

Golang Notes

Sabrina Jiang

June 6, 2017

Contents

1	Basics	2
2	Flow Control	2
3	More Data Types	4

1 Basics

- Packages

- Programs start running in package `main`
- Can also import packages using the below syntax

```
import (  
    "fmt"  
    "math/rand"  
)
```

- Exported names are **capitalized** (e.g. `Pi` is exported from the package `math`)

- Functions

- Basic Function Syntax

```
func [functionName]([varOneName], [varTwoName] [varOneAndTwoType], [etc]  
    return [thing here]  
}
```

- * A return statement without arguments will return all named variables

- Variable Declaration

- Variables can be declared without a type (e.g. `var c`)
- Variables that are initialised must have a type (e.g. `var i int = 2`)
- Variables can also be declared with the `:=` shorthand (e.g. `k := 3`)
 - * Variables declared this way have their type inferred
 - * e.g. `42` is an `int` while `3.142` is a `float64`
- Constants cannot be declared with `:=`
- Variables declared with types but no values are initialized with zero values (`0` for numeric, `false` for boolean, `""` for strings)
- You can convert between types by using the type as a function (e.g. from `int` to `float64`, use `float64(i)`)

2 Flow Control

- For Loop Syntax

```

for [initializer] ; [condition] ; [post statement] {
    [code here]
}

```

- Note the lack of parentheses around the components of the for loop
- The init and post statements are optional (basically making this into a while loop)
- A for loop without a post statement is an infinite loop

- If Syntax

```

if [statement] {
    [code]
} else {
    [more code]
}

```

- Switch Statement Syntax

```

switch [to be checked against] {
    case [case1]:
        [code execution]
    case [case2]:
        [code execution]
    default:
        [code default]
}

```

- Once the code hits a case that succeeds it automatically breaks
- A switch statement without a init is defaulted to be checked against **true**

- Defer

- Arguments are evaluated immediately but the function is not called until after
- Deferred functions are pushed onto a stack and executed in a **last-in-first-out**

3 More Data Types

- Pointers

- A type `*T` is a pointer to the value of `T`
- It's zero value is `nil`
- The `&` operator generates a pointer to its operand

```
i := 42
p = &i
```

- Struct

- Collection of fields
- Constructed via the following

```
type [name] struct {
    [varName] [varType]
    [varName] [varType]
}
```

- You can create a pointer to structs but do not need to dereference them in order to change values

- Arrays

- Array declaration syntax

```
var a [10]int
```

- An array's length is part of it's type so you cannot change that
- To "change" array lengths, you need to use the **Slices**³ data type

- Slices

- Declared as `[]T`, e.g.

```
primes := [6]int{2, 3, 5, 7, 11, 13} // an array
var s []int = primes[1:4] //a slice
```

- Slices are just a view into an array
- Changes to a slice will also change the underlying array

- You can create a slice without explicitly creating the referenced array

```
[]bool{true, true, false}
```

- * This creates an array with those values and then a slice that references said array

- Slices can be created without explicitly stating the upper and lower bounds. The defaults are 0 and the highest bound of the referenced array

- Slices have both a **length** and **capacity**

length is the number of elements the slice contains (obtained through `len(s)`)

capacity is the number of elements of the *underlying* array (obtained through `cap(s)`)

- Nil slice

- * A nil slice has length of 0, capacity of 0, and no underlying array

- Creating a slice with `make`

```
[varName] := make([][varType], [varLength], [opt: varCap])
```

- Appending to a slice

```
append([slice], [valueOne], [valueTwo], ...)
```

- Iterating over a slice or `map`³

```
for [index], [copy of element] := range [slice/map] {
    [code here]
}
```

- * If you wanted to drop the index, then you can assign it to a `_`

```
for _, value := range ...
```

- Maps

- Maps keys to values

- Zero Value: A nil map has no keys nor can keys be added

- To create one

```
[varName] = make(map[[keyType][valueType]])
```

- * For Instance: `m = make(map[string] uint8)`
- * Or you can make one using Map Literals (`map[[keyDataType]] [valueType]`)
- Mutating Maps (an `elem` in map `m`)
 - Inserting/Updating** `m[key] = elem`
 - Retrieving** `elem = m[key]`
 - Deleting** `delete(m, key)`
 - Testing that a key is present** `elem, ok = m[key]`
 - * If `ok` is `true` then the key is present, if it is `false` then the key is not
 - * If `elem` and `ok` were not declared yet, you should do `elem, ok := m[key]`