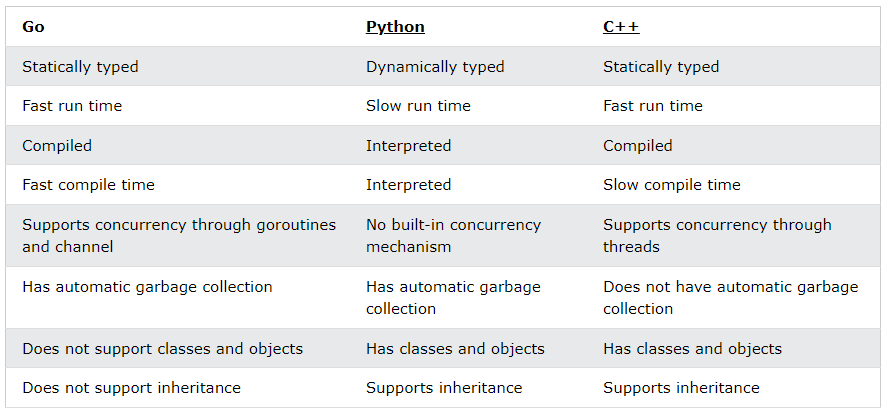
**GoLang**

What is Go?

* Go is a cross-platform, open-source programming language
* Go can be used to create high-performance applications
* Go is a fast, statically typed, **compiled language** that feels like a dynamically typed, interpreted language
* Go was developed at Google by Robert Griesemer, Rob Pike, and Ken Thompson in 2007
* Go's syntax is similar to C++

What is Go Used For?

* Web development (server-side)
* Developing network-based programs
* Developing cross-platform enterprise applications
* Cloud-native development
* System apps to web apps - cloud



**Go Syntax**

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

| package main  import ("fmt")  func main() {  fmt.Println("Hello World!")  } |
| --- |

* In Go, statements are separated by ending a line (hitting the Enter key) or by a semicolon ";".
* The left curly bracket { cannot come at the start of a line.

**Go Comments**

Go Single-line Comments

// This is a comment

Go Multi-line Comments

/\* The code below will print Hello World

to the screen, and it is amazing \*/

**Go Variables**

* 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

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 |
| --- |

**Variable Declaration With Initial Value**

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

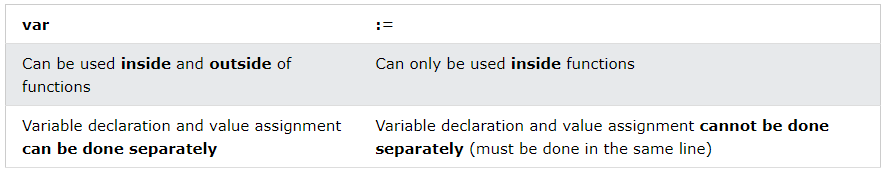
**Variable Declaration Without Initial Value**

| var a string  var b int  var c bool |
| --- |

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

**Difference Between var and :=**



**Multiple Variable Declaration**

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

**Variable Declaration in a Block**

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

**Camel Case**

myVariableName = "John"

**Pascal Case**

MyVariableName = "John"

**Snake Case**

my\_variable\_name = "John"

**Go Constants**

| const CONSTNAME type = value |
| --- |

Typed Constant

| const A int = 1 |
| --- |

Untyped Constant

| const A = 1 |
| --- |

Multiple Constants Declaration

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

**Go Output**

* Print()- The Print() function prints its arguments with their default format.
* Println() -a whitespace is added between the arguments, and a newline is added at the end
* Printf()- 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

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

**Go 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 |
| --- |

**Go Arrays**

Declare an Array

**1. With the var keyword:**

| var array\_name = [length]datatype{values} // here length is defined |
| --- |

or

| var array\_name = [...]datatype{values} // here length is inferred |
| --- |

**2. With the := sign:**

| array\_name := [length]datatype{values} // here length is defined |
| --- |

or

| array\_name := [...]datatype{values} // here length is inferred |
| --- |

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

**Access Elements of an Array**

| fmt.Println(arr1[0]) |
| --- |

**Change Elements of an Array**

| arr1[2] = 50 |
| --- |

**Array Initialization**

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

Result:

[0 0 0 0 0]

[1 2 0 0 0]

[1 2 3 4 5]

**Initialize Only Specific Elements**

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

Result:

[0 10 40 0 0]

**Length of an Array**

| fmt.Println(len(arr1)) |
| --- |

**Go 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.

Ways to create a slice:

* Using the []datatype{values} format

| slice\_name := []datatype{values} |
| --- |

* Create a slice from 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

* Using the make() function

| slice\_name := make([]type, length, capacity) |
| --- |

**Access Elements**

| fmt.Println(slice\_name[0]) |
| --- |

**Change Elements**

| slice\_name[2] = 50 |
| --- |

**Append Elements**

| slice\_name = append(slice\_name, element1, element2, ...) |
| --- |

**Append One Slice To Another Slice**

| slice3 = append(slice1, slice2...) |
| --- |

**Change The Length**

| arr1 := [6]int{9, 10, 11, 12, 13, 14} // An array  slice\_name = arr1[1:3] // Change length by re-slicing the array  slice\_name2 = append(slice\_name, 20, 21, 22, 23) // Change length by appending items |
| --- |

**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.

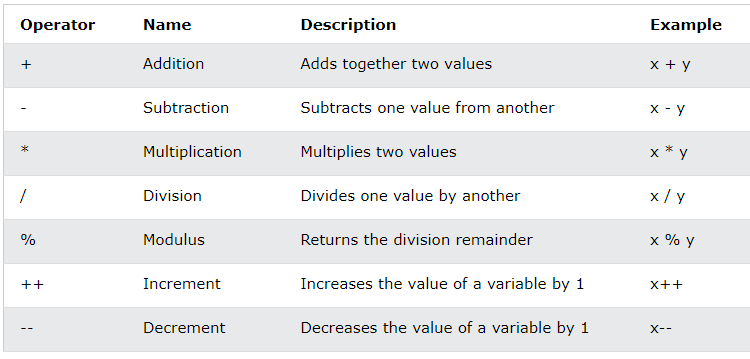
The copy() function creates a new underlying array with only the required elements for the slice. This will reduce the memory used for the program.

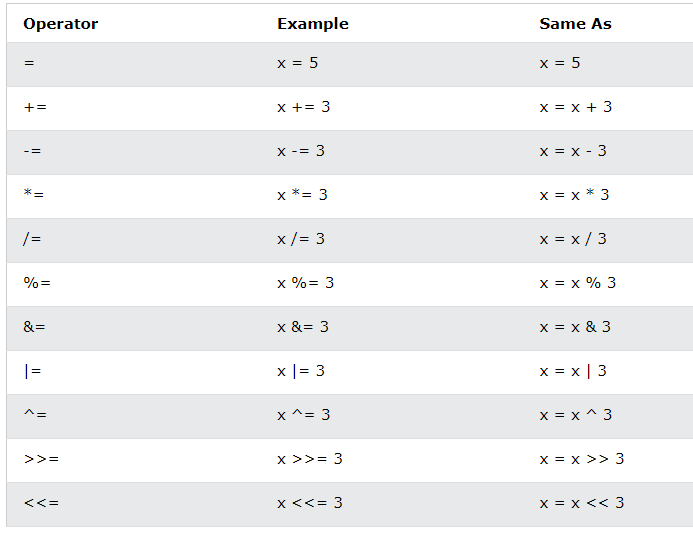
copy(dest, src)

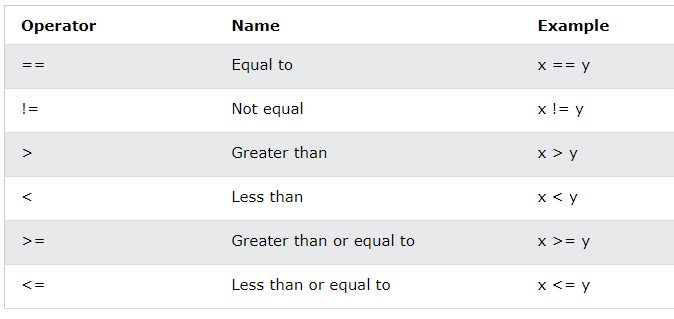
The copy() function takes in two slices dest and src, and copies data from src to dest. It returns the number of elements copied.

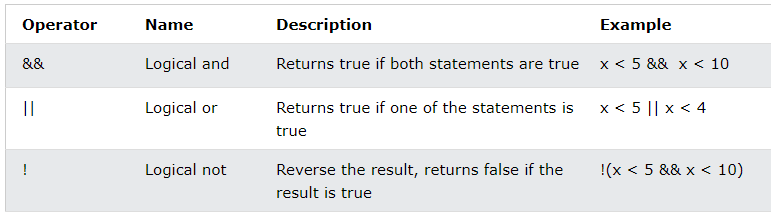
**Go Operators**

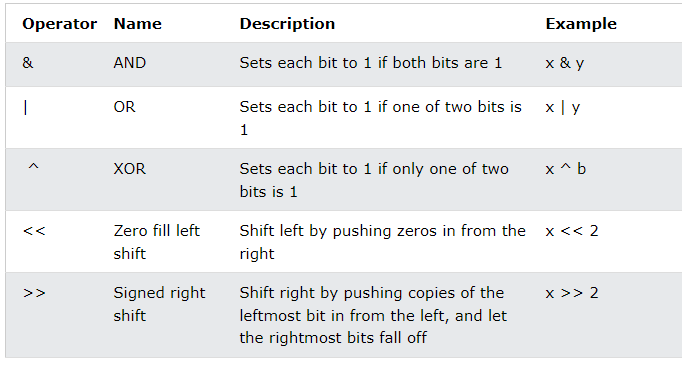
var a = 15 + 25











**Go Conditions**

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

**Nested if**

| 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.")  } |
| --- |

**Go Switch**

**Single-Case switch**

| switch expression {  case x:  // code block  case y:  // code block  case z:  ...  default:  // code block  } |
| --- |

**The Multi-case switch**

| 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")  } |
| --- |

**Go Loops**

**for Loop**

| for i:=0; i < 5; i++ {  fmt.Println(i)  } |
| --- |

**continue Statement**

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

**break Statement**

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

**Nested Loops**

| adj := [2]string{"big", "tasty"}  fruits := [3]string{"apple", "orange", "banana"}  for i:=0; i < len(adj); i++ {  for j:=0; j < len(fruits); j++ {  fmt.Println(adj[i],fruits[j])  }  } |
| --- |

Result :

big apple

big orange

big banana

tasty apple

tasty orange

tasty banana

**Range Keyword**

It returns both the index and the value.

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

Result :

0 apple

1 orange

2 banana

**Go Functions**

| package main  import ("fmt")  func myMessage() {  fmt.Println("I just got executed!")  }  func main() {  myMessage() // call the function  } |
| --- |

**Parameters and Arguments**

| func familyName(fname string) {  fmt.Println("Hello", fname)  }  func main() {  familyName("World!")  } |
| --- |

**Return Values**

| func myFunction(x int, y int) int {  return x + y  }  func main() {  fmt.Println(myFunction(1, 2))  } |
| --- |

**Go 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**

| type Person struct {  name string  age int  job string  salary int  } |
| --- |

**Access Struct Members**

| package main  import ("fmt")  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  // Access and print Pers1 info  fmt.Println("Name: ", pers1.name)  fmt.Println("Age: ", pers1.age)  fmt.Println("Job: ", pers1.job)  fmt.Println("Salary: ", pers1.salary)  // Access and print Pers2 info  fmt.Println("Name: ", pers2.name)  fmt.Println("Age: ", pers2.age)  fmt.Println("Job: ", pers2.job)  fmt.Println("Salary: ", pers2.salary)  } |
| --- |

Result:

Name: Hege

Age: 45

Job: Teacher

Salary: 6000

Name: Cecilie

Age: 24

Job: Marketing

Salary: 4500

**Go 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.

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

**Creating Maps Using Using make()Function:**

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

**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.

**Accessing Map Elements**

| value = map\_name[key] |
| --- |

**Updating and Adding Map Elements**

| map\_name[key] = value |
| --- |

**Remove Element from Map**

| delete(map\_name, key) |
| --- |

**Check For Specific Elements in a Map**

| val, ok :=map\_name[key] |
| --- |