Go

Syntax

A Go file consists of the following parts:

- Package declaration
- Import packages
- Functions
- Statements and expressions

Example:

```
package main
import "fmt"

func main() {
  fmt.Println("Hello World!")
}
```

Comments

Go supports single-line or multi-line comments.

Single-line Comments

```
// This is a comment
```

Multi-line Comments

```
/*
 * This is a Multi-line comment
 */
```

Variables

Types

- `int` - stores integers (whole numbers), such as 123 or -123
- `float32` - stores floating point numbers, with decimals, such as 19.99 or -19.99
- `string` - stores text, such as "Hello World". String values are surrounded by double quotes
- `bool` - stores values with two states: true or false

Declaring variables

In Go, there are two ways to declare a variable:

1. With the `var` keyword:

   ```
   var variablename type = value
   ```

   Note: You always have to specify either `type` or `value` (or both).

2. With the `:=` sign:

   ```
   variablename := value
   ```

   Note: In this case, the type of the variable is **inferred** from the value (means that the compiler decides the type of the variable, based on the value).

   Note: It is not possible to declare a variable using `:=` without assigning a value to it.

Variable Declaration With Initial Value

```
var student1 string = "John" //type is string
var student2 = "Jane" //type is inferred
x := 2 //type is inferred
```

Note: The variable types of `student2` and `x` is **inferred** from their values.

Variable Declaration Without Initial Value

```
package main
import ("fmt")

func main() {
    var a string
    var b int
    var c bool
    fmt.Println(a)
    fmt.Println(b)
    fmt.Println(c)
}
```

By running the code, we can see that they already have the default values of their respective types:

- `a` is `""`
- `b` is `0`
- `c` is `false`

Value Assignment After Declaration

```
func main() {
    var student1 string
    student1 = "John"
    fmt.Println(student1)
}
```

Difference Between var and :=

`var`	`:=`
Can be used **inside** and **outside** of functions Variable declaration and value assignment **can be done separately**	Can only be used **inside** functions Variable declaration and value assignment **cannot be done separately** (must be done in the same line)

Multiple Variable Declaration

```
var a, b, c, d int = 1, 3, 5, 7
```

Note: If you use the `type` keyword, it is only possible to declare **one type** of variable per line.

If the `type` keyword is not specified, you can declare different types of variables in the same line:

```
var a, b = 6, "Hello"
c, d := 7, "World!"
```

Variable Declaration in a Block

Multiple variable declarations can also be grouped together into a block for greater readability:

```
var (
    a int
    b int = 1
    c string = "hello"
)
```

Variable Naming Rules

Go variable naming rules:

- A variable name must start with a letter or an underscore character (_)
- A variable name cannot start with a digit

- A variable name can only contain alpha-numeric characters and underscores (`a-z`, `A-Z`, `0-9`, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)
- There is no limit on the length of the variable name
- A variable name cannot contain spaces
- The variable name cannot be any Go keywords

Constants

If a variable should have a fixed value that cannot be changed, you can use the `const` keyword.

```
const CONSTNAME type = value
```

Note: The value of a constant must be assigned when you declare it.

```
const PI = 3.14
```

Constant Rules

- Constant names follow the same naming rules as variables
- Constant names are usually written in uppercase letters (for easy identification and differentiation from variables)
- Constants can be declared both inside and outside of a function

Constant Types

There are two types of constants:

- Typed constants
- Untyped constants

Multiple Constants Declaration

Multiple constants can be grouped together into a block for readability:

```
const (
    A int = 1
    B = 3.14
    C = "Hi!"
)
```

Output

Output Functions

Go has three functions to output text:

- `Print()`
- `Println()`
- `Printf()`

The Print() Function

The `Print()` function prints its arguments with their default format.

- If we want to print the arguments in new lines, we need to use `\n`.

```
var i,j string = "Hello","World"
fmt.Print(i, "\n")
fmt.Print(j, "\n")
```

- `Print()` inserts a space between the arguments if **neither** are strings.

The Println() Function

The `Println()` function is similar to `Print()` with the difference that a whitespace is added between the arguments, and a newline is added at the end:

```
var i,j string = "Hello","World"
fmt.Println(i, j)
```

The Printf() Function

The `Printf()` function first formats its argument based on the given formatting verb and then prints them.

Here we will use two formatting verbs:

- `%v` is used to print the **value** of the arguments
- `%T` is used to print the **type** of the arguments

```
var i string = "Hello"
var j int = 15

fmt.Printf("i has value: %v and type: %T\n", i, i)
fmt.Printf("j has value: %v and type: %T", j, j)
```

Formatting Verbs

Go offers several formatting verbs that can be used with the `Printf()` function.

General Formatting Verbs

The following verbs can be used with all data types:

Verb	Description
`%v`	Prints the value in the default format
`%#v`	Prints the value in Go-syntax format
`%T`	Prints the type of the value
`%%`	Prints the % sign

Integer Formatting Verbs

The following verbs can be used with the integer data type:

Verb	Description
`%b`	Base 2
`%d`	Base 10
`%+d`	Base 10 and always show sign

Verb	Description
`%o`	Base 8
`%O`	Base 8, with leading 0o
`%x`	Base 16, lowercase
`%X`	Base 16, uppercase
`%#x`	Base 16, with leading 0x
`%4d`	Pad with spaces (width 4, right justified)
`%-4d`	Pad with spaces (width 4, left justified)
`%04d`	Pad with zeroes (width 4)

String Formatting Verbs

The following verbs can be used with the string data type:

Verb	Description
`%s`	Prints the value as plain string
`%q`	Prints the value as a double-quoted string
`%8s`	Prints the value as plain string (width 8, right justified)
`%-8s`	Prints the value as plain string (width 8, left justified)
`%x`	Prints the value as hex dump of byte values
`% x`	Prints the value as hex dump with spaces

Boolean Formatting Verbs

The following verb can be used with the boolean data type:

Verb	Description
`%t`	Value of the boolean operator in true or false format (same as using `%v`)

Float Formatting Verbs

The following verbs can be used with the float data type:

Verb	Description
`%e`	Scientific notation with ‘e’ as exponent
`%f`	Decimal point, no exponent
`%.2f`	Default width, precision 2
`%6.2f`	Width 6, precision 2
`%g`	Exponent as needed, only necessary digits

Data Types

Data type is an important concept in programming. Data type specifies the size and type of variable values.

Basics

Go is statically typed, meaning that once a variable type is defined, it can only store data of that type. It has three basic data types:

- **bool**: represents a boolean value and is either true or false
- **Numeric**: represents integer types, floating point values, and complex types
- **string**: represents a string value

```
var a bool = true    // Boolean
var b int = 5        // Integer
var c float32 = 3.14  // Floating point number
var d string = "Hi!"  // String
```

Boolean

A boolean data type is declared with the `bool` keyword and can only take the values `true` or `false`.

The default value of a boolean data type is `false`.

```
var b1 bool = true // typed declaration with initial value
var b2 = true // untyped declaration with initial value
var b3 bool // typed declaration without initial value
b4 := true // untyped declaration with initial value
```

Note: Boolean values are mostly used for conditional testing.

Integer

Integer data types are used to store a whole number without decimals, like 35, -50, or 1345000.

The integer data type has two categories:

- **Signed integers** - can store both positive and negative values
- **Unsigned integers** - can only store non-negative values

Tip: The default type for integer is **int**. If you do not specify a type, the type will be **int**.

Signed Integers

Signed integers, declared with one of the `int` keywords, can store both positive and negative values:

```
var x int = 500
var y int = -4500
```

Go has five keywords/types of signed integers:

Type	Size	Range
`int`	Depends on platform: 32 bits in 32 bit systems and 64 bit in 64 bit systems	-2147483648 to 2147483647 in 32 bit systems and -9223372036854775808 to 9223372036854775807 in 64 bit systems
`int8`	8 bits/1 byte	-128 to 127
`int16`	16 bits/2 byte	-32768 to 32767
`int32`	32 bits/4 byte	-2147483648 to 2147483647

Type	Size	Range
`int64`	64 bits/8 byte	-9223372036854775808 to 9223372036854775807

Unsigned Integers

Unsigned integers, declared with one of the `uint` keywords, can only store non-negative values:

```
var x uint = 500
var y uint = 4500
```

Go has five keywords/types of signed integers:

Type	Size	Range
`uint`	Depends on platform: 32 bits in 32 bit systems and 64 bit in 64 bit systems	0 to 4294967295 in 32 bit systems and 0 to 18446744073709551615 in 64 bit systems
`uint8`	8 bits/1 byte	0 to 255
`uint16`	16 bits/2 byte	0 to 65535
`uint32`	32 bits/4 byte	0 to 4294967295
`uint64`	64 bits/8 byte	0 to 18446744073709551615

Float

The float data types are used to store positive and negative numbers with a decimal point, like 35.3, -2.34, or 3597.34987.

The float data type has two keywords:

Type	Size	Range
`float32`	32 bits	-3.4e+38 to 3.4e+38
`float64`	64 bits	-1.7e+308 to +1.7e+308

Tip: The default type for float is `float64`. If you do not specify a type, the type will be `float64`.

The float32 Keyword

```
var x float32 = 123.78
var y float32 = 3.4e+38
```

The float64 Keyword

The `float64` data type can store a larger set of numbers than `float32`.

```
var x float64 = 1.7e+308
```

String

The `string` data type is used to store a sequence of characters (text). String values must be surrounded by double quotes:

```
var txt1 string = "Hello!"
var txt2 string
txt3 := "World 1"
```

Arrays

Arrays are used to store multiple values of the same type in a single variable, instead of declaring separate variables for each value.

Declare an Array

In Go, there are two ways to declare an array:

1. With the `var` keyword:

```
var array_name = [length]datatype{values} // length is defined
var array_name = [...]datatype{values} // length is inferred
```

2. With the `:=` sign:

```
array_name := [length]datatype{values} // length is defined
array_name := [...]datatype{values} // length is inferred
```

Note: The length specifies the number of elements to store in the array. In Go, arrays have a fixed length. The length of the array is either defined by a number or is inferred (means that the compiler decides the length of the array, based on the number of values).

Example:

```
var arr1 = [3]int{1, 2, 3}
var arr1 = [...]int{1, 2, 3}
arr2 := [5]int{4, 5, 6, 7, 8}
arr2 := [...]int{4, 5, 6, 7, 8}
```

Access Elements of an Array

You can access a specific array element by referring to the index number.

In Go, array indexes start at 0. That means that [0] is the first element, [1] is the second element, etc.

```
prices := [3]int{10, 20, 30}
fmt.Println(prices[0])
fmt.Println(prices[2])
```

Change Elements of an Array

You can also change the value of a specific array element by referring to the index number.

```
prices := [3]int{10, 20, 30}
prices[2] = 50
```

Array Initialization

If an array or one of its elements has not been initialized in the code, it is assigned the default value of its type.

Tip: The default value for int is `0` and the default value for string is `""`.

```
arr1 := [5]int{} // not initialized
arr2 := [5]int{1, 2} // partially initialized
arr3 := [5]int{1, 2, 3, 4, 5} // fully initialized
```

Initialize Only Specific Elements

It is possible to initialize only specific elements in an array.

```
arr1 := [5]int{1:10, 2:40}
```

The array above has 5 elements.

- `1:10` means: assign `10` to array index `1` (second element).
- `2:40` means: assign `40` to array index `2` (third element).

Find the Length of an Array

The `len()` function is used to find the length of an array:

```
arr1 := [4]string{"Volvo", "BMW", "Ford", "Mazda"}
arr2 := [...]int{1, 2, 3, 4, 5, 6}

fmt.Println(len(arr1))
fmt.Println(len(arr2))
```

Slices

Slices are similar to arrays, but are more powerful and flexible.

Like arrays, slices are also used to store multiple values of the same type in a single variable.

However, unlike arrays, the length of a slice can grow and shrink as you see fit.

Create a Slice

In Go, there are several ways to create a slice:

- Using the `[]datatype{values}` format
- Create a slice from an array
- Using the `make()` function

[]datatype{values}

```
slice_name := []datatype{values}
```

A common way of declaring a slice is like this:

```
myslice := []int{}
```

The code above declares an empty slice of 0 length and 0 capacity.

To initialize the slice during declaration, use this:

```
myslice := []int{1, 2, 3}
```

The code above declares a slice of integers of length 3 and also the capacity of 3.

In Go, there are two functions that can be used to return the length and capacity of a slice:

- **`len()` function** - returns the length of the slice (the number of elements in the slice)
- **`cap()` function** - returns the capacity of the slice (the number of elements the slice can grow or shrink to)

Example:

```
package main
import ("fmt")

func main() {
    myslice1 := []int{}
    fmt.Println(len(myslice1))
    fmt.Println(cap(myslice1))
    fmt.Println(myslice1)

    myslice2 := []string{"Go", "Slices", "Are", "Powerful"}
    fmt.Println(len(myslice2))
    fmt.Println(cap(myslice2))
    fmt.Println(myslice2)
}
```

Result:

```
0
0
[]
4
4
[Go Slices Are Powerful]
```

In the example above, we see that in the first slice (myslice1), the actual elements are not specified, so both the length and capacity of the slice will be zero. In the second slice (myslice2), the elements are specified, and both length and capacity is equal to the number of actual elements specified.

Create a Slice From an Array

You can create a slice by slicing an array:

```
var myarray = [length]datatype{values} // An array
myslice := myarray[start:end] // A slice made from the array
```

```
arr1 := [6]int{10, 11, 12, 13, 14,15}
myslice := arr1[2:4]

fmt.Printf("myslice = %v\n", myslice)
fmt.Printf("length = %d\n", len(myslice))
fmt.Printf("capacity = %d\n", cap(myslice))
```

Result:

```
myslice = [12 13]
length = 2
capacity = 4
```

In the example above `myslice` is a slice with length 2. It is made from `arr1` which is an array with length 6.

The slice starts from the second element of the array which has value 12. The slice can grow to the end of the array. This means that the capacity of the slice is 4.

If `myslice` started from element 0, the slice capacity would be 6.

Create a Slice With The make() Function

The `make()` function can also be used to create a slice.

```
slice_name := make([]type, length, capacity)
```

Note: If the capacity parameter is not defined, it will be equal to *length*

```
package main
import ("fmt")

func main() {
    myslice1 := make([]int, 5, 10)
    fmt.Printf("myslice1 = %v\n", myslice1)
    fmt.Printf("length = %d\n", len(myslice1))
    fmt.Printf("capacity = %d\n", cap(myslice1))

    // with omitted capacity
    myslice2 := make([]int, 5)
    fmt.Printf("myslice2 = %v\n", myslice2)
    fmt.Printf("length = %d\n", len(myslice2))
    fmt.Printf("capacity = %d\n", cap(myslice2))
}
```

Result:

```
myslice1 = [0 0 0 0 0]
length = 5
capacity = 10
myslice2 = [0 0 0 0 0]
length = 5
capacity = 5
```

Access, Change, Append and Copy Slices

Access Elements of a Slice

You can access a specific slice element by referring to the index number.

In Go, indexes start at 0. That means that [0] is the first element, [1] is the second element, etc.

```
prices := []int{10, 20, 30}

fmt.Println(prices[0])
fmt.Println(prices[2])
```

Change Elements of a Slice

You can also change a specific slice element by referring to the index number.

```
prices := []int{10, 20, 30}
prices[2] = 50
```

Append Elements To a Slice

You can append elements to the end of a slice using the **`append()`** function:

```
slice_name = append(slice_name, element1, element2, ...)
```

```
myslice1 := []int{1, 2, 3, 4, 5, 6}
fmt.Printf("myslice1 = %v\n", myslice1)
fmt.Printf("length = %d\n", len(myslice1))
fmt.Printf("capacity = %d\n", cap(myslice1))

myslice1 = append(myslice1, 20, 21)
fmt.Printf("myslice1 = %v\n", myslice1)
fmt.Printf("length = %d\n", len(myslice1))
fmt.Printf("capacity = %d\n", cap(myslice1))
```

Append One Slice To Another Slice

To append all the elements of one slice to another slice, use the **`append()`** function:

```
slice3 = append(slice1, slice2...)
```

Note: The '**`...`**' after *slice2* is **necessary** when appending the elements of one slice to another.

```
myslice1 := []int{1, 2, 3}
myslice2 := []int{4, 5, 6}
myslice3 := append(myslice1, myslice2...)
```

Change The Length of a Slice

Unlike arrays, it is possible to change the length of a slice.

```
arr1 := [6]int{9, 10, 11, 12, 13, 14} // An array
myslice1 := arr1[1:5] // Slice array
fmt.Printf("myslice1 = %v\n", myslice1)
fmt.Printf("length = %d\n", len(myslice1))
fmt.Printf("capacity = %d\n", cap(myslice1))

/* Change length by re-slicing the array */
myslice1 = arr1[1:3]
fmt.Printf("myslice1 = %v\n", myslice1)
fmt.Printf("length = %d\n", len(myslice1))
fmt.Printf("capacity = %d\n", cap(myslice1))

/* Change length by appending items */
myslice1 = append(myslice1, 20, 21, 22, 23)
fmt.Printf("myslice1 = %v\n", myslice1)
fmt.Printf("length = %d\n", len(myslice1))
fmt.Printf("capacity = %d\n", cap(myslice1))
```

Memory Efficiency

When using slices, Go loads all the underlying elements into the memory.

If the array is large and you need only a few elements, it is better to copy those elements using the `copy()` function.

The `copy()` function creates a new underlying array with only the required elements for the slice. This will reduce the memory used for the program.

```
copy(dest, src)
```

The `copy()` function takes in two slices dest and src, and copies data from *src* to *dest*. It returns the number of elements copied.

```
numbers := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}
// Original slice
fmt.Printf("numbers = %v\n", numbers)
fmt.Printf("length = %d\n", len(numbers))
fmt.Printf("capacity = %d\n", cap(numbers))

// Create copy with only needed numbers
neededNumbers := numbers[:len(numbers)-10]
numbersCopy := make([]int, len(neededNumbers))
copy(numbersCopy, neededNumbers)

fmt.Printf("numbersCopy = %v\n", numbersCopy)
fmt.Printf("length = %d\n", len(numbersCopy))
fmt.Printf("capacity = %d\n", cap(numbersCopy))
```

Result:

```
numbers = [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15]
length = 15
capacity = 15
numbersCopy = [1 2 3 4 5]
length = 5
capacity = 5
```

The capacity of the new slice is now less than the capacity of the original slice because the new underlying array is smaller.

Operators

Operators are used to perform operations on variables and values.

The `+` **operator** adds together two values, like in the example below:

```
var a = 15 + 25
```

Although the `+` operator is often used to add together two values, it can also be used to add together a variable and a value, or a variable and another variable:

```
var (
    sum1 = 100 + 50 // 150 (100 + 50)
    sum2 = sum1 + 250 // 400 (150 + 250)
    sum3 = sum2 + sum2 // 800 (400 + 400)
)
```

Go divides the operators into the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Bitwise operators

Arithmetic Operators

Arithmetic operators are used to perform common mathematical operations.

Operator	Name	Description	Example
`+`	Addition	Adds together two values	`x + y`
`-`	Subtraction	Subtracts one value from another	`x - y`
`*`	Multiplication	Multiplies two values	`x * y`
`/`	Division	Divides one value by another	`x / y`
`%`	Modulus	Returns the division remainder	`x % y`
`++`	Increment	Increases the value of a variable by 1	`x++`
`--`	Decrement	Decreases the value of a variable by 1	`x--`

Assignment Operators

Assignment operators are used to assign values to variables.

In the example below, we use the **assignment** operator (`=`) to assign the value **10** to a variable called **x**:

```
var x = 10
```

The **addition assignment** operator (`+=`) adds a value to a variable:

```
var x = 10
x +=5
```

A list of all assignment operators:

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Comparison Operators

Comparison operators are used to compare two values.

> **Note**: The return value of a comparison is either true (`1`) or false (`0`)

In the following example, we use the **greater than** operator (`>`) to find out if 5 is greater than 3:

```
var x = 5
var y = 3
fmt.Println(x > y) // returns 1 (true) because 5 is greater than 3
```

A list of all comparison operators:

Operator	Name	Example
==	Equal to	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y

Operator	Name	Example
`>=`	Greater than or equal to	`x >= y`
`<=`	Less than or equal to	`x <= y`

Logical Operators

Logical operators are used to determine the logic between variables or values:

Operator	Name	Description	Example
`&&`	Logical `and`	Returns true if both statements are true	`x < 5 && x < 10`
`||`	Logical `or`	Returns true if one of the statements is true	`x < 5 || x < 4`
`!`	Logical `not`	Reverse the result, returns false if the result is true	`!(x < 5 && x < 10)`

Bitwise Operators

Bitwise operators are used on (binary) numbers:

Operator	Name	Description	Example
`&`	AND	Sets each bit to 1 if both bits are 1	`x & y`
`|`	OR	Sets each bit to 1 if one of two bits is 1	`x | y`
`^`	XOR	Sets each bit to 1 if only one of two bits is 1	`x ^ b`
`<<`	Zero fill left shift	Shift left by pushing zeros in from the right	`x << 2`
`>>`	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off	`x >> 2`

Conditions

Conditional statements are used to perform different actions based on different conditions.

A condition can be either `true` or `false`.

Go supports the usual comparison operators from mathematics:

- Less than `<`
- Less than or equal `<=`
- Greater than `>`
- Greater than or equal `>=`
- Equal to `==`
- Not equal to `!=`

Additionally, Go supports the usual logical operators:

- Logical AND `&&`
- Logical OR `||`
- Logical NOT `!`

You can use these operators or their combinations to create conditions for different decisions.

Example
`x > y`
`x != y`
`(x > y) && (y > z)`
`(x == y) || z`

Go has the following conditional statements:

- **`if`** - specify a block of code to be executed, if a specified condition is true
- **`else`** - specify a block of code to be executed, if the same condition is false
- **`else if`** - specify a new condition to test, if the first condition is false
- **`switch`** - specify many alternative blocks of code to be executed

if Statement

Use the `if` statement to specify a block of Go code to be executed if a condition is `true`.

```
if condition {
    /*
     * code to be executed if condition is true
     */
}
```

Note: `if` is in lowercase letters. Uppercase letters (If or IF) will generate an error.

```
if 20 > 18 {
    fmt.Println("20 is greater than 18")
}

x:= 20
y:= 18
if x > y {
    fmt.Println("x is greater than y")
}
```

if else Statement

else Statement

Use the **`else`** statement to specify a block of code to be executed if the condition is **`false`**.

```
if condition {
    // code to be executed if condition is true
} else {
    // code to be executed if condition is false
}
```

```
time := 20
if (time < 18) {
    fmt.Println("Good day.")
} else {
    fmt.Println("Good evening.")
}
```

Note: The brackets in the `else` statement should be like `} else {`

```
if (temperature > 15) {
    fmt.Println("It is warm out there.")
} /* this raises an error */
else {
    fmt.Println("It is cold out there.")
}
```

else if Statement

Use the `else if` statement to specify a new condition if the first condition is `false`.

```
if condition1 {
    /*
     * code to be executed if condition1 is true
     */
} else if condition2 {
    /*
     * code to be executed if condition1 is false and
     * condition2 is true
     */
} else {
    /*
     * code to be executed if condition1
     * and condition2 are both false
     */
}
```

```
time := 22
if time < 10 {
    fmt.Println("Good morning.")
} else if time < 20 {
    fmt.Println("Good day.")
} else {
    fmt.Println("Good evening.")
}
```

Note: If condition1 and condition2 are BOTH true, only the code for condition1 are executed

Nested if Statement

You can have `if` statements inside `if` statements, this is called a nested if.

```
if condition1 {
    // code to be executed if condition1 is true
    if condition2 {
         /*
          * code to be executed if both condition1
          * and condition2 are true
          */
    }
}
```

```
num := 20
if num >= 10 {
    fmt.Println("Num is more than 10.")
        if num > 15 {
            fmt.Println("Num is also more than 15.")
        }
    } else {
        fmt.Println("Num is less than 10.")
}
```

switch Statement

Use the `switch` statement to select one of many code blocks to be executed.

The `switch` statement in Go is similar to the ones in C. The difference is that it only runs the matched case so it does not need a `break` statement.

Single-Case switch Syntax

```
switch expression {
case x:
    // code block
case y:
...
default:
    // code block
}
```

This is how it works:

- The expression is evaluated once
- The value of the `switch` expression is compared with the values of each `case`
- If there is a match, the associated block of code is executed
- The `default` keyword is optional. It specifies some code to run if there is no `case` match

```
day := 3
switch day {
case 1:
    fmt.Println("One")
case 2:
    fmt.Println("Two")
case 3:
    fmt.Println("Three")
}
```

default Keyword

The default keyword specifies some code to run if there is no case match:

```
day := 4
switch day {
case 1:
    fmt.Println("One")
case 2:
    fmt.Println("Two")
case 3:
    fmt.Println("Three")
default:
    fmt.Println("Not in the [1-3] range")
}
```

Note: All the `case` values should have the same type as the `switch` expression. Otherwise, the compiler will raise an error

Multi-case switch Statement

It is possible to have multiple values for each case in the switch statement:

```
switch expression {
case x,y:
    // code block if expression is evaluated to x or y
case v,w:
    // code block if expression is evaluated to v or w
case z:
...
default:
    // code block if expression is not found in any cases
}
```

```
day := 5

switch day {
case 1, 3, 5:
    fmt.Println("Odd weekday")
case 2, 4:
    fmt.Println("Even weekday")
case 6, 7:
    fmt.Println("Weekend")
default:
    fmt.Println("Invalid day of day number")
}
```

Loops

For Loop

The `for` loop loops through a block of code a specified number of times. And `for` loop is the only loop available in Go.

Loops are handy if you want to run the same code over and over again, each time with a different value.

Each execution of a loop is called an **iteration**.

The `for` loop can take up to three statements:

```
for statement1; statement2; statement3 {
    // code to be executed for each iteration
}
```

- **statement1** - Initializes the loop counter value.
- **statement2** - Evaluated for each loop iteration. If it evaluates to TRUE, the loop continues. If it evaluates to FALSE, the loop ends.
- **statement3** - Increases the loop counter value.

  Note: These statements don't need to be present as loops arguments. However, they need to be present in the code in some form.

```
for i:= 0, i < 5, i++ {
    fmt.Println(i)
}
```

continue Statement

The `continue` statement is used to skip one or more iterations in the loop. It then continues with the next iteration in the loop.

```
for i:=0; i < 5; i++ {
    if i == 3 {
        continue
    }
    fmt.Println(i)
}
```

break Statement

The `break` statement is used to break/terminate the loop execution.

```
for i:=0; i < 5; i++ {
    if i == 3 {
        break
    }
    fmt.Println(i)
}
```

Note: `continue` and `break` are usually used with conditions.

Nested Loops

It is possible to place a loop inside another loop.

Here, the “inner loop” will be executed one time for each iteration of the “outer loop”:

```
for i:= 0, i < 5, i++ {
    for j:= 0, j < 5, j++ {
        fmt.Printf("%d\t",i * j)
    }
    fmt.Println()
}
```

Range Keyword

The `range` keyword is used to more easily iterate over an array, slice or map. It returns both the index and the value.

The `range` keyword is used like this:

```
for index, value := array|slice|map {
    // code to be executed for each iteration
}
```

- This example uses `range` to iterate over an array and print both the indexes and the values at each (`idx` stores the index, `val` stores the value):

```
fruits := [3]string{"apple", "orange", "banana"}
for idx, val := range fruits {
    fmt.Printf("%v\t%v\n", idx, val)
}
```

Tip: To only show the value or the index, you can omit the other output using an underscore (_).

Functions

A function is a block of statements that can be used repeatedly in a program. A function will not execute automatically when a page loads. A function will be executed by a call to the function.

Create a Function

To create (often referred to as declare) a function, do the following:

- Use the `func` keyword.
- Specify a name for the function, followed by parentheses ().
- Finally, add code that defines what the function should do, inside curly braces {}.

```
func FunctionName() {
    // code to be executed
}
```

Call a Function

Functions are not executed immediately. They are "saved for later use", and will be executed when they are called.

In the example below, we create a function named "myMessage()". The opening curly brace ({) indicates the beginning of the function code, and the closing curly brace (}) indicates the end of the function. The function outputs "I just got executed!". To call the function, just write its name followed by two parentheses ():

```
func myMessage() {
    fmt.Println("I just got executed!")
}

func main() {
    myMessage() // call the function
}
```

- A function can be called multiple times.

Naming Rules for Go Functions

- A function name must start with a letter
- A function name can only contain alpha-numeric characters and underscores (`A-z`, `0-9`, and `_`)
- Function names are case-sensitive

- A function name cannot contain spaces
- If the function name consists of multiple words, techniques introduced for multi-word variable naming can be used

Tip: Give the function a name that reflects what the function does!

Parameters and Arguments

Information can be passed to functions as a parameter. Parameters act as variables inside the function.

Parameters and their types are specified after the function name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma:

```
func FunctionName(param1 type, param2 type, param3 type) {
    // code to be executed
}
```

Function With Parameter Example

The following example has a function with one parameter (`fname`) of type `string`. When the familyName() function is called, we also pass along a name (e.g. Liam), and the name is used inside the function, which outputs several different first names, but an equal last name:

```
func familyName(fname string) {
    fmt.Println("Hello", fname, "Refsnes")
}

func main() {
    familyName("Liam")
    familyName("Jenny")
    familyName("Anja")
}
```

Result:

```
Hello Liam Refsnes
Hello Jenny Refsnes
Hello Anja Refsnes
```

Note: When a **parameter** is passed to the function, it is called an **argument**. So, from the example above: `fname` is a **parameter**, while `Liam`, `Jenny` and `Anja` are **arguments**.

Multiple Parameters

Inside the function, you can add as many parameters as you want:

```
func familyName(fname string, age int) {
    fmt.Println("Hello", age, "year old", fname, "Refsnes")
}
```

Note: When you are working with multiple parameters, the function call must have the same number of arguments as there are parameters, and the arguments must be passed in the same order.

Function Returns

Values

If you want the function to return a value, you need to define the data type of the return value (such as `int`, `string`, etc), and also use the `return` keyword inside the function:

```
func FunctionName(param1 type, param2 type) type {
    // code to be executed
    return output
}
```

```
func myFunction(x int, y int) int {
    return x + y
}

func main() {
    fmt.Println(myFunction(1, 2))
}
```

In Go, you can name the return values of a function.

Here, we name the return value as `result` (of type `int`), and return the value with a naked return (means that we use the `return` statement without specifying the variable name):

```
func myFunction(x int, y int) (result int) {
    result = x + y
    return
}

func main() {
    fmt.Println(myFunction(1, 2))
}
```

The example above can also be written like this. Here, the return statement specifies the variable name:

```
func myFunction(x int, y int) (result int) {
    result = x + y
    return result
}

func main() {
    fmt.Println(myFunction(1, 2))
}
```

Store the Return Value in a Variable

You can also store the return value in a variable, like this:

```
func myFunction(x int, y int) (result int) {
    result = x + y
    return
}

func main() {
    total := myFunction(1, 2)
    fmt.Println(total)
}
```

Multiple Return Values

Go functions can also return multiple values.

```
func myFunction(x int, y string) (result int, txt1 string) {
    result = x + x
    txt1 = y + " World!"
    return
}

func main() {
    fmt.Println(myFunction(5, "Hello"))
}
```

If we (for some reason) do not want to use some of the returned values, we can add an underscore (_) to omit this value.

```
func myFunction(x int, y string) (result int, txt1 string) {
    result = x + x
    txt1 = y + " World!"
    return
}

func main() {
     _, b := myFunction(5, "Hello")
    fmt.Println(b)
}
```

Recursion Functions

Go accepts recursion functions. A function is recursive if it calls itself and reaches a stop condition.

In the following example, `testcount()` is a function that calls itself. We use the `x` variable as the data, which increments with 1 (`x + 1`) every time we recurse. The recursion ends when the `x` variable equals to 11 (`x == 11`).

```
package main
import ("fmt")

func testcount(x int) int {
    if x == 11 {
        return 0
    }
    fmt.Println(x)
    return testcount(x + 1)
}

func main(){
    testcount(1)
}
```

Recursion is a common mathematical and programming concept. This has the benefit of meaning that you can loop through data to reach a result.

The developer should be careful with recursion functions as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.

In the following example, `factorial_recursion()` is a function that calls itself. We use the `x` variable as the data, which decrements (-1) every time we recurse. The recursion ends when the condition is not greater than 0 (i.e. when it is 0).

```
package main
import ("fmt")

func factorial_recursion(x float64) (y float64) {
    if x > 0 {
         y = x * factorial_recursion(x-1)
    } else {
         y = 1
    }
    return
}

func main() {
    fmt.Println(factorial_recursion(4))
}
```

To a new developer it can take some time to work out how exactly this works, best way to find out is by testing and modifying it.

Struct

A struct (short for structure) is used to create a collection of members of different data types, into a single variable.

While arrays are used to store multiple values of the same data type into a single variable, structs are used to store multiple values of different data types into a single variable.

A struct can be useful for grouping data together to create records.

Declare a Struct

To declare a structure in Go, use the `type` and `struct` keywords:

```
type struct_name struct {
    member1 datatype;
    member2 datatype;
    member3 datatype;
    ...
}
```

Here we declare a struct type `Person` with the following members: `name`, `age`, `job` and `salary`:

```
type Person struct {
    name string
    age int
    job string
    salary int
}
```

Tip: Notice that the struct members above have different data types. `name` and `job` is of type string, while `age` and `salary` is of type int.

Access Struct Members

To access any member of a structure, use the dot operator (.) between the structure variable name and the structure member:

```
type Person struct {
    name string
    age int
    job string
    salary int
}

func main() {
    var pers1 Person
    var pers2 Person
    /* Pers1 specification */
    pers1.name = "Hege"
    pers1.age = 45
    pers1.job = "Teacher"
    pers1.salary = 6000
    /* Pers2 specification */
    pers2.name = "Cecilie"
    pers2.age = 24
    pers2.job = "Marketing"
    pers2.salary = 4500
}
```

Pass Struct as Function Arguments

You can also pass a structure as a function argument, like this:

```
type Person struct {
    name string
    age int
    job string
    salary int
}

func main() {
    var pers1 Person
    /* Pers1 specification */
    pers1.name = "Hege"
    pers1.age = 45
    pers1.job = "Teacher"
    pers1.salary = 6000
    /* Print Pers1 info by calling a function */
    printPerson(pers1)
}

func printPerson(pers Person) {
    fmt.Println("Name: ", pers.name)
    fmt.Println("Age: ", pers.age)
    fmt.Println("Job: ", pers.job)
    fmt.Println("Salary: ", pers.salary)
}
```

Maps

Maps are used to store data values in `key:value` pairs. Each element in a map is a `key:value` pair. A map is an unordered and changeable collection that does not allow duplicates.

The length of a map is the number of its elements. You can find it using the `len()` function. The default value of a map is nil. Maps hold references to an underlying hash table.

Go has multiple ways for creating maps.

Creating Maps Using `var` and `:=`

```
var a = map[KeyType]ValueType{key1:value1, key2:value2,...}
b := map[KeyType]ValueType{key1:value1, key2:value2,...}
```

- This example shows how to create maps in Go. Notice the order in the code and in the output

```
package main
import ("fmt")

func main() {
    var a = map[string]string{"brand": "Ford", "model": "Mustang"}
    b := map[string]int{"Oslo": 1, "Bergen": 2, "Trondheim": 3}

    fmt.Printf("a\t%v\n", a)
    fmt.Printf("b\t%v\n", b)
}
```

Result:

```
a       map[brand:Ford model:Mustang]
b       map[Bergen:2 Oslo:1 Trondheim:3]
```

Note: The order of the map elements defined in the code is different from the way that they are stored. The data are stored in a way to have efficient data retrieval from the map.

Creating Maps Using Using `make( )` Function

```
var a = make(map[KeyType]ValueType)
b := make(map[KeyType]ValueType)
```

```
func main() {
    var a = make(map[string]string) // The map is empty now
    a["brand"] = "Ford"
    a["model"] = "Mustang"
    a["year"] = "1964"

    b := make(map[string]int)
    b["Oslo"] = 1
    b["Bergen"] = 2
    b["Trondheim"] = 3
    b["Stavanger"] = 4
}
```

Creating an Empty Map

There are two ways to create an empty map. One is by using the `make()` function and the other is by using the following syntax.

```
var a map[KeyType]ValueType
```

Note: The `make()` function is the right way to create an empty map. If you make an empty map in a different way and write to it, it will causes a runtime panic.

```
func main() {
    var a = make(map[string]string)
    var b map[string]string

    fmt.Println(a == nil)
    fmt.Println(b == nil)
}
```

Result:

```
false
true
```

Allowed Key Types

The map key can be of any data type for which the equality operator (==) is defined. These include:

- Booleans
- Numbers
- Strings
- Arrays
- Pointers
- Structs
- Interfaces (as long as the dynamic type supports equality)

Invalid key types are:

- Slices
- Maps
- Functions

These types are invalid because the equality operator (==) is not defined for them.

Allowed Value Types

The map values can be **any** type.

Accessing Map Elements

You can access map elements by:

```
value = map_name[key]
```

```
var a = make(map[string]string)
a["brand"] = "Ford"
a["model"] = "Mustang"
a["year"] = "1964"

fmt.Printf(a["brand"])
```

Updating and Adding Map Elements

Updating or adding an elements are done by:

```
map_name[key] = value
```

```
var a = make(map[string]string)
a["brand"] = "Ford"
a["model"] = "Mustang"
a["year"] = "1964"

fmt.Println(a)

a["year"] = "1970" // Updating an element
a["color"] = "red" // Adding an element

fmt.Println(a)
```

Remove Element from Map

Removing elements is done using the `delete()` function.

```
delete(map_name, key)
```

```
var a = make(map[string]string)
a["brand"] = "Ford"
a["model"] = "Mustang"
a["year"] = "1964"

fmt.Println(a)

delete(a,"year")

fmt.Println(a)
```

Check For Specific Elements in a Map

You can check if a certain key exists in a map using:

```
val, ok :=map_name[key]
```

If you only want to check the existence of a certain key, you can use the blank identifier (_) in place of val.

```
func main() {
    var a = map[string]string{
        "brand": "Ford", "model": "Mustang",
        "year": "1964", "day":"",
    }

    /* Checking for existing key and its value */
    val1, ok1 := a["brand"]
    /* Checking for non-existing key and its value */
    val2, ok2 := a["color"]
    /* Checking for existing key and its value */
    val3, ok3 := a["day"]
    /* Only checking for existing key and not its value */
    _, ok4 := a["model"]

    fmt.Println(val1, ok1)
    fmt.Println(val2, ok2)
    fmt.Println(val3, ok3)
    fmt.Println(ok4)
}
```

Result:

```
Ford true
 false
 true
true
```

Maps Are References

Maps are references to hash tables.

If two map variables refer to the same hash table, changing the content of one variable affect the content of the other.

```
package main
import ("fmt")

func main() {
    var a = map[string]string{
        "brand": "Ford", "model": "Mustang",
        "year": "1964",
    }
    b := a

    fmt.Println(a)
    fmt.Println(b)

    b["year"] = "1970"
    fmt.Println("After change to b:")

    fmt.Println(a)
    fmt.Println(b)
}
```

Result:

```
map[brand:Ford model:Mustang year:1964]
map[brand:Ford model:Mustang year:1964]
After change to b:
map[brand:Ford model:Mustang year:1970]
map[brand:Ford model:Mustang year:1970]
```

Iterating Over Maps

You can use `range` to iterate over maps.

Note the order of the elements in the output.

```
package main
import ("fmt")

func main() {
    a := map[string]int{"one": 1, "two": 2, "three": 3, "four": 4}

    for k, v := range a {
        fmt.Printf("%v : %v, ", k, v)
    }
}
```

Result:

```
one : 1, two : 2, three : 3, four : 4,
```

Iterate Over Maps in a Specific Order

Maps are unordered data structures. If you need to iterate over a map in a specific order, you must have a separate data structure that specifies that order.

```
a := map[string]int{"one": 1, "two": 2, "three": 3, "four": 4}
var b []string                // defining the order
b = append(b, "one", "two", "three", "four")

for k, v := range a {        // loop with no order
    fmt.Printf("%v : %v, ", k, v)
}
fmt.Println()
for _, element := range b {  // loop with the defined order
    fmt.Printf("%v : %v, ", element, a[element])
}
```

Source

- w3schools.com