GOLANG

(RAFAEL ABREU DE CRISTO) && (JULIEN DOIRON)

DECLARING VARIABLES

2 keywords

var

- Decalres a mutable variable
- Specify type or value (or both)

- Same as declaring a var
- Only specify Value

const

- Declares an immutable variable
- Must specify value

Declaring Variable Examples

var

```
var num1 = 3

var num2 int
num2 = 4

var num3 int = 3
```

:=

```
num4 := 4
```

const

```
const NUM5 = 5
const NUM6 int = 6
```

DATA TYPES

4 main data types

bool

- True or False

int

- Whole number
- Unsigned and Signed
- Default bits depend on system (64 bits for 64 bit systems)

float

- Decimal point numbers
- Default is float64 if not specified

string

- stores characters
- Only double quotes

Data Types Examples

bool

```
var x bool = false
var y bool = true
```

float

```
var num1 float32 = 12.3
var num2 float64 = 65.1234
var num3 = 34.5 // defaults to float64
```

string

```
var text string = "hello"
```

Data Types Examples

int

```
var num4 int8 = 127
var num5 int16 = 32767
var num6 int32 = 2147483647
var num7 int64 = 9223372036854775807

var num8 uint8 = 255
var num9 uint16 = 65535
var num10 uint32 = 4294967295
var num11 uint64 = 18446744073709551615

var num12 int = 9223372036854775807 // Defaults to int64
```

ARRAYS

2 ways to declare (cannot be const)

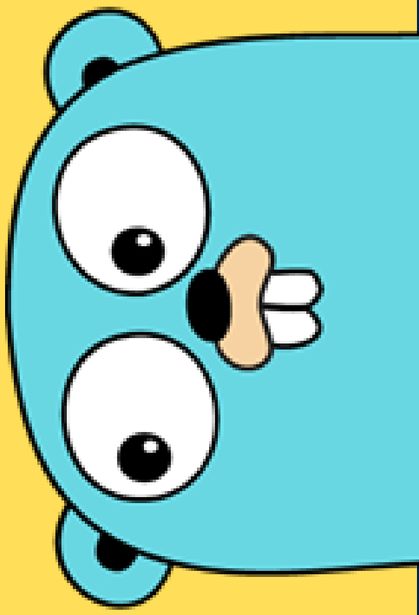
- var *array_name* = [*length*]*datatype*{*values*}
- var *array_name* = [...]*datatype*{*values*}

```
var names = [2]string{"Julien", "Rafael"}
```

```
var names = [...]string{"Julien", "Rafael"}
```

```
var names = [2]string{}
names[0] = "Julien"
names[1] = "Rafael"
```

function that works with arrays: len()



SLICES

Like arrays but more flexible

- var *slice_name* = []*datatype*{*values*}
- var *slice_name* = *array_name*[*start*: *end?*] // slice from array
- var *slice_name* = make([]*datatype*, *length*, *capacity*?)

```
var slice1 = []int{3, 4, 5, 5} // [3,4,5,5]
var slice2 = []int{}           // []
var slice3 = make([]int, 5)    // [0,0,0,0,0]
var slice4 = names[0:]         // ["Julien", "Rafael"]
```

Modifiying Slices

append()

slice_name = apend(*slice_name*, *element1*, *element2*, ...)

```
var slice5 = []int{1, 2, 3}
var slice6 = append(slice5, 4, 5, 6) // [1,2,3,4,5,6]
```

```
var slice5 = []int{1, 2, 3}
var slice6 = []int{4, 5, 6}
var slice7 = append(slice5, slice6...) // [1,2,3,4,5,6]
```

spread operator

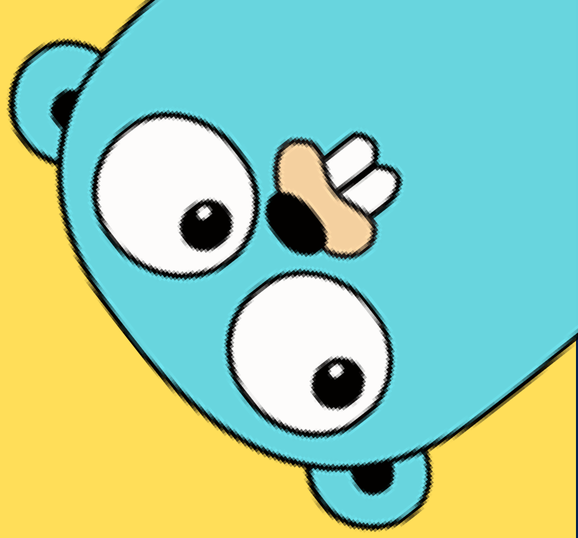
IF AND SWITCH

```go
if 10 > 9 {
    fmt.Println("10 is greater")
}
```

```go
var number int = 1

switch number {
case 1:
    fmt.Println("The number is 1")
case 2:
    fmt.Println("The number is 2")
case 3:
    fmt.Println("The number is 3")
default:
    fmt.Println("The number was none of the above")
}
```

no break; needed



LOOPS

Only for loops in go

- Simple For Loop

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

LOOPS

Continue and Break keyword

```go
for i := 0; i < 10; i++ {
    if i == 5 {
        continue // Will skip the current iteration
    }

    fmt.Println(i)
}
```

LOOPS

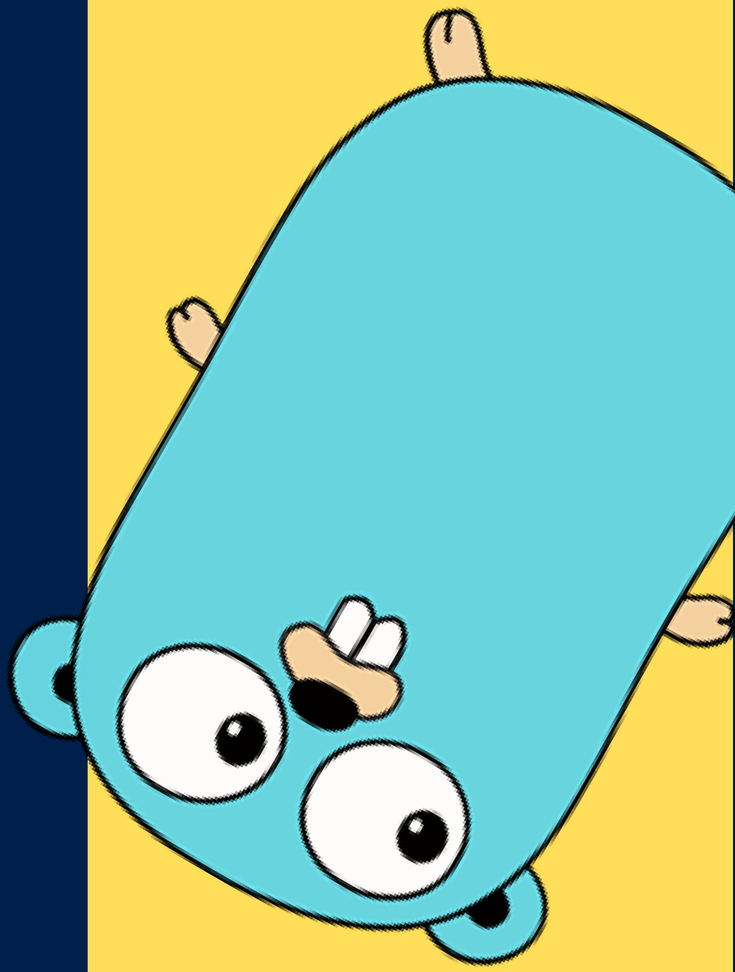
Range keyword

- for *index*, *value* := *array*|*slice*|*map*

```
var names = [3]string{"Julien", "Rafael", "Samuel"}

for idx, val := range names {
    fmt.Printf("index: %d value: %s\n", idx, val)
}
```

```
index: 0 value: Julien
index: 1 value: Rafael
index: 2 value: Samuel
```



FUNCTIONS

func keyword

- func *function_name*(*param1 type*, ...) *type*

```
func addNumbers(num1 int, num2 int) int {
    return num1 + num2
}
```

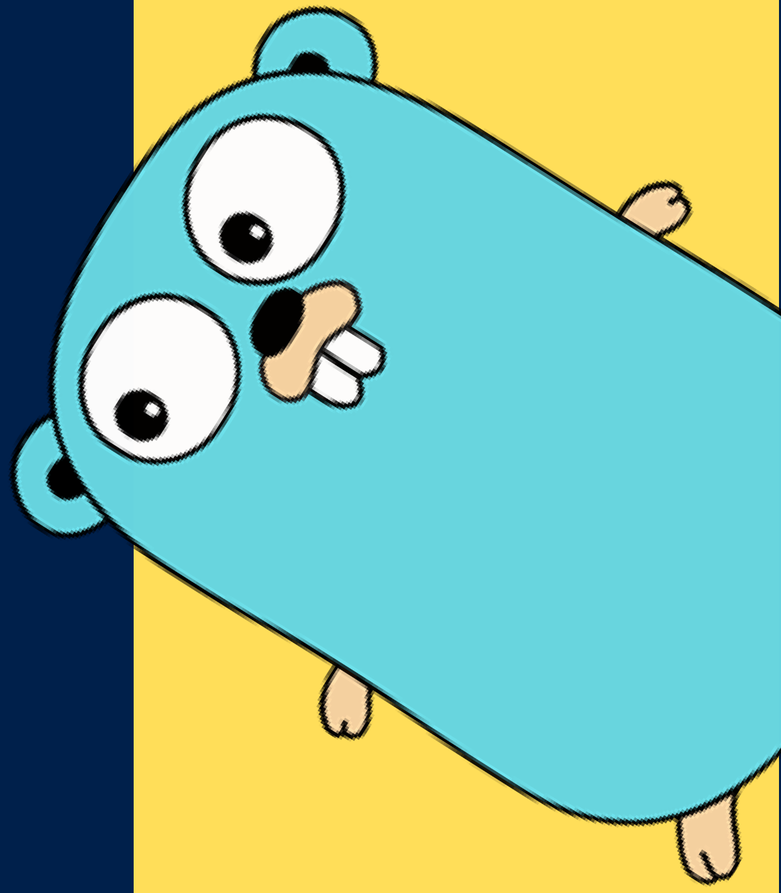
FUNCTIONS

- func *function_name*(*param1 type*, ...) (*variable_name type*)

```
func addNumbers(num1 int, num2 int) (result int) {
    result = num1 + num2
    return
}
```

- Add any number of return values

```
func addNumbers(num1 int, num2 int) (result1 int, result2 int) {
    result1 = num1 + num2
    result2 = num1 - num2
    return
}
```



STRUCT

Like a Class

- type *struct_name* struct {
  member1 datatype
  ...
  }

```
type Snowboard struct {
    length int
    style  string
    brand  string
    color  string
}
```

Struct

Accessing properties

Access properties with dot operator

```
var snowboard1 Snowboard

snowboard1.length = 154
snowboard1.style = "Regular"
snowboard1.brand = "Capix"
snowboard1.color = "Brown"
```

Printing properties

```
fmt.Println("Length: ", snowboard1.length)
fmt.Println("Riding Style: ", snowboard1.style)
fmt.Println("Brand: ", snowboard1.brand)
fmt.Println("Color: ", snowboard1.color)
```

```
Length:  154
Riding Style:  Regular
Brand:  Capix
Color:  Brown
```

STRUCT

Creating functions for structs

```go
func (s Snowboard) printDetails() {
    fmt.Println("Length: ", s.length)
    fmt.Println("Riding Style: ", s.style)
    fmt.Println("Brand: ", s.brand)
    fmt.Println("Color: ", s.color)
}
```

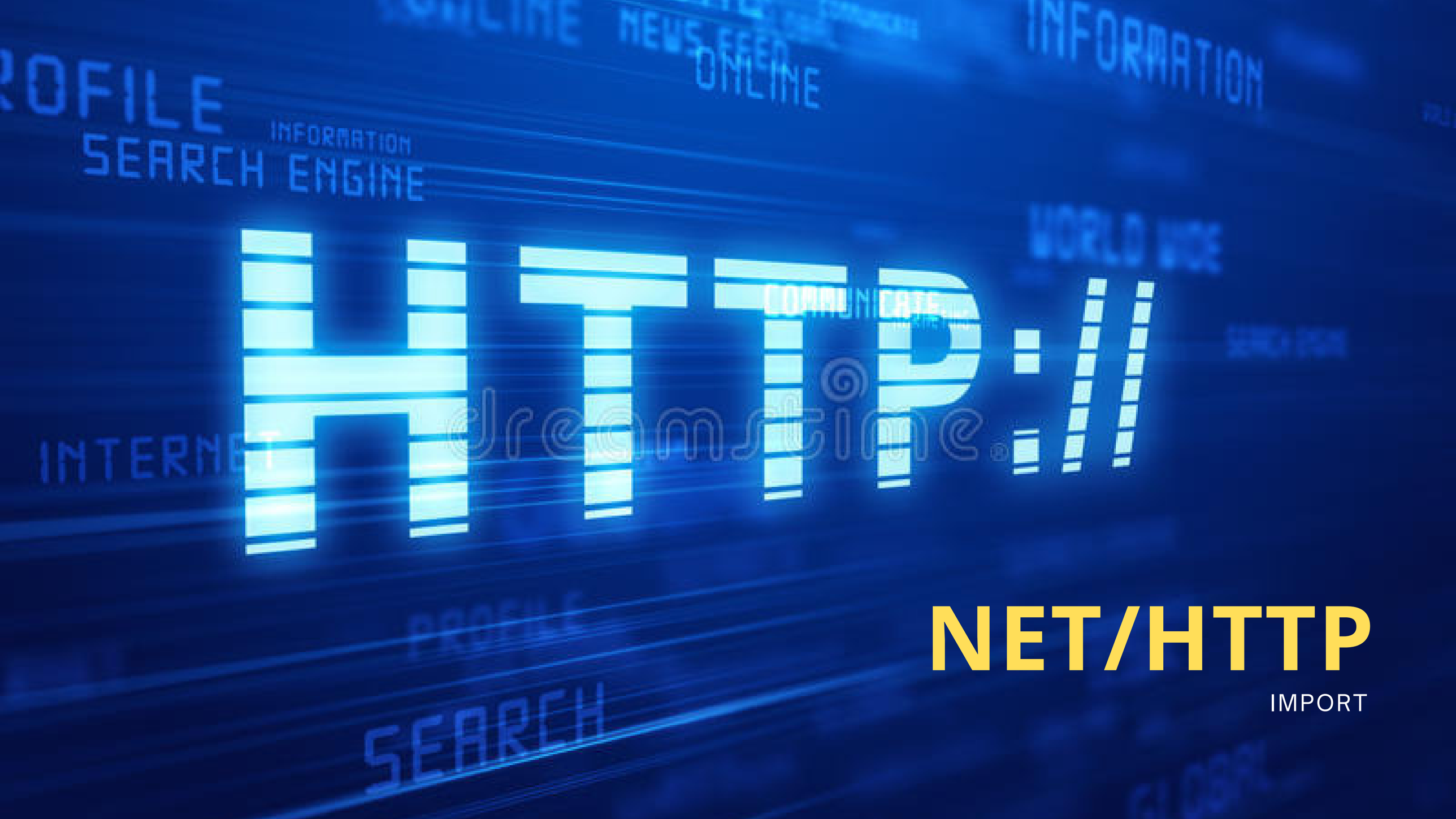
STRUCT

Creating functions that will make changes to the object

Place star here to work with persisting data

```
func (s *Snowboard) addLength(length int) int {
    s.length += length
    return s.length
}
```

This will make sure you are changing the values of the original object, and not a copy of it


HTTP://
NET/HTTP
IMPORT

LISTEN AND SERVE

http.ListenAndServe(*address* string, *handler* Handler) error

```
http.ListenAndServe("localhost:3000", nil);
```

Handler is usually nil, making it default (DefaultServeMux)

Listen And Serve with error handling

```
if err := http.ListenAndServe("localhost:3000", nil); err != nil {
    log.Fatal(err)
}
```

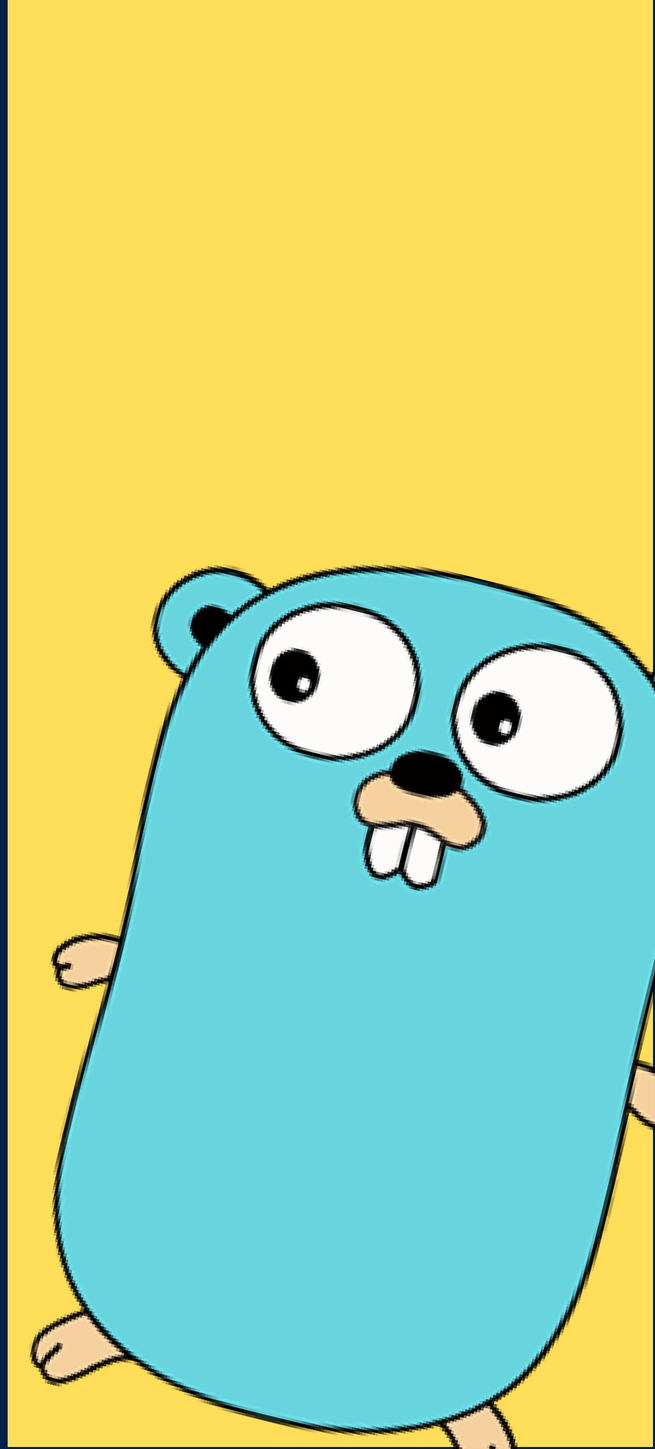
HANDLE FUNC

http.HandleFunc(*pattern* string, *handler* func(ResponseWriter, *Request))

Registers the handler passed in for the given pattern

```
http.HandleFunc("/", handleHome)
http.HandleFunc("/add", handleAdd)
```

```
func handleHome(w http.ResponseWriter, r *http.Request)
```



SERVE FILE

http.ServeFile(*w* ResponseWriter, *r* *Request, *name* string)

Basically redirects to the given file/path and passing in the request and response.

```
http.ServeFile(w, r, "index.html")
```

FORMS

Request.ParseForm()

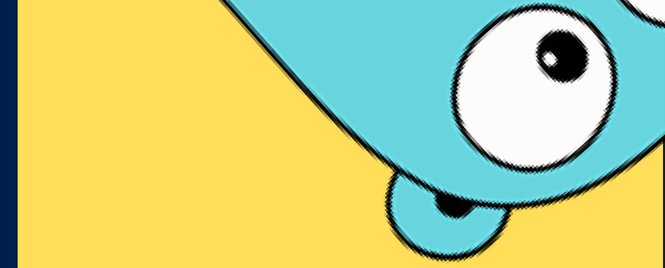
Reads the raw query from URL and populates *request*.Form and *request*.PostForm()

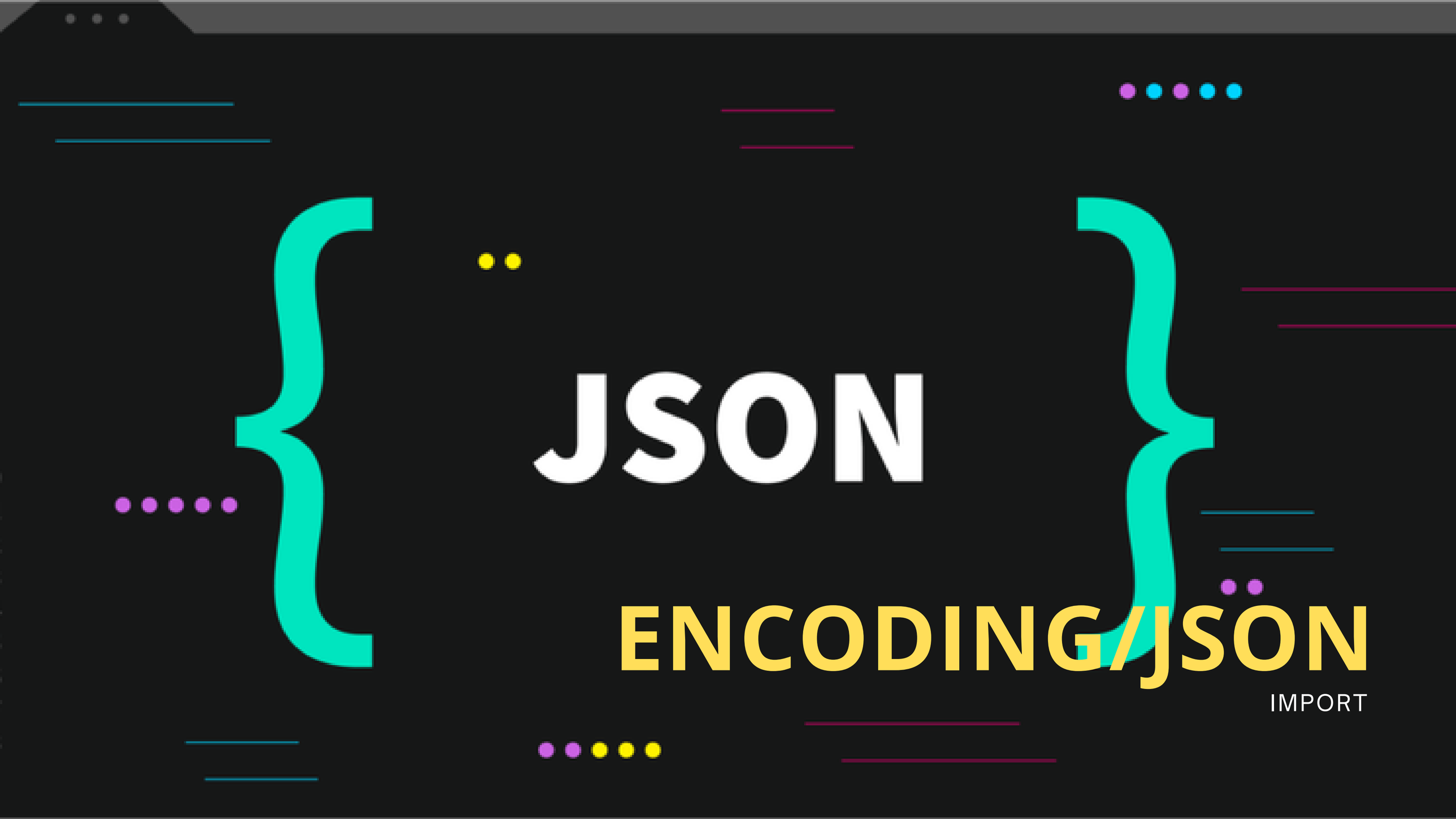
```
r.ParseForm();
```

Request.FormValue(*key* string)

Gets the data from the form input where the key is the name

```
name := r.FormValue("name")
```




JSON
ENCODING/JSON
IMPORT

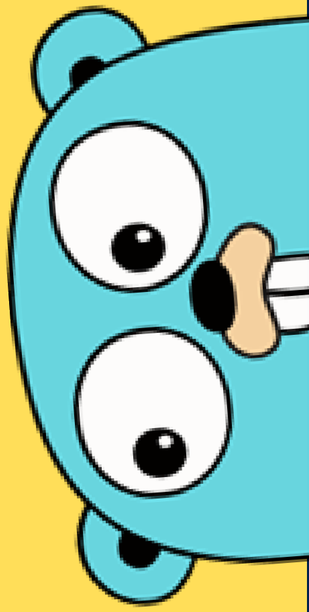
OPEN

os.Open(*fileName* string) (*os.File, error)

Opens the file for reading (read only)

```
readFile, _ := os.Open("people.json")
```

will not store the returned errors



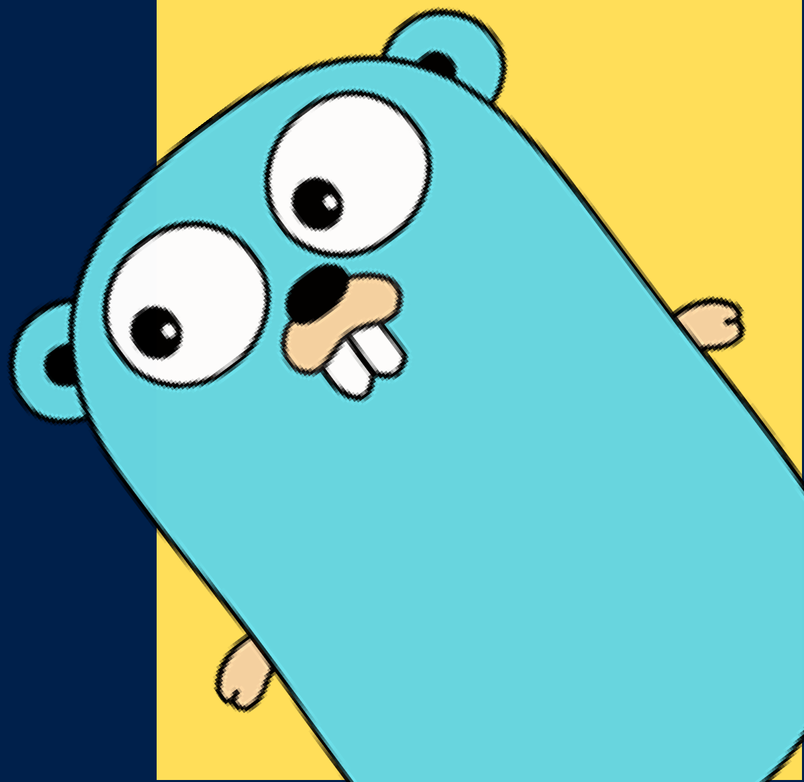
CREATE

os.Create(*fileName* string) (*os.File, error)

Creates a new file or rewrites it if it already exists

```
writeFile, _ := os.Create("people.json")
```

will not store the returned errors



READING FROM THE FILE

io.ReadAll(*r* io.Reader) ([]bytes, error)

Returns the data it read from the file as bytes

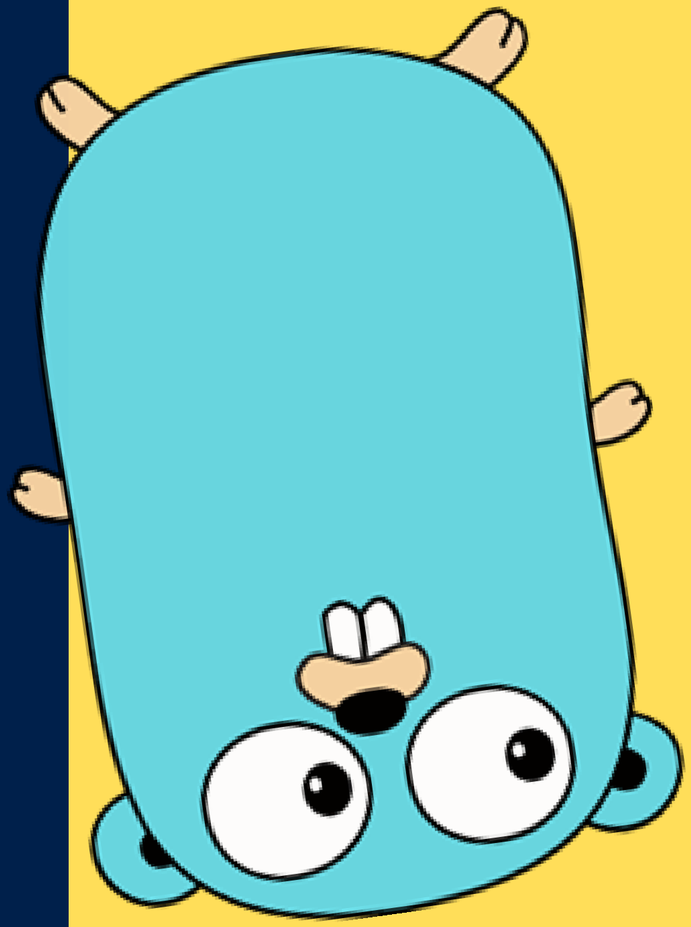
```
bytes, _ := io.ReadAll(readFile)
```

json.Unmarshall(*data* []bytes, *v* any) error

Parses the data and stores the result into the second parameter

```
var existingPeople []Person
```

```
json.Unmarshal(bytes, &existingPeople)
```



WRITING TO THE FILE

json.MarshalIndent(*v* any, *prefix* string, *indent* string) ([]byte, error)

- Basically a json.stringify()
- The orignal method is Marshal() (will not format)

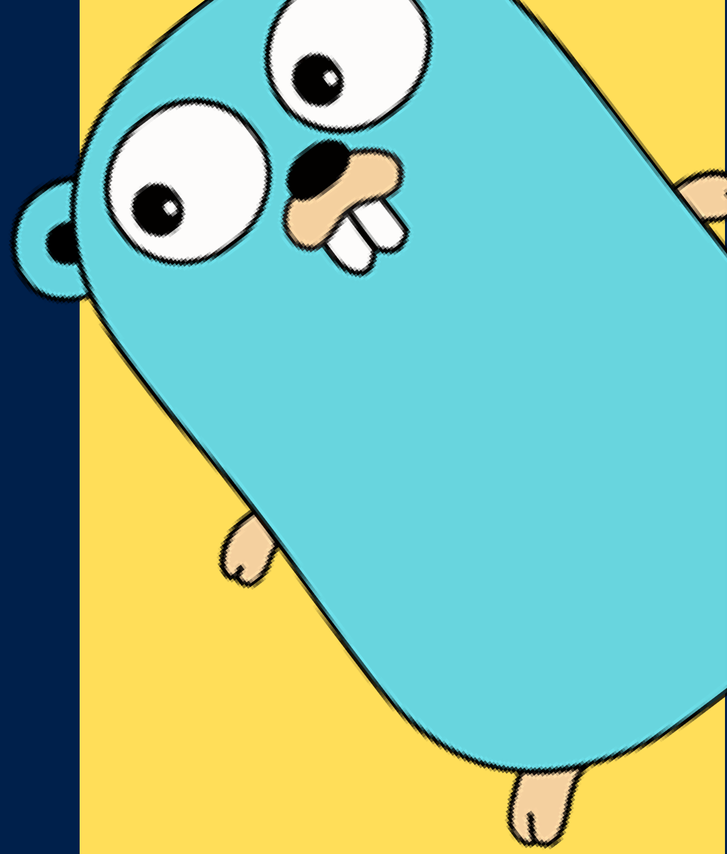
```
jsonData, _ := json.MarshalIndent(people, "", "   ")
```

this is a slice

file.Write(*b* []byte) (*n* int, *err* error)

Will write the passed in bytes to the file

```
writeFile.Write(jsonData)
```

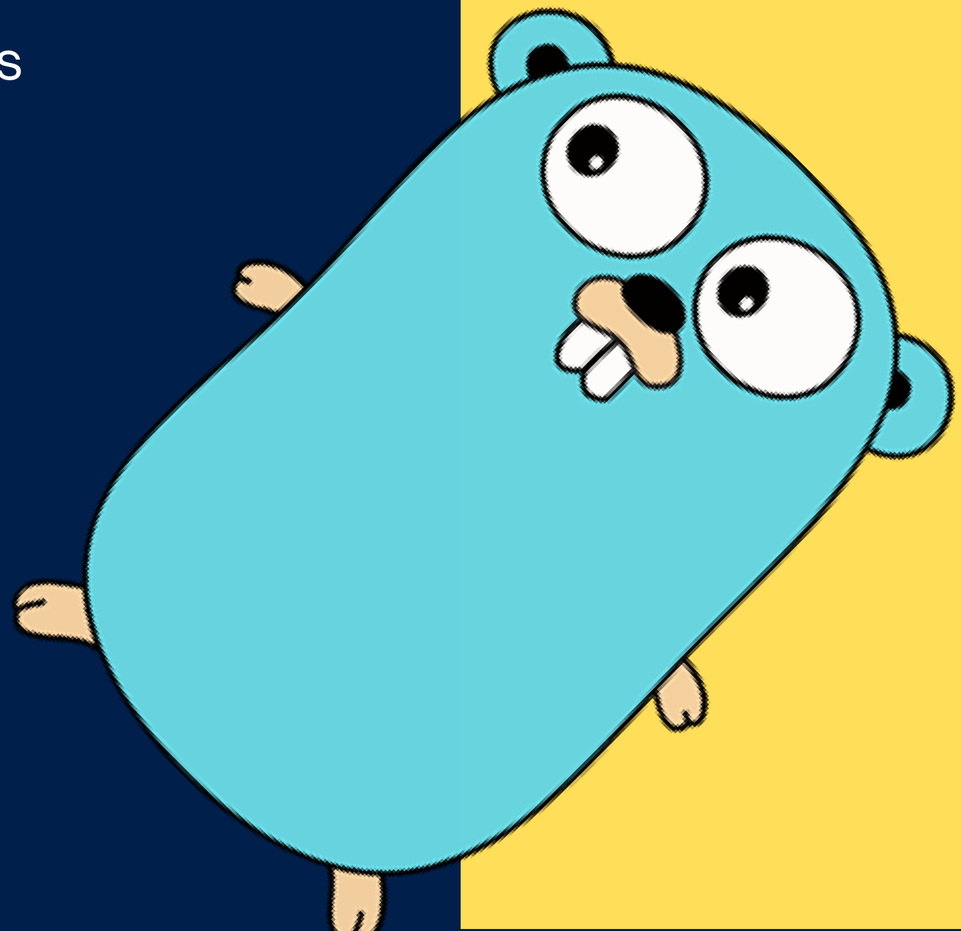


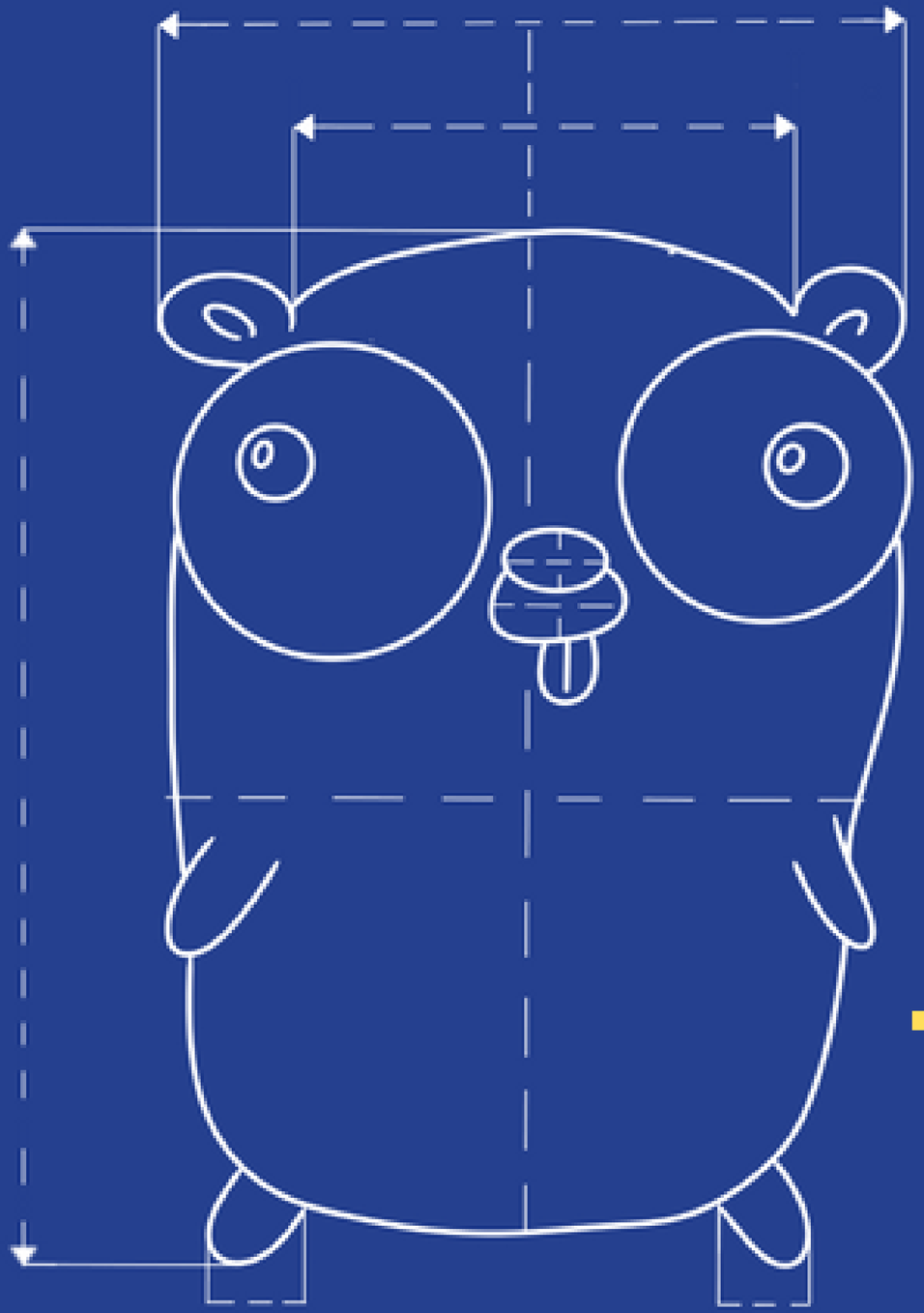
CLOSING A FILE

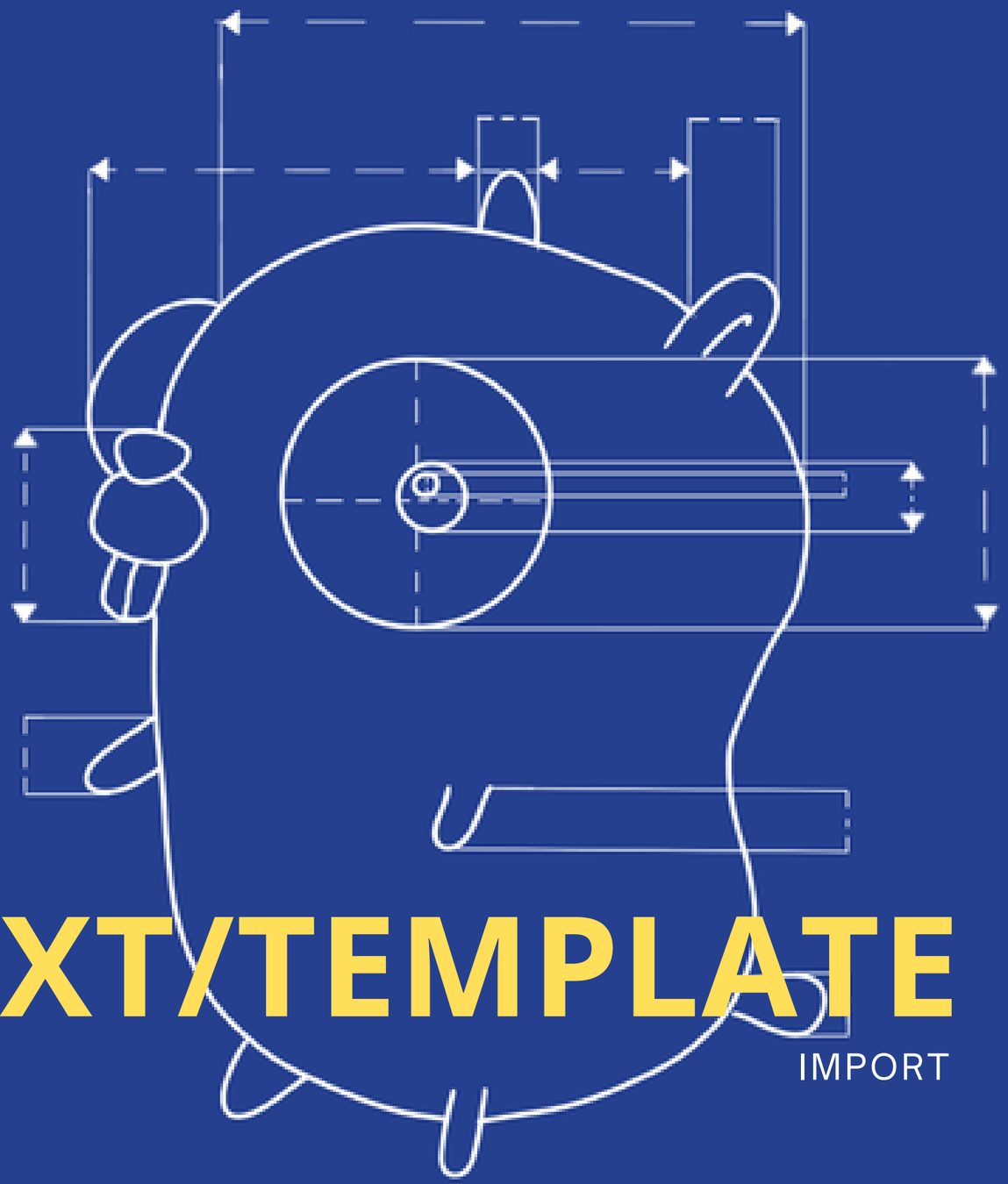
defer *file*.Close()

Closes an open file after it is done with it's operations

```
defer readFile.Close()
defer writeFile.Close()
```







TEXT/TEMPLATE

IMPORT

GLOBAL VARIABLE

var tmplt *template.Template

This is the first step to work with templates.
This way you can access the same instance of tmplt throughout your code.

PARSE FILES

template.ParseFiles(*filenames* ...string) (*template.Template, error)

Creates a new template

```
tmplt, _ = template.ParseFiles("list.html")
```

HTML from list.html

```
<h1>List of People</h1>
<ol>
    {{range .people}}
    <li>Name :{{.Name}}</li>
        Age :{{.Age}}
    {{end}}
</ol>
```

EXECUTE

template.Execute(*wr* io.Writer, *data* any) error

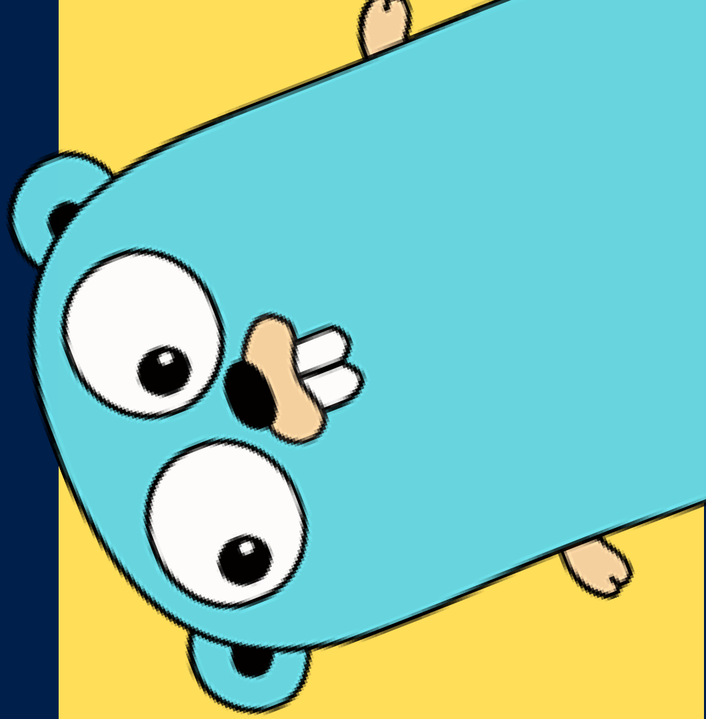
Applies the template to the data object

```
data := map[string]interface{}{
    "people": existingPeople,
}
```

slice of people

```
tmplt.Execute(w, data)
```

http.ResponseWriter



Thank you

References

- The Go Programming Language (n.d). Tutorial: Get started with Go https://go.dev/doc/tutorial/getting-started
- The Go Programming Language (n.d). Documentation https://go.dev/doc/
- The Go Programming Language (n.d). Effective Go https://go.dev/doc/effective_go
- The Go Programming Language (n.d). Standard Library https://pkg.go.dev/std
- W3 Schools (n.d). Go Syntax https://www.w3schools.com/go/go_syntax.php
- Youtube (n.d). Web application development with Buffalo, in Golang https://www.youtube.com/watch?v=1mXWtP3EkLk