Guide to Golang

This is a quick whirlwind tour of the Go programming language! Please feel free to use this as reference for your assignments. We **highly** recommend starting with A Tour of Go to get through the basics quickly and interactively; however, this guide will seek to be as comprehensive as possible. For other resources to learn Go, check out:

- Go by Example

Setup

To get started with Go, first ensure that you have the dev-environment properly installed by running `go version` in the terminal in your dev-environment. You should see the version of Go 1.23.0.

To set up a project, navigate to your folder and run `go mod init <mod_name>`. Your GitHub setup generally determines the module name. If your GitHub username is `jswaggin` and your repository name is `dinodb`, you should run `go mod init github.com/jswaggin/dinodb`. This creates a **Go module**, a virtual environment complete with a package system. Most Go projects should be in a Go module.

You can import external packages into your module for use in your code. Running `go get <pkg>` will add the package to your `go.mod` file and create a checksum file, `go.sum`. Then the package will then be accessible to your code. Try it out on `github.com/couchbase/vellum`!

Some useful commands:

- `go mod tidy` cleans up your packages and downloads newly added packages.
- `go clean -modcache` removes cached packages.

Building and Running

Now that you've installed and initialized a Go project, it's time to learn how to run it!

- `go build <dir>` will build a `main` package into a runnable executable.
- `go run <file.go>` will run a particular `main` package.
- `go install <dir>` will build a `main` package and link it to your `$PATH`, so it is callable from your entire machine.

We'll explain what a `main` package is in the next section. In this class, building, and running are handled by our Makefile. We typically build the package into an executable to avoid creating too many binaries in your `$PATH`.

Project Structure

The basic structure of every Go project is about the same. Your root directory will be your `go.mod` file alongside any other configuration files or build scripts, like a Makefile or README. There will be two or three top-level directories: `cmd` and `pkg`, and potentially `internal`.

`cmd` is where your project's primary entrypoint(s) will live, separated into subfolders containing a `main` package. Each `main` package is an entrypoint to your application and can be compiled into an executable. Let's say you had a file: `cmd/db/main.go` - running `go build ./cmd/db` would create an executable `./db` that would run the code inside the associated `main.go` file. Similarly, `go run ./cmd/db/main.go` would run the package. ·

`pkg` is where your project's packages live. A package is a collection of related Go files, typically implementing a single piece of logic. To use a package in your binaries, you have to import them using the module name you defined above (more on this later). Packages are self-contained, and everything capitalized will be exported from the package (e.g. `func Sample()` is exported, `func secret()` is not). The `main` package is specially designed to be compiled into a binary. You can have multiple `main` packages, each in a subfolder in the cmd directory (detailed above).

`internal` is where private packages live; the compiler guarantees privacy, and code here can't be imported outside the module. In this course, we don't use the `internal` folder.

Hello, World!

The following is a classic "Hello World" program:

```
package main

import "fmt"

func main() {
	fmt.Println("Hello, world!")
}
```

The first line of every Go file must be the package it belongs in; in this case, `main`. Next are the packages that are imported using the `import` keyword. You can (and should) import multiple packages, both from the standard library and from external sources, using the following syntax:

```
import (
    "fmt"
    "os"

    db "github.com/jcarberry/db/pkg"
)
```

You will have to import packages from your own module into other parts of your module. However, all code in a module is accessible to all other files in that module, even unexported values.

Printing

The `fmt` package is the main package for printing content out to the terminal:

```
fmt.Println("Three, Two, One")
fmt.Printf("%d, %d, %d\n", 3, 2, 1)
```

Types

Go's basic types are:

```
bool
string
int  int8  int16  int32  int64
uint uint8 uint16 uint32 uint64 uintptr
byte // alias for uint8
rune // alias for int32, represents a Unicode code point
float32 float64
complex64 complex128
```

and their pointer variants, which are prefixed with a `*`. Each type has a zero value, which is `0` for numeric types, `false` for boolean types, and `""` for strings.

You can convert between types using `Type(v)`. However, be aware of how data representation may affect the underlying data.

Variables

Variables can be declared using the **`var`** keyword or the `:=` operator:

```
var x int
x = 10
var y = 11  // Types are inferred.
z := 12     // Types are inferred.
a, b := 13, 14 // Can initialize two at a time!
```

Note that the `:=` operator is not available outside of functions.

There is also a **`const`** keyword that can be used to declare constants outside of a function. By convention, constants should be named using **`SCREAMING_SNAKE_CASE`**.

Functions

Functions can be declared like so:

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

Use the keyword func, give the function a name, and type each parameter and output. Functions can have multiple outputs and named outputs:

```
func split(sum int) (x, y int) {
       x = sum * 4 / 9
       y = sum - x
       return
}
```

Calling functions is rather straightforward as well:

```
fmt.Println(add(10, 11)) // Prints 21
```

You can also define functions anonymously and inline:

```
isEven := func(x int) bool { return x % 2 == 0 }
```

Loops

Go loops are declared using the **`for`** keyword. There are two main ways to write a for loop, traditionally and using the **`range`** keyword:

```
for i := 0; i < 10; i++ {
    fmt.Println(i)
}

primes := [6]int{2, 3, 5, 7, 11, 13}
for idx, val := range primes {
    fmt.Println(idx, val)
}
```

While loops are written as a for loop with no condition:

```
for {
    fmt.Println("ever")
}
```

Control Flow

if statements are written as follows:

```
x := 10
if x < 8 {
    return 1
} else if x > 10 {
    return 2
} else {
    return 3
}
```

You can declare variables to be used in the `if` statement, but variables declared like this will be scoped to the if block, and be inaccessible outside of it:

```
if v := f(); v > 10 {
    return true
}
fmt.Println(v) // will error
```

To check a number of cases, use the `switch` statement:

```go
switch v {
case 10:
    return false
case 12:
    return true
default:
    return false
}
```

To have a function be invoked only when the calling function returns, use the **`defer`** keyword:

```go
func sum() {
    defer fmt.Println("Yoohoo")
    fmt.Println("Yahoo")
    // Prints "Yahoo" then "Yoohoo."
}
```

Pointers

Go has pointers that hold memory addresses. If you have never worked with pointers before, we recommend reading up about C pointers to get an idea of how they work (Go pointers are similar to C pointers, just without pointer arithmetic). The zero value of a pointer is **`nil`**. Define and dereference a pointer like so:

```go
x := 10      // x holds 10
ptr := &x    // ptr holds a reference to x
val := *ptr    // val holds the value of x
```

You'll often see constructors that create pointers:

```go
func NewDog() *Dog {
    return &Dog{paws: 4, rating: 10}
}
```

Arrays, Slices, and Maps

Arrays are fixed-size composite data types. The type [n]T is an array of n values of type T. Declare an array, filled with its zero value or initialized yourself, like so:

```
var names [2]string
primes := [6]int{2, 3, 5, 7, 11, 13}
```

Slices are dynamically-sized views of an array; they are much more commonly used than arrays. The type []T is a slice of values of type T. While there are many ways to define a slice, the most useful one uses the make function, while using the append function to add more elements to the slice:

```
names := make([]string, 0)
names = append(names, "sparky")
```

Use the len function to find the lengths of slices (and many other datatypes).

```
names := make([]string, 8)
length := len(names) // 8
```

Maps are key-value stores like Python dictionaries or Javascript objects. Declare and use a map like so:

```
m = make(map[string]int)
m["ten"] = 10
fmt.Println(m["ten"])

ten := m["ten"]
fmt.Println(ten)

delete(m, "ten")
fmt.Println(m["ten"]) // errors

ten_check, ok := m["ten"]
fmt.Println(ok) // false
fmt.Println(ten_check) // 0
```

Structs

A struct is a collection of fields:

```
struct Dog {
    name string
    legs int
}
```

To initialize and print a struct, see the following example:

```
sparky := Dog{name: "sparky", legs: 4}
fmt.Printf("%+v \n", sparky)
```

You can create pointers to structs and access their fields in the exact same way (automatic dereferencing):

```
sparky := Dog{name: "sparky", legs: 4}
ptr := &sparky
fmt.Println(ptr.name)
```

You can define methods on structs (functions with a struct as a receiver) like so:

```
func (d Dog) bark() {
    fmt.Println("Bark")
}

sparky := Dog{name: "sparky", legs: 4}
sparky.bark() // prints "Bark"
```

Only pointer methods can mutate a struct:

```
// Won't do anything
func (d Dog) growLeg() {
    d.legs += 1
}

// Euruka!
func (d *Dog) growLeg() {
    d.legs += 1
}
```

Interfaces

An interface is a set of method signatures. There is no implements keyword; any struct with a method for every method signature automatically implements the interface.

```go
type Animal interface {
    walk()
    name() string
}
```

The empty interface interface{} is implemented by every type, and is useful for when a type is unknown. We can cast from an interface{} type to another type (unsafely) using the i.(type) syntax:

```go
anyMap := make(map[string]interface{}) // Can put anything in this map
anyMap["one"] = 1
anyMap["two"] = "two"
anyMap["three"] = Number{value: 3}

one := anyMap["one"].(int)
one := anyMap["one"].(string) // This will panic!
```

One thing to note about interfaces is that an interface can be implemented by either a struct or a pointer to that struct; as a result, you should rarely use a pointer to an interface in a function header, as it can cause some confusion:

```go
func Wrong(i *SomeInterface) {} // This will not work as expected, even if your struct has
pointer receiver methods
```

Errors

The error type is used to express errors. It is often returned by functions to signal whether or not the function ran as expected. The following is a very common pattern in Go:

```go
func mightFail(input int) (int, error) {
    if input == 0 {
        return -1, errors.New("Can't use 0") // Insert Error
    } else {
        return 10 / input
    }
}

func main() {
    result, err := mightFail(0)
    if err != nil {
        return err
    }
    fmt.Println(result)
}
```

The errors package is useful for creating errors.

Concurrency

This section will be populated later in the course. Important topics to review include:

- Goroutines
- Channels
- sync

Testing and Benchmarking

Unit tests in Go are written in files that end in _test.go. Typically, unit tests for a given package live in the same folder as the package itself. Unit tests are simply functions that begin with Test and take one parameter of type *testing.T. You can run all of the tests for a given package using go test [-v]. The following is an example test:

```go
func TestAdd(t *testing.T) {
    a, b := 10, 11
    if a + b != 21 {
        t.Error("Addition is broken")
    }
}
```

Benchmarks in Go are the same as tests, except that they must be prefixed with Benchmark and take one parameter of type *testing.B. To run benchmarks, run go test -bench=.. Benchmark code must be run multiple times; be sure to wrap your benchmarked code in a for loop that runs at most b.N times. To have control over the benchmarking timer, for instance, to allow for setup or teardown, use b.ResetTimer, b.StopTimer, and b.StartTimer. To initiate cleanup, use b.CleanUp.

To check your test's code coverage, run go test -coverprofile=.... In general, strive to cover most of your code with tests, about 80%. Note that high test coverage does not necessarily mean good testing, and good testing does not always result in high test coverage; given that this class also uses system testing, it may be that the majority of your testing infrastructure lies there. That is perfectly fine. What's important is that you are confident that your software works as intended.

Style & Other Tips

- You might notice that Go doesn't require semicolons; it is considered poor style to include them unless when necessary.

- The interface{} type is the "Any" type in Go. All other types implement it. To cast an interface{} variable to another type, use the dot syntax. e.g. x.(int) casts interface{} variable x to an int.
- Typically, you want to access struct attributes using getters and setters; avoid accessing them directly with the dot operator.
- To return an error, use the errors package: errors.New("this is an error"). Our autograder does not care what error string you write, just that an error is thrown when expected.
- To cast any value to a string, use fmt.Sprintf("%v\n", x). This code returns a string version of x.