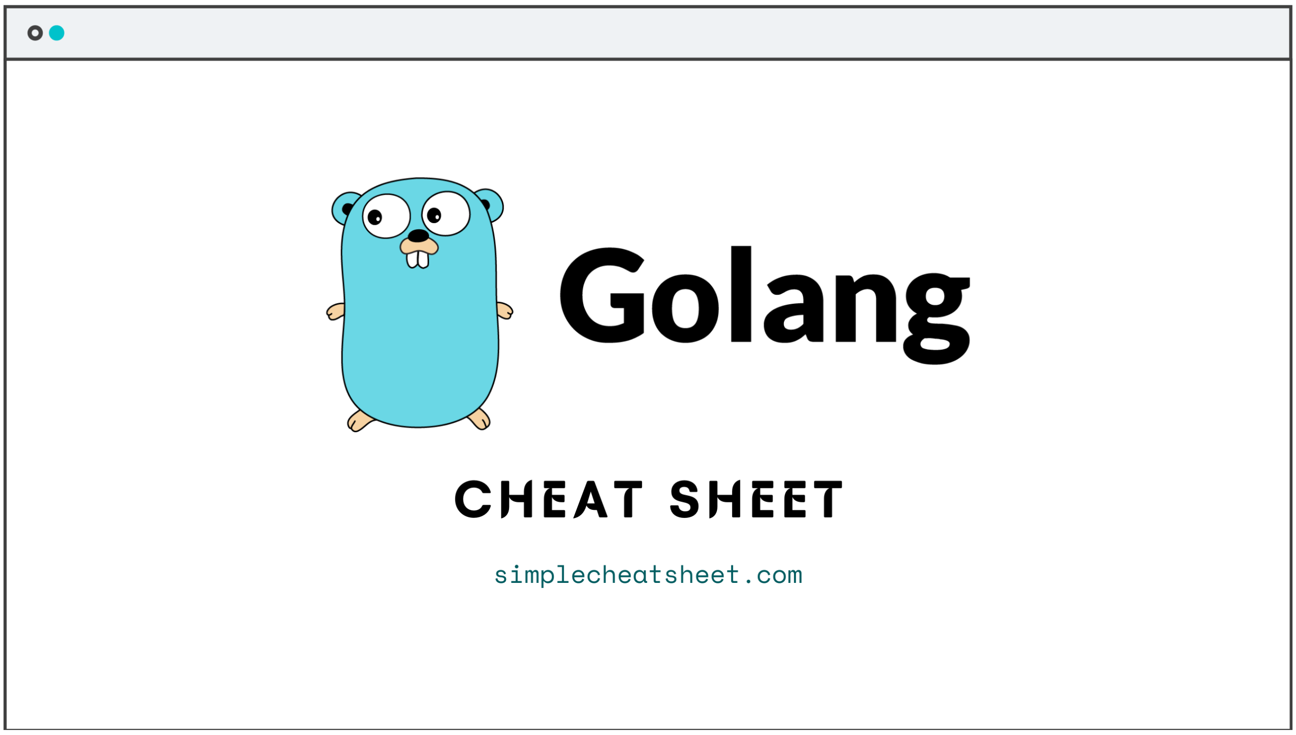
Golang Cheat Sheet



This cheat sheet provided basic syntax and methods to help you using **Golang**. **Go** is a statically typed, compiled programming language designed at Google by Robert Griesemer, Rob Pike, and Ken Thompson. Go is syntactically similar to C, but with memory safety, garbage collection, structural typing, and CSP-style concurrency. The language is often referred to as "Golang" because of its domain name, golang.org, but the proper name is Go.

Go: How to setup < https://simplecheatsheet.com/go-how-to-setup/>

Download Go Archive

Go 1.14 < https://golang.org/doc/go1.14> (February 2020)

Up ↑

OS	**Archive name**
Windows	go1.14.windows-amd64.msi
Linux	go1.14.linux-amd64.tar.gz
Mac	go1.14.darwin-amd64-osx10.8.pkg
FreeBSD	go1.14.freebsd-amd64.tar.gz

Installation

Download the archive < https://golang.org/dl/> and extract it into `/usr/local`, creating a Go tree in `/usr/local/go`. For example:

```
tar -C /usr/local -xzf go$VERSION.$OS-$ARCH.tar.gz
```

Add /usr/local/go/bin to the PATH environment variable.

OS	**Output**
Linux	export PATH = $PATH:/usr/local/go/bin
Mac	export PATH = $PATH:/usr/local/go/bin
FreeBSD	export PATH = $PATH:/usr/local/go/bin

On Windows:

Create a go file named test.go in **C:\>Go_WorkSpace**.

File: test.go

```
package main

import "fmt"

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

Now run test.go to see the result –

```
C:\Go_WorkSpace>go run test.go
```

Up ↑

=> Output

```
Hello, World!
```

Go: Program Structure < https://simplecheatsheet.com/go-program-structure/>

Hello World

Now, first add the package main in your program:

```
package main
```

After adding main package import “fmt” package in your program:

```
import "fmt"
```

Now write the code in the main function to print hello world in Go language:

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

Go: Identifiers < https://simplecheatsheet.com/go-identifiers/>

In Go language, an identifier can be a variable name, function name, constant, statement labels, package name, or types.

Example:

```
package main
import "fmt"

func main() {

 var name = "GeeksforGeeks"

}
```

There is total of three identifiers available in the above example:

- **main:** Name of the package
- **main:** Name of the function

Up ↑

- **name:** Name of the variable

Go: Keywords < https://simplecheatsheet.com/go-keywords/>

There are total **25 keywords** present in the Go language as follows:

break	default	func	in
case	defer	Go	m
chan	else	Goto	pa
const	fallthrough	if	ra
continue	for	import	re

Go: Data Types < https://simplecheatsheet.com/go-data-types/>

Basic Data Types

Types	**Description**
Boolean types	They are boolean types and consists of the two predefined constants: (a) true (b) false
Numeric types	They are again arithmetic types and they represent a) integer types or b) floating-point values throughout the program.
String types	A string type represents the set of string values. Its value is a sequence of bytes. Strings are immutable types that are once created, it is not possible to change the contents of a string. The predeclared string type is a string.
Derived types	They include (a) Pointer types, (b) Array types, (c) Structure types, (d) Union types and (e) Function types f) Slice types g) Interface types h) Map types i) Channel Types

Up ↑

Integer Types

uint8	Unsigned 8-bit integers (0 to 255)
uint16	Unsigned 16-bit integers (0 to 65535)
uint32	Unsigned 32-bit integers (0 to 4294967295)
uint64	Unsigned 64-bit integers (0 to 18446744073709551615)
int8	Signed 8-bit integers (-128 to 127)
int16	Signed 16-bit integers (-32768 to 32767)
int32	Signed 32-bit integers (-2147483648 to 2147483647)
int64	Signed 64-bit integers (-9223372036854775808 to 9223372036854775807)

Floating-Point Types

float32	IEEE-754 32-bit floating-point numbers
float64	IEEE-754 64-bit floating-point numbers
complex64	Complex numbers with float32 real and imaginary parts
complex128	Complex numbers with float64 real and imaginary parts

---

Go: Variables < https://simplecheatsheet.com/go-variables/>

There basic variable types:

Up ↑

Type	Description
byte	Typically a single octet(one byte). This is a byte type.
int	The most natural size of integer for the machine.
float32	A single-precision floating-point value.

Declaring a Variable

Using var keyword:

```
var variable_name type = expression
```

Using short variable declaration:

```
variable_name:= expression
```

Go: Constants < https://simplecheatsheet.com/go-constants/>

Declare

Example:

```
package main

import "fmt"

const PI = 3.1412

func main() {
    const SC = "Simplecheatsheet"
    fmt.Println("Hello", SC)

    fmt.Println("Happy", PI, "Day")

    const Correct= true
    fmt.Println("Go rules?", Correct)
}
```

=> Output

Up ↑

```
Hello Simplecheatsheet
Happy 3.14 Day
Go rules? true
```

Integer Constant

some examples of Integer Constant:

```
85          /* decimal */
0213        /* octal */
0x4b        /* hexadecimal */
30          /* int */
30u         /* unsigned int */
30l         /* long */
30ul        /* unsigned long */
212          /* Legal */
215u         /* Legal */
0xFeeL       /* Legal */
078          /* Illegal: 8 is not an octal digit */
032UU        /* Illegal: cannot repeat a suffix */
```

Floating Type Constant

Following are the examples of Floating type constant:

```
3.14159       /* Legal */
314159E-5L    /* Legal */
510E          /* Illegal: incomplete exponent */
210f          /* Illegal: no decimal or exponent */
.e55          /* Illegal: missing integer or fraction */
```

String Literals

Syntax

```
type _string struct {
    elements *byte // underlying bytes
    len      int   // number of bytes
}
```

Some examples of string literals:

```
"hello, simplecheatsheet"
```

```
"hello, \
simplecheatsheet"
```

```
"hello, " "simple" "cheatsheet"
```

Up ↑

The *const* Keyword

```
const variable type = value;
```

Example:

```
package main

import "fmt"

func main() {
   const LENGTH int = 4
   const WIDTH int = 5
   var area int

   area = LENGTH * WIDTH
   fmt.Printf("value of area : %d", area)
}
```

=> Output

```
value of area : 20
```

Escape Sequence

A list of some of such escape sequence codes –

Up ↑

Escape sequence	Meaning
\\	\ character
\’	‘ character
\”	” character
\?	? character
\a	Alert or bell
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\ooo	Octal number of one to three digits
\xhh . . .	Hexadecimal number of one or more digits

Go: Operators < https://simplecheatsheet.com/go-operators/>

Arithmetic Operators

Up ↑

Operator	Description	Example
+	Adds two operands	x + y
–	Subtracts second operand from the first	x – y
*	Multiplies both operands	x * y
/	Divides the numerator by the denominator.	x / y
%	Modulus operator; gives the remainder after an integer division.	x % y
++	Increment operator. It increases the integer value by one.	x++
—	Decrement operator. It decreases the integer value by one.	x–

Relational Operators

Assume variable **x** holds 15 and variable **y** holds 45, then:

Up ↑

Operator	Description	Example
==	It checks if the values of two operands are equal or not; if yes, the condition becomes true.	(A == B) is not true.
!=	It checks if the values of two operands are equal or not; if the values are not equal, then the condition becomes true.	(A != B) is true.
>	It checks if the value of left operand is greater than the value of right operand; if yes, the condition becomes true.	(A > B) is not true.
<	It checks if the value of left operand is less than the value of the right operand; if yes, the condition becomes true.	(A < B) is true.
>=	It checks if the value of the left operand is greater than or equal to the value of the right operand; if yes, the condition becomes true.	(A >= B) is not true.
<=	It checks if the value of left operand is less than or equal to the value of right operand; if yes, the condition becomes true.	(A <= B) is true.

Logical Operators

Assume variable **x** holds 1 and variable **y** holds 0, then:

Up ↑

Operator	**Description**	**Example**
&&	Called Logical **AND** operator. If both the operands are non-zero, then the condition becomes true.	(x && y) is false.
||	Called Logical **OR** Operator. If any of the two operands are non-zero, then the condition becomes true.	(x || y) is true.
!	Called Logical **NOT** Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(x && y) is true.

Bitwise Operators

Bitwise operators work on bits and perform the bit-by-bit operation. The truth tables for &, |, and ^ are as follows:

p	**q**	**p & q**	**p**
0	0	0	0
0	1	0	1
1	1	1	1
1	0	0	1

In Go language, there are 6 bitwise operators which work at bit level or used to perform bit by bit operations:

Operator	Description	Example
&	Binary **AND** Operator copies a bit to the result if it exists in both operands.	(x & y) will give 12, which is 0000 1100
|	Binary **OR** Operator copies a bit if it exists in either operand.	(x | y) will give 61, which is 0011 1101
^	Binary **XOR** Operator copies the bit if it is set in one operand but not both.	(x ^ y) will give 49, which is 0011 0001
<<	Binary **Left Shift** Operator. The left operand’s value is moved left by the number of bits specified by the right operand.	x << 2 will give 240 which is 1111 0000
>>	Binary **Right Shift** Operator. The left operand’s value is moved right by the number of bits specified by the right operand.	x >> 2 will give 15 which is 0000 1111
&^	Binary **AND NOT** Operator. This is a bit clear operator.	

Assignment Operators

Up ↑

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C – A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2

Up ↑

&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

Miscellaneous Operators

- &: This operator returns the address of the variable.
- *: This operator provides a pointer to a variable.
- <-:The name of this operator is received. It is used to receive a value from the channel.

Operators Precedence

Up ↑

Category	Operator	Associativity
Postfix	() [] -> . ++ – –	Left to right
Unary	+ – ! ~ ++ – – (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ –	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Go: Loops < https://simplecheatsheet.com/go-loops/>

for Loop

It executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

```
for [condition |  ( init; condition; increment ) | Range] {
   statement(s);
}
```

Up ↑

Nested for Loops

These are one or multiple loops inside any for loop.

```
for [condition |  ( init; condition; increment ) | Range] {
   for [condition |  ( init; condition; increment ) | Range] {
      statement(s);
   }
   statement(s);
}
```

Simple range in for loop

```
for i, j:= range rvariable{
   // statement..
}
```

Using for loop for strings

A for loop can iterate over the Unicode code point for a string.

```
for index, chr:= range str{
     // Statement..
}
```

For Maps

A for loop can iterate over the key and value pairs of the map.

```
for key, value := range map {
     // Statement..
}
```

For Channel

A for loop can iterate over the sequential values sent on the channel until it closed.

```
for item := range Chnl {
     // statements..
}
```

break statement

It terminates a **for loop** or **switch** statement and transfers execution to the statement immediately following the for loop or switch.

Up ↑

```
break;
```

continue statement

It causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

```
continue;
```

goto statement

It transfers control to the labeled statement.

```
goto label;
..
.
label: statement;
```

Go: Functions < https://simplecheatsheet.com/go-functions/>

Function Declaration

Syntax:

```
func function_name( [parameter list] ) [return_types]
{
   body of the function
}
```

The declaration of the function contains:

- `func`**:** It is a keyword in Go language, which is used to create a function.
- `function_name`**:** It is the name of the function.
- `Parameter-list`**:** It contains the name and the type of function parameters.
- `Return_type` : It is optional and it contains the types of values that function returns. If you are using return_type in your function, then it is necessary to use a return statement in your function.

Function Arguments

Up ↑

Call by value

By default, Go programming language uses *call by value* method to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function

```
func swap(int x, int y) int {
   var temp int

   temp = x /* save the value of x */
   x = y    /* put y into x */
   y = temp /* put temp into y */

   return temp;
}
```

Call by reference

The *call by reference* method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call.

```
func swap(x *int, y *int) {
   var temp int

   temp = *x    /* save the value at address x */
   *x = *y      /* put y into x */
   *y = temp    /* put temp into y */
}
```

Go: Arrays < https://simplecheatsheet.com/go-arrays/>

Declaring Arrays

```
var array_name[length]Type

or

var array_name[length]Typle{item1, item2, item3, ...itemN}
```

Multi-Dimensional Array

```
Array_name[Length1][Length2]..[LengthN]Type
```

Example:

Up ↑

```
a := [2][3]int{
   {0, 1, 2} ,   /*  initializers for row indexed by 0 */
   {3, 4, 5} ,   /*  initializers for row indexed by 1 */
}
```

Go: Switch Statement < https://simplecheatsheet.com/go-switch-statement/>

Expression Switch

Syntax:

```
switch optstatement; optexpression{
case expression1: Statement..
case expression2: Statement..
...
default: Statement..
}
```

Example:

```
package main

import "fmt"

func main() {

        switch color:=3; color{
        case 1:
        fmt.Println("Red")
        case 2:
        fmt.Println("Blue")
        case 3:
        fmt.Println("Black")
        case 4:
        fmt.Println("Pink")
        case 5:
        fmt.Println("Orange")
        default:
        fmt.Println("Invalid")
   }
}
```

=> Output

Up ↑

```
Black
```

Type Switch

Syntax:

```
switch optstatement; typeswitchexpression{
case typelist 1: Statement..
case typelist 2: Statement..
...
default: Statement..
}
```

Example:

```
package main

import "fmt"

func main() {
   var value interface{}
   switch q:= value.(type) {
      case bool:
      fmt.Println("value is of boolean type")
      case float64:
      fmt.Println("value is of float64 type")
      case int:
      fmt.Println("value is of int type")
      default:
      fmt.Printf("value is of type: %T", q)
   }
}
```

Output:

```
value is of type: <nil>
```

Go: Pointers < https://simplecheatsheet.com/go-pointers/>

Consider the following example, which will print the address of the variables defined

Up ↑

```
package main

import "fmt"

func main() {

   i, j := 42, 2701

   p := &i         // point to I
   fmt.Println(*p) // read i through the pointer
   *p = 21         // set i through the pointer
   fmt.Println(i)  // see the new value of I

   p = &j         // point to j
   *p = *p / 37   // divide j through the pointer
   fmt.Println(j) // see the new value of j
}
```

=> Ouput:

```
42
21
73
```

Declaring a pointer:

```
var var_name *var-type
```

Example:

```
var s *string /* pointer to a string */
```

Initialization of Pointer:

```
// normal variable declaration
var a = 45

// Initialization of pointer s with
// memory address of variable a
var s *int = &a
```

Nil Pointers:

The nil pointer is a constant with a value of zero defined in several standard libraries.

Example:

Up ↑

```
package main

import "fmt"

func main() {
   // taking a pointer
   var  s *int

   // displaying the result
   fmt.Printf("s = ", s)
}
```

=> Output

```
s =  <nil>
```

[Go: Defer Keyword < https://simplecheatsheet.com/go-defer-keyword/>](https://simplecheatsheet.com/go-defer-keyword/)

You can create a deferred method, or function, or anonymous function by using the defer keyword.

```
// Function
defer func func_name(parameter_list Type)return_type{
// Code
}

// Method
defer func (receiver Type) method_name(parameter_list){
// Code
}

defer func (parameter_list)(return_type){
// code
}()
```

[Go: Structures < https://simplecheatsheet.com/go-structures/>](https://simplecheatsheet.com/go-structures/)

Defining a Structure

Syntax:

Up ↑

```
type struct_variable_type struct {
   member definition;
   member definition;
   ...
   member definition;
}
```

Example:

```
 type Address struct {
      name string
      street string
      city string
      state string
      Pincode int
}
```

Or

```
type Address struct { name, street, city, state string Pincode int }
```

Go: Slices < https://simplecheatsheet.com/go-slices/>

Declaration of Slice

Syntax:

```
[]T
or
[]T{}
or
[]T{value1, value2, value3, ...value n}
```

Here, T is the type of the elements. For example:

```
var my_slice[]int
```

Components of Slice

Three components of a slice:

- **Pointer:** The pointer is used to points to the first element of the array that is accessible through the slice. Here, it is not necessary that the pointed element is

Up ↑

the first element of the array.

- **Length:** The length is the total number of elements present in the array.
- **Capacity:** The capacity represents the maximum size up to which it can expand.

Go: Maps < https://simplecheatsheet.com/go-maps/>

You must use **make** function to create a map.

```
/* declare a variable, by default map will be nil*/ var map_variable map

/* create a map as nil map can not be assigned any value*/
map_variable = make(map[key_data_type]value_data_type)
```

delete() Function

```
/* delete an entry */
   delete(map_variable,value_data);
```

Go: Goroutines < https://simplecheatsheet.com/go-goroutines/>

Create a Goroutine

Syntax:

```
func name(){
// statements
}

// using go keyword as the
// prefix of your function call
go name()
```

Anonymous Goroutine

Syntax:

```
// Anonymous function call
go func (parameter_list){
// statement
}(arguments)
```

Go: Recursion < https://simplecheatsheet.com/go-recursion/>

Recursion is the process of repeating items in a self-similar way.

Syntax:

```
func recursion() {
   recursion() /* function calls itself */
}
func main() {
   recursion()
}
```

Go: Channel Cheat Sheet < https://simplecheatsheet.com/go-channel-cheat-sheet/>

Creating a Channel

Syntax:

```
var Channel_name chan Type
```

Or you can also create a channel using make() function

```
channel_name:= make(chan Type)
```

Closing a Channel

```
close(channelname)
```

Go: Interfaces Cheat Sheet < https://simplecheatsheet.com/go-interfaces-cheat-sheet/>

Syntax

Define an interface

Up ↑

```
type interface_name interface {
   method_name1 [return_type]
   ...
   method_namen [return_type]
}
```

Define a struct

```
type struct_name struct {
   /* variables */
}
```

Implement interface methods

```
func (struct_name_variable struct_name) method_name1() [return_type] {
   /* method implementation */
}
...
func (struct_name_variable struct_name) method_namen() [return_type] {
   /* method implementation */
}
```

Up ↑