Intro To the Go Language and Basic Types

Admin

- Syllabus
- Anyone new this class? Check the Syllabus
- Any trouble installing the tool chains?

Go Headline Features

- **Go (Sometimes calls Golang)**
- **Headline features:**
  - Compiled
  - Statically typed
    - Variable will always refer to same type of value
  - Structurally Typed
    - Type equivalence by definition not name
  - Memory safe
    - No buffer overflows, unsafe pointer operations
  - Garbage collected
  - Focus on concurrency
    - One of Go’s claims to fame

Where did it come from?

- **Came from google**
  - Just as java came from Sun(Oracle)
- **Originally by**
  - Robert Griesemer, Rob Pike, and Ken Thompson
- **Open source**
  - Originally: https://go.googlesource.com/go
  - But opensource is on Microsoft's github so https://github.com/golang/go
  - git clone https://go.googlesource.com/go or git clone https://github.com/golang/go
- **"BSD-style" licence**
  - https://golang.org/LICENSE

Go: A first impression beyond ‘Hello World’

- **From original go book (Donovan and Kernighan)– chapter 1 (not updated since 2016)**
  - Note package main function main is entry point
  - Also several early approaches have since been deprecated.
    - We will **not** use deprecated functions/methods even if they are often suggested by LLM/AIs

```go
package main

import (
    "fmt"
    "io/ioutil"
    "net/http"
    "os"
)

// Fetch prints the content found at a URL.
func main() {
    for _, url := range os.Args[1:] {
        resp, err := http.Get(url)
        if err != nil {
            fmt.Fprintf(os.Stderr, format: "fetch: %v\n", err)
            os.Exit( code: 1)
        }
        b, err := ioutil.ReadAll(resp.Body)
        resp.Body.Close()
        if err != nil {
            fmt.Fprintf(os.Stderr, format: "fetch: reading %s: %v\n", url, err)
            os.Exit( code: 1)
        }
        fmt.Printf( format: "%s", b)
    }
}
```

Go: A first impression

- **When I first looked at Go**
  - It looked like python and C++ had a baby
  - Of course I don’t know algol
- **(Original) Python philosophy :**
  - There is one ‘right’ way to do things
  - (harder to see in the last 10 years)
  - This is pythonic, but not enforced by compiler/interpreter.
- **With go – often is enforced by compiler**

Go and Style

- **Go ‘Good Style’ is often compiler enforced**
  - Unused variables are a compiler error
  - Unused imports are a compiler error
- **Online flame wars are often about what “good style” is**
- **Go often settles these by making the compiler only work for the approved style**
  - Your book explains some of this in a little more in chapter 1 (eg go fmt section page 5)

gofmt

- **Go helps you be compile ready with gofmt**
  - Can run on command line
    - Or let goland do it for you.
  - Gofmt pronunciation
    - Do you go with the majority?
    - Or with the crusading minority?
  - Gofmt sort of like python-black
    - Simply rewrites your code to be ‘proper’ (idiomatic) go

Go, Syntactically

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

Comments

- **Comments are really useful when learning a language**
- **Comments in Go same as C++**
  - Go took them just like java did
    - // line comments
    - /*
    - Multiline comments
    - */

Types

- **Go, like java, has distinction between basic/predefined types and all other types**
  - Basic/predefined types:
    - Boolean
    - string
    - and number (several number types)
      - uint8/byte, uint16, uint32, uint64, int8, int16, int32 and int64, etc see chap 2 in learning go book.

Literals

- **What do we mean by ‘literals’ in programming?**
  - Lucky volunteer?

Literals

- **What do we mean by ‘literals’ in programming?**
  - Values typed directly into a program
  - Integer literals in go
    - By default are int
    - By default are base 10
    - 0x: Hexadecimal, 0b : binary, 0o octal (only used for really old stuff)
  - Floating point literals
    - Are float64 by default

String Literals

- **There are two types of string literals**
  - Interpreted string literals (the traditional strings)
    - Surrounded by double quotes : “
    - Escape characters are replaced
      - Eg
      - “this \t is a string\n” :this string has 19 characters including a tab and a newline in it.
  - Raw string literals
    - Surrounded by backticks: `
    - No replacement, no escapes
      - Eg
      - `this \t is a string\n` : this string has 21 characters

Runes

- **A single ‘character’ from another language is a ‘rune’ in go.**
  - Rune type is alias for int 32

Type conversion

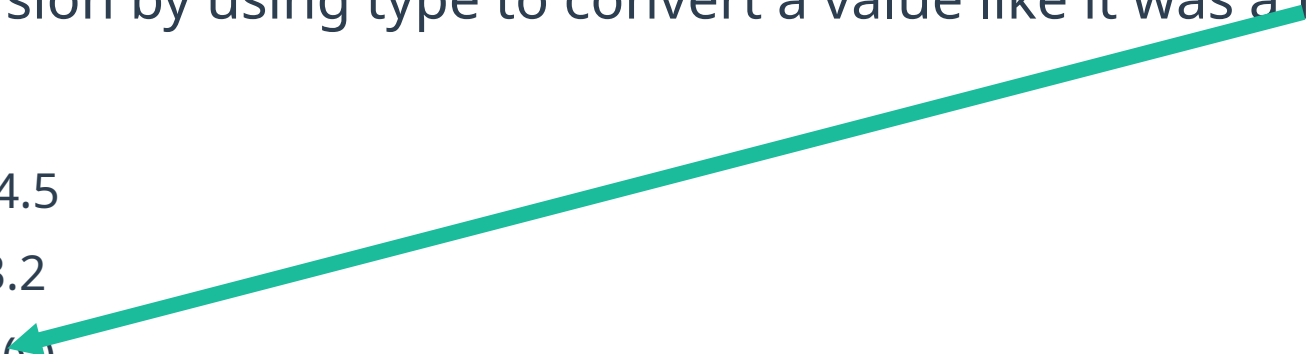
- **Python is dynamically typed**
  - When doing arithmetic automatically coverts up to “BigNum” which will go to arbitrary length.
  - 45**25 for example
- **Java?**
  - Lucky volunteer?

Type conversion

- **Python is dynamically typed**
  - When doing arithmetic automatically coverts up to “BigNum” which will go to arbitrary length.
- **Java?**
  - Will automatically convert less precise types to more precise types.
  - Left side of operation is really important.
    - int x = 4;
    - float y = 3.2;
    - float z = y+x; //x is converted to float - but the other way around would be error

Type conversion in go

- **Go makes you do explicit type conversion**
  - Go has a philosophy of making everything explicit
    - “go is boring”
  - Explicit conversion by using type to convert a value like it was a function
  - So
    - x float64 = 34.5
    - y float32 =13.2
    - z = x+float64(y)



Literals and types

- **Literals are untyped (before being assigned to a variable)**
- **But still can only be coerced into a more precise type from a less precise type**
  - So we can multiply a number literal by any int or float type
  - float64 x = 3.5
  - float64 y = x * 10 //this is fine
  - int z = 10.4 // this is **not** fine
  - int a = “wait a string” //this is **also not** fine

Variables

- **Several ways to create variables in go**
  - Break into two primary approaches
  - More variation of these two – see chapter 2 page 28-29 of book
  - First approach:
    - x,y:= 12, 20 //use := to create local variable in function, at least one must be new variable (others can be new values of existing variable), type inference
    - var x, y int = 12, 20 //used to create new variable in or out of function
      - All variables must be new and of same type
      - If no value assigned, then uses zero value automatically
      - var x,y int //x and y both now have the value zero

Const

- **Use const keyword to create constants**
  - Immutable labeled values.
  - Similar to const in java and c++
  - Value must be computed at compile time.
  - const x int32 = 50
  - const(
    - left = true
    - Right = false
    -)

Const naming conventions

- What are the naming conventions for constants in java/c++ etc?

Const naming conventions

- **What are the naming conventions for constants in java/c++ etc?**
  - Usually all caps
  - Go uses same conventions as variables

Const naming conventions

- **What are the naming conventions for constants in java/c++ etc?**
  - Usually all caps
  - Go uses same conventions as variables
  - Because Capitalization is akin to public/private in go
    - More on this later.

Warnings and Compile errors

- **Go treats several potential issues that are warnings in other languages as compile errors**
  - Unused imports
  - Unused **local** variables
    - Unused package wide variables are not an error.

Multiple data items

- **How do we store multiple related data items in <your favorite language here> (usually data items of the same type)**
  - Lucky volunteer?

Multiple data items

- **How do we store multiple related data items in <your favorite language here> (usually data items of the same type)**
  - Linear collection
    - Python: list
    - Java ArrayList
    - c/c++ (closer to ‘bare metal’ and faster) arrays
  - Go also has arrays (with a big giant **BUT** which we will get to)

Arrays

- **Someone remind us (especially for the people who have mostly done python)**
  - What is an array?

Arrays in Go

- **Arrays are linear collections of elements of the same type which are placed in sequential memory locations.**
  - Will always have the same number of ‘slots’ in the array

```
var directions = [4]int{1,2,3,4} //create an array of four ints and initialize the four spots with 1,2,3, and 4
```

  - We can change the four values in directions, but it will always have four values

```
directions[1] = 10
```

  - To create an array of four ints but initialize the values to the zero value

```
var directions [4]int
```

Arrays in Go

- **Arrays are linear collections of elements of the same type which are placed in sequential memory locations.**
  - Will always have the same number of ‘slots’ in the array
  - In fact, the number of items in the array is part of the type in go
  - [4]int and [10] int are different types!!
  - This makes it **really** efficient!
    - But really restrictive
  - Any function which took an array as a parameter would have to have multiple copies of the function, one for each size of array

Arrays and Slices in Go

- **Hardly anyone uses raw arrays in go**
  - Too restrictive.
- **Most people/programs use slices**
  - A ‘view’ into a sequence that works much more like java’s array list
    - Usually arrays, but also strings
  - Can grow as needed (up to the limit of the underlying array)
  - Every slice has an array under it, but slices grow and have variable size.
  - Every slice has:
    - pointer to an array element (first item in slice)
    - len (how many elements in slice
    - cap (how many elements till end of underlying array)

Slices

- **Creating the zero value slice**
  - `var emptySlice []int`
- **Zero value for slices is nil**
  - nil means no value.
  - More like None in python than Null in C++ or java
- **len(emptySlice) will yield zero**

Create a Slice with items in it

- Make is a really important function in go – more on it later.
- `names := make([]string, 5, 10)`
- Makes a sequence of type <first param> with len <second param> and capacity <third param>
  - If cap isn't specified, len and cap are same
- Going past len in a slice expands the slice
- Going past cap, causes *panic*
- *Update values like array.*
  - `names[3] = "slices are zero-based"`

Slices as Parameters

- **Since slices are just the three values (data, len and cap)**
  - The data pointer points at the data in the array
  - Slices are passed by value
    - Like all parameters
  - Like java (and python), you can’t change what the slice points at in the caller, but you can change the value of the slice for the caller.

Adding to a slice

- **To add (a) value(s) on to a slice**
  - Possibly increasing the size and the size of the underlying array
  - Use the append function
    - `names := make([]string, 5, 10)`
  - You can append one thing at a time
    - `names = append(names, “Prof Santore”)`
  - or many
    - `names = append(names, “Sattar”, “Santore”, “Jung”, “Black”)`
  - Unpack another sequence to append many (assume names2 is slice)
    - `names = append(names,names2...)`

Reading from Command line

- **Now we have most of the basic types**
- **So let's get info from user and tell the user**
  - Write to the console using **fmt.Println( ) or fmt.Print( )**
    - Put as many 'things' as you want to print in parens separated by commas
  - Read from console in two steps – first create a bufio reader

```
reader := bufio.NewReader(os.Stdin)  //os.Stdin will make this reader read from command line instead of file
```

  - Then call one of the readXXX methods

```
author, err := reader.ReadString('\n')
```

  - This will read till the return/enter key is pressed like input in python

In class exercise

- **Let’s try something**
  - Create a project in goland
  - Create a main function
  - And in that, create a slice of strings then
  - Ask the user four times for classes they are taking this semester
  - Add each one to the slice
  - Finally print the slice to the screen.
- **Let me know if you run into issues.**

Maps

- **Maps in Go**
  - Are hash tables. Fast access given key to find value O(n) space.
  - Work (almost) just like dictionaries in python
  - Keys must be comparable
    - a type you can use == with
    - Values can be any type
  - Trying to retrieve a key/value not in the map returns the zero value
    - Check ok if you really need to know (see next slide)

Making Maps

- **We can make a map with make**
  - `wages:= make(map[string]float32)`
- **Or with a literal map**
  - `wages := map[string]:float32{`
  - `“Ed”: 450.17,`
  - `“Ann”: 375.99,`
  - `}`
- **Check ok:**
  - `earnings, ok := wages[“John”]`
    - Earnings will be 0.0, ok will be false.

Structs

- **Structs in Go are aggregate data types**
  - If you squint hard enough they look like c(++)-structs
  - A collection of named typed fields
  - Eg:
    - ```
      type Player struct {
            Name string
            Health int
            jumpDistance int
      }
      ```
  - Creates a struct type with 3 fields,
    - `var player1 Player //creates a variable player1 of type Player with zero value for the fields`
  - Access fields in a C-like manner
    - `player1.jumpDistance = 3`
      ```

Structs II

- **Can create struct with literal**
  - `var player2 = Player{“Mario”, 1, 2}`
- **Or**
  - `var player3 = Player{name: “Luigi”, health:1}`
  - `//note jump distance not supplied so zero.`
- **Looking at this code (pay special attention to the highlighted code), what can you tell me about the package visibility for this code and the Player struct?**

Structs II

- `var player3 = Player{name: “Luigi”, health:1}`
- `//note jump distance not supplied so zero.`

- **Looking at this code (pay special attention to the highlighted code), what can you tell me about the package visibility for this code and the Player struct?**
  - They have to be in the same package
  - The struct name is Capitalized and exported
  - But the field names are not – so can’t access fields from another package.
  - Of course don’t do this in ‘real life’

Structs III

- **Structs can have another struct as a member**
  - But no recursive definitions
  - Must use pointer for recursive.
- **Embedded structs have a “lazy programmer” hack**
  - `type saveGame struct{`
  - `p Player`
  - `Size int`
  - `}`

Structs IV

- **Suppose someone hands me a saveGame from another method**
  - MySave := *<some function call here>*
- **I want to find and display the name of the player from the save**
  - `Access with MySave.p.name`
- **This could be a pain if there are lots of embedded structs – so see next slide**

Structs V

- **Use struct with anonymous fields**
  - Eg
    - `type SaveGame struct{`
    - `Player`
    - `Size int`
    - `}`
  - Now
    - `MySave := <some function call here>`
    - `MySave.name`
  - Ahh “programmers are lazy” works as long as there are no fields with same name in anonymous fields

In class exercise

- **Let’s extend our last exercise**
  - Create a struct representing a class with
    - Class name
    - Instructor name
    - Number of credits
  - Now instead of creating a slice of strings, make it a slice of this struct
    - Ask the user for all of this data
    - Create a struct and add it to the slice
  - Finally print out the whole slice.

Slices of Strings and Slices

- **In addition to array-simplifying slices, go supports python-like slicing of strings and slices.**
  - Use <start>:<end> inside of the []
    - Skip start starts at beginning, skip end goes to end.
  - Eg
    - `names := make([]string, 0, 10)`
    - `names= append(names, “Sattar”, “Santore”, “Jung”, “Black”, “Khojasteh”, “Li”, “Kim”, “Yeo”, “Kumari”)`
    - `senior_faculty=names[:5]//will have from Sattar to Li inclusive`
  - `Slice is new variable, but points to same underlying data.`
    - `Change to senior_faculty will change names`

Reading

- **Please read Bodner's Learning Go chapter's 2-3**
- **I've covered what I think are the highlights, but you really need to read to get the rest of the nuances.**