spring":
    fmt.Println("Plant some bulbs.")
  case "summer":
    ...
}
```

Go Switch Statement

The Go `switch` statement can be used as an alternative to a set of `if` followed by `else if` statements. The `switch` statement compares the expression inside a condition with a set of values encapsulated in `case` s. The code inside a matched `case` value is executed and the `switch` block terminates. A `default` case without a value can be appended to the last `case` and its code executed if no prior match is found.

```go
day := "Tuesday"
switch day {
  case "Monday":
    fmt.Println("Monday is magnificent.")
  case "Tuesday":
    fmt.Println("Tuesday is terrific.")
  case "Wednesday":
    fmt.Println("Wednesday is wacky.")
  default:
    fmt.Println("We survived.")
}
```

Go Seed Value

A seed value in Go is used for generating random numbers. By default, the seed value is `1` and this leads to a predictable number instead of random. To make the seed value unique, call the seed function, `rand.Seed()` , with the argument `time.Now().UnixNano()` to return the difference in time (in Nanoseconds) since Janurary 1st 1970.

```
rand.Seed(time.Now().UnixNano())
```

Go Random Number Generator

Go provides a function, `math.rand.Intn()` , in the `math.rand` package to generate a random number. To generate such a number between `0` to `99` , pass `100` as the function argument.

```
number := math.rand.Intn(100)
```

Learn Go: Functions

Go Pass by Value Parameter

When a Go function parameter is passed by value, it means only a copy of the value is accessed and manipulated inside the function. The original value of the variable that is passed as an argument to the function remains intact.

```go
func makeMeOlder(age int) {
  age += 5
}

func main() {
  myAge := 10
  makeMeOlder(myAge)
  fmt.Println(myAge)
  // myAge is still 10
}
```

Go Variable Address

A Go variable occupies a slot in virtual memory and is accessible via an *address*. To access the address of a variable, type `&` followed by the variable name. The value of a variable address is in the form of a hexadecimal number, such as `0x414020` .

```go
name := "Codecademy"
fmt.Printf("Address of %v is %v", name, &name)
// Address of Codecademy is 0xc000010210
```

Go Pointer Dereferencing

The `*` operator preceding a data type is describing the data type for a Go pointer. For example:

```go
var pointerToInt *int
// a pointer to a variable of type int
```

The `*` operator preceding a variable is used to *dereference* a pointer variable. For example, the pointer variable, `x`, is assigned the address of variable, `y`. We dereference `x` by typing `*x`. By doing so, we can access and change the value of `y`. For example:

```go
y := 3
var x *int = &y
*x = 5
fmt.Println(y)
// y is now 5
```