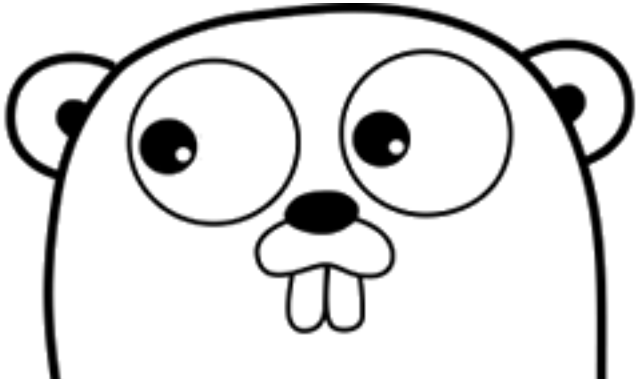


# Building (Awesome) Forensics Tools in Go

Joe Sylve, M.S. @jtsylve

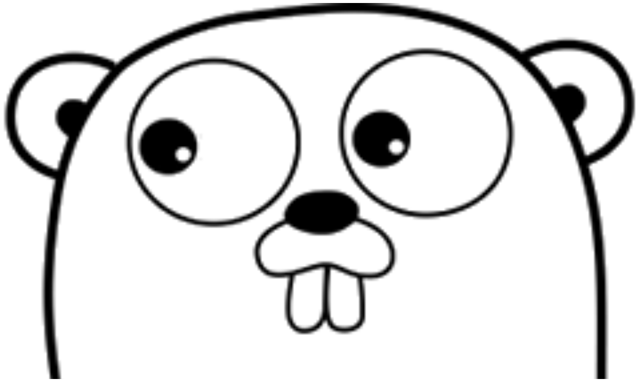
Vico Marziale, Ph.D. @vicomarziale

DFRWS 2016

BlackBag™ TECHNOLOGIES

**Grab an Updated Copy of These Slides**

https://github.com/jtsylve/slides



# Intro

# $whoarewe

- Senior Research Developers at BlackBag
- Owners 504ENSICS Labs
- UNO graduates (PhD and ABD)
- BSides NOLA organizers
- Margarita enthusiasts
- Developers for open source stuff
  - Scalpel (c)
  - LiME (c)
  - Registry Decoder (python)
  - Spotlight Inspector (python)
  - DAMM (python)
  - Golang contributor (go)

# Why Are We Here

- Not to start a flame war …
- Programming languages all have advantages and disadvantages
    - C: memory management, unsafe typing, close to the metal, compile time, but get speed, control
    - Python: higher level constructs, interpreted, just plain easier to read and write (for most tasks), slower because of the above (and don’t even get me started on the GIL)

# Specific Challenges in Forensics

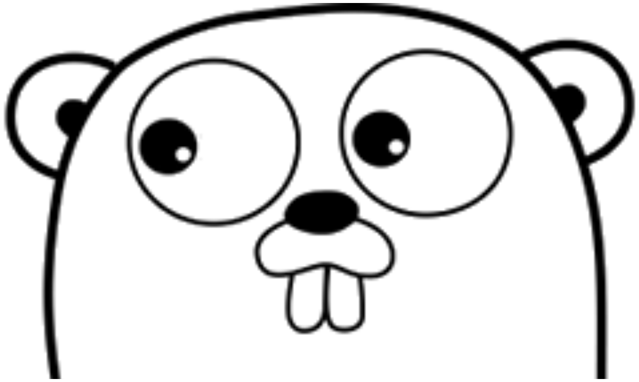
- Need the ability to do rapid prototyping (for that rad-new, or crazy-old format that is super important at the moment)
- Need to easily interface with on-disk c-based structures that we didn’t put there
- Need to be fast – eat TiB of data potentially
  - Related: need to use all those cores
- Would like something friendly to part-time devs (forensics research types)
- Cross platform
- Stable (parsing broken files, making sense of binary blobs)

# The Basics

- Fully free, open source
- Created by Google in 2007
- Compiled (for speed) but so fast as to seem interpreted
  - Even large projects can build in seconds
- Statically, safely typed
  - No: x = 5 + "N"
  - No: adding uint32 and int64
  - No: adding int + int64 - even if they are the same size
  - But with type inference
- Garbage collected:
  - No manual memory management

# More Basics

- Fast: not quite c-speed (yet), but getting there
- Expressive: First class functions, closures, function literals, maps, slices, multiple assignment/returns, etc.
- Built-in concurrency primitives (goroutines, channels)
- Syntax is brief, familiar to c users
- Simple: only ~25 keywords
- Built-in C-like structs
- (Relatively) simple Interfaces to and from C-code
- Awesome, well documented standard library



# Lab 0

Setup time!

Like "party time" but with more software installs.

Follow instructions @ https://goo.gl/na68Wz

# More Why Go: Toolchain

- *build*
- *run*: build and run in one step
- *fmt*: no arguing about where the braces go
- *vet*: find unsafe things
- *lint*: find non-idiomatic things
- *doc*
- *test*
- *get*: retrieve, compile, install external packages

# fmt

- Braces
- Spacing
- Line breaks
- Makes (git) diff far more useful
- See: http://gofmt.com
- Also enables go fix to update API calls

# vet

- Unreachable code
- Shifts equal to or longer than the variable's length
- Variables that may have been unintentionally shadowed
- Locks that are erroneously passed by value
- Check for useless assignments

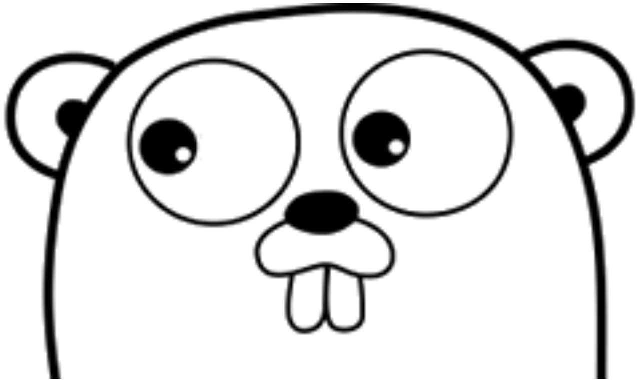
# lint

"Writing well formatted and documented software should be the lowest common denominator. Moving beyond that, software should be written to follow best practices, use common conventions, and shouldn't try to be tricky unless it's one of those rare times to be tricky."

- forces comment on Public constructs
- enforces other go-specific idioms

# doc

- Automagically extracts documentation for source files
- Yours if you do things the go way (easy)
- The standard library
- try: "go doc fmt"
- godoc -http=:6060
  - – yes, your go installation comes with a webserver



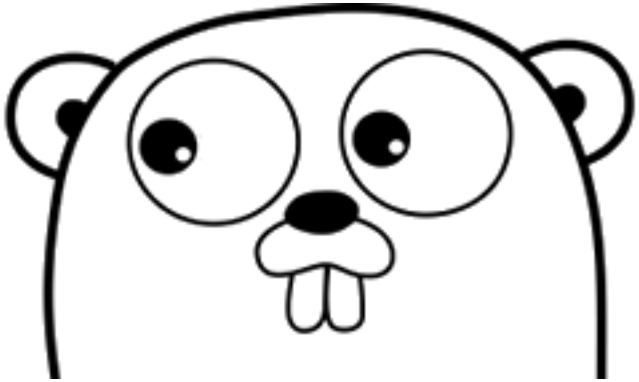
# Tour of go

# Obligatory

```go
// goodbye.go
package main

import (
   "fmt"
)

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

```
#go run goodbye.go
Goodbye, cruel world.
```

https://play.golang.org/p/ZGM1bVjC-M



# Built-in Types

# Basic Types

- int, int8, int16, int32, int64
- uint, uint8, uint16, uint32, uint64, uintptr
- byte: alias for uint8
- float32, float64
- complex64, complex128

- bool: **true**, **false**

# Variables

```
var x int // declare
x = 64 // then assign

var x int = 64 // both at once
```

Or rely on type inference

```
var x = 42
x := 42 // short version, can be used inside functions
```

Multiple assignment

```
j, k := 2, 8 // both are ints

var done bool = false
var b byte // what happens here?
```

# Strings

All utf-8 internally. Means can has all of Unicode for identifiers. *Please don’t*.

```
s := "monkeys!"

t := "Dance, " + s
```

```
import "strings"
```

For ton’s of built-in strings functions

# Zero Values

Each type has a value it’s guaranteed to be when declared. Don’t do this:

```
var i int = 0
```

because it’s the same as:

```
var i int
```

(says the linter to me over and over again)

```
var b byte // all numeric types are 0
var lame bool // lame is false
var words string // words is ""
```

# Type Conversion

No:

```go
i := 21 // an int
var j int64 = 11 // an int64
x := i +j
```

"invalid operation: i + j (mismatched types int and int64)"

Yes:

```go
x := int64(i) + j
```

Also no:

```go
type myInt int
var m myInt = 33 // an int under the covers
k := 89 // an int
y := m + k // error
```

Yes:

```go
y := int(m) + k
```

# Complex Types

- Array: [128]byte, [16]int, etc.
  - Fixed size
  - Each is a different type, even [4]int and [8]int
- Slice: []byte, []int
  - Dynamically sized "lists"
- Map: map[string]int
  - Key, value store
- Channel
  - Synchronized queue
- We’ll talk about each of these more coming up

# Array

Fixed size … hard to describe without using the word "array," so there.

```
var ints [8]int // array of 8 int, all zero
var bytes [32]byte // array of 32 byte, all zero
ints[3] = 6
anInt := ints[3] // anInt is type int value 6
```

Not used often at all in go (because we have slices). For us though they serve a useful purpose when reading C-structs from a file/blob of bytes, etc. We’ll get into this later.

# Slice

Dynamically sized lists of a type

```
var buf []byte // zero value is zero length slice
```

Can allocate initial size with built-in function *make*

```
buf := make([]byte, 10) // 10 byte slice, all zeros
buf := make([]byte, 20, 10) // 10 byte slice, 20 byte capacity, all zeros
```

Add elements with built-in function *append*

```
buf = append(buf, 42)
```

Access elements

```
second := buf[0] // second is type byte value 42
```

Take slice of slice

```
firstFew := buf[:3]
lastFew := buf[len(buf)-3:]
middleFew := buf[5:7]
```

# Slice (more)

Can take slices of arrays of the same type

```
var numbers [16]uint32
// throw some uint32s into the array
sliceNumbers := numbers[:]
```

Get the length of a slice

```
sliceLen := len(sliceNumbers)
```

Passing around slices is super cheap - they're basically just pointer to underlying arrays.

Many go libraries will take slices as parameters. Basically none take arrays. Why?

# Map

Key / value store. Must use *make* to initialize.

…

```
m := make(map[int]string)
m[3] = "three"
m[6] = "not three"
fmt.Println(m)
```

Prints: "map[3:three 6:not three]"

```
for k, v := range m {
        // do stuff with keys and values
}
```

# Struct and Type

```
type House struct {
        bedrooms   int
        baths      int
        sqFootage  int
        address    string
}
myHouse := House{}
myHouse := House{2, 2, 1000, "1123 Royal St."}
myHouse := House{sqFootage: 1000}
```

Can also declare custom types:

```
type guid int64
```

# Struct Composition

```
type Student struct {
    name string
    age  int
    classes []string
}

type Professor struct {
    name string
    age  int
    cv string
}
```

Code duplication sucks

# Embedded Structs

```
type Human struct {
    name string
    age  int
}

type Student struct {
    Human
    classes []string
}

type Professor struct {
    Human
    cv string
}

func main() {
    p := Professor{ Human{" Bob", 42}, "my CV stuff"}
    s := Student{ Human{" Sue", 19}, []string{" class1", " class2"} }
    fmt.Printf("%+ v %+v\n", p, s)
}
```

# Pointers and Deferencing

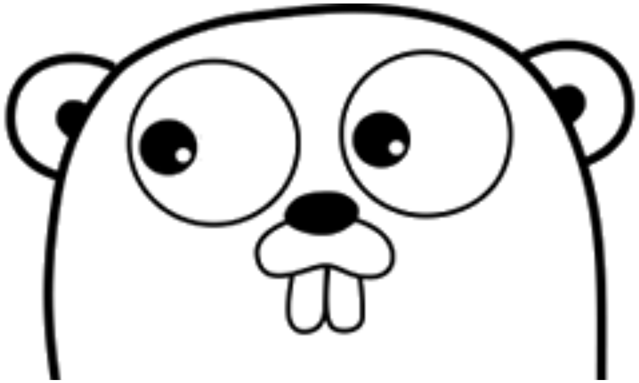
- For any type can declare pointer of type
  - var myHouse *House // pointer to House struct
    - Note: this only makes a pointer, not a struct
  - myHouse := &House{}
    - Note: this actually allocates the struct too
  - like in C, but *no pointer arithmetic*
- And can deference the pointer
  - anotherHouse := *myHouse // copy of myHouse

# Errors

Idiomatic go demands using multiple returns to return an error value in addition to whatever the function should normally return. A ‘nil’ value for the error indicates no error occurred.

```
…
fd, err := os.Open(" file.txt")
if err != nil {
      // do something maybe
      return err
}
// do something with fd
…
```

You will get tired of typing "if err != nil;" no PL is perfect.



# Control Flow

# Conditional

```
if somethingIsTrue {
        // do some stuff
}

if somethingIsTrue {
        // do some stuff
} else {
        // do something else
}
```

# Switch

```
switch {
case condition1:
        // do some condition1 thing
        // this does not fallthrough
case variable > 0:
        // do some condition2 thing
        // this does fallthrough
        fallthrough
case condition3:
        // do some condition3 thing
case condition4, condition5:
         // do some condition4 or 5 thing
default:
        // do if no case hits
```

# Loop

```
max := 10
for i:=0; i<max, i++ {
    // do some stuff max times
}

condition := true
for condition {
    // do stuff till condition is false
}

for {
    // do something till break or return
    // or crash, lol
}
```

Also have **continue**

# Range

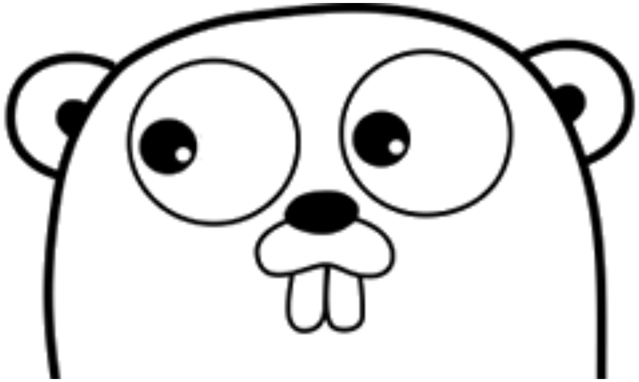
Loop alternative, like in python

```
for idx, elem := range mySlice {
        // do stuff to each elem of mySlice
        // idx is a loop counter
}
```

Go enforces that declared variables are used. If you don’t need the loop counter (idx), then use the *blank identifier* "_"

```
for _, elem := range mySlice {
        // do stuff to each elem of mySlice
}
```

Can also use for maps and arrays. For maps, you get k,v pairs.
"_" can be used in other assignment contexts as well.



# Functions

# Functions

```
func funcName (argName argType) returnType { }
```

Can has multiple arguments:

```
func funcName(x int, y int, z int) { }
```

Can has multiple returns:

```
func funcName() (int, int, int) {
    // need to declare x, y, z to return
    return x, y, z
}
```

Can has named returns:

```
func funcName() (x int, y int, z int) { // kills ambiguity
    // no need to declare x, y, z
    return x, y, z
}
```

# Method Receivers

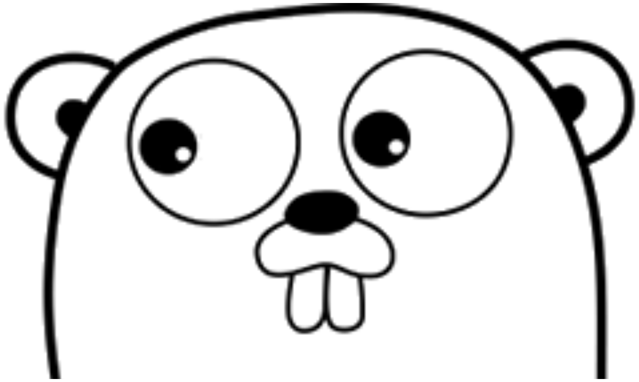
Any package local type (we’ll explain this a bit later) can have a method defined on it. E.g., define the function "square" on int, that takes no arguments and returns an int.

```
type myInt int // a type I have defined locally
func (i myInt) square() myInt {
    return i*i
}

func main() {
    var i myInt = 23
    fmt.Println( i.square() )
}
```

Prints: "529""

https://play.golang.org/p/AQknQIXY9a

# Crazy Standard Library Dance Party Time!?!

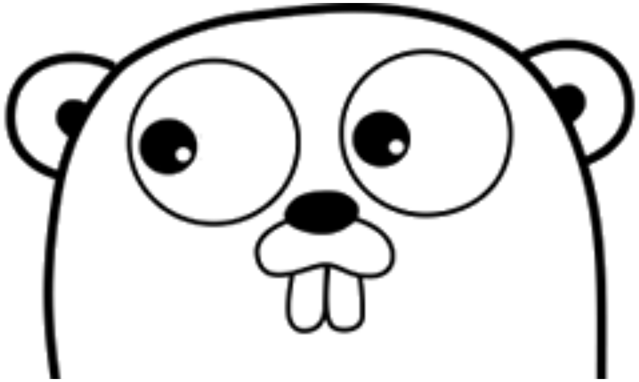


# Package *time*

https://golang.org/pkg/time/

# Labs!

- Labs can be found on Github
  - https://github.com/jtsylve/DFRWS-Go-Workshop
  - Skeleton files and tests are provided to help you
  - Solutions are in a separate "solutions" branch
- Checkout Repository
  - go get github.com/jtsylve/DFRWS-Go-Workshop
- Edit your solutions in your $GOPATH
  - $GOPATH/src/github.com/jtsylve/DFRWS-Go-Workshop



# Lab 1

Exactly how many kinds of timestamps do we have to deal with daily?

When reading from some forensics artifact, they're never the format we want. In that they're often just a blob of bytes.

In this first lab, you'll convert a blob of bytes (an int) into the go representation of time.

# Lab Advice

Remember:
fmt.Println(someGoThing)
is your friend.

Google (or duckduckgo) is also your friend.

So is the standard library documentation and the examples .

Go cheat sheet
https://github.com/a8m/go-lang-cheat-sheet

A list of 50 go "gotchas"
http://devs.cloudimmunity.com/gotchas-and-common-mistakes-in-go-golang/index.htm

# What are Objects?

- As far as OO programming
  - A collection of data:
  - And functions that operate on that data
- We have structs for collections of data
- But what about operations on structs?

# Method Receivers

Generally used for functions on structs

```
type attendee struct { // defined a local type like myInt before
	name string
	funds int
}

func (a *attendee) payRegistration() { // defined a method on the type
	a.funds -= 10000                    // like square before
}

func main() {
	me := attendee{name: "vico", funds: 10000 }
	me.payRegistration()
	fmt.Printf("% v has %v funds\n", me.name, me.funds)
}
```

Prints: "vico has 0 funds"

https://play.golang.org/p/5DSFFiedu5

# Interface

An interface is named collection of methods signatures. It specifies the *behavior* of an object.

```
type Reader interface {
        Read(p []byte (n int, err error)
}
```

Anything that implements the function with the above signature is a *Reader*.

We can write a function that takes a Reader as a parameter, and we know for sure that whatever is passed in can be used as:

```
func whatevs(thing Reader) {
    p := make([]byte, 1024)
    n, err := thing.Read(p)
    // do stuff with n, err, p
}
```

```
type Animal interface {
    Speak() string
}

type Dog struct { /* dog stuff */ }
func (d Dog) Speak() string {
    return "ruff"
} // implements Animal

type Cow struct { /* cow stuff */}
func (c Cow) Speak() string {
    return "moo"
} // implements Animal

func makeNoise(a Animal) string { // takes type Animal
    return a.Speak()
}

func main() {
    d := Dog{}
    fmt.Println(makeNoise(d))
    c := Cow{}
    fmt.Println(makeNoise(c))
}
```

https://play.golang.org/p/JBpRkN39oC

# sort Library

The sort package implements a sort function for a slice of ints.

```
import (
	"sort"
	"fmt"
)

func main() {
	things := []int{7, 17, 11, 53}
	sort.Ints(things) // sorts in-place
	fmt.Println(things)
}
```

Prints: "[ 7, 11, 17, 53]"

https://play.golang.org/p/QHEqx_9Ww1

# More Interfaces

What if you want to sort int64? No implementation provided. There is interface *Sortable* in the package *sort*, and function *Sort* takes a *Sortable* and sorts the slice.

```
func Sort(s Sortable) { … }
type Sortable interface {
    Len() int // number of elements in the collection.
    Less(i, j int) bool  // does element at index i
                         // sort before with index j.
    Swap(i, j int) // swaps elements with indexes i and j.
}
```

# More Implementing Interfaces

```
type sort64 []int64 // new collection type, slice of int64

// Implement the functions required by the interface
func (s sort64) Len() int { return len(s) }
func (s sort64) Swap(i, j int) { s[i], s[j] = s[j], s[i] }
func (s sort64) Less(i, j int) bool { return s[i] < s[j] }
// Now type sort64 is a Sortable

func main() {
        things := sort64{57, 41, 89}
        sort.Sort(things)
        fmt.Println(things)
}
```

Nowhere have we implemented a *Sort* function.

Prints: "[ 41, 57, 89]"

https://play.golang.org/p/-bZEA1OubM

# Defer

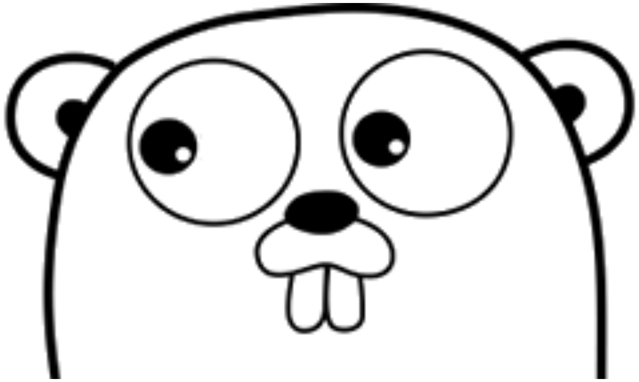
The defer keyword place before a function call ensures that that function is called just before the enclosing function exits regardless of how/where the function exits.

The example below makes way more sense.

```
import "os"

fd, err := os.Open(" somefile.txt")
If err != nil {
      // do some error stuff
}
defer fd.Close()
… do stuff with file …
```

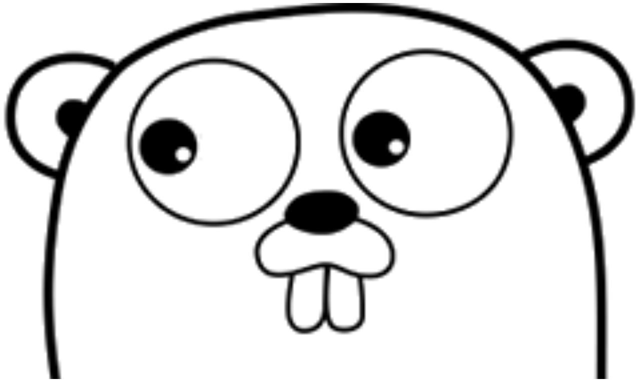
Useful for any type of cleanup code that must have a chance to execute.

# Crazy Standard Library Dance Party Time!?!



# Package *strings*

https://golang.org/pkg/strings/



# Package *encodings/binary*

[https://golang.org/pkg/encoding/binary/](https://golang.org/pkg/encoding/binary/)

# Package *encodings/binary*

Useful for reading structured data from input (a file or otherwise)

```
import (
	"encodings/binary"
	"fmt"
)

type wineCellar struct {
	Temperature int32
	Humidity [6]byte
}
// open some file, fd
wc := wineCellar{}
binary.Read(fd, binary.LittleEndian, &wc)
fmt.Printf("%+ v\n", wc)
```

# Package *encodings/binary* (cont.)

Note: binary.Read() only likes to read into fixed size constructs.

```
type someStruct struct {
        Size int
        Name string
        Count int
}
```

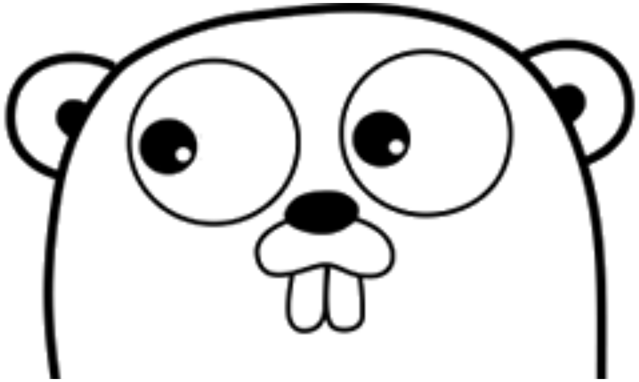
Is not going to work because "string" is variable size. If you happen to know the size of the string field, use:

```
Name [stringFieldSize]byte
```

And then convert later

```
strName := string(x.Size[:])  // yes the syntax is weird, but it makes sense
```

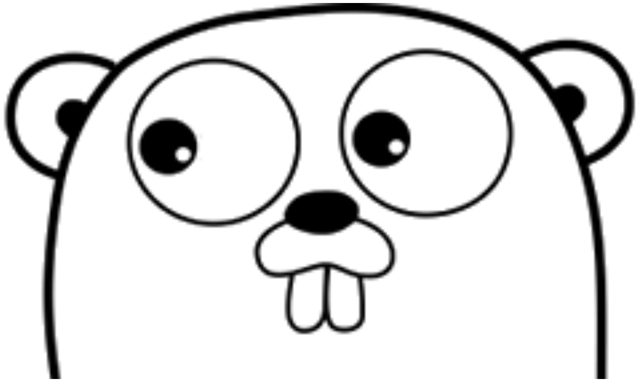
Note: binary.Read() can only read into Public fields; struct fields names must be upper case.



# Lab 2

Often times file-based forensic artifacts are a collection of binary C-based structures.

Snarfing them out of files, and formatting them into something readable is our bread and butter. For this lab, we'll play with structs.



# Packages

# Packages

- Working with go requires specific directory structure and setting environment variables
- $GOPATH
  - Where your work resides
  - E.g., /Users/you/go
- $GOROOT
  - Where go is installed
  - Defaults to /usr/local/go or c:\go
- Make sure these are in you $PATH
  - $GOPATH/bin
  - $GOROOT/bin

# More Packages

- Directory structure based
- $GOPATH/src/someproject
  - Where someproject code resides
  - Can contain a **package** main // compiles to executable
  - Or **package** someproject // compiled to a library
  - And any number of subpackages
- $GOPATH/src/someproject/someproject.go
  - package *main*
- $GOPATH/src/someproject/mylib/mylib.go
  - Package *mylib* (imported into package *main*)
- If someproject.go has: **import** "someproject/mylib"
- go run someproject.go compiles all and executes func *main*

# someproject.go

```
// $GOPATH/src/someproject/someproject.go
package main

import (
    "someproject/mylib"
)

func main() {
    //do stuff with public things from mylib.go
    x, err := mylib.DoStuff()
    // do stuff with x and err
}
```

# mylib.go

```
// $GOPATH/src/someproject/mylib/mylib.go
package mylib

func DoStuff() (int, error) {
    var err error
    var x int
    // do stuff with err and x
    return x, err
}
```

# Cross Package Visibility

Case is important in go. Generally, capitalized means accessible outside the package. Lower case constructs are not.

```
package mine

func privateFunc() { }
func PublicFunc() { }
type private struct {
        privInt int
}
type Public struct {
        PubInt int
}
type Public struct {
        privInt int
}
```

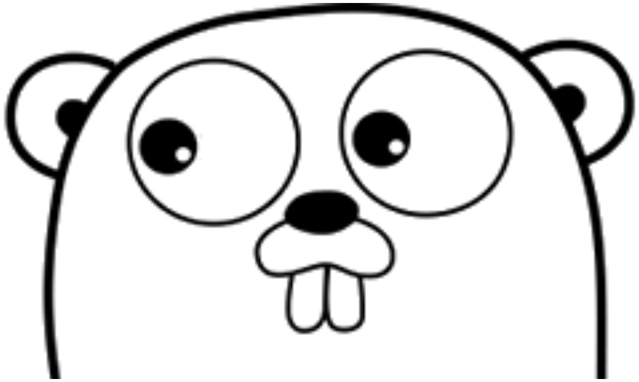
# Argument Parsing

The simple version

```
import (
    "fmt"
    "os"
)

func main() {
    for _, a := range os.Args[1:] {
        fmt.Println(a)
}
```

Works like the *echo* command

# Crazy Standard Library Dance Party Time!?!



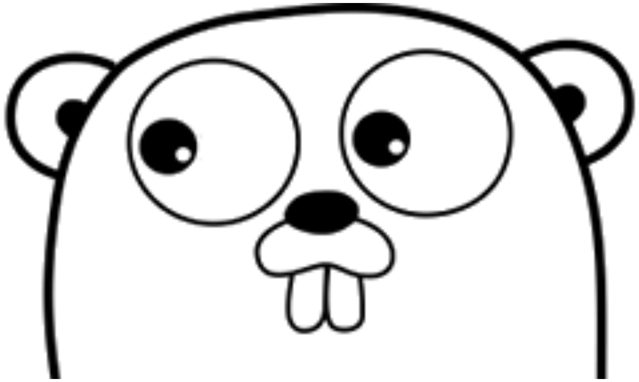
# Package *flag*

https://golang.org/pkg/flag/

# Package *flag*

```
import (
	"flag"
	"fmt"
)

func main() {
	dbg := flag.Bool("dbg", false, "print debugging")
	size := flag.Int("s", 128, "size of something")
	db := flag.String("db", "", "db to use")
	flag.Parse()
	fmt.Printf("%v %v %v", *dbg, *size, *db)
}
```

```
#go run flagtest.go --dbg –s 11 --db sql.db
```



# Packages *os, io, io/ioutil*

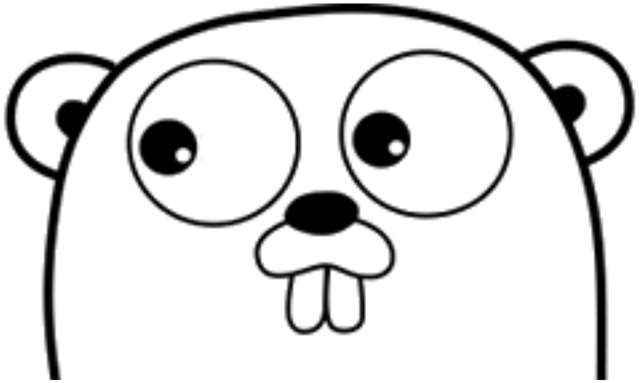
https://golang.org/pkg/os/

https://golang.org/pkg/io/

https://golang.org/pkg/io/ioutil/

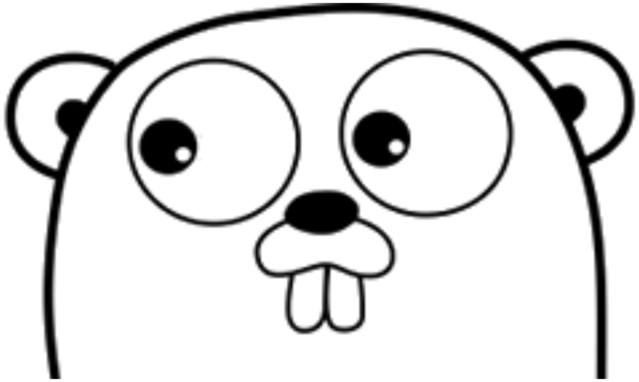
# FileI/O

```
import "os"
func Open(name string) (*File, error)
func (f *File) Close() error

import "io"
func Copy(dst Writer, src Reader) (written int64, err error)
func ReadFull(r Reader, buf []byte) (n int, err error)

import "io/ioutil"
func ReadAll(r io.Reader) ([]byte, error)
```



# Lab 3

So far we've been working inside of packages, but not made a usable program. In this lab that all changes. You'll finally define a *main* package, read an input file, and use the libraries you've created to make a useful whole.



# Concurrency

# Concurrency

Built-in concurrency in go leverages *goroutines* and *channels*, **no messy manual locking** (unless you're into that sort of thing; I'm not judging), and no explicit thread management. Communication between goroutines is accomplished via channels.

To make a function call run in another goroutine, use the keyword *go* in front of the function call.

```
go myFunc()
```

Goroutines are super lightweight, Can spawn millions if you want.

Channels are synchronized queues. Multiple goroutines can dump stuff in and read from without manual locking.

```
ch := make(chan int, 10)
ch <- someInt // put someInt into channel
myInt := <- ch // read someInt from channel into myInt
```

# Concurrency

```
func double(i int, c chan int) {
        c <- i*2
}

func main () {
        c := make(chan int)
        go double(10, c)
        go double(11, c)
        fmt.Println(<-c)
        fmt.Println(<- c)
}
```

Channels have a capacity, 0 by default. With only one goroutine this is ok, but if we wanted many goroutines running *double*, say 10, better to:

```
c := make(chan int, 10)
```

https://play.golang.org/p/TIo364Si2I

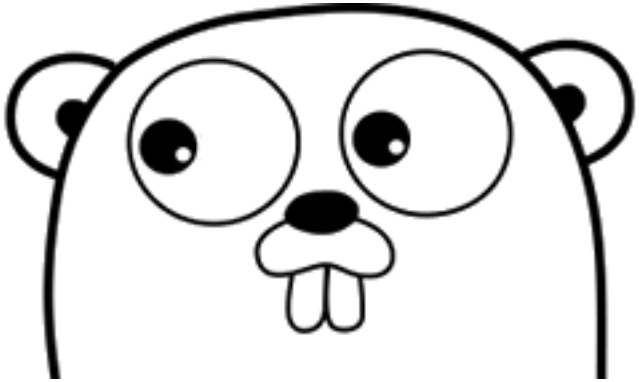
# Bad Concurrency

```
func double(i int) {
    fmt.Printf("%v\n", i*2)
}

func main() {
    // spawns 100 goroutines, each executing func double
    for j:=0; j<100; j++ {
        go double(j)
    }
    // program exits (without waiting for goroutines to finish!)
}
```

Prints: <nothing>

https://play.golang.org/p/-l-EszWnW4

# Crazy Standard Library Dance Party Time!?!



# Package *sync*

https://golang.org/pkg/sync/

# WaitGroups

```
func double(i int) { fmt.Println(i*2) }

func main() {
    var wg sync.WaitGroup
    for j := 0; j < 100; j++ {
        wg.Add(1)
        go func(j int) {
            defer wg.Done()
            double(j)
        }(j)
    }
    wg.Wait()
}
```

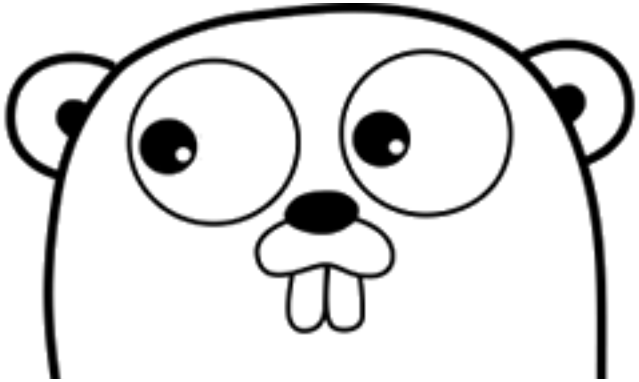
Prints: 0 – 198 in *some* order

https://play.golang.org/p/Ls2w2Dg155

# Producer/ Consumer with Channels

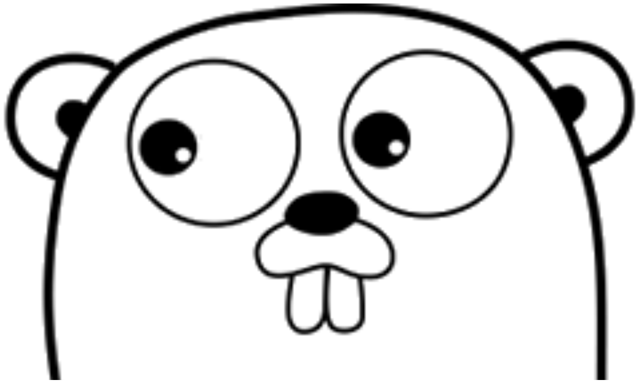
```
var msgs = make(chan int)

func produce () {
        for i := 0; i < 10; i++ {
                msgs <- i
        }
        close(msgs)
}

func consume () {
        for msg := range msgs {
                fmt.Println(msg)
        }
}

func main () {
        go produce()
        consume()
}
```

https://play.golang.org/p/XQ7SaZ7JJj



# Package *archive/zip*

https://golang.org/pkg/archive/zip/



# Lab 4

We have a usable app now, but let's make it a little more interesting with concurrency, multiple input files, and have them zipped, just for fun.

# Happy Hunting!

# Questions / Comments / Problems

- Joe T. Sylve, M.S.
  - joe.sylve@gmail.com
  - @jtsylve

- Vico Marziale, Ph.D.
  - vicodark@gmail.com
  - @vicomarziale