Go Basic: control flow and functions

Admin

- Assignment read chapters 4-5 in Learning Go book
- And

Go Syntactically

- **So lets take a “brief” look at how go implements most of the programming constructs you’ll need**

Selection

- **Selection in Go (AKA if)**
  - if <condition/Boolean>{
    - <do this if true>
    - }
- **Or**
  - if <condition/Boolean>{
    - <do this if true>
    - }else{
    - <do this if false>
    - }
  - No parens around condition, but must have braces {} around body even if one line

Statements

- **How does a java statement end?**

Statements

- **How does a java statement end?**
  - ;
  - Same as c/c++/C# and other “c-like” languages
- **How does a python statement end?**

Statements

- **How does a java statement end?**
  - ;
  - Same as c/c++/C# and other “c-like” languages
- **How does a python statement end?**
  - With the end of the line except for special circumstances

Go Statements

- How Does a go Statement end?
- Reminder:
- ```
  package main

  import (
  	"fmt"
  	"io/ioutil"
  	"log"
  	"net/http"
  )
  func main() {
  	response, err := http.Get("https://news.ycombinator.com/")
  	if err != nil{
  		log.Fatal(err)
  	}
  	defer response.Body.Close()
  	dataAsBytes, err := ioutil.ReadAll(response.Body)
  	if err != nil{
  		log.Fatal(err)
  	}
  	fmt.Print(string(dataAsBytes))
  }
  ```
- Code is a mangling of https://www.devdungeon.com/content/web-scraping-go
  ```

Go Statements

- **How Does a go Statement end?**
- **Reminder:**
-

```
package main

import (
	"fmt"
	"io/ioutil"
	"log"
	"net/http"
)
func main() {
	response, err := http.Get("https://news.ycombinator.com/")
	if err != nil{
		log.Fatal(err)
	}
	defer response.Body.Close()
	dataAsBytes, err := ioutil.ReadAll(response.Body)
	if err != nil{
		log.Fatal(err)
	}
	fmt.Print(string(dataAsBytes))
}
```

- **More like python (remember compiler puts ; in for you)**
  - End of line except for special circumstances

Selection II

- A more complicated selection example
- ```
  if num := 9; num < 0 {
      fmt.Println(num, "is negative")
  } else if num < 10 {
      fmt.Println(num, "has 1 digit")
  } else {
      fmt.Println(num, "has multiple digits")
  }
  ```
- Notice two statements in first condition
  - Also variables created in condition are available in all later branches
  ```

Repetition

- **In programming theory, two types of repetition, definite and indefinite**
  - for and while in most languages
- **Go has only for – which it uses for both**

Basic For

- **Basic for loop looks a lot like C-like language for loop**
- ```
  func countDown(start int){ //in honor of falcon heavy launch
  for counter := start; counter >0; counter--{
      fmt.Println(counter)
  }
  fmt.Println("blastoff")
  }
  ```
- **Again**
  - no parens around setup, but required braces
  - Scope of variables created in initialization statement only that for-loop
  ```

For II

- **You can omit the initialization and post part of the for (not the condition)**
- **makes it functionally what other languages use while**
  - func main() {
    - sum := 1
    - for ; sum < 1000; {
    - sum += sum
    - }
    - fmt.Println(sum)
  - } //From https://tour.golang.org/flowcontrol/2
- **Semi colons are optional – can be dropped – and will be by gofmt**

The forever loop

- **Go doesn’t have while – it has what the punny original go designers (last one retired in late 2025) called for-ever**
- **If we want to loop forever (till break)**
  - for{
  - //Do something forever
  - }

Break and Continue

- **As with most languages**
  - Go has *break*
    - Breaks out of the innermost for, switch or select
      - *For* we saw.
      - *Switch* works like C-like switch (mostly)
      - *Select* we'll *defer* (all puns intended) on (used like switch but for messages)
  - And *continue*
    - Goes to next iteration of loop without finishing this one

In class exercise

- **Let’s update our previous exercise once again**
  - Now instead of the repeated code we had to have last time
  - Let’s adjust the code to use a loop
    - Read the information needed to create the struct about your classes in a loop
    - Keep reading now till the user chooses to stop
    - Then print the slice of structs
    - Then end the program.

For-range

- **Go also has the iterating over a collection modern for loop**
  - Like python and modern java

```
func main() {
	names := make([]string, 0, 10)
	names = append(names, "Sattar", "Santore", "Jung", "Black", "Khojasteh", "Li", "Kim", "Yeo", "Kumari")
	for i , name := range names {
		fmt.Println(name, i)
	}

}
```

  - Use two variables, and range keyword to iterate over collection.
    - First variable is position in collection (or key for maps)
    - Second is value
  - Special: iterating over a string has runes for values not bytes (might skip a position)

Final update on original in class project

- **Let’s make one final change to the program we have been working on**
  - Now instead of printing the whole slice
  - Iterate over the slice and print each struct on its own line

Digression: Reading a text file

- **A few ways to read from a text file in go**
  - One good way (extension of https://golangbot.com/read-files/)

```
func getFile() []string{
	contents, err := os.ReadFile("test.txt")
	if err != nil {
		fmt.Println("File reading error", err)
		return nil
	}
	bigString := string(contents)
	allLines := strings.Split(bigString, "\n")
	return allLines
}
```

We will look at functions more later – but this will return a slice of strings, one for each line like python readlines

In Class Exercise

- **As an in class exercise,**
  - Create a new program for this exercise.
  - Lets grab the silly ‘recommendation’ from the resources page
  - Write a program which opens the text file reads it in and prints out every other line to the screen.
  - Simple, but gives us a chance to actually write some more interesting go.

Security, for-range, and Maps

- **Iterating over a map (using for-range) will return the items in a different order each run**
  - Intended to prevent a hash-DOS attack
  - By preventing people from entering many keys that would hash to the same hash-bucket.
  - See chapter for more.

For-range is a copy

- **The values you get from the for-range construct is a copy of the location and a copy of the value in the collection**
  - If you try to update the original value, you **must** use the location the update the original collection
  - Otherwise the changes are immediately lost next time through the loop

The _ special variable

- **If you create a local variable and don’t use it, go calls that a what?**
  - Lucky volunteer?

The _ special variable

- **If you create a local variable and don’t use it, go calls that a what?**
  - Compile error!!
- **What if we don’t want all of the variable (maybe the location when iterating over a slice)**
  - Use the _ special variable. It tells go that you don’t want this value and you don’t get the compile error
  - **NEVER** use it for error return values (to be discussed soon)

switch

- **Switch statement in go**
  - Used for multiple related if/else if else constructs in go like in c/java
  - Can use any type that can be a key value in map as switch value
    - Any type that can use == to compare
  - Or can use “blank switch” and use comparisons
  - No default fall-through of conditions
    - So no need for break as in c/c++/java

Basic Switch example

- **Here is a default switch for a dean asking you to be part of a committee (or manager asking you to ‘volunteer’)**

```
answer := getAnswer()
switch answer {
case "yes": {
        fmt.Println("You have answered correctly")
}
case "no": {
        fmt.Println("You should consider being a team player")
}
case "maybe":{
        fmt.Println("I will ask again after lunch")
}
```

- **Note required braces around switch body.**
  - Also possible to have new variable declaration on switch line as in if statement

‘Blank’ Switch

- The ‘blank’ switch switches on condition rather than value

```
answer := getAnswer()
switch {
case strings.Contains(answer,"yes"):
        {
                fmt.Println("You have answered correctly")
        }
case strings.Contains(answer,"no"):
        {
                fmt.Println("You should consider being a team player")
        }
case strings.Contains(answer,"maybe"):
        {
                fmt.Println("I will ask again after lunch")
        }
}
```

- This one will match any substring of answer and not just an exact match

Functions

- **Create a function in go using keyword func,**
  - `func <function name>(<param list>) <ret type>{`
    - `<function body>`
    - `}`
  - A few things to look at here:
    - Return type is after param list (unlike java/c/C++, but like python/swift)
    - Param list can be empty, when not, param name first then type
    - And that opening brace? It must be there. Compile error for being on next line.
      - Avoids one of the favorite java flame wars

Function Returns

- **If function has return value/type**
  - Must use return keyword to send value back
  - Can use return keyword to exit function early even if no return value
- **A simple example function from the golang tour**
  - func add(x int, y int) int {
    - return x + y
  - }

Multi-Value Returns

- **Some language (eg python, lisp) support multi value returns**
  - Mostly interpreted languages
- **Go embraces multi-value returns and really uses it.**
- **From the http standard library:**
  - `func Get(url string) (resp *Response, err error) {`
    - `return DefaultClient.Get(url)`
  - `}`
  - resp is a Response pointer, second return value as error is the go way.

Functions II

- **A function using http.Get**
- ```
  package main

  import (
       "fmt"
       "log"
       "net/http"
  )
  func main() {
       response, err := http.Get("https://news.ycombinator.com/")
       if err != nil{
           log.Fatal(err)
       }
       defer response.Body.Close()
       fmt.Print(response.Body)
  }
  ```
- A few things:
  - First the multiple returns are captured by two new variables
  - The err is checked first, then ignored.
    - No exceptions in go - discuss
  ```

Functions III – Return variables

- **A feature seeing less use as go gets older: Return variables**
- ```
  func factorial(n int) (answer int, err error){
  if n<0{
      answer = -1; err = errors.New("can use negative number for factorial"); return
  }
  answer = 1
  for ;n>0; n--{
      answer *= n
  }
  return //known as ‘naked return’ is a frowned upon pattern put return vars here
  }
  ```
  - answer and err are return variables
    - Initialized to zero values when function starts
    - Need to give useful values before function returns
    - Functions that have return types must end in return
  ```

Errors

- **Will look more deeply at errors later but need to know basics**
- **As discussed**
  - No exceptions in go
  - Errors as (traditionally/conventionally) the last return value in multi value return
  - Since you can’t have an unused variable, must handle the error
  - What about _ ?
    -

Errors

- **As discussed**
  - No exceptions in go
  - Errors as (traditionally/conventionally) the last return value in multi value return
  - Since you can’t have an unused variable, must handle the error
  - What about _ ?
    - don’t, just don’t, it would be a terrible, horrible, no good very bad idea.
      - Technically called ‘an antipattern’
    - And most of the time will not compile anyway

Functions are first class

- **Functions are first class values in Go**
  - Like python and rust (but not java and C/C++)
  - Can assign a function (not the result but the function itself) to a variable
  - Functions are not comparable (no ==)
    - So not as keys to map.
    - But can be compared to nil (zero value for function)

Closures (functions in functions

- **You can declare functions in other functions**
  - And use variables from the outer function in the inner function
  - ```
    func main() {
      a := 5
      add := func(x, y int) int {
        return a*(x + y)
      }
      fmt.Println(add(1,1))
    }
    ```
  - *Parameter types and return type signature defines go function types*
  -
    ```

Function Types

- **Given the assignment to add in previous slide, one of these will compile and one will error. Which is which?**
  - Reminder: go is a strongly typed language

```
add= func(x, y int64) int64{
 return x+y
}

add = func(first, second int) int{
 return first + second
}
```

Functions can be parameters

- **Since functions are first class: can use functions as parameters**
  - Example from go standard library strings.Map
  - Signature of strings.Map
    - `func Map(mapping func(rune) rune, s string) string`
    -

```
func main() {
    rot13 := func(r rune) rune {
        switch {
        case r >= 'A' && r <= 'Z':
            return 'A' + (r-'A'+13)%26
        case r >= 'a' && r <= 'z':
            return 'a' + (r-'a'+13)%26
        }
        return r
    }
    fmt.Println(strings.Map(rot13, "'Twas brillig and the slithy gopher..."))
}
```

Anonymous functions

- **Functions passed as parameters or assigned to variables (or as return values) can be anonymous**
  - Simply provide no name when declaring the function as in example below from golang.org

```
// This function `intSeq` returns another function, which
// we define anonymously in the body of `intSeq`. The
func intSeq() func() int {
    i := 0
    return func() int {
        i++
        return i
    }
}
```

defer

- **Defer keyword**
  - Used to execute a function after the current function returns
    - Before control returns to calling function
  - Why: usually used to release a resource
    - Call defer <release function> immediately after acquiring resource
    - Multiple defers possible in a single function – executed as a stack, most recent first.
  - Use defer instead of having close code like kudzu all through your code.
  - This way you only have to put the close call in once
    - (Almost) **Never call defer in a loop**
      - Can stack overflow

Defer and loops

- **Multiple sources caution about using defer in a loop, in example below, no close happens till function ends**

```
func() {
  for {
    row, err := db.Query("SELECT ...")
    if err != nil {
      ..
    }
    defer row.Close()
    ..
  }
  // deferred funcs run here
}
```

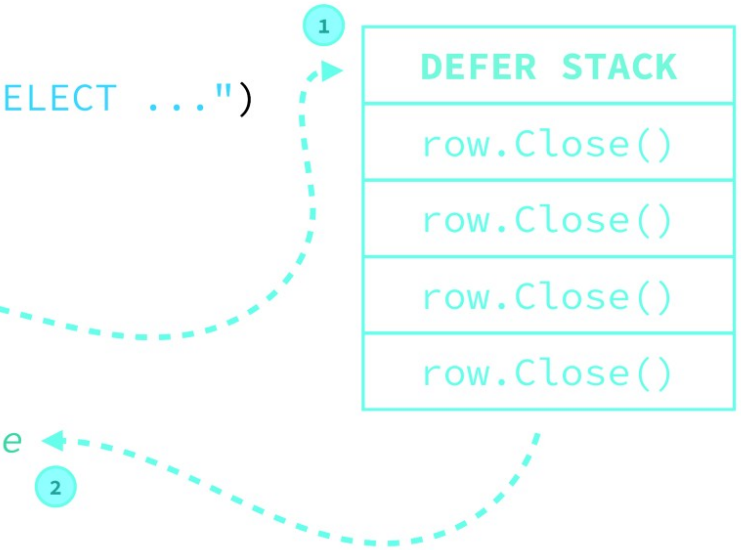

Image credit: 5 Gotchas of Defer in Go Part I by Inanc Gumus (possibly defuct)
https://blog.learngoprogramming.com/gotchas-of-defer-in-go-1-8d070894cb01

Deferred Functions

- **Deferred calls execute right after function returns**
  - From return statement
  - ‘falling through’ (closing ‘}’ of function with no return value)
  - Or panicking (more in a couple slides)
- **Any function can be deferred – even an anonymous one.**

Clearing the junk

- **Another digression to add useful utilities**
- **To clear all punctuation from a string**
  - Use regular expression like in python

```
var nonAlphanumericRegex = regexp.MustCompile(`[^a-zA-Z0-9 ]+`)
func clearString(str string) string {
    return nonAlphanumericRegex.ReplaceAllString(str, "")
}
```

  - Replaces all of the non (^) alphanumeric chars multiple times (+) with nothing “”
- **Example from: https://gosamples.dev/remove-non-alphanumeric/**

Another In Class exercise

- **My favorite exercise to demonstrate all concepts in a new language:**
  - Get a text file – maybe war and peace
    - https://www.gutenberg.org/cache/epub/2600/pg2600.txt
  - Then write a go program which will read the file in
  - count all of the times every word is used
  - Then report that count on the command line

Please read chapters 4-5 in Learning Go