trainologic

Introduction to Go

trainologic

Go

- Go (or GoLang) is a general purpose programming language created by Google.
- First version appeared at 2009.
- Some well known open-source projects using Go:
  - Docker.
  - Kubernetes.
  - OpenShift.

trainologic

Go

- Go language characteristics:
  - Staticly-typed.
  - Compiled.
  - Garbage-collected.
  - Memory-safe.

trainologic

Go

- There is no large runtime to install like in Java.
- Great concurrency model (goroutines & channels).
- Extremely efficient.
- Large standard library.
- AOT compilation.
- Fast compilation.
- Small and clear language spec.
- Integrates well with C.

trainologic

Go vs. Java

- Java is definitely more high level.
- Provides generics and classes.
- However, comes with a large runtime (less in Java 9).
- Containers become large.
- Not suited for command line utilities.
- First interpreted, then compiled.
- Large external dependencies.

trainologic

Go vs. Python

- Performance, Performance, Performance.
- Python is dynamically typed.
- However, Python has its use-cases (e.g., machine learning).

trainologic

Go vs. C

- Go is safer.
- Provides superior concurrency utilities.
- More productive.

trainologic

Setup

trainologic

Environment Variables

- There are 2 important environment variables crucial to the operation of the Go toolchain:
  - GOROOT.
  - GOPATH.

trainologic

GOROOT

- By default, Go assumes that its installation is under /usr/local/go.
- If you install to a custom location, you have to define GOROOT to point it to that location.
- Note that you don’t need to define this environment variable when default location is used.
- Make sure to add the $GOROOT/bin folder to your PATH.

trainologic

GOPATH

- The path used to resolve ‘import’ statements.
- A colon separated list (Unix/Linux).
- Each directory must have the following structure:
  - src directory holds the Go source code in a directory structure according to the import path (more about this later).
  - pkg directory holds installed package objects (shared libs).
  - bin directory holds compiled commands.
  - internal subdirectory holds code that is can only be imported at its parent directory.
  - vendor subdirectory is like internal, but the import statement starts from this point.
- We’ll discuss ‘internal’ and ‘vendor’ later.

trainologic

Sanity Check

- Invoke the “go env” command to list the Go environment variables’ values.

- You can also invoke “go version” to display the installation version.

trainologic

Workspace

- In Go, usually all projects reside under a single workspace.
- A workspace contains:
  - src – directory containing repositories.
  - pkg – directory containing package objects.
  - bin – directory containing binaries.
- GOPATH points to the location of the workspace.

trainologic

Workspace

- Each workspace contains repositories.
- Each repository contains one or more package.
- The import path (discuss later) specifies the path relative to the workspace.
- The last directory on the path is the package name.
- Each package has one or more Go files in a single directory!

trainologic

Basic Syntax

trainologic

Comments

- It is customary for programming language syntax reference to start with comments.
- A single-line comment starts with **//**.
- A multi-line comment starts with /* and ends with */.
- Comments can’t be nested.

```
// this is a comment

/*
This is also a comment

*/
```

trainologic

Packages

- Each Go source file starts with a package declaration.
- Packages are the most important encapsulation mechanism in Go.
- You can export the API of your package and import the API of another package.
- All programs start from from the *main()* function in the package *main*.
- The *main()* function must not accept arguments and doesn't return a value.

trainologic

Imports

- You can import the API of another package by using the **import** keyword.
- For example, let's print PI:

```
package main

import "math"
import "strconv"



func main()  {
  print("PI: " +  strconv.FormatFloat(math.Pi,'f', -1, 64 ))
}
```

trainologic

Discussion

- Without the import statements the program will not compile.
- The ‘math’ import is required for PI and the ‘strconv’ is required for the *FormatFloat*.
- Instead of writing multiple import statements, we can (and should) combine them:

```
import (
  "math"
  "strconv"
)
```

trainologic

FormatFloat

- Unlike other programming languages (e.g., Java), you can't automatically get a string from a float.
- FormatFloat converts a float to a string.
- Accepts 4 parameters:
  - The float value.
  - The formatting (explained next slide).
  - The precision (the number of digits after the decimal point, excluding the exponent). The value of -1 is special and uses the smallest number of digits necessary.
  - The bit-size of the value (32 or 64).

trainologic

Formatting

- A single character:
  - ‘f’, ‘F’ – decimal point without exponent.
  - ‘e’ – scientific notation with lowercase ‘e’.
  - ‘E’ – scientific notation with uppercase ‘E’.
  - ‘g’ – like ‘e’ for large exponents, ‘f’ otherwise.
  - ‘G’ – like ‘E’ for large exponents, ‘F’ otherwise.
  - ‘b’ – decimalless scientific notation with exponent in the power of 2.

trainologic

Variable Declarations

- A variable is declared using the ***var*** keyword.
- Followed by the variable name, its type and an optional initializer.
- E.g.:

```
var someInt int = 6

var someString string = "hello"
```

trainologic

Variable Declarations

- If you don’t provide an initial value, a ‘zero’ value is used.
- Zero values:
  - Numerics – 0.
  - Booleans – false.
  - Strings – “”.

trainologic

strings

- Strings in Go are built-in types.
- They are not pointers (like in Java) but values.
- They are never null (or nil in Go).

- Note that strings are immutable in Go.

trainologic

strings

- A string in Go represents a sequence of runes(int32) and in turn bytes!
- Not characters.

- You can specify string literals with either double quotes (supports escaping characters) or raw strings with back-quotes.

trainologic

strings

- Often you'll hear the strings in Go are UTF-8 encoded.
- That is not true.
- The source file must be UTF-8 encoded.
- Thus, a string literal is the UTF-8 bytes representation of the characters.

- Remember, strings in Go store bytes!

trainologic

runes

- Go provides the built-in type ***rune*** which is an alias for int32.
- Rune literals are defined with single quotes.
- Supports escaping.

- Prefer using the rune type when working with characters.
- Remember that **int** is defined as either 32-bit or 64-bit.
- You don’t want to waste additional 32bit on characters.

trainologic

Variable Declarations

- You can skip the type when using initializer and it would be inferred:

```
var someInt = 6

var someString = "hello"
```

trainologic

Variable Declarations

- Short syntax: you can skip the *var* keyword and use := instead of = to get a variable declaration with implicit type.
- Only inside functions.
- Not available in top-level code.

```
func main()  {


   someInt := 6

   someString := "hello"
```

trainologic

Constants

- Constants are declared with the **const** keyword.
- Note that constants can't be declared with the short syntax (:=)
- Example:

```
const fixed = 80

// compilation error
fixed = 4399
println(fixed)
```

trainologic

Const Blocks

- Instead of declaring many constants in separate expressions, you can (and should) declare them in a const block.
- Example:

```
const (
  fixed = 80
  fixed2 = 9.99
)

println(fixed)
println(fixed2)
```

trainologic

(Const Blocks (detailed

- Const blocks are composed of a list of ConstSpecs.
- A ConstSpec is composed of a list of identifiers and an ExpressionList.
- Example:

```
const (
  // ConstSpec
  fixed, i = 80, 10  // ExpressionList
  // ConstSpec
  fixed2 = 7.7 // ExpressionList
)
```

trainologic

Constants

- The first *ConstSpec* must have an expression list.
- If the expression list is omitted in a *ConstSpec*, the identifiers get the same expression list from the preceeding *ConstSpec*.
- Example:

```
const (
  fixed, i = 80, 10  // ExpressionList
  fixed2, j  // same values as above
)
```

trainologic

iota

- iota is a constant generator that generates untyped integer constants.
- It is incremented automatically after each **ConstSpec.**
- Whenever the keyword **const** appears in the source, *iota* is reset to zero.
- Very useful for defining lightweight enums.

- Let’s see it in action…

trainologic

iota

- What will be the value of each constant in the following code?

```
const (
  a = iota
  b, c = iota, iota * 100
  d, e
  f = 800
  g = iota
)
```

trainologic

Constants

- Constants in Go must be of one of the following types:
  - Numeric.
  - String.
  - Boolean.
  - Character.

trainologic

Arrays

- Arrays have fixed size in Go.
- Declared with [n]T.
- Where *n* is the size and *T* is the element type.
- The built-in *len* function returns the size of the array.
- Accessing an element is done with arr[idx].
- Index is zero-based.
- Let’s see some examples…

trainologic

Arrays

```
func main()  {

  // declaring an array of ints with size 10
  var arr [10]int

  // assigning 5 to index 0
  arr[0] = 5
  // assigning 1 to index 6
  arr[6] = 1
  // prints 5
  println(arr[0])
  // prints 1
  println(arr[6])
  // prints 0
  println(arr[7])
  // prints 10
  println("Array size: " + strconv.Itoa(len(arr)))

}
```

trainologic

Arrays

- The length of the array is part of its type.
- For example:

```
var arr1 [10]int
var arr2 [10]int
var arr3 [4]int

arr1 = arr2
arr2 = arr1
// Compilation error
// not of the same type
arr3 = arr2
```

trainologic

Arrays

- You can initialize arrays on the spot.
- Example:

```
myarr := [3]int {10,20,30}

fmt.Println(myarr)
```

trainologic

Arrays

- When initializing the array on the declaration site, you can let the compiler figure out the size by itself.
- Example:

```
myarr := [...]int {10,20,30}

fmt.Println(myarr)
```

trainologic

Arrays

- Important: arrays are values. They are copied on assignment or when sent as a function argument.
- In most cases this will result in an incorrect result or in a performance hit.

trainologic

Slices

- Usually, you'll not work with arrays directly (due to the fact that they are values).
- You'll work with slices.
- A slice is a reference to an array.
- It is analogous to an array pointer.

trainologic

Slices

- A slice holds a pointer inside the array, a length and the capacity (the maximum allowed length inside the underlying array).
- Example:

```
// array
var arr4 [10]int
// slice
var slice1 []int = arr4[0:2]
// prints 2
println(len(slice1))
// prints 10
println(cap(slice1))
  // slice
slice2 := arr4[2:4]

// prints 2
println(len(slice2))
// prints 8
println(cap(slice2))
```

trainologic

Slices

- Slices, like arrays can also be initialized on declaration:

```
msgs := []string {
  "hello",
  "go",
  "python",
  "java",
}
```

trainologic

Slices Representation

- A slice, behind the scenes, contains a pointer into the array, a length and capacity.
- The capacity is the maximum allowed growth of the slice (from current location until the end of the underlying array).
- s2 = s2[:cap(s2)+1]
- Runtime error!

trainologic

Working with Slices

- Usually, you create slices with the ***make*** function (more about it later).
- The *make* function accepts a type, length and capacity.
- Go provides with 2 additional functions to work with slices:
  - copy – accepts destination and source and returns the number of elements copied (min(len(dst), len(src))).
  - append – appends elements to a slice, possibly allocates a new underlying array.

trainologic

Important

- Note that the underlying array will not be collected (disposed) as long as there is a slice pointing into it.
- May cause memory problems in some cases.

- Consider using copy/append to avoid these situations!

trainologic

For Loops

- Go doesn’t have a *while* loop.
- Only a *for* loop.
- The general syntax for ”*for loops*”:
- For loops have 3 parts:
  - Initializers.
  - Condition.
  - Step.
- Note that you don’t put parenthesis around these and you must provide curly braces for the loop’s body.

trainologic

For Loops

- Example:

```
for i := 0; i < 10; i++ {
   println(i)
}

for i , j := 0, 0; i < 10; j,i = j+1,i+1 {
   println(i * j)
}
```

trainologic

For Loops

- If you only specify the condition part, you get the equivalent of a *while* loop from other programming languages.
- If you don’t specify any part except the *for* loop’s body, you have an endless loop.

trainologic

Ranges

- Using the ***range*** keyword you can iterate over: slices, array, string or map.
- Produces two iteration variables:
  - Arrays, slices, pointers to arrays – idx, value.
  - maps – key, value.
  - strings – byte index, rune.
- Can also work on channels (discussed later).

trainologic

Ranges

- Important, the range expression is evaluated once.
- Constant-sized arrays will not cause any *range* evaluation.
- *nil* values will produce zero iterations!

trainologic

if/else

- If statements resemble *for* loops (without the *step* clause).
- The initializer part is optional. There are no parenthesis.
- Example:

```
x := 5
if y := x * x; y < 20 {
   println("if")
} else {
   println("else")
}

if x < 10 {
   println("again")
}
```

trainologic

switch

- Go provides a *switch* statement.
- No fall-through unless explicitly stated.
- Example:

```
x := 5

switch z := x*x; z {
case 25:
  println("hello")
  println("world")
  fallthrough
case 30:
  println("false")
}
```

trainologic

switch

- If you drop the condition, you get a concise syntax for if-else chains.
- Due to the fact that case clauses can contain expressions.
- Example:

```
x := rand.Int()

switch {
case x < 10:
  println("x < 10")
case x > 10:
  println("x > 10")
default:
  println("x == 10")
}
```

trainologic

Structs

- A struct is a group of fields.
- General syntax: *type* name *struct* { fields }.
- Example:

```
type Point struct {
  x int
  y int
}
```

trainologic

Structs

- Struct instances are created like the following:

```
var p1 Point = Point{10,20}
p2 := Point{10,20}

p3 := Point{y:20, x: 10}
```

trainologic

Structs

- You can access a struct field by using dot (.) followed by the field name.
- Example:

```
var p1 Point = Point{10,20}

// print the x+y value:
println(p1.x + p1.y)
```

trainologic

Functions

- Function declaration in Go starts with the ***func*** keyword.
- General syntax is: *func name (args) return_type { body}.*
- Example:

```
func greaterThan10(x int) bool {
  return x > 10
}
```

trainologic

Functions

- Functions can have zero or more arguments.
- When you have several consecutive arguments with the same type, you may state the type only on the last one.
- Example:

```
func someFunc(y, x int, h ,o bool) bool {
  return …..
}
```

trainologic

Functions

- Functions in Go can return multiple results.
- You can even name the results.
- If you name the results, it defines the variables at the beginning of the function.
- A *return* without values, will automatically return those variables.

- Let’s see an example…

trainologic

Multiple Results

```
func DivAndMod(a,b int) (q,r int) {
  q = a / b
  r = a % b
  return
}

func main()  {
  q,r := DivAndMod(10,3)
  // prints 3
  println(q)
  // prints 1
  println(r)
```

trainologic

Naming Conventions

- As you may have noticed, Go uses CamelCase notation.
- Functions that are exported (i.e., part of your public API) should start with an uppercase letter.
- If it's part of your internal API, it should start with a lowercase letter.
- The same goes for variables.

trainologic

Exports

- So, what is exported from your package (i.e., visible to other packages)?
- Everything that starts with an uppercase.
- “export” is not a keyword.

trainologic

main & init

- The entry point in every Go application is the function *main* in the package *main*.
- The function *main* takes no arguments and has no return value.
- If a package contains the *init()* function, the function will be run before the *main* function in the main package.

- Allows for initialization and creating pluggable architectures.

trainologic

init

- Note that each file can have its own *init*() function.
- Actually there can be several *init*() functions in the same file.
- They will be executed by order of appearance in the file.
- But only after global variables have been initialized!

trainologic

Pointers

- Go has pointers.
- A pointer holds the memory address of a variable.
- Note that Go doesn’t provide pointer arithmetic like C.
- The zero value for a pointer is: ***nil***.

trainologic

Pointers

- Note that Go is a **by-value** language.
- It means that when an argument is passed to a function, it will pass by value.
- I.e., its value will be copied.
- Pointers are just a memory address number.

trainologic

Pointers

- Go guarantees that the object pointed by the pointer will be ”alive” and reachable as long as it is referenced.
- Due to the fact that there is no pointer arithmetic, pointers and arrays are not equal like in C.

trainologic

Pointers Syntax

- Pointer types are regular types preceded by an asterisk (*).
- Example:

```
func f1(a int) {
  a = 8
}

func f2(a *int) {
  *a = 8
}

func main()  {
  xx := 10
  f1(xx)
  println(xx)  // prints 10
  // passing the pointer to xx
  f2(&xx)
  println(xx) // prints 8
```

trainologic

Defer

- The *defer* keyword allows to provide some code that will be executed when the function returns.
- When several *defer* statements are provided, they are executed in a LIFO manner.
- One classic usecase is to provide resource-cleaning operations.

- Let's see an example…

trainologic

Defer

```
func boo() {
  defer println("This line is deffered 1!")
  defer println("This line is deffered 2!")
  println("This line is NOT deferred!")
}
```

trainologic

Defer

- It is important to note that the arguments to deferred functions are evaluated when the *defer* statement is evaluated (i.e., not at the return site).
- Also, you must provide a function invocation for *defer*.

```
func boo() {
   i := 0
   // will print 0 at the end
   defer println(i)
   i++
}
```

trainologic

The Blank Identifier

- The blank identifier, ‘_’, is used as a write-only, immediately discarded value.
- Let’s discuss several use-cases for it:
  - Discard some values from multiple results of a function.
  - Silence work-in-progress, unused imports.
  - Imports for side-effects.
  - Interface checks.

trainologic

The Blank Identifier

- When invoking a function that returns multiple results, you can ignore some results by assigning them to the blank identifier.
- Example:

```
func multipleResults() (a,b,c int) {
  return 1,2,3
}


func main()  {

  // must use unused1,unused2, need
  var unused1,need,unused2 = multipleResults()
  // must use only need
  var _,need2, _ = multipleResults()
```

trainologic

Maps

- One of the basic data-structures in Go is: ***map***.
- General syntax for the type: *map[keyType]valueType.*
- For example, a map from *int* to *string* is the type: map[int]string.
- Note that the key type must be a *comparable* type (more about this later).

trainologic

Map Initialization

- Maps should be created with *make*.
- Example:

```
m := make(map[int]string)

m[1] = "hello"
println(m[1])
```

trainologic

Maps

- If the key doesn't exist, we get the zero value (pun intended).
- The built-in function *len* returns the number of elements in the map.
- The *delete* function removes a key-value pair from the map.
- You can use two-value assignment to test if the key exists:

```
m := make(map[int]string)

m[1] = "hello"
s1 := m[2]
s2, ok := m[2]

// prints false
println(ok)
```

trainologic

Comparable Types

- Comparable types in Go are:
  - string, numbers, booleans, pointers, channels, interfaces.
  - Structs and arrays that are composed only of these types.
- Non-comparable: functions, slices, maps.

- Comparable types can be compared using ==.

trainologic

Types

- You can introduce new types with the ***type*** keyword.
- Note that two types are distinct even if they share the same layout.
- Example:

```
type XY int
type ZZ int


func main()  {


  var uu XY = 40
  var ii ZZ = 80
  // doesn't compile
  ii = uu
```

trainologic

Type Conversion

- Conversion is always explicit in Go.
- General syntax: T(v).
- Where *T* is the type and *v* is the value to convert.
- Example:

```
type XY int
type ZZ int


func main()  {


  var uu XY = 40
  var ii ZZ = 80

  ii = ZZ(uu)
```