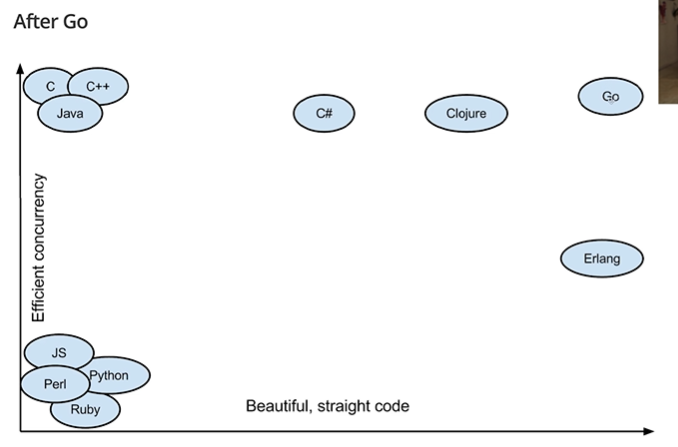
# Introduction:

* 1. Valuable resources:
* https://goo.gl/Tbz6Xf : presentations
* <https://goo.gl/KbUroF> : Github code
* Youtube: <https://www.youtube.com/user/toddmcleod>
* Twitter: <https://twitter.com/Todd_McLeod>
* Book:
  + The go programming language



* 1. Why go?

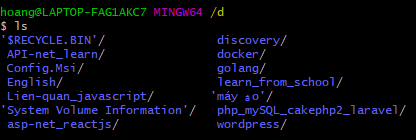
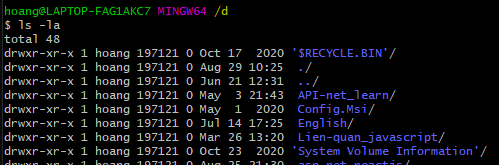
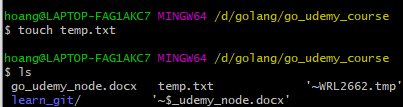
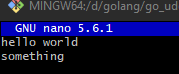
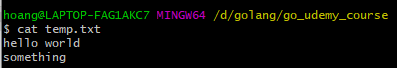
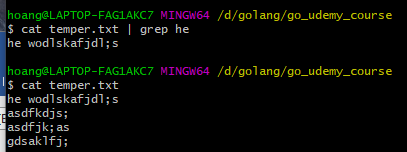
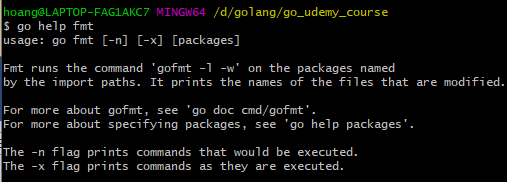


* 1. How to succeed:

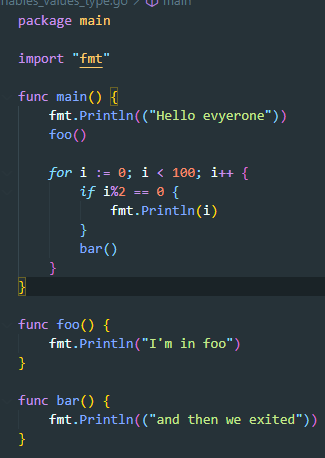
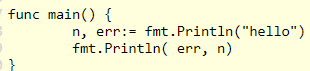
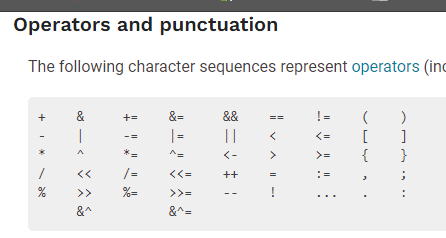
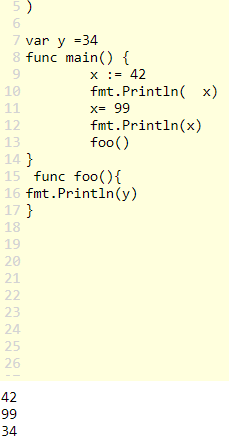
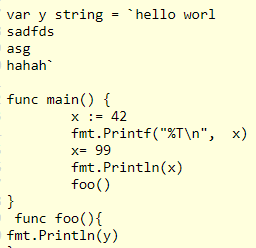
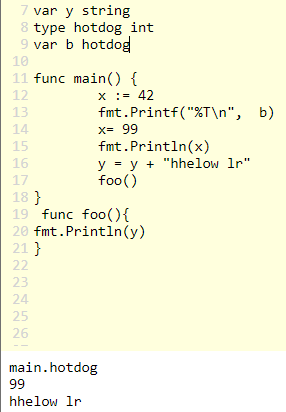
# Course Overview:

* 1. Course resources
  2. Documentation
  3. Acelerate learning

# Your development environment

* 1. The terminal
  2. Bash on windows
* Download git
* Pwd: show link recent
* 
  1. Shell/bash commands I
* Ls: 
* Clear
* Ls -la
* 
* Cd
  1. Shell/ bash commands II
* Mkdir
* 
* Touch temp.txt : create temp.txt
* 
* Nano temp.txt
*  Ctrl + x  Y -> enter
* Cat temp.txt
* 
* Clear
* Rm <file or folder name>
  + Rm -rf <file or folder name>
* .bash\_profile & .bashrc:
  + .bash\_profile is excecuted for login shells, while .bashrc is executed for interactive non-login shells. When you login (type username and password) via console, either sitting at the machine, or remove, via ssh: .bash\_profile is executed to configure your shell shell before the initial command prompt.
* Grep
  + Cat temp2.txt | grep enter
  + Ls | grep -I documents
* 
*  find folder, file
  1. Installing Go
* https://golang.org/doc/install
  1. Go modules – introduction
  2. Go workspace
* Bin, pkg, src
  1. Environment variables
  2. IDE’s
* Vs code
* Goland
* Atom.io
* sublime
  1. Go commands
* Go help
* 
* Go install main.go
  1. Github repos
  2. Github explored
  3. Package management
* Godoc.org/
  1. Go modules overview
* Blog.golang.org
  1. Creating a go module
* Go mod init
* Go build, go test
* Go list -m all
* Go get
* Go mod tidy
  1. Adding a dependency
  2. Upgrading dependencies

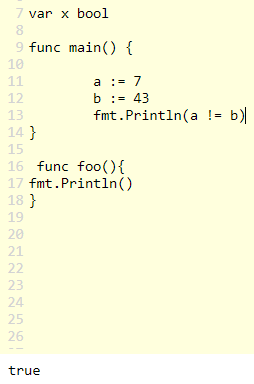
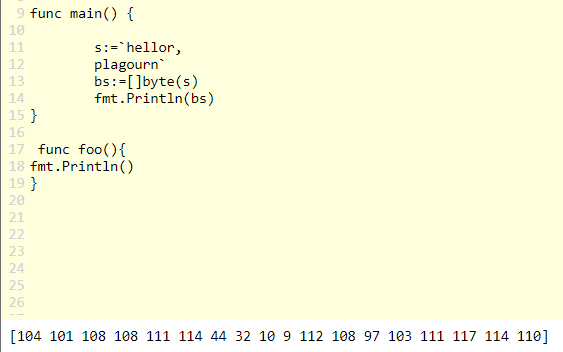
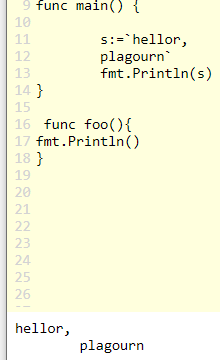
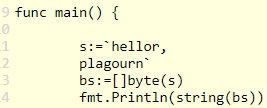
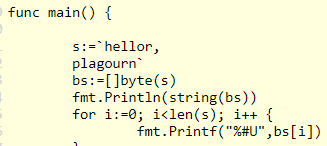
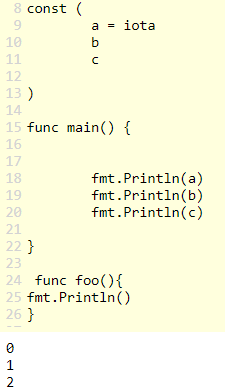
# Variables, values, & type:

* 1. Playground
* https://play.golang.org/
  1. Hello world
* 
  1. Introduction to packages
* Godoc.org/fmt
*  
  1. Short declaration operator
* https://golang.org/ref/spec#Operators\_and\_punctuation
* 
  1. The var keyword
* Declare a variable and assign a value
* global, default value variable type int is 0
* False for Booleans, 0 for integers, 0.0 for floats, “” for strings, and nil for pointers, functions, interfaces, slices, channels, and maps.
* Declare and assign = initilization
  1. Exploring type
* `` 
  1. Zero value
  2. The fmt package
* https://pkg.go.dev/fmt
  1. Creating your own type
*  type
  1. Conversion, not casting

# Exercises – Ninja level 1

* 1. Hands-on exercise #1
  2. Hands-on exercise #2
  3. Hands-on exercise #3
  4. Hands-on exercise #4
  5. Hands-on exercise #5

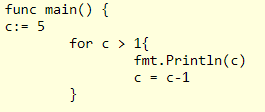
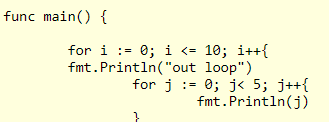
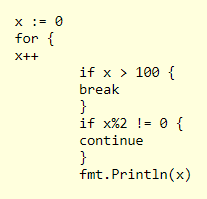
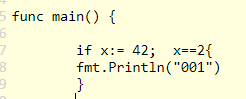
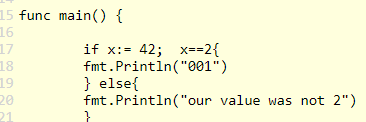
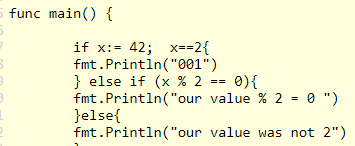
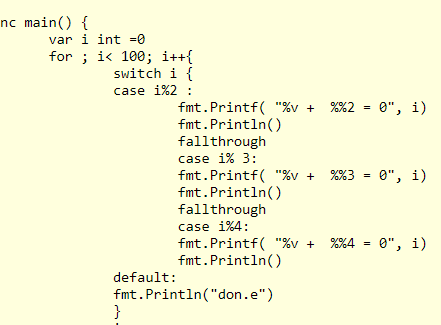
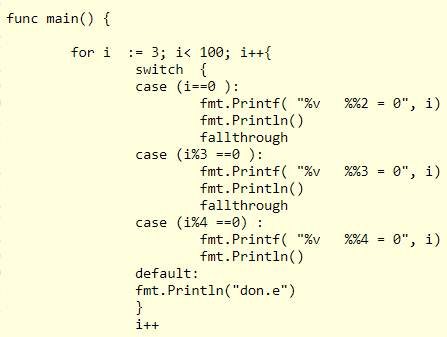
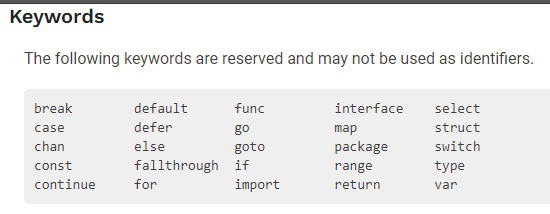
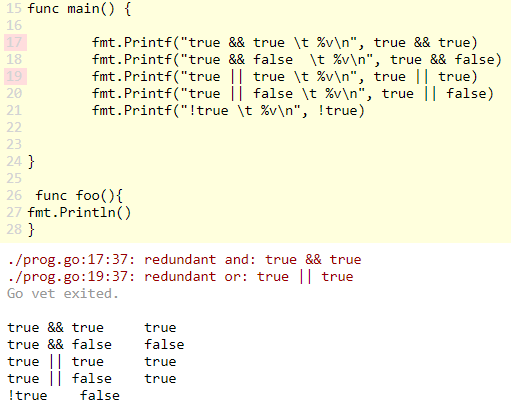
# Programming fundamentals

* 1. Bool type
* 
  1. How computers work
* Computers run on electricity
* Electricity can be on or off
* Coding schemes
  + Assign meaning to on or off state
  + Porch light on Halloween 1 porch light = 2 messages
  + 2 porch lights = 4 messages 2^n
* 000: A
* 001: B
* 010: C
* 100: D
  1. Numeric types
* Unint8(8bit) uint16 uint32 uint64 (0 -> …)
* Int8(8bit) int16 int32 in64
  1. String type
* 
* binary string
* 
* 
* 
  1. Numeral systems
  2. Constants
  3. Lota
* 
  1. Bit shifting

# Exercises – Ninja level 2

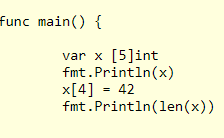
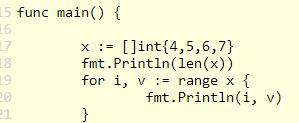
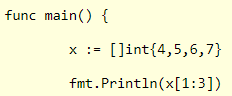
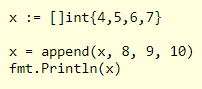
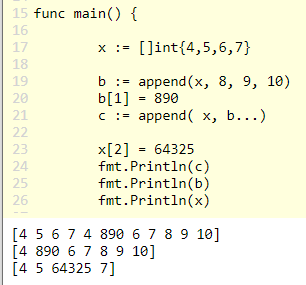
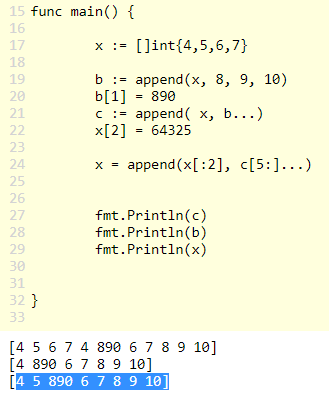
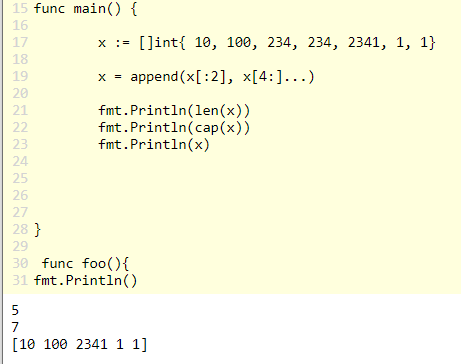
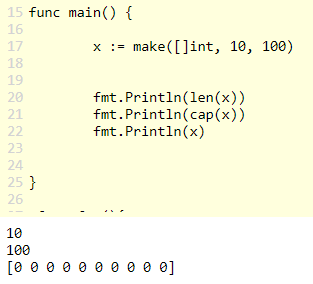
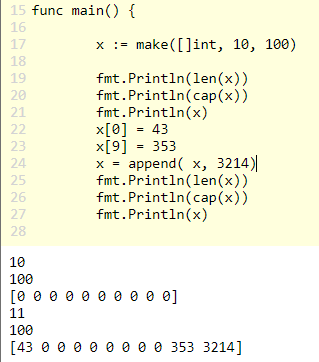
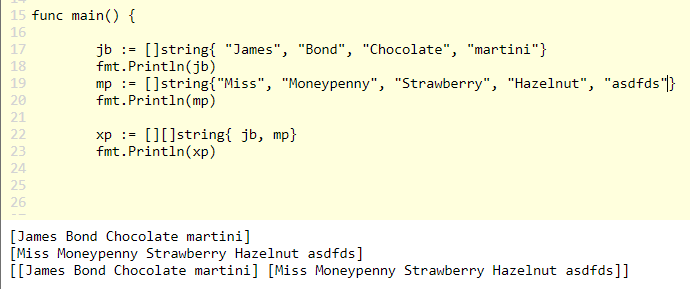
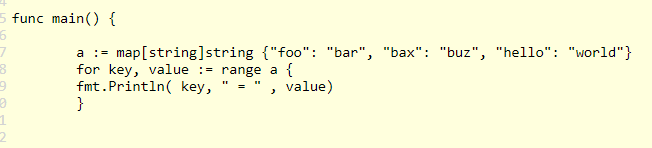
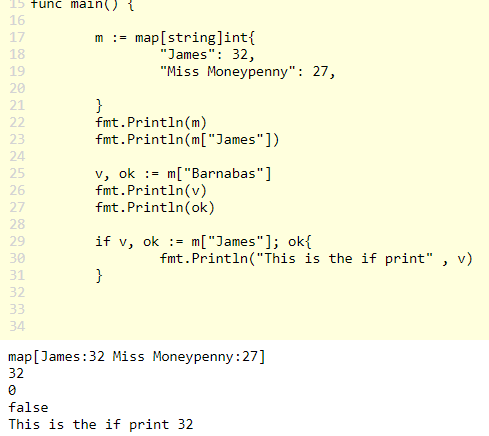
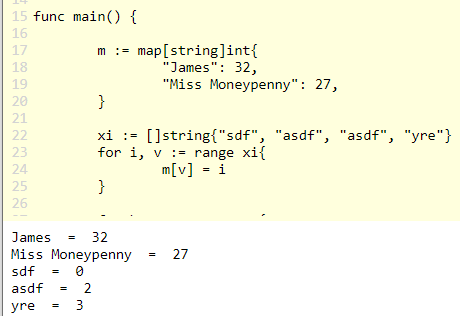
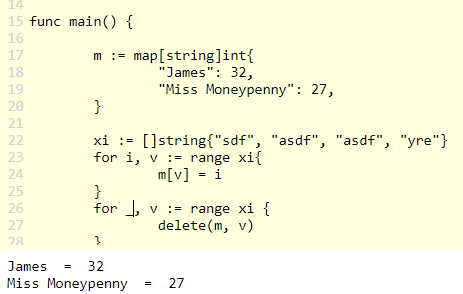
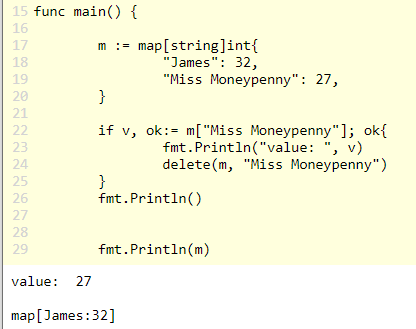
* 1. Hands-on exercise #1
  2. Hands-on exercise #2
  3. Hands-on exercise #3
  4. Hands-on exercise #4
  5. Hands-on exercise #5
  6. Hands-on exercise #6
  7. Hands-on exercise #7

# Control flow

* 1. Understanding control flow
  2. Loop – init, condition, post
* 
  1. Loop – nesting loops
* 
  1. Loop – for statement
* For ( condition | forClause | rangeclass)
* For [InitStmt] “;” [Condition] “;” [PostStmt]
* Like while : for condition {}
  1. Loop – break & continue
* 
  1. Loop – printing ascii
* Godoc.org/fmt
  1. Conditional – if statement
* 
  1. Conditional – if, else if, else
* 
  1. Loop , conditional, modulus
* 
* Don’t create new line when use else{
  1. Conditional – switch statement
* 
  1. Conditional – switch statement documentation
* 
  1. Conditional logic operators
* Keyword:
* 
* 

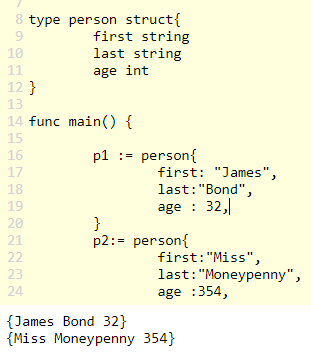
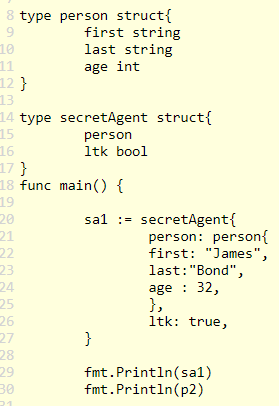
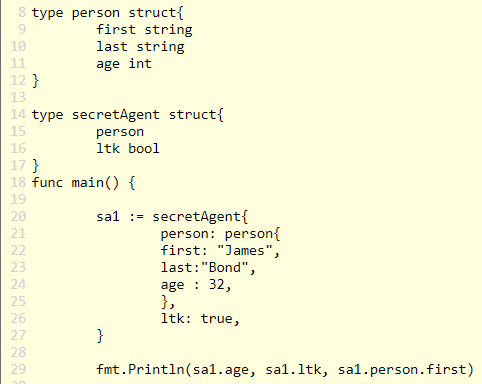
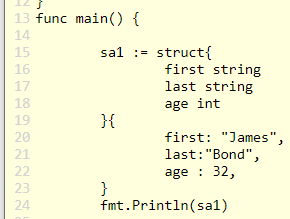
# Exercises – Ninja level 3

# Grouping data

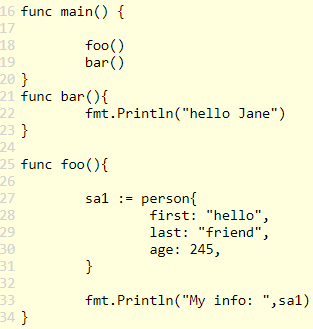
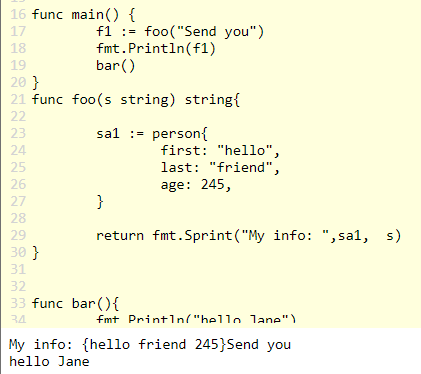
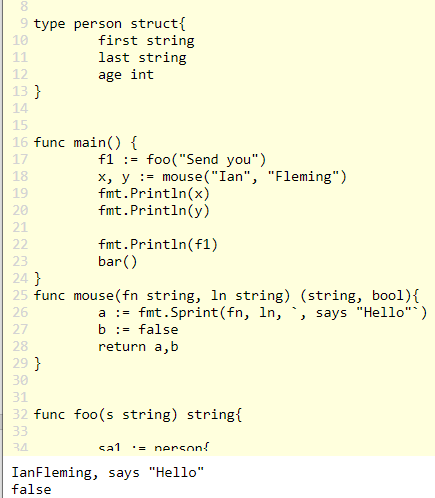
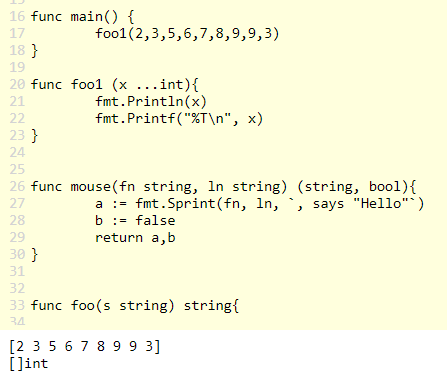
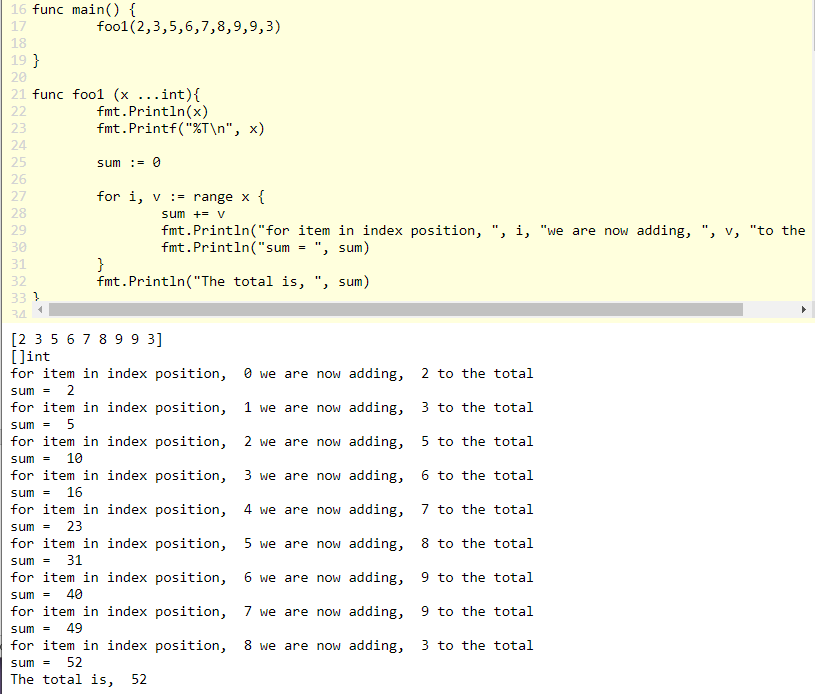
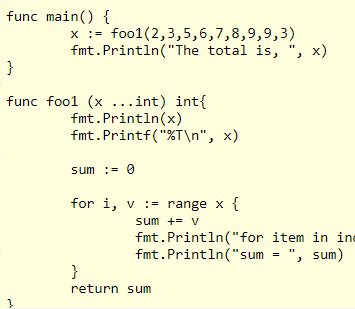
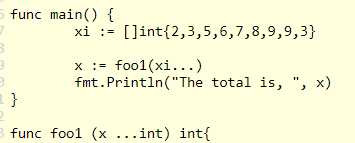
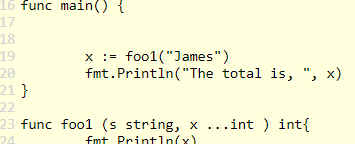
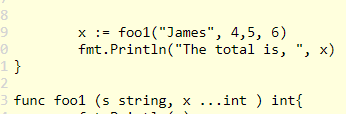
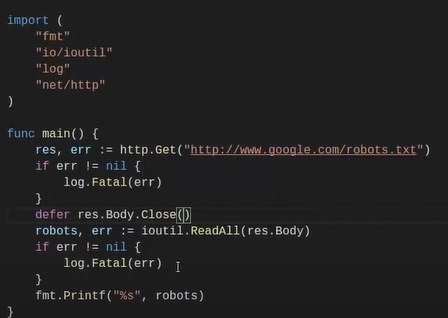
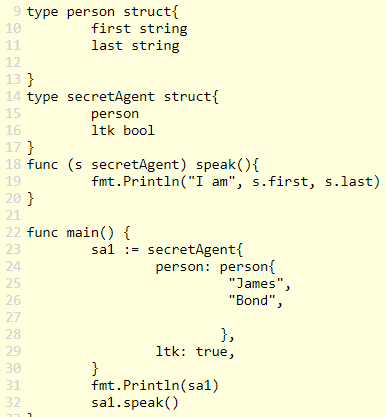
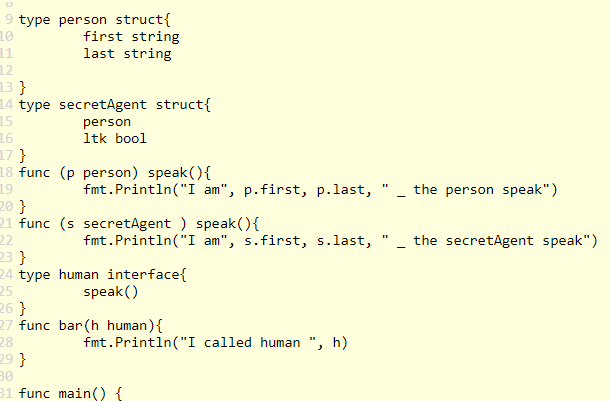
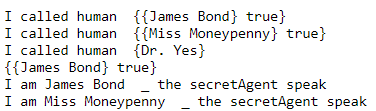
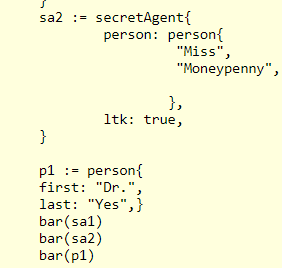
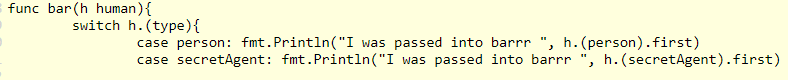
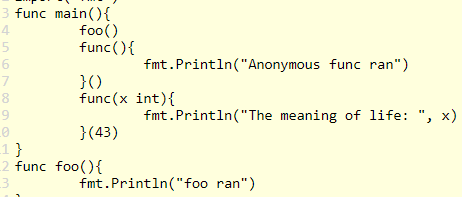
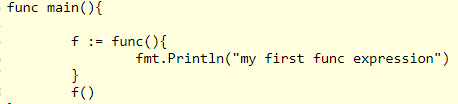
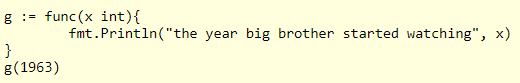
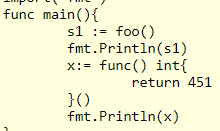
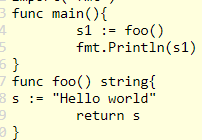
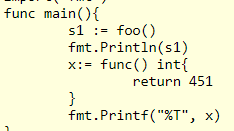
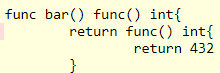
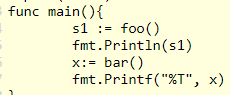
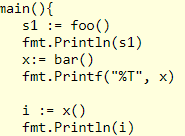
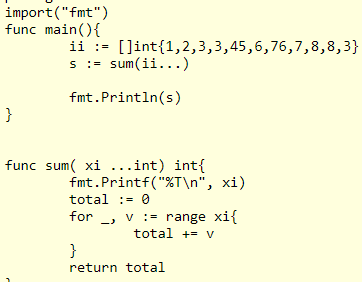
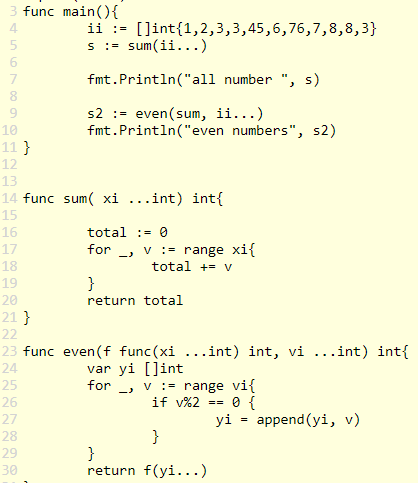
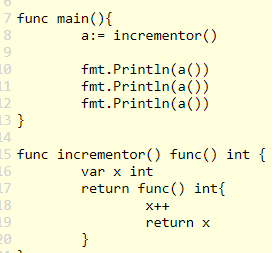
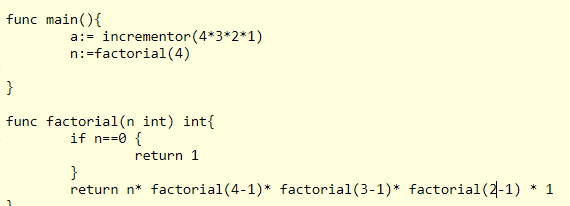
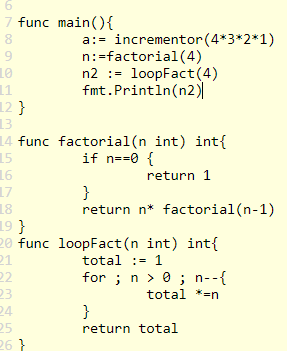
* 1. Array
* 
  1. Slice – composite literal
* X := []int {2,2,3,4}
  1. Slice – for range
* 
  1. Slice – slicing a slice
* 
  1. Slice – append to a slice
* Golang.org/doc/effective.go.html/#append
* 
* Func append(slice []T, elements …T) []T
* The underlying array may change
* … : mean, I take an unlimited number of these of this type
* 
  1. Slice – append to a slice
* Godoc.org/builtin
* 
  1. Slice – deleting from a slice
* <https://golang.org/doc/effective_go>
* 
  1. Slice – make
* 
  1. Slice – multi-dimensional slice
* 
  1. Map – introduction
* 
* Println with variable
* 
  1. Map – add element & range
* 
  1. Map – delete
* 
* delete with if

# Exercise – Ninja Level 4

# Structs

* 1. Struct
* 
  1. Embedded structs
* 
* 
  1. Reading documentation
* <https://golang.org/doc/effective_go>
* <https://golang.org/ref/spec#struct.type>
* Golang.org/doc
  1. Anonymous structs
* 
  1. Hoursekeeping
* Anonymous types are indeterminate. They have not been declared as a type yet. The compiler has flexibility with anonymous types. You can assign an anonymous type to a variable decalred as a certain type. If the assignment can occur, the compiler will figure it out; the compiler will do an implicit conversion. You cannot assign a named type to a different named type
* Padding & architectural alignment
* Convention: logically organize your fields together. Readability & clarity trump performance as a design concern. Go will be performant. Go for readability first. However, if you are in a situation where you need to prioritize performance: lay the fields out from largest to smallest, eg, int 64, float32, bool

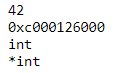
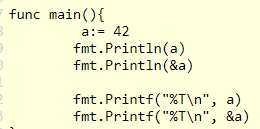
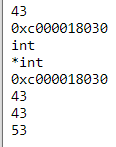
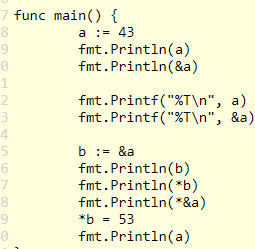
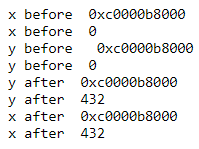
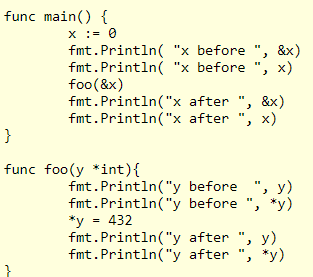
# Functions

* 1. Syntax
*  
* Return string
* Multiple argument
* 
  1. Variadic parameter
* …. In argument, array in argument
* 
* 
  1. Unfurling a slice
* …xi
* 
* <https://golang.org/ref/spec#passing_arguments_to_..._parameters>
*  
* 
  1. Defer
* 
  1. Methods
* Func (r receiver) identifier(parameters) (return(s)) {<code>
* Struct fill value
* 
  1. Interfaces & polymorphism
* Variable more one type
* 
* 
* Get type
* 
* 
  1. Anonymous func
* 
  1. Func expression
* Variable func
* 
* 
  1. Returning a func
* 
* 
* 
* 
* 
* 
  1. Callback
* 
* 
  1. Closure
* 
  1. Recursion
* 
* 

# Exercises – Ninja level 6

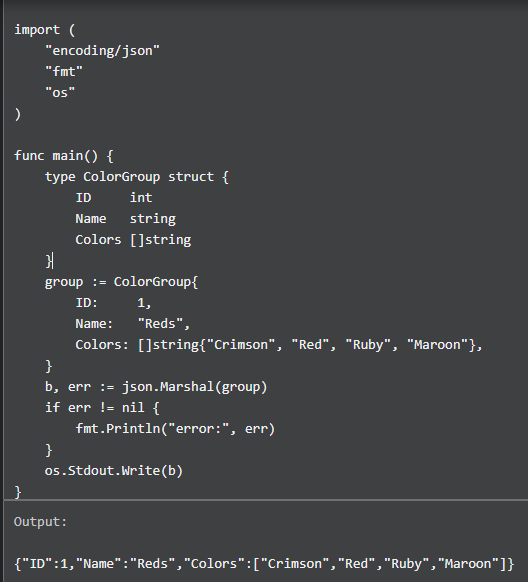
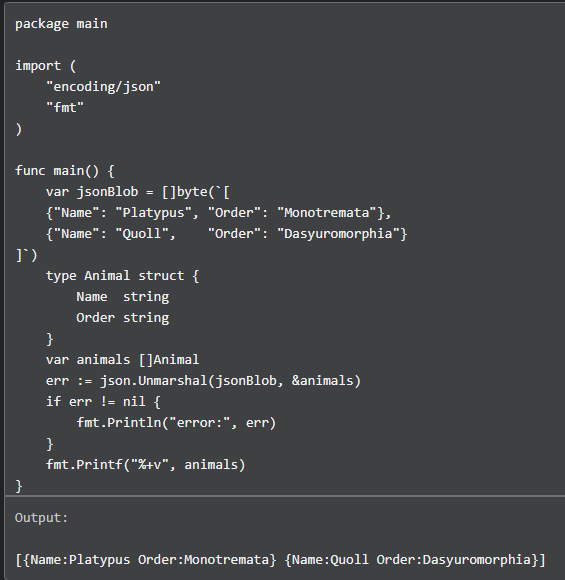
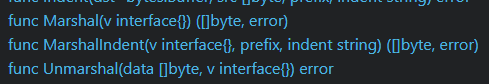
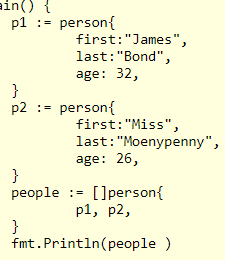
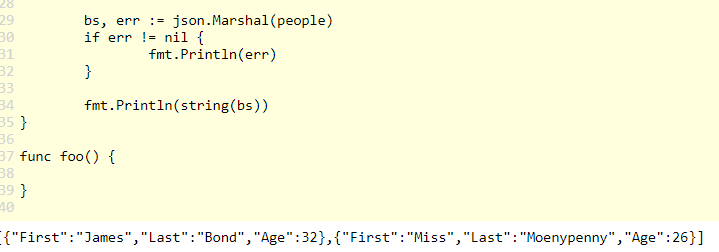
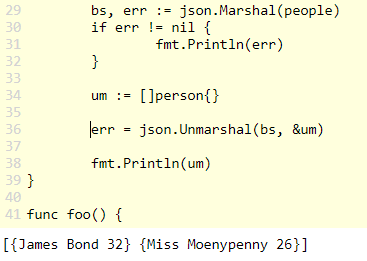
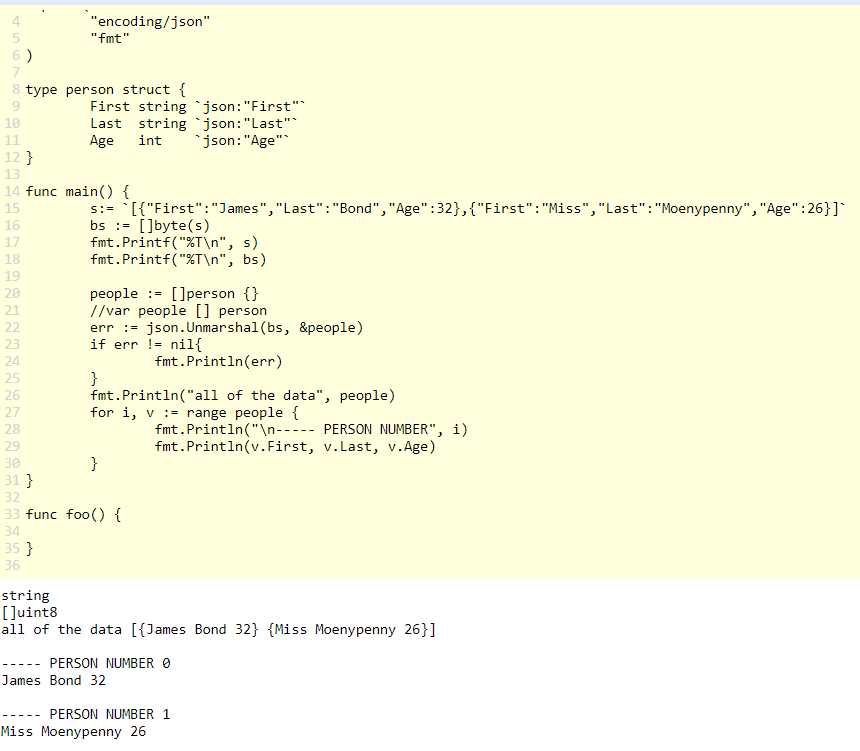
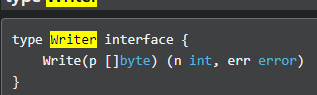
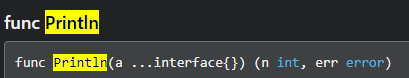
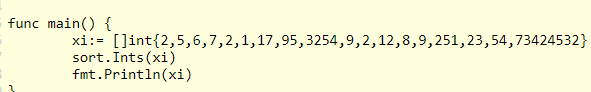
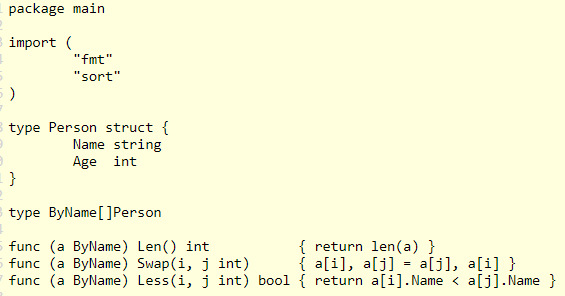
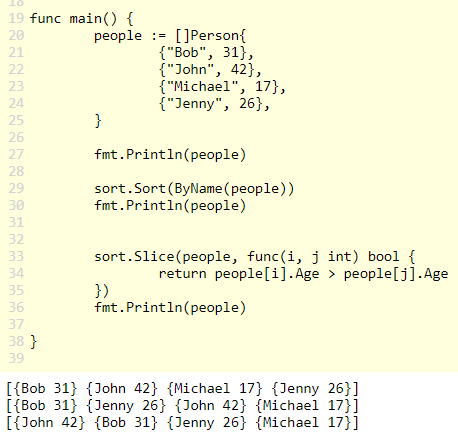
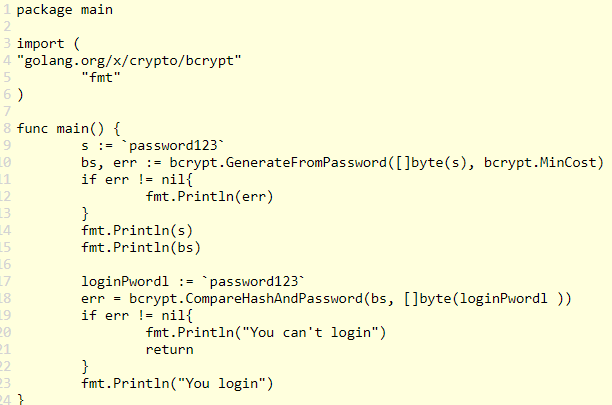
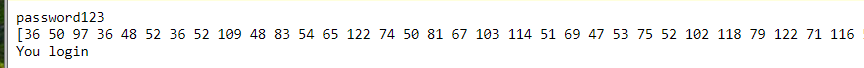
* 1. Review
* Functions:
  + Purpose of functions
    - Abstract code
    - Code reusability
      * DRY – don’t repeat Yourself
  + Func, receiver, identifier, params, returns
  + Parameters vs arguments
  + Variadic func
    - Multiple “variadic params”
    - Multiple “variadic” args
  + Returns
    - Multiple returns
    - Named returns – yuck!
  + Func expressions
    - Assigning a func to a variable
  + Callbacks
    - Passing a func as an argument
  + Closure
    - One scope enclosing another
    - Variables declared in the outer scope are accessible in inner scopes
    - Closure helps us limit the scope of variables
  + Recursion
    - factorial

# Pointers

* 1. What are pointers?
* 
* Give me a address and give me a value: \*&a
* 
  1. When to use pointers
* 
  1. Methods sets
* Golang.org/ref/spec#Method\_sets

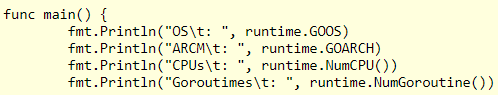
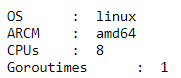
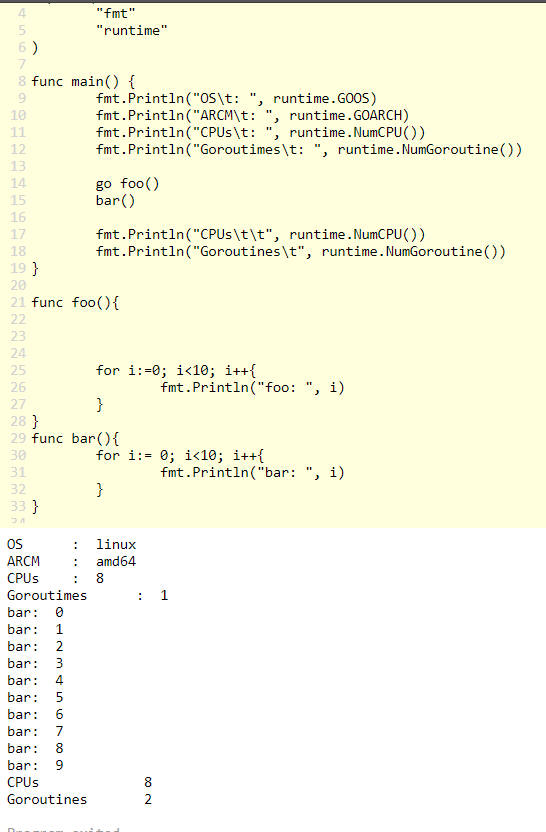
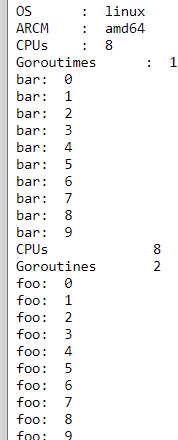
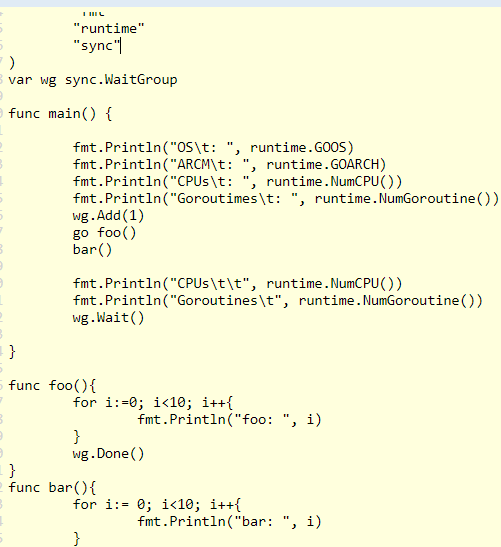
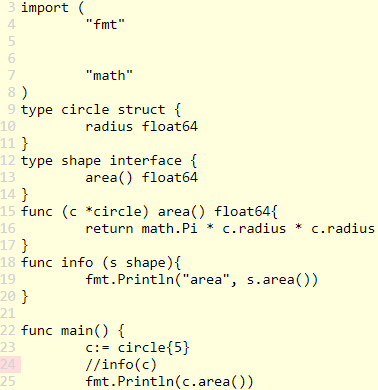
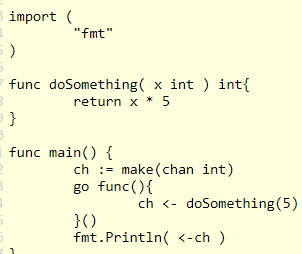
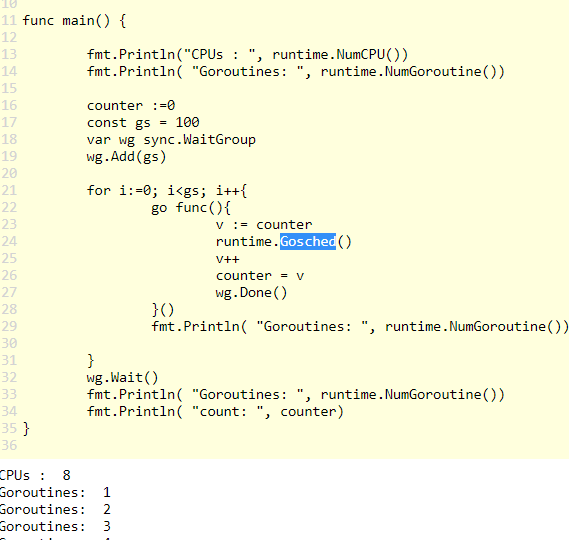
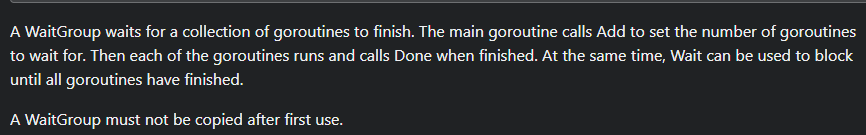
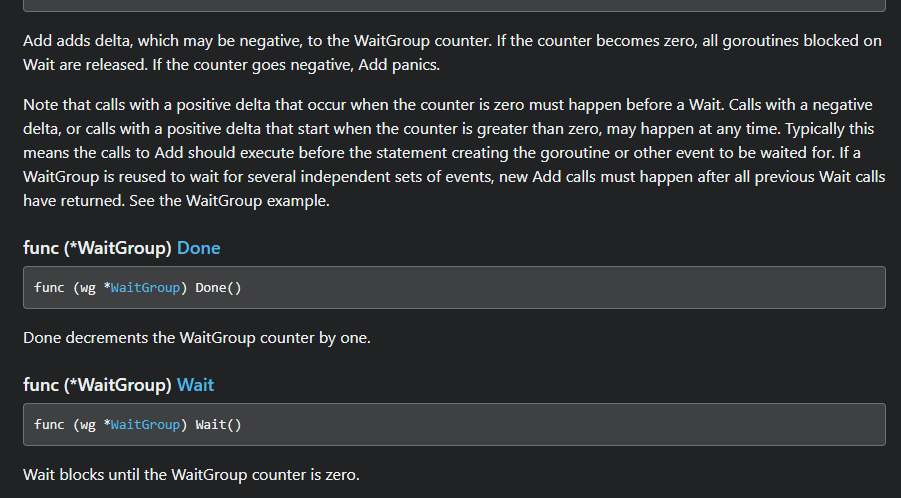
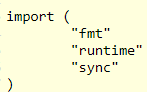
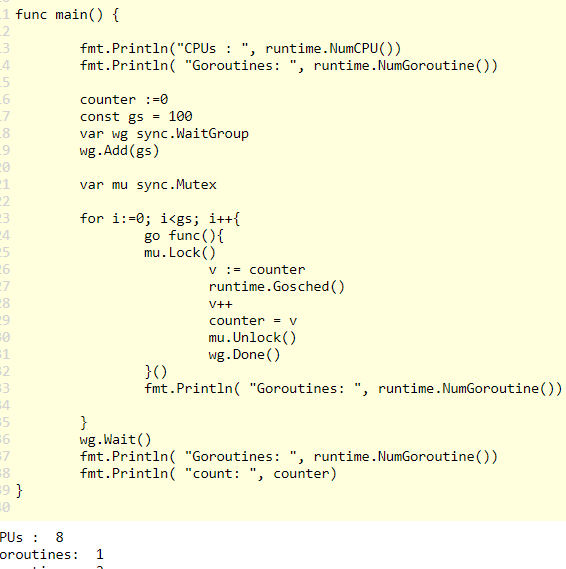
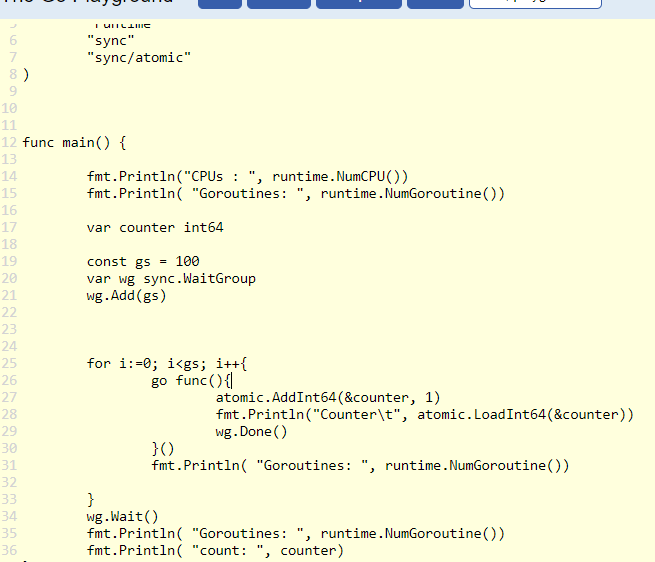
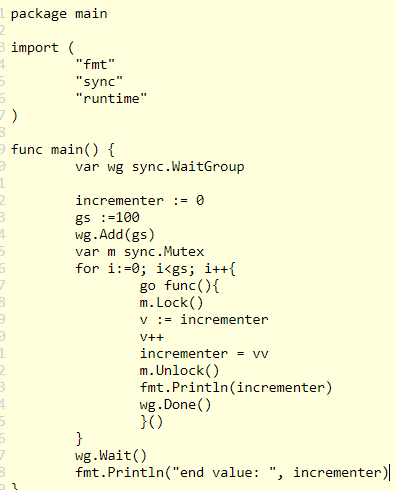
# Exercises – Ninja level 7

# Applications

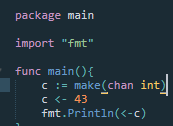
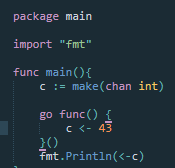
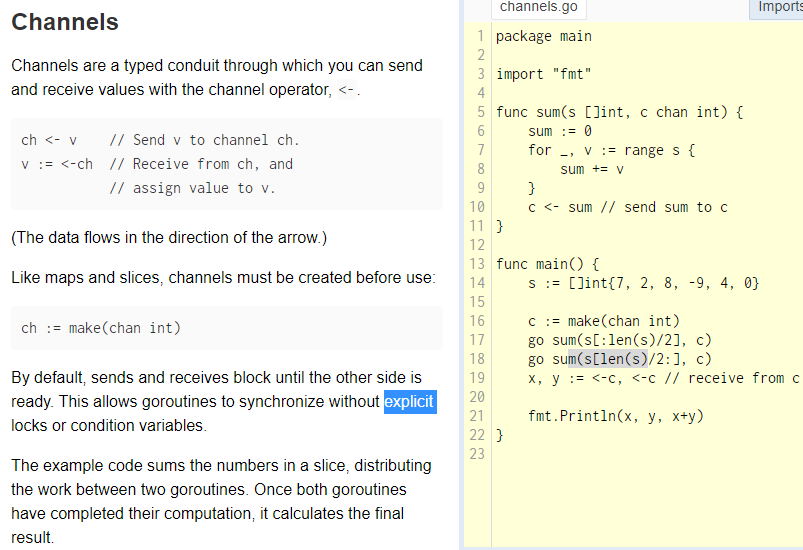
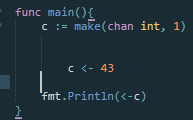
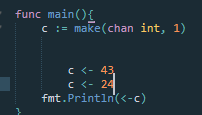
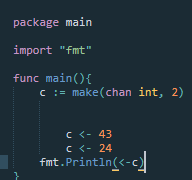
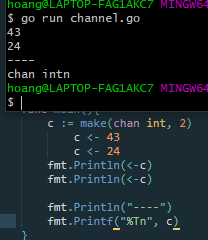
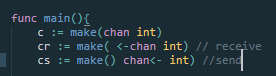
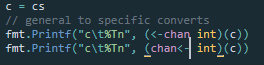
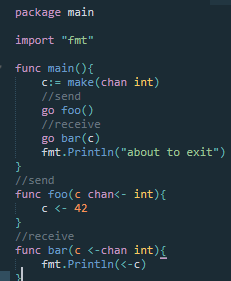
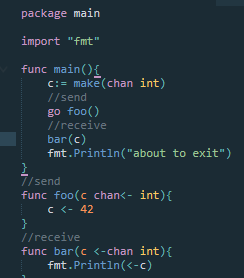
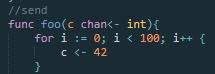
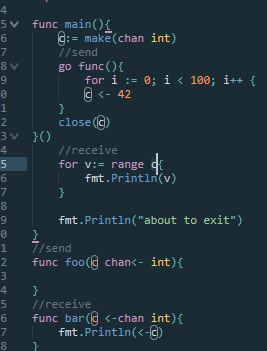
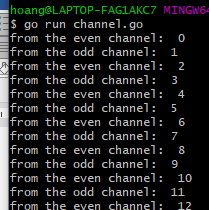
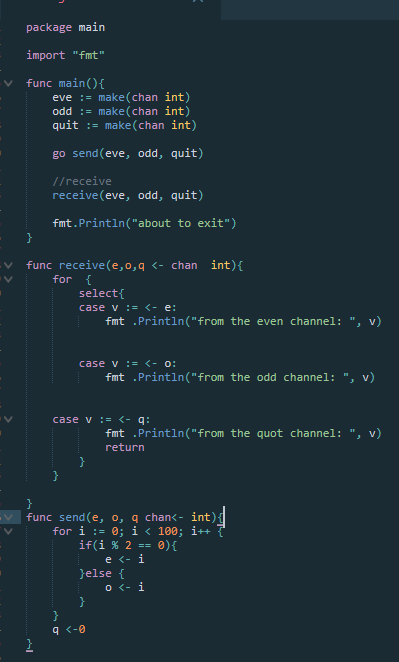
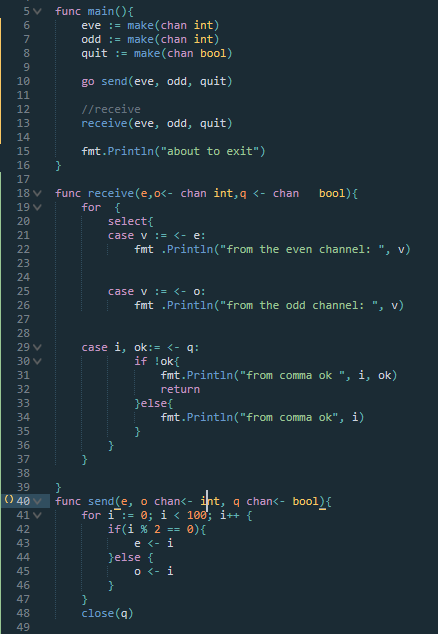
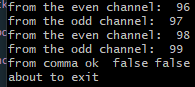
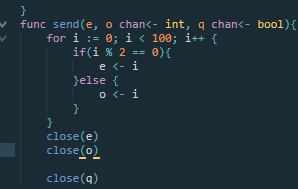
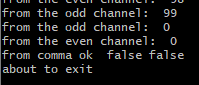
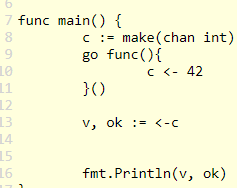
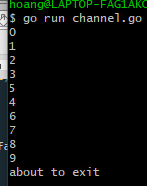
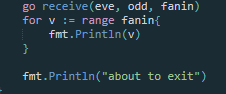
* 1. JSON documentation
* <https://golang.org/ref/spec>
* <https://pkg.go.dev/search?q=encoding>
* <https://pkg.go.dev/encoding/json>
* <https://pkg.go.dev/encoding/json#example-Marshal>
* 
* 
* 
* Empty interface means it takes a value of any type and it return a slice of bytes and an error, marshal
  1. JSON marshal
* Object in array:
* 
* 
  1. JSON unmarshal
* 
* Json to go
* <https://mholt.github.io/json-to-go/>
* 
* Json
* 
  1. Writer interface
* <https://pkg.go.dev/io>
* <https://pkg.go.dev/io#Writer>
* 
* 
  1. Sort
* Sort array
* 
* 
* 
* 
  1. Sort custom
* <https://pkg.go.dev/sort#example-package>
* Đổi giá trị 2 số
* 
* 
* 
  1. Bcrypt
* <https://pkg.go.dev/golang.org/x/crypto/bcrypt>
* Update
* Go get -u golang.org/x/crypto/bcrypt
* 
* 

# Exercises – Ninja level 8

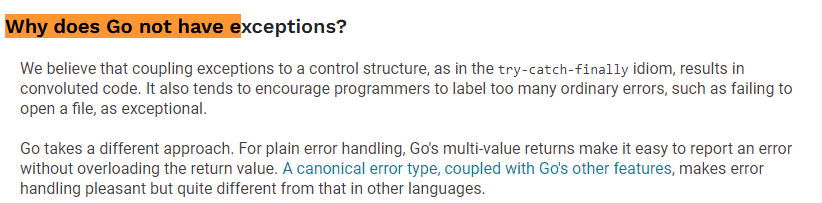
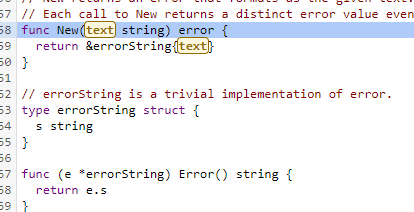
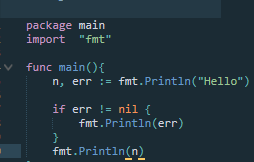
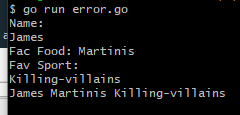
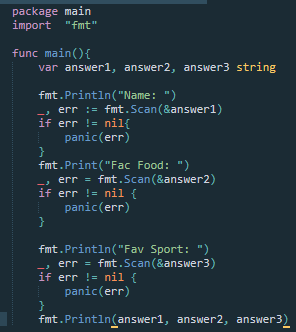
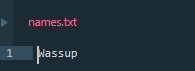
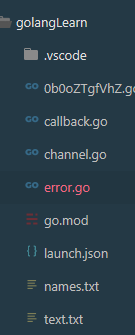
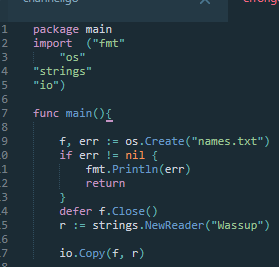
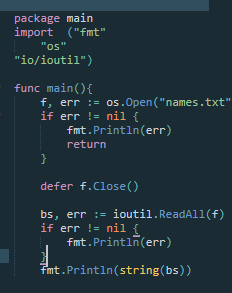
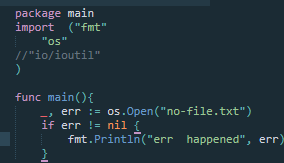
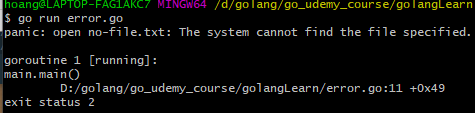
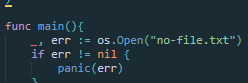
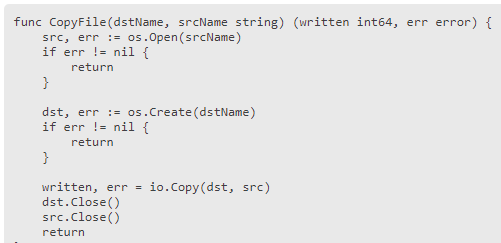
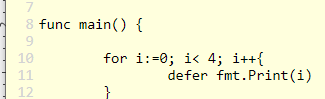
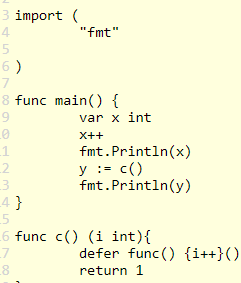
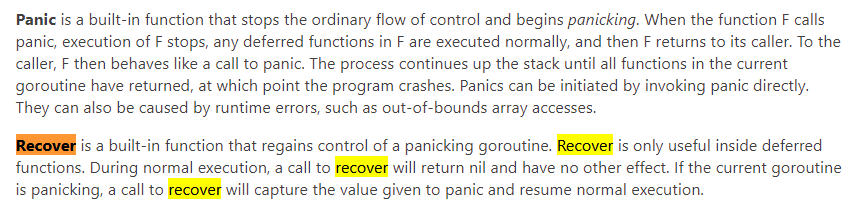
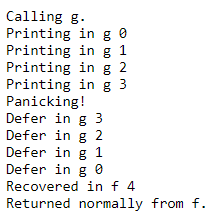
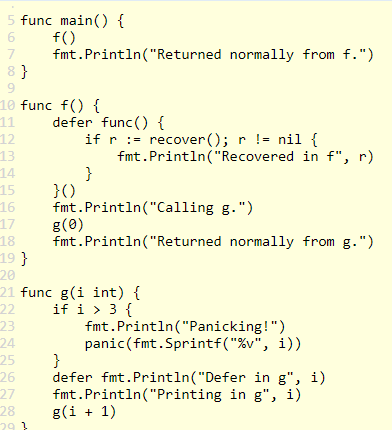
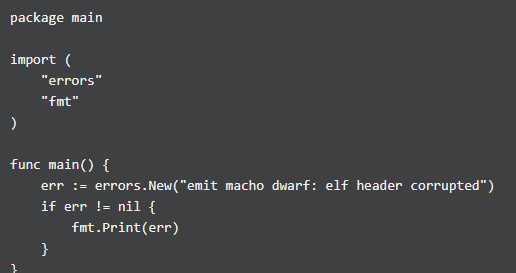
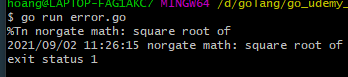
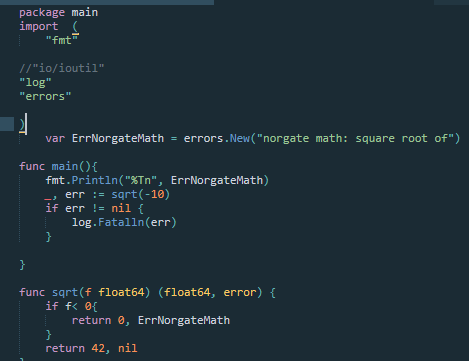
# Concurrency

* 1. Concurrency vs parallelism
* Parallelism: song song
  1. WaitGroup
* <https://pkg.go.dev/runtime>
* 
* 
* 
* 
  1. Method sets revisited
* Pointer
* Receivers values
* (t T) T and \*T
* (t \*T) \*T
* 
  1. Documentation
* Go statements
* A “go” statement starts the execution of a function call as an independent concurrent thread of control, or goroutine, within the same address space.
* The expression must be a function or method call
* The function value and parameters are evaluated as usual in the calling goroutine, but unlike with a regular call, program execution does not wait for the invoked function to complete. Instead, the function begins executing independently in a new goroutine. When the function terminates, its goroutine also terminates. If the function has any return values, they are discarded when the function completes
* 
* Create multiple variable with initialize value
* 
* Goroutines: is a function or method which executes independently and simultaneously in connection with any other Goroutines present in your program. Or in other words, every concurrently executing activity in Go language is known as a Goroutines
  1. Race condition
* 
* 
* 
  1. Mutex
* Go run -race hello.go
* <https://pkg.go.dev/sync#Mutex.Lock>
* 
* 
  1. Atomic
* <https://pkg.go.dev/sync/atomic>
* Go run -race main.go
* 
* 

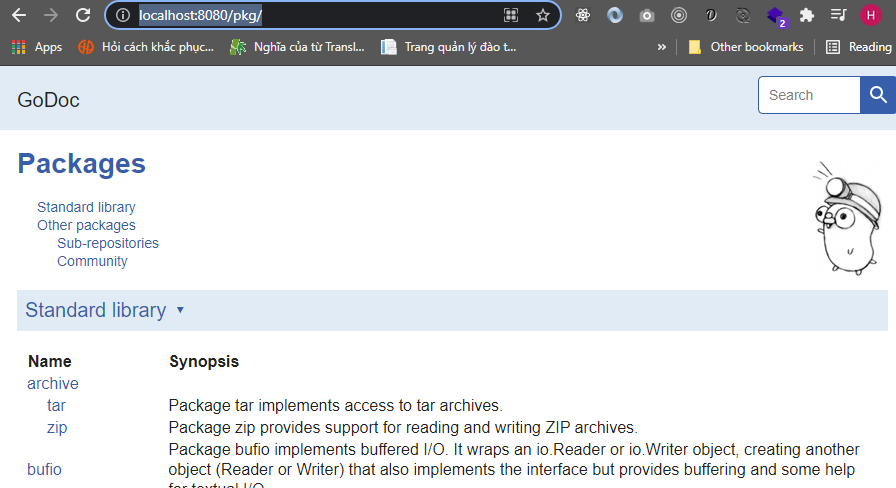
# Channels

* 1. Understanding channels
* Channel block, and this is blocking because my program enter main makes the channel gets right here and there’s nothing to pull it of
*  when you send and receive on a channel, it’s like relay racers, racers in a track race that have they have to pass a baton, and they have to pass It hand to hand and the transaction cannot occur. It can occur until both send and receive can happen at the same time
* If they can’t happen at the same time, it blocks the send and receive blocks until the receivers are ready to pull it off
* that can run, because here is the entry point to my program, func main I make my channel to this channel launches off and it’s goroutine launch Is off and is running/
* Because this goroutine(go func(){}()) has just been sent off to run on its own, the other go routine, the main goroutine comes down here, fires off this one and then the flow continues. (fmt.prin…), and then this one blocks until it takes the value off. And then so this one’s over there ready to put it on that one down here on line twelve(fmt.Println…) right there ready take it off. And they are running at the same time. They interblock, they pass the baton. And we have program coordination with concurrent design
* About go keyword:
  + Go starts a goroutine, which is managed by golang run-time
  + It can either run on the current OS thread, or it can un on a different OS thread automatically
* 
*  it run, it call buffers
* Deadlock, like your code, hit a block and nothing’s happening, and there’s no way it’s going to release, we’re done.
*  run 
  1. Directional channels
*  
* Receiving from a channel event
* And below, sending on a channel is going to be sent.
* 
* false
*  
  1. Using channels
* not work, might not have time to happen
* work
  1. Range
* Because this channel here is like now a new memory area and it’s been assigned to another, it’s a channel that sends an aunt onto it. so it’s a second channel. We’re sending values onto the channel, which is what we’re doing right here. 
* it pulls off of a channel until the channel is closed and it just is they’re pulling way to pull until the channel is closed and the channel is closed. It pulls any last value or values that are still on the channel. And then when there are no values on the closed channel, it leaves the range loop. So that’s how you close a channel.
  1. Select
* Range loop and closing the channel and the range loop is going to continue to hang out right there until the channel is closed. And so you can launch off some go routine that’s going to put values on to a channel. And then in your main go routine, you can have a range. Claw’s pulling the values off the channel and then you’re done putting them on. Close the channel over in this go routine and then this one will pull all the values off that are still there now that the channel is closed and then one last phase off, it’s going to continue its flow.
* Parameter in function
* 
*  
* Using a select statement to pull values off of multiple channels and a select statement will pull whatever value is ready to be pulled off of that channel.
  1. Comma ok idiom
*  
*  
*  
  1. Fan in
*  
* These are both running at the same time( waitgroup), and then when those are done right, we do the wg.Wait(), this is waiting until those are both finished and then we close fan in because up here we’re ranging  over farin until it’s close or printing out the values and then we print about to exit.
  1. Fan out
  2. Context
* In go servers, each incoming request is handled in its own goroutine. Request handlers often start additional goroutines to access backends such as databases and RPC services. The set of goroutines working on a request typically needs access to request- specific values such as the identity of the end user, authorization tokens, and the request’s deadline. When a request is canceled or times out, all the goroutines working on that request should exit quickly so the system can reclaim any resources they are using.
* At google, we developed a context package that makes it easy to pass request-scoped values, cancelation signals, and deadlines across API boundaries to all the goroutines involved in handling a request. The package is publicly available as context. This article describes how to use the package and provides a complete working example.
* But there are some classes of information for which a context is necessary. This is so-called request scoped data, i.e, information that can only exist once a request has begun. Good examples of request scoped data include user IDs extracted from headers, authentication tokens tied to cookies or session IDs, distributed tracing IDs, and so on.
* <https://pkg.go.dev/context>
* https://go.dev/blog/context

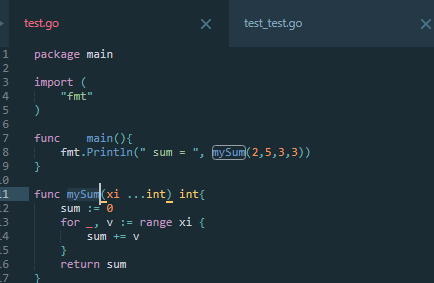
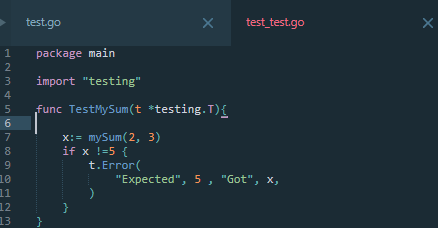
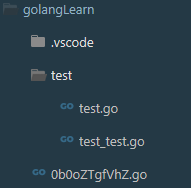
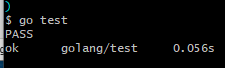
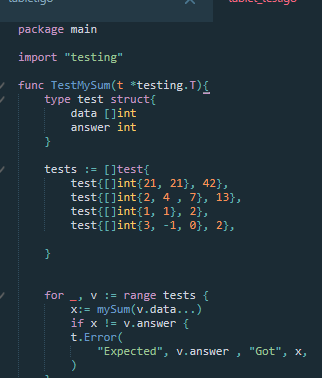
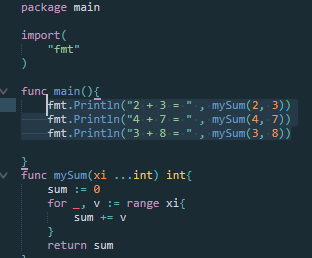
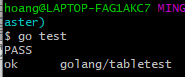
# Error handling

* 1. Understanding
* <https://golang.org/doc/faq#exceptions>
* 
* <https://go.dev/blog/error-handling-and-go>
* <https://pkg.go.dev/builtin#error>
* <https://pkg.go.dev/errors#New>
* 
  1. Checking errors
* Always, always, always check your error
* Nil: Nil is nothing 0
* 
* Lấy giá trị từ người dùng
* 
* 
* 
  1. Printing and login
* Fmt.Println()
* Log.Println()
* Log.Fatalln()
  + Os.Exit()
* Log.Panichln()
  + Deferred functions run
  + Can use “recover”
* Panic() https://pkg.go.dev/log
* 
* 
  1. Recover
* <https://go.dev/blog/defer-panic-and-recover>
* File
* 
* Defer
* 
* 
* We’re going to defer func and it’s an anonymous self executing whatever call it what you will, we’re returning one then this runs.
* And strangely enough, since it’s the return, it’ll take one and add one to it, which give us two
* <https://pkg.go.dev/builtin#recover>
* https://go.dev/blog/defer-panic-and-recover
* 
* 
* When go to panicking!
* Panic run, we go up the call stack, executing any deferred statements first. So those go in last in first out of order, three to one zero
* And then up to defer in f()
* This has been deferred and since we’re recovering, we’re going to recover our panic, and so hey, here’s Rickover as it’s not equals nil, we’re actually in a panic that’s running, printline recovered enough 4, done. Returned normally from g.
* So we never get down to return normally from g. it just runs the defer, so that’s an interesting thing to note recovered an f 4 and return normally from f is right up here
  1. Errors with info
* 
* Function with error
* 

# Writing documentation

* 1. Introduction
  2. Go doc
* Go doc <pkg> vd: go doc fmt
* Go doc fmt Println
  1. Godoc
* Godoc fmt Printf
* Godoc -http=:8080
* Localhost:8080
* <http://localhost:8080/pkg/>
* 
  1. Godoc.org
  2. Writing documentation

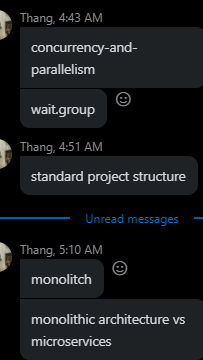
# Testing & benchmarking

* 1. Introduction
* <https://www.golang-book.com/>
* <https://www.golang-book.com/books/intro/12>
* <https://pkg.go.dev/testing>
* Go test
* Go test -v
* 
* 
* In one folder
*  
  1. Table tests
* 
*  name file, file.go file\_test.go
  1. Example tests
  2. Golint
* Formats go code
* Go vet
  + Reports suspicious constructs
* Golint reports poor coding style
  1. Benchmark
* Go test -bench .
* Go help testflag
  1. Coverage
* Go test -cover
  + Go test -coverprofile c.out
    - Show in browser:
      * Go tool cover -html=c.out
    - Learn more
      * Go tool cover -h
  + https://github.com/GoesToEleven/go-programming
  1. Benchmark examples
  2. Review

# Farewell

* 1. Farewell

# Learn

* Monnolism: để trong 1 mục
* 
* lập trình nhúng, xây dựng API nhanh chóng
* build ra file binary, 1 project gồm các file nhẹ
* quy tắc ông viết golang:
* không muốn phức tạp hóa ngôn ngữ, php, c# phải học framwork
* golang biết được chính xác bên dưới làm gì,
* C# cho 1 framework để viết ra thuật toán
* golang phải tự viết, viết function đơn giản.
* sử lý đa luồng thresh java, C#
* concurrency: 1 thresh có thể có vài trăm ngàn
* so sánh concurrency vs pallelism