Golang Basics Cheat Sheet

by Dmytro (Dmy) via cheatography.com/163964/cs/34356/

Declarations

```
var foo string // declaration
with zero value
var foo int = 191 // declar -
ation with initial value
foo := 191 // shorthand
foo, bar := 191 , 23 //
shorthand multiple
const c = "This is a consta nt"
const (
        width = 1920
        height= 1080
)
```

Types

Basic Types:

int, int8, int16, int32, int64
uint, uint8, uint16, uint32,
uint64, uintptr
float32, float64
complex64, complex128
byte // alias for uint8. ASCII
char. var a byte = 'a'
rune // alias for int32. Unicode
Char
string
bool

Composit Types

struct, array, slice, map,
channel

Interfaces

Describe behavior of data type

Functions

```
func main() { // main func
contains no params and no
returns
```

Functions (cont)

```
func foo(param0 int, param1,
param2 string, ...) (int,
string, ...) { // func can take
multiple params and return
multiple values
func foo() (res int, code int) {
// named return
   res = 2
 code = 3
    return
}
res, code := foo() // execute
func and store returned values
func foo(a, b, arg ...int) {} //
variable number of parameters
// factory
func add(a int) (func(b int)
int) { // return a function
        return func(b int) int {
                return a + b
       }
}
// defer - executed when func
exits
// When many defer’s are
declared in the code, they are
executed in the inverse order.
func foo() {
        fmt.Pr int f("1 ")
        defer f() // function
deferred, will be executed latst
and print 3.
        fmt.Pr int f("2 ")
}
// The defer allows us to
guarantee that certain clean-up
tasks are performed before we
return from a function, for
example:
```

Functions (cont)

```
// Closing a file stream,
Unlocking a locked resource (a
mutex), log, ,losing a database
connection
// Tracing example
func foo() {
        cal lTr ace ("fo o")
        defer callUn tra ce( " -
foo ") // untracing via defer
        fmt.Pr int ln( "foo is
being execut ed")
}
// Debugging values
func foo(s string )(res int, err
error) {
        defer func() {
                log.Pr int -
f("f oo(%q) = >%d, %v", s, res,
err)
       }()
        return 5, nil
}
// No Method overload, But we
can overload receiver
func (a *Obj1) Add(b Int) Int
func (a *Obj2) Add(b Int) Int
```

Misc

Go doesn't have casting. Go uses
Conversion - bytes need to be
copied to a new
memory location for the new
repres ent ation.

Enums

```
const (
   ONE= iota
   TWO
    THREE
)
// Print file full path and line
where the code is executed
```

Misc (cont)

```
log.Se tFl ags (lo g.L lon -
gfile)
        whe reAmI := log.Print
        whe reAmI()
// Size
size := unsafe.Si zeo f(T{})
```

Interface

```
type ReadWrite interface {
    Read(b Buffer) bool
    Write(b Buffer) bool
}
type Lock interface {
    Lock()
    Unl ock()
}
switch t := someVa r.( type) {
       case *Game:
            fmt.Pr int f("G -
ame ")
       case *Unit:
            fmt.Pr int f("U -
nit ")
        def ault:
            fmt.Pr int f("U -
nex pected type"t)
       }
}
type File interface {
    Rea dWrite
   Lock
    Close()
}
// Check dynamic var if is of
type
if v, ok := someVa r.(T); ok {
// ...
```

Interface (cont)

```
}
```

Fmt

TBA

String and Rune

```
s := "line one \n line two" //
interpreted string \n - newline
s :=`line one \n line two` //
raw string. no interp ret ation.
no new line for \n
l := len(s) // get string length
```

strings pkg contains utilities

```
// HasPrefix, HasSuffix,
Contains, Index, LastIndex,
Replace,
// Count, Repeat, ToLower,
ToUpper, Trim, TrimSpace,
// Fields /Split, Join, Read,
ReadByte, ReadRune
```

strconv pkg contains convertion utilities

```
// Itoa, Format Float, Atoi,
ParseFloat
v, err := strcon v.A toi(s)
! Go strings are immutable -
behave like read-only byte
slice.
// To insert /modify use:
rune(u nic ode )/b yte (ASCII)
slice
bs := []byte (str)
bs[0] = bs[1]
str = string(bs)
```

Struct

```
type foo struct { // define new
type
   name string
   age int
}
var f foo // declare instance
and init to zero value
f := foo { // create type
instance with literal cunstr -
uction type
    name: " Joh n",
   age: 30
}
f := struct { // declare a var
of unnamed type
   age: int
}{
   age: 35
}
// Compos ition
type s1struct {
   age int
   name string
}
type s2 struct {
    sec ondName string
   s1
}
```

Array

```
var foo [7]int // create zero
value array length of 7
foo := [3]int{10, 20, 30} //
define array length of 3 with
values
```

Array (cont)

```
foo := [...]i nt{10, 20} //
compiler calculate length of the
array
foo := [10]st ring{3: " Thr ee",
4: " Fou r"}
foo[1] = 193 // write value
a := foo[1] // read value
```

Time

```
t := time.Now()
// Duration, Since, Location,
Format
t := time.N ow( ).UTC()
fmt.Pr int ln( t.F orm at( "01
Jan 2022 18:34")) // 12 Oct 2022
01:23
```

Slice

Slice - is a data structure with 3 fields:

1. * - a pointer to the underlying array
2. len - the length of the slice (indexable by len)
3. cap - capacity of the slice (how long a slice can become)

```
var s []string // zero value
slice
var s []string= arr[st -
art :end] // slice from array
s := []int{ 1,2,3} // literal
s:= make([ ]st ring, len, cap)
// create with make
```

Multid ime ntional

```
screen:= [][]int{}
var screen [w][h]int
screen [2][5] = 1 // write
v := screen [2][5] // read
s = s[1:] // cut 1st element
s = s[:len (s)-1] // cut last
element
```

Slice (cont)

```
s = s[3:5] // cut slice - new
one from 3rd inclusive to 5th
exclusive
```

COPY

```
func copy(dst, src []T) int
s := copy(sTo, sFrom)
```

ADD/append

```
func append (s[]T, x ...T) []T
s = append(s, " add Val ue") //
append value
s = append(s1, s2...) // append
slice
```

OPERATIONS

```
// Delete at idex i (from i to
j)
s = append (s[:i], s[i+1: ]...)
s = append (s[:i], s[j:]...)
// Insert a value/ range at
index i
s = append (s[:i], append ([] -
str ing {"va lue ToI nse rt"},
s[i:]...)...)
s = append (s[:i], append ([] -
str ing {"va lue ToI nse rt",
" val ue2 ToI nse rt"},
s[i:]...)...)
s = append (s[:i], append(s2,
s[i:]...)...) // insert existing
lice s2
// Stack
val, s = s[len( s)-1:], s[:len -
(a)-1] // Pop
s = append(s, val) // Push
// Queue
val, q = q[:1], q[1:] // Dequeu
q = append ([] str ing {"va -
lue "}, q...) // Enqueue
// Swap
s[j], s[i] = s[i], s[j]
```

Sort

```
sort pkg. ex: sort.S tri ngs(s)
```

Maps

```
var m map[string]int // zero
value - nil
m := make(m ap[ str ing ]int) //
short
m := map[in t]s tri ng{ 1:"O -
ne", 2: " Two "}
m[key] = val
val:= m[key]
if _, ok := m[key]; ok { // if
key is present
}
delete(m, key) // delete record
// we can range on map with key,
value
len(map) // get length
// Unlike arrays, maps grow
dynami cally to accomm odate new
key-values that are added;
// they have no fixed or maximum
size. However, you can
optionally indicate
// an initial capacity cap for
the map, as in:
m := make(m ap[ str ing ]int,
cap) // m := make(m ap[ str -
ing ]int, 100)
```

Bytes

TBA

Control Structures (if, switch)

If

```
if x < 0 {
        return err
} else if x == 0 {
        f.C lose()
} else {
     a = 5
}
// Pattern to use with
true/false condition
if n % 2 == 0 {
```

Control Structures (if, switch) (cont)

```
        return " eve n"
}
return " odd "
// with initia liz ation
statement
if val := getVal ue(); val > max
{
    return err
}
Switch
switch var1 {
case " a":
   f()
case " b", " c", " d": //
multiple values
   f2()
default:
   f3()
}
switch a,b := 3, 4; { // with
initia liz ation
case a > 0: fallth rough// to
execte next
case b < 3:
       f()
}
// switch with multiple
conditions
switch {
case i < 0:
       f1()
case i == 0 && a > 3:
       f2()
case i > 0:
       f3()
}
```

Loop

```
For
for i := 0; i < 100; i++ {}
for i, j := 0, N; i < j; i, j =
i+1, j-1 { } // multiple
counters
// nested
for i:=0; i<5; i++ {
       for j:=0; j<10; j++ {
       }
}
// While like
for i < 5 {
   i += 1
}
// infinite loop
for { } // another form: for
true { }
//range
for i, val:= range str {
    fmt.Pr int f("C har acter on
position %d is: %c \n", i, val)
}
// we can use break and continue
to skip execution
// there is also goto and label
```