

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
fruit = "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.

```
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 itself.

```
// 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:

```
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.

```
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.

```
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.

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.
```

```
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
```