Go Language Overview

CMPSC443 Fall 2017 – Professor McDaniel
November 14, 2017

This document outlines the Go programming language largely extracted from the text and examples of the online tutorial (https://tour.golang.org/list). Please see other online sources and project notes for additional information.

The Go Language Basics

- Every program is made up of packages
- Programs start with a package main, and within it the package main
- The last part of the path is the package name "math/rand" - means the rand path
- Imports import the packages, just like python
- Capitalization is really important, names that are capitalized are exported, lower are not, e.g., **Pizza** would be exported, but **pizza** would not
- Scoping - anything defined within the package but not inside a function has a global score, anything inside has a local scope.

Interesting packages

"fmt" — formatting package
"math" — math functions
"math/rand" — random function

Basic Types

bool - boolean (true, false)
string - string ("" for the empty string)
int int8 int16 int32 int64 uint uint8 uint16 uint32 uint64 - integers
uintptr - a pointer type
byte - alias for uint8
rune - alias for int32, represents a Unicode code point
float32 float64 - floating point numbers
complex64 complex128 - complex numbers

"=" vs ":=" - these are two versions of the assignment, but the latter defines a variable as you make the assignment (it infers the type)

```
i := uint(4.589) // Creates a new variable
i = 10 // Just does an assignment for existing variable
```

Typecasting All variable translation between types must be explicitly cast. You do this by using the type itself as a pseudo-function, e.g.,

```
i := uint(4.589)
```

Pointers - pointers look a lot like pointers in C. A zero value is `nil`. You can get a pointer using the basic address of "`&`" annotation. Dereferencing is also like C, with the "`*`" syntax. There is no pointer arithmetic.

```
var p *int
P = &g // Where g is an integer variable
```

Arrays - you create and reference arrays largely using the same syntax as C. The length is part of the type, so you cannot resize them.

```
var arr [10]int
arr[0] = 1
arr[9] = 8
fmt.Println(arr)
```

You can grab a **slice** of an array using the syntax from Python. Slices also have a length (number of elements), and a capacity (size of the underlying array).

```
var slce []int = arr[1:4]
slce2 := arr[7:9]
slc3 := arr[4:]
slc4 := arr[:5]
fmt.Println(arr, slce, slce2, slc3, slc4)
fmt.Printf("Slice 4 -> %d, %d — %v", len(slc4), cap(slc4),
slc4)
```

You can also create literal arrays on the fly

```
var ltr1 []int = []int{0, 1, 2, 3}
ltr2 := []int{0, 9, 11, 14}
fmt.Println(ltr1, ltr2)
```

The make function creates a dynamically allocated array and returns a slice to it.

```
slc5 := make([]int, 9)
fmt.Println(slc5)
```

Multi-dimensional arrays/slices are also possible

```
slc6 := [][]int{
    []int{0,1},
    []int{2,3},
    []int{4,5},
}
fmt.Println(slc6)
```

You can also append to slices

```
slc7 = append(slc5, 10, 11, 12, 13, 14)
fmt.Println(slc7)
```

Maps - maps are associative arrays that have key values, where the key type and a value type. Note that this is just a reference until you "make" it.

```
var mp map[string]string
mp = make(map[string]string)
mp["one"] = "hello"
```

```
mp["two"] = "world"
fmt.Printf("%s, %s.\n", mp["one"], mp["two”])
```

They can be created as literals as needed.

```
var mp = map[string]string{
    "One": "Hello",
    "Two": "World",
}
fmt.Println(mp)
```

The basic associative array operations are also available.

```
mp["three"] = "more" // Add an element
ele := mp["three"]   // Get an element
fmt.Println(ele)
fmt.Println(mp)
delete(mp, "three")  // Remove an elemetn
fmt.Println(mp)
```

Testing for an element has some strange syntax, where you do a element, boolean assignment to test it

```
one, ok1 := mp["One"]
thr, ok2 := mp["three"]
fmt.Println(one, ok1, thr, ok2)
```

Structures - structures are defined as a collection of fields that can be assigned individually or as a group. The fields adopt default (zero) values.

```
Type Vertex struct {
    X int
    Y int
}
var vert Vertex
vert.X = 1
vert.Y = 2
va = Vertex{1, 2}
vb = Vertex{3}
vc = Vertex{X: 1, Y: 2}
fmt.Println(va, vb, vc). // You can print out structs
```

When printing the above, you get:

```
{1 2} {0 0} {1 2}
```

When you have a pointer to a structure, you can access the field values with the same dereferencing notation (it does this for you).

```
vert := Vertex(1, 2)
p = &vert
p.X = 1
```

```
p.Y = 2
```

Language elements and syntax

Import — language part to include other packages.

```
Import (
    "One"
    "Two"
)
```

Or (less preferred)

```
import "one"
Import "two"
```

Function declaration -

start with keywords,
has zero or more typed parameters (name then types,
has return type after the parameters
has curly braces

```
func party(a int, b int) int {
    ...
    return a + b
}
```

Alternately, if two consecutive parameters share a type, then you can omit the first type, e.g.,

```
func party(a, b int) int {
```

Also, return types can have multiple outputs, e.g.,

```
func party(a, b int) (int, int) { ..
x, y = party(11, 12)
```

If the return variables are named in the declaration, you don't pass them out at the return. This is called a naked return

```
func party(a, b int) (x, y int) {
    x = 1
    y = 2
    return
}
```

Function pointers can be used by referencing the name and declaring as a parameter to a function. These are very similar to C

```
func myfunc(a int, b int) int {
    return a + b
}
func party(mfc func(int, int) int) int {
    return mfc(1, 2)
}
```

```
...
fmt.Println(party(myfunc))
```

A closure is a function that returns a function bound to a specific variable.

```
func mclose() func(int) int {
    mult := 1
    return func(x int) int {
        mult = mult * x
        return mult
    }
}
cls := mclose()
cls(7)
fmt.Println(cls(8)) // Would print 56
```

Variable declaration - you define the variables at package or function level with the var keyword and provide type last

```
var k, l int;
```

The variables can also include initializers—interestingly if the initializer is there you don't have to include the type (it it automatically inferred)

```
var k, l int = 5, 6
var m, n = 5, 6
```

There are also "short" declarations within functions that do not need the var syntax

```
p := 3
```

Constants are just declared as normal variables, with the word const. Note that numeric constants take on the type of the given context — so we are able to convert

```
const p int = 3
```

For statements (pascal-esque) - this is the looping function of the language, no parenthesis and the curly braces are always required, the init an post are optional

```
for i :=1; i<10; i++ {
    ...
}
for ; i<10; {
    ...
}
```

Note that the for loop is really the while of the go language. If you drop the semicolons then the for acts exactly like while

```
for i<10 {
    ...
}
```

Forever loop just drops the for parameters altogether.

```
for {
    ...
}
```

For loops can be used on slices as well using the range keyword. You declare the index (idx below) and an element (ele), which are used at each iteration for the number index and the element of the slice itself. Note that you can skip the element or index by replacing it with a "_".

```
for idx, ele := range slc6 {
    fmt.Printf("arr[%d] = %v\n", idx, ele)
}
for _, ele := range slc6 {
    fmt.Printf("%v\n", ele)
}
```

If statements - parenthesis not required, curly braces required, else available

```
if x==10 {
    ...
} else {
    …
}
```

Strangely, the if statement can start with a precondition statement that is executed before the condition. Note here that the scope of the variable created in the precondition is only the conditional statement—in this case, the variable **y** is only usable within the conditional and conditional block.

```
var p int = 20
if y := p/10; y==2 {
    fmt.Println("LA LA")
}
```

Switch statements - very similar to those used in C. Note that in go, the "case" does not fall through to the following cases (which was weird to begin with).

```
switch x {
case "first":
    …
case "second":
    …
default:
    ...
```

Moreover, the cases do not have to be constants, but can be function calls, conditions, etc. without a specific variable it is switching on.

```
switch {
case t.Hour() == 12:
    …
```

```
case x + y == 14:
    …
default:
    ...
```

Defer statement - this is a statement that waits until the function returns (kinda like a finally). Where multiple defer functions are used, they stack in LIFO (stack).

```
defer fmt.Println("At the very end of this function")
```

Methods - Go does not have classes, but you can define methods that receiver arguments. This allow you to call the method on the type. Pointer types are also legal, and the pass by reference behavior works. Also note that if you use a receiver you can pass an object and it will automatically get the reference.

```
type MyType struct {
    x, y float64
}
func (m MyType) Must() float64 {
    return m.x * m.y
}
func (m *MyType) MustMod() float64 {
    m.x = m.x * m.x
    m.y = m.y * m.y
    return m.x * m.y
}
mt := MyType{9.4, 8.3}
fmt.Println(mt.Must())
fmt.Println(mt.MustMod(), mt)
```

Interfaces - Interfaces are collections of method signatures that perform some set of functions (often related). Note that you can provide the implementation of the functions anywhere, and there is no syntax for stating that particular implementation is for the interface (e.g., MyType:: in C++).

```
type I64 int64
type F64 float64
type MyIface interface {
    MethOne() int64
}
func (x I64) MethOne() int64 {
    return int64(x)
}
func (x F64) MethOne() int64 {
    return int64(x)
}
```

Now interestingly, you can create a variable of the type of interface and then call that. The context of the call sets which function gets called. This is a bit like overriding in C++.

```
    var iface MyIface
    iface = I64(3498)
    fmt.Println(iface.MethOne())
    iface = F64(9.3874)
    fmt.Println(iface.MethOne())
```

A special kind of interface use is the Stringer methods, which implement a toString kind of interface that can be used by things like Printf.

```
    func (ad IPAddr) String() string {
        return fmt.Sprintf("%d.%d.%d.%d", ad[0], ad[1], ad[2],
ad[3])
    }
    add := IPAddr{192, 168, 0, 1}
    fmt.Printf("%v", add)
```

There are many other uses of interfaces like errors, reading, etc. Understanding how to use interfaces is key to getting proficient at go.

--

Tutrial Problem Solutions

Tour of go, section, "More types: structs, slices, and maps", exercise on slices: (page 18/27 - on slices

```
func Pic(dx, dy int) [][]uint8 {

    // Create the array
    retar := make([][]uint8, dy)
    for i := range retar {
        retar[i] = make([]uint8, dx)
    }

    // Now make the function values
    for j := 0; j < dy; j++ {
        for k := 0; k < dx; k++ {
            retar[j][k] = uint8(j^k)
        }
    }
    return retar
}
```

Tour of go, section, "More types: structs, slices, and maps", exercise on slices: (page 23/27) - on maps

```
func WordCount(s string) map[string]int {

    wdmap := make(map[string]int)
    words := strings.Fields(s)
    for i := range words {
        _, ok := wdmap[words[i]]
        if ok {
            wdmap[words[i]] = wdmap[words[i]] + 1
        } else {
            wdmap[words[i]] = 1
        }
    }
    return wdmap
}
```

Tour of go, section, "More types: structs, slices, and maps", exercise on slices: (page 23/27) - on closures

```
// fibonacci is a function that returns
// a function that returns an int.
func fibonacci() func() int {
```

```
        last := -1
        curr := 0
        return func() int {
            if last == -1 {
                last = 0
            } else if last == 0 && curr == 0 {
                curr = 1
            } else {
                val := last + curr
                last = curr
                curr = val
            }
            return curr
        }
    }
```

Tour of go, section, “Methods and Interfaces”, exercise on stringers: (page 18/26) - on stringers

```
    func (ad IPAddr) String() string {
        return fmt.Sprintf("%d.%d.%d.%d", ad[0], ad[1], ad[2],
ad[3])
    }
```

Tour of go, section, "Methods and Interfaces”, exercise on readers: (page 20/26) - on errors

```
    type ErrNegativeSqrt float64
    func (e ErrNegativeSqrt) Error() string {
        return fmt.Sprintf("cannot Sqrt negative number: %f\n",
e)
    }
    func Sqrt(x float64) (float64, error) {
        if x < 0 {
            return 0, ErrNegativeSqrt(-2)
        }
        return 0, nil
    }
```

Tour of go, section, "Methods and Interfaces”, exercise on readers: (page 22/26) - on closures

```
    func (r MyReader) Read(b []byte) (n int, err error) {
        for i := 0; i < len(b); i++ {
            b[i] = 'A'
        }
        return len(b), nil
    }
```

Tour of go, section "Methods and Interfaces", exercise on readers (page 23/26)

```
func rotSub(r byte) byte {
    if 'a' <= r && r <= 'z' {
        return (r-'a'+13)%26 + 'a'
    } else if 'A' <= r && r <= 'Z' {
        return (r-'A'+13)%26 + 'A'
    }
    return r
}

func (r rot13Reader) Read(p []byte) (n int, err error) {
    n, err = r.r.Read(p)
    if err != nil {
        return
    }
    for i := range p[:n] {
        p[i] = rotSub(p[i])
    }
    return
}
```

Tour of go, section "Methods and Interfaces", exercise on images and interfaces (page 25/26)

```
type Image struct {
    width, height int
}

func (i Image) ColorModel() color.Model {
    return color.RGBAModel
}

func (i Image) Bounds() image.Rectangle {
    return image.Rect(0, 0, i.width, i.height)
}

func (i Image) At(x, y int) color.Color {
    val := uint8(x ^ y)
    return color.RGBA{val, val, 255, 255}
}

func main() {
    m := Image{500, 500}
    pic.ShowImage(m)
}
```