Golang Cheat Sheet

Dainish Jabeen

March 14, 2023

1 Packages

Go uses packages, which can contain multiple files. **The app will start running in the main application.**

Names exported outside the packages, must use a Capital letter.

Listing 1: Golang basics

```
1
2       package main
3
4       import "fmt"
5
6       func main(){
7               fmt.Println("Hello World")
8       }
```

2 Basics

2.1 Types

Listing 2: Golang types

```
1
2       := //Declare and initialize non explicit type
3       >> //Shift bitwise right
4       <<  //Shift bitwise left
5
6       //ARRAY
7       name []string
8       var := []string("blah","meh")
9
10      //MAPS
11      map[key type]val type //Dict has to be made
12      m := make(map[string]String)
13
14      elem, ok = m[key] // check key exists
15
16      // Initialization
17
18      var i int // initializes as 0
19
20      // Constants
21
22      const(
23              x=1 // the type of this can change on context
24      )
25
26      p := &i //point to i
27      *p //value of i, changes will change also change i
28
29      type Name struct{
30              x int
31              y int
32      }
33
34      // Pointers to structs
35
36      v := StructName{1,2}
37
38      p = &v
39      p.x = //Will change the value of v
40
41      // Struct constructors
42
43      v := StructName{x:1} //Others members made 0
44
45      //SLICES
```

```
46
47          //Slices acts as pointers
48
49          a[1:] // slice to end
50          s := a[:3] // slice start to 3
51
52          cap(s) // Capacity, elements in underlying array
53
54          // Dynamically size arrays
55
56          a := make(type, len, cap)
57          append(arr, val)
```

2.2 Functions

Listing 3: Functions

```
1
2       func Name(name type) type{}
3
4       //FUNC PARAMS
5
6       func name(x int,y int)
7       func name(x,y int)
8
9       //NAKED RETURN
10
11      func() (x,y int) {
12              x:=1
13              y:=2
14              return
15      } // Will return x and y
```

2.2.1 Funcs as params

Listing 4: Params

```
1
2       func compute(fn func(float64, float64) float64) float64 {
3       return fn(3, 4)
4       }
5
6       func main() {
7               hypot := func(x, y float64) float64 {
8                       return math.Sqrt(x*x + y*y)
9               }
10              fmt.Println(hypot(5, 12))
11
12              fmt.Println(compute(hypot))
13              fmt.Println(compute(math.Pow))
14      }
```

2.2.2 Funcs as closures

Reference var from outside the body and be bound to that value, which when called includes any prev changes to those vars.

Listing 5: Params

```
1
2       func adder() func(int) int {
3               sum := 0
4               return func(x int) int {
5                       sum += x
6                       return sum
```

```
7                 }
8         }
9
10        func main() {
11                pos, neg := adder(), adder()
12                for i := 0; i < 10; i++ {
13                        fmt.Println(
14                                pos(i),
15                                neg(-2*i),
16                        )
17                }
18        }
```

2.3 Control

Listing 6: Control

```
1
2         for i:=0;i<10;i++{}
3
4         for x<100 {} //Same as while loop
5
6         if statement; cond {}
7
8         switch statement; val {
9
10                case x: //x same as val == x
11
12                case y: //y same as val == y
13
14                default:
15        }
16
17        defer expr //execute expr at the end of func, can stack defers
18
19        for i,v := range arr {} // Loops through array (can do _,v or i,_)
```

3 Methods

Function with a receiver argument. Generally a type through which the function allows for polymorphism.

Listing 7: Methods

```
1
2         type blah struct{
3                 x int
4         }
5
6         func (v blah) Name() int {} // v can access all members of the struct.
```

```
7
8       func (v *blah) Name() int {} //Pointer allows changing of structs.
9
10      // v.Name() interpreted as &v.Name()
```

- Pointers good for performance as copying is avoided
- Should avoid mixing methods of val and pointers

3.1 Interfaces

Interfaces allow holding any value used to interface with methods.

Listing 8: Interfaces

```
1
2       type Name interface{
3               method()
4       }
5
6       interface{} // Empty interface can hold any type
7
8       // Type Assertion
9
10      var := interface{} = "h"
11
12      s := var.(string) // Gives the value in the interface
13      s := var.(int) // Causes panic
14
15      s,ok := i.(string) // Check
16
17      switch v := i.(type){
18              case int:
19
20              case string:
21      }
```

Fmt uses Stringer interface, that allows the method String() to be accessed via new types.

4 Error

Error method can be extended to use new types, which is auto called if the return type is error.

Listing 9: Error

```
1
2       type ErrNegativeSqrt float64
3
4       func (e ErrNegativeSqrt) Error() string {
5               return fmt.Sprint("cannot Sqrt negative number:",
6               float64(e))
7       }
8
```

```
9         func Sqrt(x float64) (float64, error) {
10                if x > 0{
11                        return x*x, nil
12                }else {
13                        return 0,ErrNegativeSqrt(x)
14                }
15        }
16
17        func main() {
18                fmt.Println(Sqrt(2))
19                fmt.Println(Sqrt(-2))
20        }
```

5 Type Parameters

Can use T as a generic type in parameters with a constraint.

Listing 10: Type Parameters

```
1
2         //s can be any type that is comparable
3         func Name[T comparable](s T) int{}
4
5         type[T any] struct{
6                 next *List[T]
7                 val
8         }
9
10        v := List[int]{nil,3} // Before use need to define T
```

6 Concurrency

6.1 Go routines

Lightweight thread managed by the Go routine.

$$go\ f(x,y,z)$$

Eval of f happens in current routine, execution happens in new routine. Happens in the same address space.

6.2 Channels

Type conduit to send/receive values through the routines.

$$ch <- v,\ v := <- ch$$

The data flows in arrows direction. Like maps and slices need to make before use, send and receive will block until channel is ready. Add length to make buffered lengths.

Sends can close the channel so no more vals will be sent. Use ,ok to check if channel is closed, sending on a closed channel causes a panic.

Listing 11: Channels

```
1
2       func fibonacci(n int, c chan int) {
3       x, y := 0, 1
4       for i := 0; i < n; i++ {
5               c <- x
6               x, y = y, x+y
7       }
8       close(c)
9       }
10
11      func main() {
12              c := make(chan int, 10)
13              go fibonacci(cap(c), c)
14              for i := range c { // Receives until c is closed
15                      fmt.Println(i)
16              }
17      }
```

Select with a channel will block until one of its cases can run (a channel is not empty), unless theres a default.

6.3 Mutual exclusion

Listing 12: Mutal exclusion

```
1
2       var mu sync.Mutex
3
4       mu.Lock
5       mu.Unlock
```

Lock will wait for another lock to unlock before continuing, can use defer to unlock. Locks are queued in order.