ganesh-19 / **golang-cheat-sheet**

forked from a8m/golang-cheat-sheet

An overview of Go syntax and features.

☆ 1 star   737 forks

Unstar   Watch

Code   Pull requests   Actions   Projects   Wiki   Security   Insights

master

This branch is even with a8m:master.   Pull request   Compare

a8m Merge pull request a8m#58 from a8m/codingsince1985-pat… … on Feb 18, 2019   **108**

View code

README.md

Go Cheat Sheet

Index

Credits

Most example code taken from A Tour of Go, which is an excellent introduction to Go. If you're new to Go, do that tour. Seriously.

Go in a Nutshell

- Imperative language
- Statically typed
- Syntax tokens similar to C (but less parentheses and no semicolons) and the structure to Oberon-2
- Compiles to native code (no JVM)
- No classes, but structs with methods
- Interfaces
- No implementation inheritance. There's type embedding, though.
- Functions are first class citizens
- Functions can return multiple values
- Has closures
- Pointers, but not pointer arithmetic
- Built-in concurrency primitives: Goroutines and Channels

Basic Syntax

Hello World

File `hello.go`:

```go
package main

import "fmt"

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

```
$ go run hello.go
```

Operators

Arithmetic

Operator	Description
+	addition

Operator	Description
`-`	subtraction
`*`	multiplication
`/`	quotient
`%`	remainder
`&`	bitwise and
`|`	bitwise or
`^`	bitwise xor
`&^`	bit clear (and not)
`<<`	left shift
`>>`	right shift

Comparison

Operator	Description
`==`	equal
`!=`	not equal
`<`	less than
`<=`	less than or equal
`>`	greater than
`>=`	greater than or equal

Logical

Operator	Description
`&&`	logical and
`||`	logical or
`!`	logical not

Other

Operator	Description
`&`	address of / create pointer
`*`	dereference pointer
`<-`	send / receive operator (see 'Channels' below)

Declarations

Type goes after identifier!

```
var foo int // declaration without initialization
var foo int = 42 // declaration with initialization
var foo, bar int = 42, 1302 // declare and init multiple vars at once
var foo = 42 // type omitted, will be inferred
foo := 42 // shorthand, only in func bodies, omit var keyword, type is always imp
const constant = "This is a constant"

// iota can be used for incrementing numbers, starting from 0
const (
    _ = iota
    a
    b
    c = 1 << iota
    d
)
    fmt.Println(a, b) // 1 2 (0 is skipped)
    fmt.Println(c, d) // 8 16 (2^3, 2^4)
```

Functions

```
// a simple function
func functionName() {}

// function with parameters (again, types go after identifiers)
func functionName(param1 string, param2 int) {}

// multiple parameters of the same type
func functionName(param1, param2 int) {}
```

```
// return type declaration
func functionName() int {
    return 42
}

// Can return multiple values at once
func returnMulti() (int, string) {
    return 42, "foobar"
}
var x, str = returnMulti()

// Return multiple named results simply by return
func returnMulti2() (n int, s string) {
    n = 42
    s = "foobar"
    // n and s will be returned
    return
}
var x, str = returnMulti2()
```

Functions As Values And Closures

```
func main() {
    // assign a function to a name
    add := func(a, b int) int {
        return a + b
    }
    // use the name to call the function
    fmt.Println(add(3, 4))
}

// Closures, lexically scoped: Functions can access values that were
// in scope when defining the function
func scope() func() int{
    outer_var := 2
    foo := func() int { return outer_var}
    return foo
}

func another_scope() func() int{
    // won't compile because outer_var and foo not defined in this scope
    outer_var = 444
    return foo
}
```

```
// Closures
func outer() (func() int, int) {
    outer_var := 2
    inner := func() int {
        outer_var += 99 // outer_var from outer scope is mutated.
        return outer_var
    }
    inner()
    return inner, outer_var // return inner func and mutated outer_var 101
}
```

Variadic Functions

```
func main() {
        fmt.Println(adder(1, 2, 3))     // 6
        fmt.Println(adder(9, 9))        // 18

        nums := []int{10, 20, 30}
        fmt.Println(adder(nums...))     // 60
}

// By using ... before the type name of the last parameter you can indicate that
// The function is invoked like any other function except we can pass as many arg
func adder(args ...int) int {
        total := 0
        for _, v := range args { // Iterates over the arguments whatever the numb
                total += v
        }
        return total
}
```

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) - very Viking
```

```
float32 float64

complex64 complex128
```

Type Conversions

```
var i int = 42
var f float64 = float64(i)
var u uint = uint(f)

// alternative syntax
i := 42
f := float64(i)
u := uint(f)
```

Packages

- Package declaration at top of every source file
- Executables are in package `main`
- Convention: package name == last name of import path (import path `math/rand` => package `rand`)
- Upper case identifier: exported (visible from other packages)
- Lower case identifier: private (not visible from other packages)

Control structures

If

```
func main() {
        // Basic one
        if x > 10 {
                return x
        } else if x == 10 {
                return 10
        } else {
                return -x
        }

        // You can put one statement before the condition
        if a := b + c; a < 42 {
                return a
        } else {
                return a - 42
        }

        // Type assertion inside if
        var val interface{}
        val = "foo"
        if str, ok := val.(string); ok {
                fmt.Println(str)
        }
}
```

Loops

```
    // There's only `for`, no `while`, no `until`
    for i := 1; i < 10; i++ {
    }
    for ; i < 10;  { // while - loop
    }
    for i < 10  { // you can omit semicolons if there is only a condition
    }
    for { // you can omit the condition ~ while (true)
    }

    // use break/continue on current loop
    // use break/continue with label on outer loop
here:
    for i := 0; i < 2; i++ {
        for j := i + 1; j < 3; j++ {
            if i == 0 {
                continue here
            }
            fmt.Println(j)
            if j == 2 {
                break
            }
        }
    }

there:
    for i := 0; i < 2; i++ {
        for j := i + 1; j < 3; j++ {
            if j == 1 {
                continue
            }
            fmt.Println(j)
            if j == 2 {
                break there
            }
        }
    }
```

Switch

```
// switch statement
switch operatingSystem {
case "darwin":
    fmt.Println("Mac OS Hipster")
    // cases break automatically, no fallthrough by default
case "linux":
    fmt.Println("Linux Geek")
default:
    // Windows, BSD, ...
    fmt.Println("Other")
}

// as with for and if, you can have an assignment statement before the switch
switch os := runtime.GOOS; os {
case "darwin": ...
}

// you can also make comparisons in switch cases
number := 42
switch {
    case number < 42:
        fmt.Println("Smaller")
    case number == 42:
        fmt.Println("Equal")
    case number > 42:
        fmt.Println("Greater")
}

// cases can be presented in comma-separated lists
var char byte = '?'
switch char {
    case ' ', '?', '&', '=', '#', '+', '%':
        fmt.Println("Should escape")
}
```

Arrays, Slices, Ranges

Arrays

```
var a [10]int // declare an int array with length 10. Array length is part of the
a[3] = 42     // set elements
i := a[3]     // read elements

// declare and initialize
var a = [2]int{1, 2}
a := [2]int{1, 2} //shorthand
a := [...]int{1, 2} // elipsis -> Compiler figures out array length
```

Slices

```
var a []int                              // declare a slice - similar to an array
var a = []int {1, 2, 3, 4}               // declare and initialize a slice (backe
a := []int{1, 2, 3, 4}                   // shorthand
chars := []string{0:"a", 2:"c", 1: "b"}  // ["a", "b", "c"]

var b = a[lo:hi]        // creates a slice (view of the array) from index lo to h
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([]byte, 5, 5)  // first arg length, second capacity
a = make([]byte, 5)     // capacity is optional

// create a slice from an array
x := [3]string{"Лайка", "Белка", "Стрелка"}
s := x[:] // a slice referencing the storage of x
```

Operations on Arrays and Slices

`len(a)` gives you the length of an array/a slice. It's a built-in function, not a attribute/method on the array.

```
// loop over an array/a slice
for i, e := range a {
    // i is the index, e the element
}

// if you only need e:
for _, e := range a {
    // e is the element
}

// ...and if you only need the index
for i := range a {
}

// In Go pre-1.4, you'll get a compiler error if you're not using i and e.
// Go 1.4 introduced a variable-free form, so that you can do this
for range time.Tick(time.Second) {
    // do it once a sec
}
```

Maps

```
var m map[string]int
m = make(map[string]int)
m["key"] = 42
fmt.Println(m["key"])

delete(m, "key")

elem, ok := m["key"] // test if key "key" is present and retrieve it, if so

// map literal
var m = map[string]Vertex{
    "Bell Labs": {40.68433, -74.39967},
    "Google":    {37.42202, -122.08408},
}

// iterate over map content
for key, value := range m {
}
```

Structs

There are no classes, only structs. Structs can have methods.

```
// A struct is a type. It's also a collection of fields

// Declaration
type Vertex struct {
    X, Y int
}

// Creating
var v = Vertex{1, 2}
var v = Vertex{X: 1, Y: 2} // Creates a struct by defining values with keys
var v = []Vertex{{1,2},{5,2},{5,5}} // Initialize a slice of structs

// Accessing members
v.X = 4

// You can declare methods on structs. The struct you want to declare the
// method on (the receiving type) comes between the the func keyword and
// the method name. The struct is copied on each method call(!)
func (v Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

// Call method
v.Abs()

// For mutating methods, you need to use a pointer (see below) to the Struct
// as the type. With this, the struct value is not copied for the method call.
func (v *Vertex) add(n float64) {
    v.X += n
    v.Y += n
}
```

Anonymous structs: Cheaper and safer than using `map[string]interface{}` .

```
point := struct {
        X, Y int
}{1, 2}
```

Pointers

```
p := Vertex{1, 2}  // p is a Vertex
q := &p            // q is a pointer to a Vertex
r := &Vertex{1, 2} // r is also a pointer to a Vertex
```

```go
// The type of a pointer to a Vertex is *Vertex

var s *Vertex = new(Vertex) // new creates a pointer to a new struct instance
```

Interfaces

```go
// interface declaration
type Awesomizer interface {
    Awesomize() string
}

// types do *not* declare to implement interfaces
type Foo struct {}

// instead, types implicitly satisfy an interface if they implement all required
func (foo Foo) Awesomize() string {
    return "Awesome!"
}
```

Embedding

There is no subclassing in Go. Instead, there is interface and struct embedding.

```go
// ReadWriter implementations must satisfy both Reader and Writer
type ReadWriter interface {
    Reader
    Writer
}

// Server exposes all the methods that Logger has
type Server struct {
    Host string
    Port int
    *log.Logger
}

// initialize the embedded type the usual way
server := &Server{"localhost", 80, log.New(...)}

// methods implemented on the embedded struct are passed through
server.Log(...) // calls server.Logger.Log(...)
```

```
// the field name of the embedded type is its type name (in this case Logger)
var logger *log.Logger = server.Logger
```

Errors

There is no exception handling. Functions that might produce an error just declare an additional return value of type `Error`. This is the `Error` interface:

```
type error interface {
    Error() string
}
```

A function that might return an error:

```
func doStuff() (int, error) {
}

func main() {
    result, err := doStuff()
    if err != nil {
        // handle error
    } else {
        // all is good, use result
    }
}
```

Concurrency

Goroutines

Goroutines are lightweight threads (managed by Go, not OS threads). `go f(a, b)` starts a new goroutine which runs `f` (given `f` is a function).

```
// just a function (which can be later started as a goroutine)
func doStuff(s string) {
}

func main() {
    // using a named function in a goroutine
    go doStuff("foobar")

    // using an anonymous inner function in a goroutine
    go func (x int) {
        // function body goes here
    }(42)
}
```

Channels

```
ch := make(chan int) // create a channel of type int
ch <- 42             // Send a value to the channel ch.
v := <-ch            // Receive a value from ch

// Non-buffered channels block. Read blocks when no value is available, write blo

// Create a buffered channel. Writing to a buffered channels does not block if le
ch := make(chan int, 100)

close(ch) // closes the channel (only sender should close)

// read from channel and test if it has been closed
v, ok := <-ch

// if ok is false, channel has been closed

// Read from channel until it is closed
for i := range ch {
    fmt.Println(i)
}

// select blocks on multiple channel operations, if one unblocks, the correspondi
func doStuff(channelOut, channelIn chan int) {
    select {
    case channelOut <- 42:
        fmt.Println("We could write to channelOut!")
    case x := <- channelIn:
        fmt.Println("We could read from channelIn")
    case <-time.After(time.Second * 1):
        fmt.Println("timeout")
    }
}
```

Channel Axioms

- A send to a nil channel blocks forever

  ```
  var c chan string
  c <- "Hello, World!"
  // fatal error: all goroutines are asleep - deadlock!
  ```

- A receive from a nil channel blocks forever

  ```
  var c chan string
  fmt.Println(<-c)
  // fatal error: all goroutines are asleep - deadlock!
  ```

- A send to a closed channel panics

  ```
  var c = make(chan string, 1)
  c <- "Hello, World!"
  close(c)
  c <- "Hello, Panic!"
  // panic: send on closed channel
  ```

- A receive from a closed channel returns the zero value immediately

  ```
  var c = make(chan int, 2)
  c <- 1
  c <- 2
  close(c)
  for i := 0; i < 3; i++ {
      fmt.Printf("%d ", <-c)
  }
  // 1 2 0
  ```

Printing

```
fmt.Println("Hello, 你好, नमस्ते, Привет, ᎣᏏᏲ") // basic print, plus newline
p := struct { X, Y int }{ 17, 2 }
fmt.Println( "My point:", p, "x coord=", p.X ) // print structs, ints, etc
s := fmt.Sprintln( "My point:", p, "x coord=", p.X ) // print to string variable

fmt.Printf("%d hex:%x bin:%b fp:%f sci:%e",17,17,17,17.0,17.0) // c-ish format
s2 := fmt.Sprintf( "%d %f", 17, 17.0 ) // formatted print to string variable
```

```
hellomsg := `
 "Hello" in Chinese is 你好 ('Ni Hao')
 "Hello" in Hindi is नमस्ते ('Namaste')
` // multi-line string literal, using back-tick at beginning and end
```

Reflection

Type Switch

A type switch is like a regular switch statement, but the cases in a type switch specify types (not values), and those values are compared against the type of the value held by the given interface value.

```
func do(i interface{}) {
        switch v := i.(type) {
        case int:
                fmt.Printf("Twice %v is %v\n", v, v*2)
        case string:
                fmt.Printf("%q is %v bytes long\n", v, len(v))
        default:
                fmt.Printf("I don't know about type %T!\n", v)
        }
}

func main() {
        do(21)
        do("hello")
        do(true)
}
```

Snippets

HTTP Server

```
package main

import (
    "fmt"
    "net/http"
```

```
)

// define a type for the response
type Hello struct{}

// let that type implement the ServeHTTP method (defined in interface http.Handle
func (h Hello) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    fmt.Fprint(w, "Hello!")
}

func main() {
    var h Hello
    http.ListenAndServe("localhost:4000", h)
}

// Here's the method signature of http.ServeHTTP:
// type Handler interface {
//     ServeHTTP(w http.ResponseWriter, r *http.Request)
// }
```

Releases

No releases published

Create a new release

Packages

No packages published

Publish your first package