devhints.io

Go cheatsheet

7–9 minutes

#Getting started

Hello world

hello.go

```
package main

import "fmt"

func main() {
  message := greetMe("world")
  fmt.Println(message)
}

func greetMe(name string) string {
  return "Hello, " + name + "!"
}
```

```
$ go build
```

Or try it out in the Go repl, or A Tour of Go.

Variables

Variable declaration

```
var msg string
var msg = "Hello, world!"
var msg string = "Hello, world!"
var x, y int
var x, y int = 1, 2
var x, msg = 1, "Hello, world!"
msg = "Hello"
```

Declaration list

```
var (
  x int
  y = 20
  z int = 30
  d, e = 40, "Hello"
  f, g string
)
```

Shortcut of above (Infers type)

```
msg := "Hello"
x, msg := 1, "Hello"
```

Constants

```
const Phi = 1.618
const Size int64 = 1024
const x, y = 1, 2
const (
  Pi = 3.14
  E  = 2.718
)
const (
  Sunday = iota
  Monday
  Tuesday
  Wednesday
  Thursday
  Friday
  Saturday
)
```

Constants can be character, string, boolean, or numeric values.

See: Constants

#Basic types

Strings

```
str := "Hello"
```

```
str := `Multiline
string`
```

Strings are of type `string`.

Numbers

Typical types

```
num := 3          // int
num := 3.         // float64
num := 3 + 4i     // complex128
num := byte('a')  // byte (alias for uint8)
```

Other types

```
var u uint = 7        // uint (unsigned)
var p float32 = 22.7  // 32-bit float
```

Arrays

```
// var numbers [5]int
```

```
numbers := [...]int{0, 0, 0, 0, 0}
```

Arrays have a fixed size.

Slices

```
slice := []int{2, 3, 4}
```

```
slice := []byte("Hello")
```

Slices have a dynamic size, unlike arrays.

Pointers

```
func main () {
  b := *getPointer()
  fmt.Println("Value is", b)
}
```

```
func getPointer () (myPointer *int) {
  a := 234
  return &a
}
```

```
a := new(int)
*a = 234
```

Pointers point to a memory location of a variable. Go is fully garbage-collected.

See: Pointers

#Flow control

Conditional

```
if day == "sunday" || day == "saturday" {
  rest()
} else if day == "monday" && isTired() {
  groan()
} else {
  work()
}
```

See: If

Statements in if

```
if _, err := doThing(); err != nil {
  fmt.Println("Uh oh")
}
```

A condition in an `if` statement can be preceded with a statement before a `;`. Variables declared by the statement are only in scope until the end of the `if`.

See: If with a short statement

Switch

```
switch day {
  case "sunday":
    // cases don't "fall through" by default!
    fallthrough

  case "saturday":
    rest()

  default:
    work()
}
```

See: Switch

For loop

```
for count := 0; count <= 10; count++ {
  fmt.Println("My counter is at", count)
}
```

See: For loops

For-Range loop

```
entry := []string{"Jack","John","Jones"}

for i, val := range entry {
  fmt.Printf("At position %d, the character %s is present\n", i, val)
}
```

See: For-Range loops

While loop

```
n := 0
x := 42
for n != x {
  n := guess()
}
```

See: Go’s “while”

#Functions

Lambdas

```
myfunc := func() bool {
  return x > 10000
}
```

Functions are first class objects.

Multiple return types

```
a, b := getMessage()
```

```
func getMessage() (a string, b string) {
  return "Hello", "World"
}
```

Named return values

```
func split(sum int) (x, y int) {
  x = sum * 4 / 9
  y = sum - x
  return
}
```

By defining the return value names in the signature, a `return` (no args) will return variables with those names.

See: Named return values

#Packages

Importing

```
import "fmt"
import "math/rand"
```

```
import (
  "fmt"        // gives fmt.Println
  "math/rand"  // gives rand.Intn
)
```

Both are the same.

See: Importing

Aliases

```
import r "math/rand"
```

```
r.Intn()
```

Exporting names

```
func Hello () {
  ···
}
```

Exported names begin with capital letters.

See: Exported names

### Packages

```
package hello
```

Every package file has to start with `package`.

#Concurrency

Goroutines

```
func main() {
  // A "channel"
  ch := make(chan string)

  // Start concurrent routines
  go push("Moe", ch)
  go push("Larry", ch)
  go push("Curly", ch)

  // Read 3 results
  // (Since our goroutines are concurrent,
  // the order isn't guaranteed!)
  fmt.Println(<-ch, <-ch, <-ch)
}
```

```
func push(name string, ch chan string) {
  msg := "Hey, " + name
  ch <- msg
}
```

Channels are concurrency-safe communication objects, used in goroutines.

See: Goroutines, Channels

Buffered channels

```
ch := make(chan int, 2)
ch <- 1
ch <- 2
ch <- 3
// fatal error:
// all goroutines are asleep - deadlock!
```

Buffered channels limit the amount of messages it can keep.

See: Buffered channels

Closing channels

Closes a channel

```
ch <- 1
ch <- 2
ch <- 3
close(ch)
```

Iterates across a channel until its closed

```
for i := range ch {
  ···
}
```

Closed if `ok == false`

```
v, ok := <- ch
```

See: Range and close

WaitGroup

```
import "sync"

func main() {
  var wg sync.WaitGroup

  for _, item := range itemList {
    // Increment WaitGroup Counter
    wg.Add(1)
    go doOperation(&wg, item)
  }
  // Wait for goroutines to finish
  wg.Wait()

}
```

```
func doOperation(wg *sync.WaitGroup, item string)
{
  defer wg.Done()
  // do operation on item
  // ...

}
```

A WaitGroup waits for a collection of goroutines to finish. The main goroutine calls Add to set the number of goroutines to wait for. The goroutine calls `wg.Done()` when it finishes. See: WaitGroup

#Error control

Defer

```
func main() {
  defer fmt.Println("Done")
  fmt.Println("Working...")
}
```

Defers running a function until the surrounding function returns. The arguments are evaluated immediately, but the function call is not ran until later.

See: Defer, panic and recover

Deferring functions

```
func main() {
  defer func() {
    fmt.Println("Done")
  }()
  fmt.Println("Working...")
}
```

Lambdas are better suited for defer blocks.

```
func main() {
  var d = int64(0)
  defer func(d *int64) {
    fmt.Printf("& %v Unix Sec\n", *d)
  }(&d)
  fmt.Print("Done ")
  d = time.Now().Unix()
}
```

The defer func uses current value of d, unless we use a pointer to get final value at end of main.

#Structs

Defining

```
type Vertex struct {
  X int
  Y int
}
```

```
func main() {
  v := Vertex{1, 2}
  v.X = 4

  fmt.Println(v.X, v.Y)
}
```

See: Structs

Literals

```
v := Vertex{X: 1, Y: 2}
```

```
// Field names can be omitted
v := Vertex{1, 2}
```

```
// Y is implicit
v := Vertex{X: 1}
```

You can also put field names.

Pointers to structs

```
v := &Vertex{1, 2}
v.X = 2
```

Doing `v.X` is the same as doing `(*v).X`, when v is a pointer.

#Methods

Receivers

```
type Vertex struct {
  X, Y float64
}
```

```
func (v Vertex) Abs() float64 {
  return math.Sqrt(v.X * v.X + v.Y * v.Y)
}
```

```
v := Vertex{1, 2}
v.Abs()
```

There are no classes, but you can define functions with *receivers*.

See: Methods

Mutation

```
func (v *Vertex) Scale(f float64) {
  v.X = v.X * f
  v.Y = v.Y * f
}
```

```
v := Vertex{6, 12}
v.Scale(0.5)
// `v` is updated
```

By defining your receiver as a pointer (`*Vertex`), you can do mutations.

See: Pointer receivers

#Interfaces

A basic interface

```
type Shape interface {
  Area() float64
  Perimeter() float64
}
```

Struct

```
type Rectangle struct {
  Length, Width float64
}
```

Struct `Rectangle` implicitly implements interface `Shape` by implementing all of its methods.

Methods

```
func (r Rectangle) Area() float64 {
  return r.Length * r.Width
}

func (r Rectangle) Perimeter() float64 {
  return 2 * (r.Length + r.Width)
}
```

The methods defined in `Shape` are implemented in `Rectangle`.

Interface example

```
func main() {
  var r Shape = Rectangle{Length: 3, Width: 4}
  fmt.Printf("Type of r: %T, Area: %v, Perimeter:
%v.", r, r.Area(), r.Perimeter())
}
```