Cheatography

GoLang Cheat Sheet

by Andan H M (andanhm) via cheatography.com/69306/cs/17474/

Go

Go (also referred to as GoLang) is an open source and lower level programming language designed and created at Google in 2009 by Robert Griesemer, Rob Pike and Ken Thompson, to enable users to easily write simple, reliable, and highly efficient computer programs

Besides its better-known aspects such as built-in concurrency and garbage collection

Go is a statically typed language, it is anti functional programming and anti OOP, as far as the designers concerned.

https://golang.org/

Feature

Language is very concise, simple and safe.

Compilation time is very fast.

Patterns which adapt to the surrounding environment similar to dynamic languages.

Inbuilt concurrency such as lightweight processes channels and select statements.

Supports the interfaces and the embedded types.

https://golang.org/doc/faq

Lack of essential features

No ternary operator ?:

No generic types

No exceptions

No assertions

No overloading of methods and operators

~~GOPATH is a mess~~

~~Package dependence manage tool~~

https://github.com/ksimka/go-is-not-good

Companies Using Golang

Google for “dozens of systems”

Docker a set of tools for deploying linux containers

Openshift a cloud computing platform as a service by Red Hat

Dropbox migrated few of their critical components from Python to Go

Netflix for two portions of their server architecture

Soundcloud for “dozens of systems”

ThoughtWorks some tools and applications around continuous delivery and instant messages (CoyIM)

Uber for handling high volumes of geofence-based queries.

BookMyShow for handling high volume of traffic, rapidly growing customer, to adapt new business solution and (cloud solution) distribution tools

https://www.qwentic.com/blog/companies-using-golang

Install

OSX	brew install go

Run the command below to view your Go version:

```
go version
```

https://golang.org/doc/install

Directory layout

GOPATH=/home/user/go

```
/home/user/go/
  src/
    hello/
      main.go (package main)
  bin/
     hello (installed command)
  pkg/
    linux_amd64/ (installed package object)
    github.com/ (3rd party dependencies)
```

By **Andan H M** (andanhm)
cheatography.com/andanhm/
andanhm.me

Published 21st October, 2018.
Last updated 22nd October, 2018.

Hello Word

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

Create a file named main.go in the directory src/hello inside your workspace/go path
go env Default go system environment

https://tour.golang.org

Running

```
$ cd $HOME/go/src/hello
$ go run main.go
Hello, World!!
$ go build
$ ./hello
Hello, World!!
```

Package

Package declaration at top of every source file

Executables are should be in package main

Upper case identifier: public (accessible from other packages)

Lower case identifier: private (not accessible from other packages)

Built-in Types

```
bool
string
int int8 int16 int32 int64
uint uint8 uint16 uint32 uint64 uintptr
byte // alias for uint8
rune // alias for int32 ~= a character (Unicode code point)
```

Built-in Types (cont)

```
float32 float64
complex64 complex128
```

Packages and Modules

Packages

> Go packages are folders that contain one more go files.

Modules

> A modules (starting with vgo and go 1.11) is a versioned collection of packages.

```
go get github.com/andanhm/go-prettytimee
go mod init github.com/andanhm/go-prettytime
```

Variable & Function Declarations

```
const country = "india"
// declaration without initialization
var age int
// declaration with initialization
var age int = 23
// declare and init multiple at once
var age, pincode int = 23, 577002
// type omitted, will be inferred
var age = 23
// simple function
func person() {
 // shorthand, only within func bodies
 // type is always implicit
 age := 23
}
// Can have function with params
```

Variable & Function Declarations (cont)

```
func person(firstName string, lastName string) {}
// Can have multiple params of the same type
func person(firstName, lastName string) {}
// Can return type declaration
func person() int {
    return 23
}
// Can return multiple values at once
func person() (int, string) {
    return 23, "vinay"
}
var age, name = person()
// Can return multiple named results
func person() (age int, name string) {
    age = 23
    name = "vinay"
    return
}
var age, name = person()
// Can return function
func person() func() (string,string) {
     area:=func() (string,string) {
          return "street", "city"
      }
      return area
}
```

If statement

```
if age < 18 {
      return errors.New("not allowed to enter")
}
// Conditional statement
if err := Request("google.com"); err != nil {
    return err
}
// Type assertion inside
var age interface{}
age = 23
if val, ok := age.(int); ok {
    fmt.Println(val)
}
```

Loop statement

```
for i := 1; i < 3; i++ {
}
// while loop syntax
for i < 3 {
}
// Can omit semicolons if there is only a condition
for i < 10 {
}
// while (true) like syntax
for {
}
```

Go don't have **while until**

Switch statement

```
// switch statement
switch runtime.GOOS {
  case "darwin": {
// cases break automatically
   }
   case "linux": {
   }
   default:
}
// can have an assignment statement before the switch
statement
switch os := runtime.GOOS; os {
case "darwin":
default:
}
// comparisons in switch cases
os := runtime.GOOS
switch {
case os == "darwin":
default:
}
// cases can be presented in comma-separated lists
switch os {
case "darwin", "linux":
}
```

Arrays, Slices

```
var a [3]int // declare an int array with length 3.
var a = [3]int {1, 2, 3} // declare and initialize a
slice
a := [...]int{1, 2} // elipsis -> Compiler figures out
array length
a[0] = 1 // set elements
i := a[0] // read elements
var b = a[lo:hi] // creates a slice (view of the
array) from index lo to hi-1
var b = a[1:4] // slice from index 1 to 3
var b = a[:3] // missing low index implies 0
var b = a[3:] // missing high index implies len(a)
a = append(a,17,3) // append items to slice a
c := append(a,b...) // concatenate slices a and b

// create a slice with make
a = make([]int, 5, 5) // first arg length, second
capacity
a = make([]int, 5) // capacity is optional
// loop over an array/ slice / struct
for index, element := range a {

}
```

Maps & Struct

Maps

> Maps are Go's built-in associative data type (`hashes` or `dicts`)

Struct

> Structs are the way to create concrete user-defined types in Go. Struct types are declared by composing a fixed set of unique fields.

By **Andan H M** (andanhm)
cheatography.com/andanhm/
andanhm.me

Published 21st October, 2018.
Last updated 22nd October, 2018.

Example

```
type Address struct {
  Street string
  City string
}
type Employee struct {
  Name string
  Age int
  Address Address
}
// Can declare methods on structs.
func (emp Employee) Display() string {
  // accessing member
  name:=emp.Name
  return fmt.Sprintf("Name %s",name)
}
// Initialize the map with the type
// map key is city value employees working
bookmyshow := make(map[string][]Employee)
// Create new/updates the key value pair
bookmyshow["Pune"] = []Employee{}
bookmyshow["Bangalore"] = []Employee{
   Employee{
    Name: "Andan H M",
    Age: 23,
    Address: Address{
     Street: "KB Extension",
     City: "Davanagere",
   },
```

Example (cont)

```
 },
},
// Determains the the length of the map
_ = len(bookmyshow)
// read the item from the map
employees := bookmyshow["Bangalore"]
// loop over an array, slice, struct array
for index, element := range employees {
  // read the element from the struct
  fmt.Println(index, element.Display())
}
// Delete the key from the map
delete(bookmyshow, "Pune")
```

Interfaces

Interface type that specifies zero methods is known as the *empty interface*

```
var i interface{}
i = 42
// Reflection: type switch specify types
switch v := i.(type) {
 case int:
   fmt.Printf("(%v, %T)\n", i, i)
 case string:
  fmt.Printf("(%v, %T)\n", i, i)
 default:
  fmt.Printf("Unknow type %T!\n", v)
}
```

Interfaces are named collections of method signatures.

```
type error interface {
```

Interfaces (cont)

```
    Error() string
}
```

Accept interfaces, return structs

Error

The error type is an interface type.

`error` variable represents description of the error string

```
errors.New('user not found')
fmt.Errorf("%s user not found", "foo")
```

https://blog.golang.org/error-handling-and-go

HTTP Handler

```
package main
import (
 "io"
 "net/http"
)
func health(w http.ResponseWriter, r *http.Request) {
  w.WriteHeader(http.StatusOK)
  io.WriteString(w, "Ok")
}
func main() {
 http.HandleFunc("/health", health)
 http.ListenAndServe(":8080", nil)
}
```

A mini-toolkit/micro-framework to build web apps; with handler chaining, middleware and context injection, with standard library compliant HTTP handlers(i.e. http.HandlerFunc).

https://github.com/bnkamalesh/webgo

Unit Test

Go has a built-in testing command called go test and a package testing which combine to give a minimal but complete testing experience.

Standard tool-chain also includes benchmarking and code coverage

https://github.com/andanhm/gounittest

Concurrency

Goroutines

> Goroutines are lightweight threads managed by Go

Channels

> Channels are a typed conduit through which you can send and receive values with the channel operator (`<-`)

Example

```
package main
import "fmt"
func main() {
 n := 2

 // "make" the channel, which can be used
 // to move the int datatype
 out := make(chan int)
 // run this function as a goroutine
 // the channel that we made is also provided
 go Square(n, out)
 // Any output is received on this channel
 // print it to the console and proceed
 fmt.Println(<-out)
}
func Square(n int, out chan<- int) {
 result := n * n
```

Example (cont)

```
 //pipes the result into it
 out <- result
}
```

select statement lets a goroutine wait on multiple communication operations.

sync go build-in package provides basic synchronization primitives such as mutual exclusion locks.
https://golang.org/pkg/sync/

Defer, Panic, and Recover

Defer

> A defer statement pushes a function call onto a Last In First Out order list. The list of saved calls is executed after the surrounding function returns

Panic

> Panic is a built-in function that stops the ordinary flow of control and begins panicking.

Recover

> Recover is a built-in function that regains control of a panicking goroutine

```
func main() {
 defer func() {
 if r := recover(); r != nil {
   fmt.Println("Recovered", r)
  }
 }()
 panic("make panic")
}
```

Encoding

encoding is a built-in package defines interfaces shared by other packages that convert data to and from byte-level and textual representations

Go offers built-in support for encoding/gob, encoding/json, and encoding/xml

https://golang.org/pkg/encoding/

Example

```
package main
import (
 "encoding/json"
 "encoding/xml"
 "fmt"
)
type Employee struct {
 Name string `json:"name" xml:"name"`
 Age int `json:"age" xml:"age"`
}
func main() {
 emp := Employee{
  Name: "andan.h",
  Age: 27,
 }
 // Marshal: refers to the process of converting
 // the data or the objects into a byte-stream
 jsonData, _ := json.Marshal(emp)
 fmt.Println(string(jsonData))
 xmlData, _ := xml.Marshal(emp)
 fmt.Println(string(xmlData))
 // Unmarshal: refers to the reverse process of
 // converting the byte-stream back to data or object
 json.Unmarshal(jsonData, &emp)
 fmt.Println(emp)
}
```

Tool
https://godoc.org/golang.org/x/tools
https://dominik.honnef.co/posts/2014/12/an_incomplete_list_of_go_tools/
https://github.com/campoy/go-tooling-workshop

By **Andan H M** (andanhm)
cheatography.com/andanhm/
andanhm.me

Published 21st October, 2018.
Last updated 22nd October, 2018.