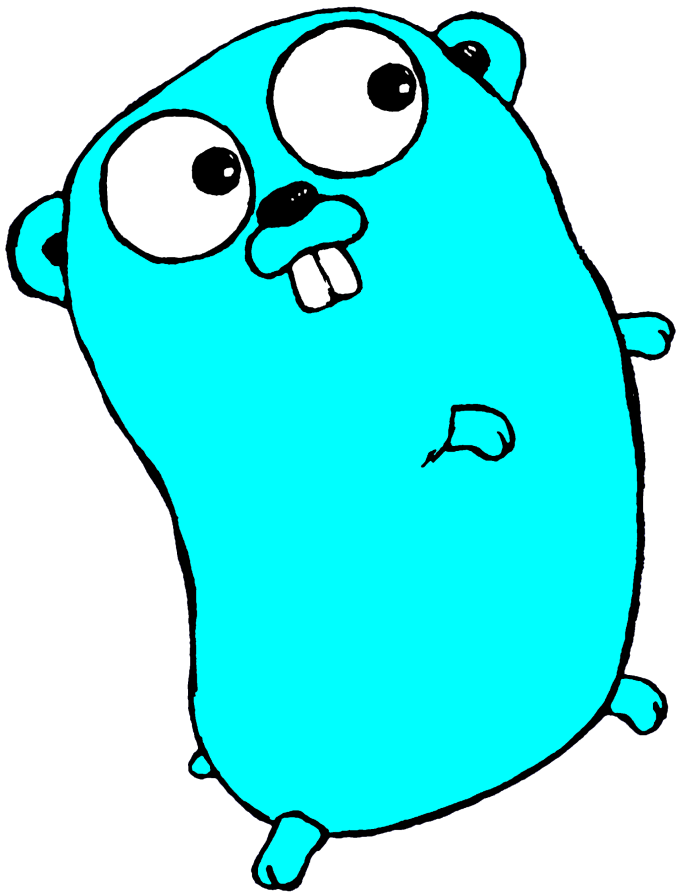


Introduction to the Go Programming Language

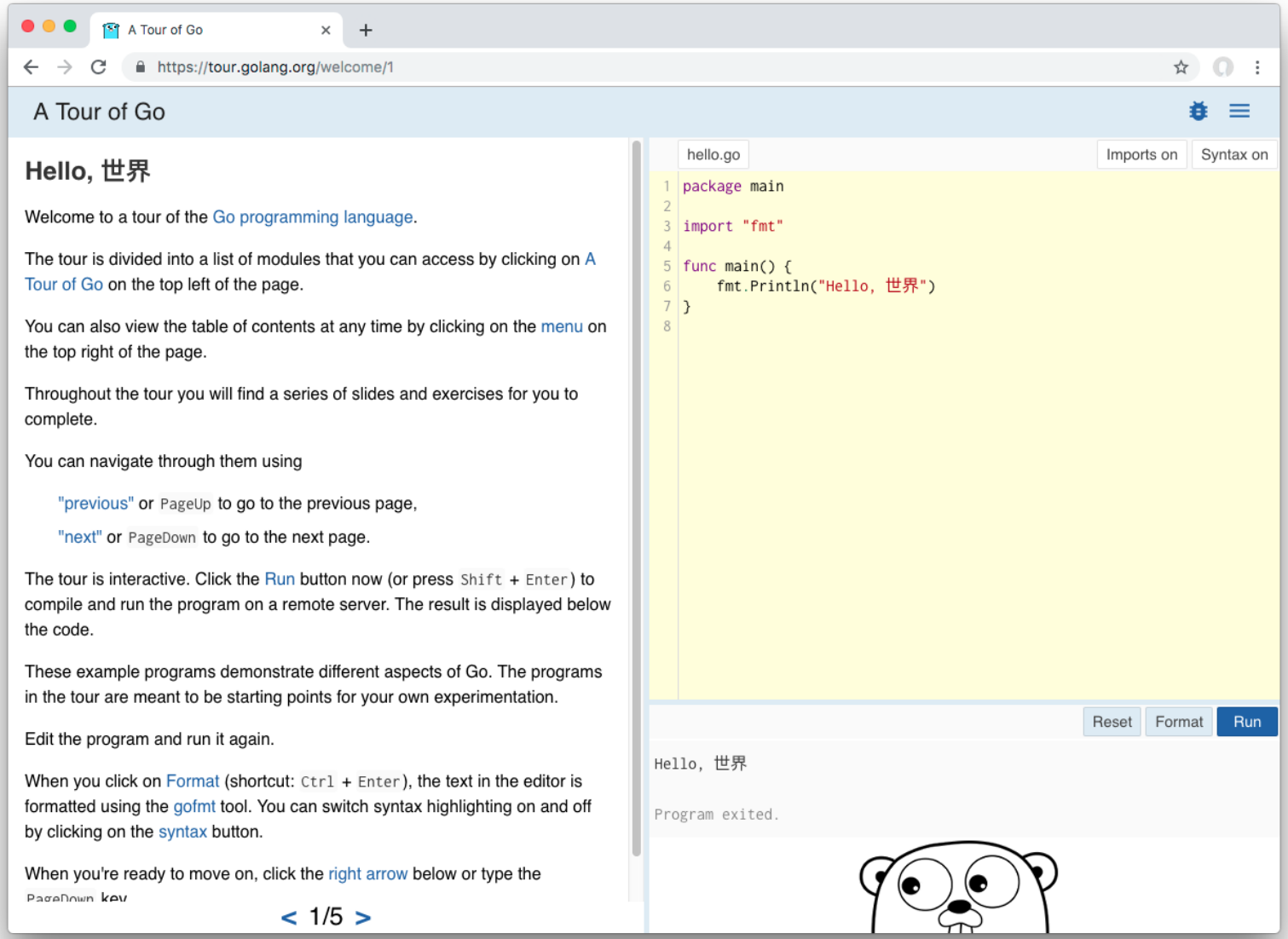
About me

- Author, *Head First Ruby* and *Head First Go*
- 4 years experience as online software development instructor
- See my recent courses at `https://teamtreehouse.com`

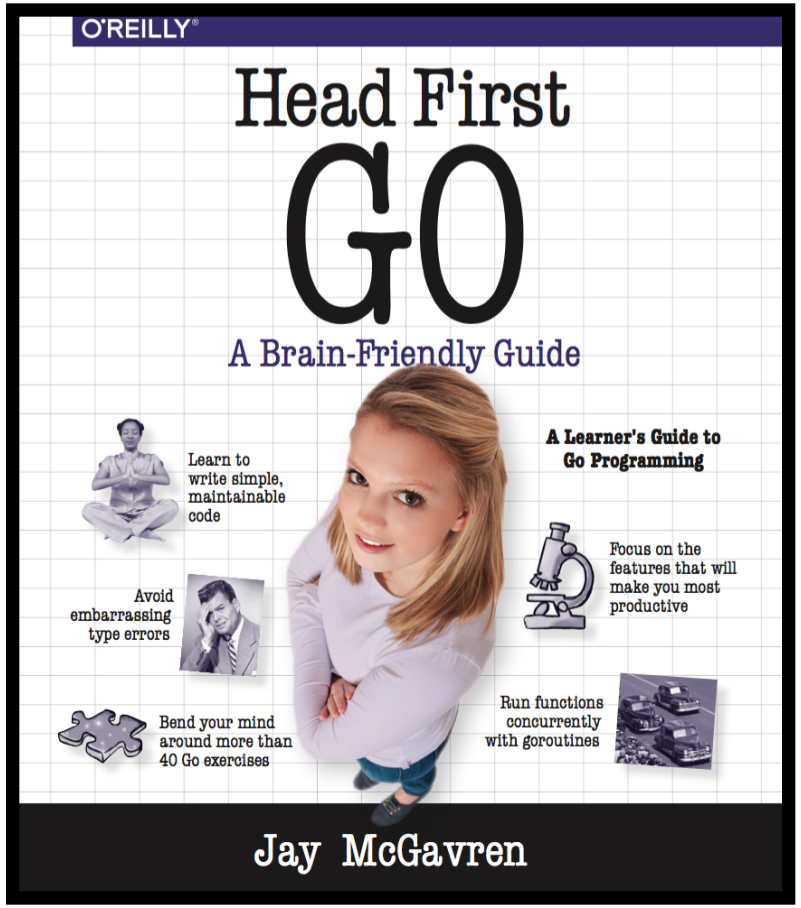
Where to Learn Go

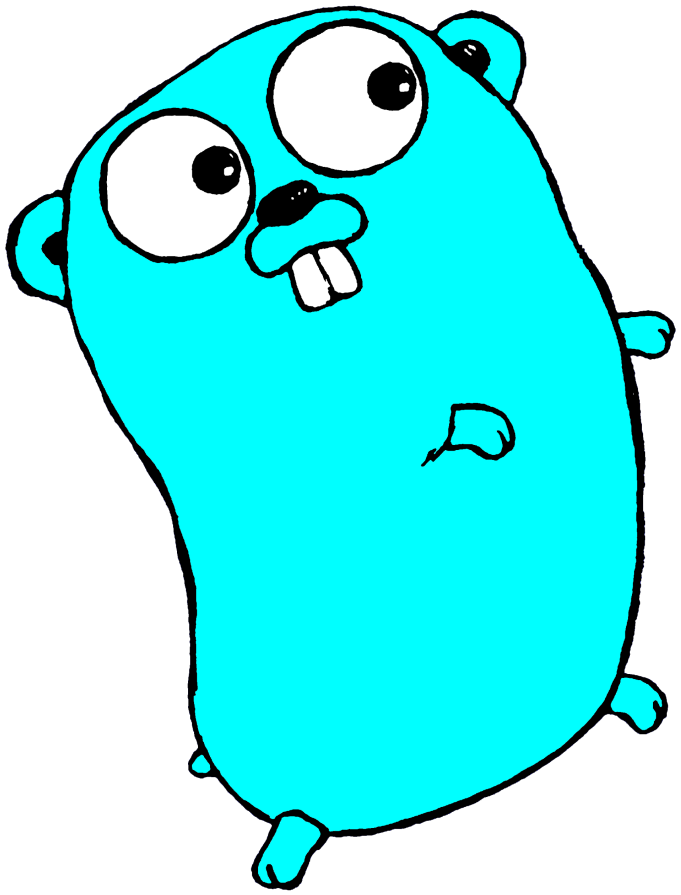
`https://tour.golang.org`

(We'll repeat that link at the end.)



Another humble recommendation





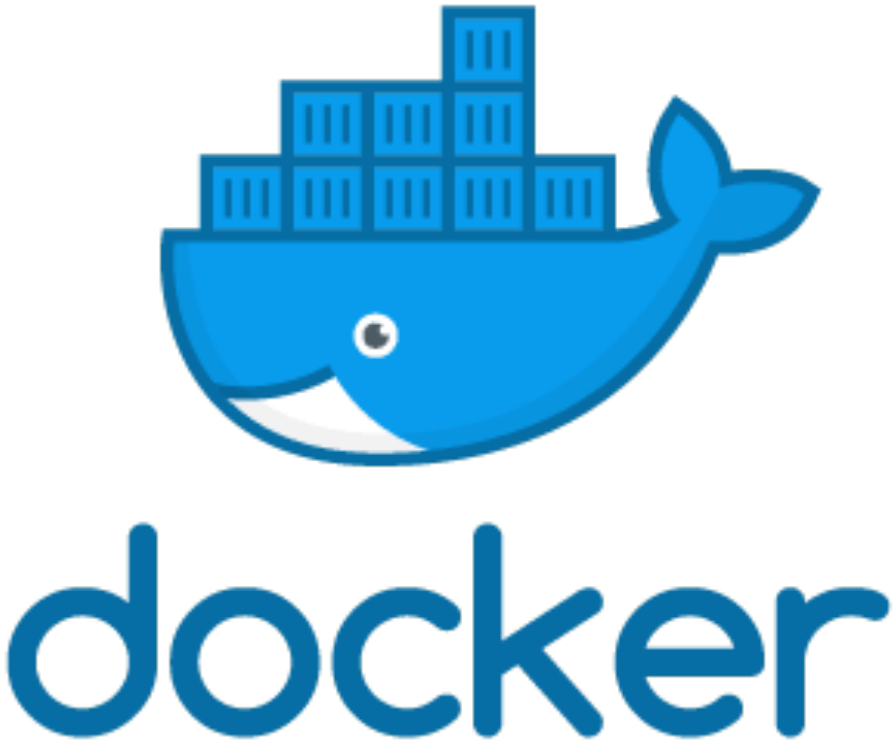
Why Go?

Go at a glance

- C-like syntax
- Compiles to native code
- Type-safe
- Garbage collected
- Concurrency built into language

OK, but what can you do with Go?

Docker



Docker

- “‘go build’ will embed everything you need. (No more ‘install this in order to run my stuff’.)”
- “Extensive standard library and data types.”
- “Strong duck typing.”

—Jérôme Petazzoni, “Docker and Go: why did we decide to write Docker in Go?”

Kubernetes

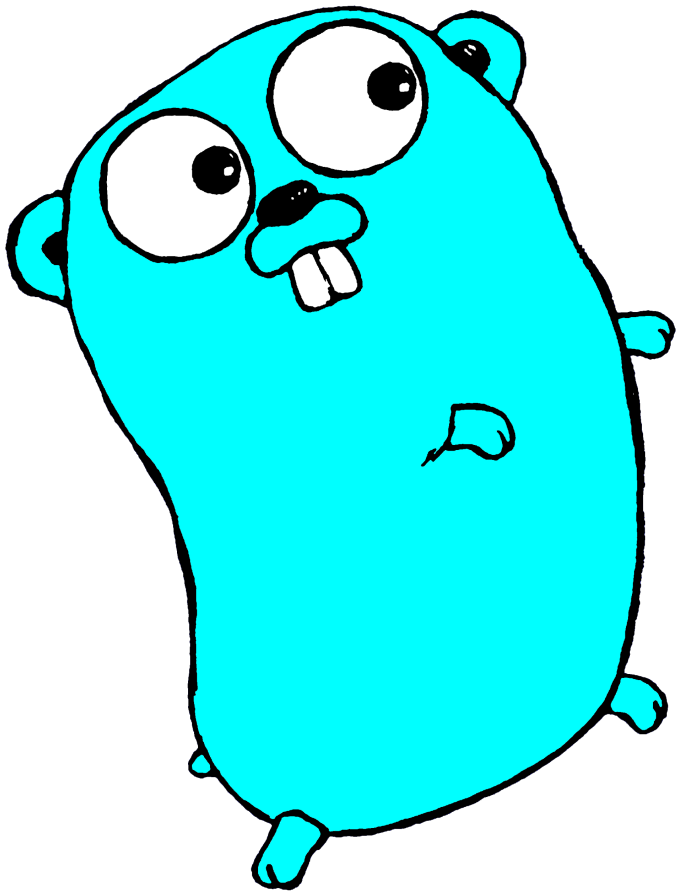
Kubernetes

- “Code in Go isn’t overly complex. People don’t create FactoryFactory objects.”
- “Something with the feel of C with more advanced features like anonymous functions is a great combo.”
- “Garbage Collection: We all know how to clean up after our selves but it is so nice to not have to worry about it.”

—Joe Beda, “Kubernetes + Go = Crazy Delicious”

Poll: What do you want to make with Go?

1. A system utility
2. A web app or service
3. Something else entirely
4. I don’t know yet



Why Go?

“go fmt”

- Automatically fixes code style
- Acts as community’s style guide
- No more arguing tabs vs. spaces!

“go fmt”

Before

```
package main

import "fmt"

func main() {
repeatLine("hello", 3     )
}

func repeatLine( line string ,times  int) {
    for i := 0; i < times; i++ {
fmt.Println(line)
    }
}
```

“go fmt”

```
$ go fmt repeat.go
```

“go fmt”

After

```
package main

import "fmt"

func main() {
    repeatLine("hello", 3)
}

func repeatLine(line string, times int) {
    for i := 0; i < times; i++ {
        fmt.Println(line)
    }
}
```

“go run”

- Compiles a Go source file and runs it
- No executable is saved

```
$ go run repeat.go
hello
hello
hello
```

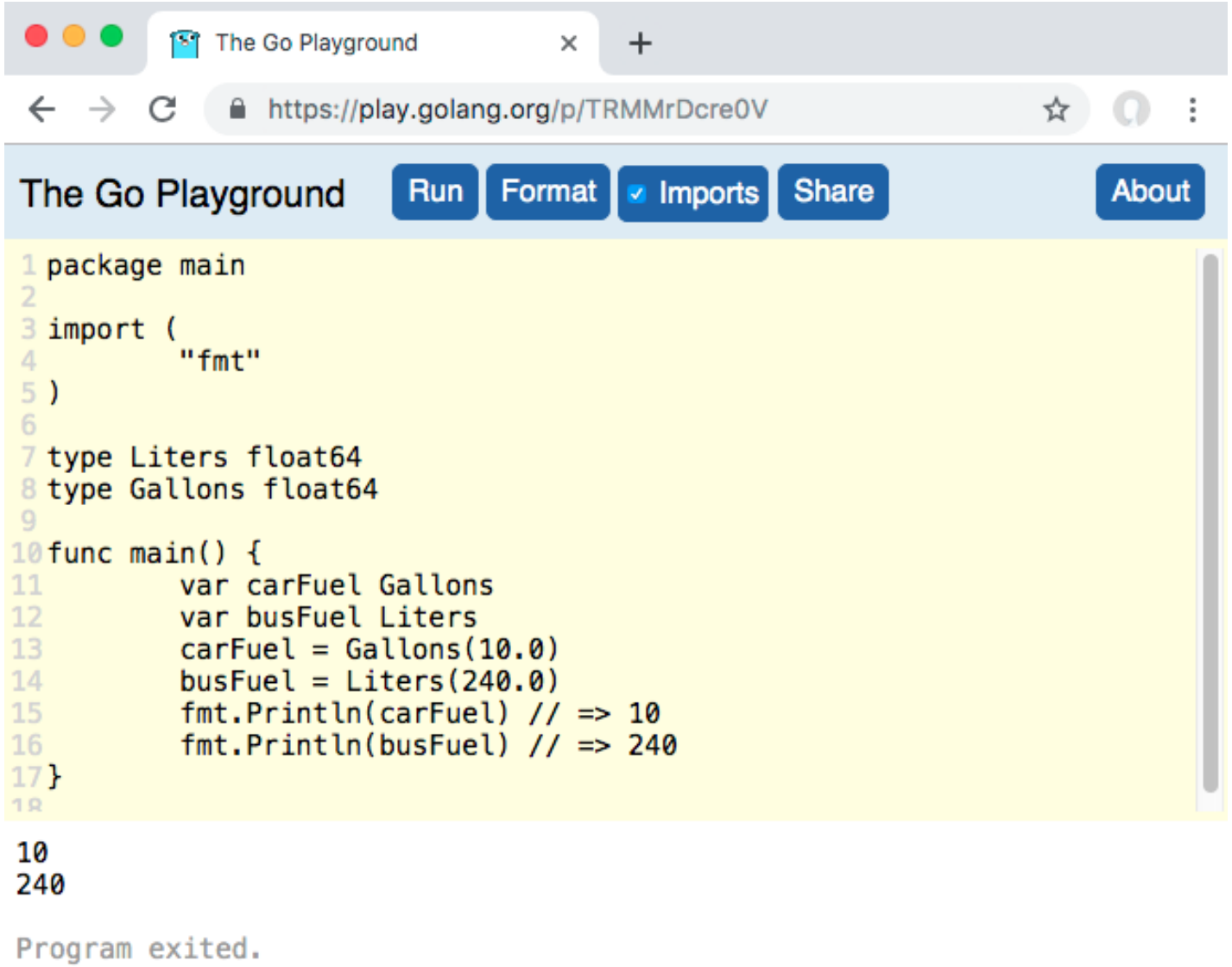
“go build”

- Compiles Go source file(s) into an executable

```
$ go build repeat.go
$ ls -l
total 2064
-rwxr-xr-x 1 jay staff 2106512 May  1 21:13 repeat
-rw-r--r-- 1 jay staff     166 May  1 21:13 repeat.go
$ ./repeat
hello
hello
hello
```

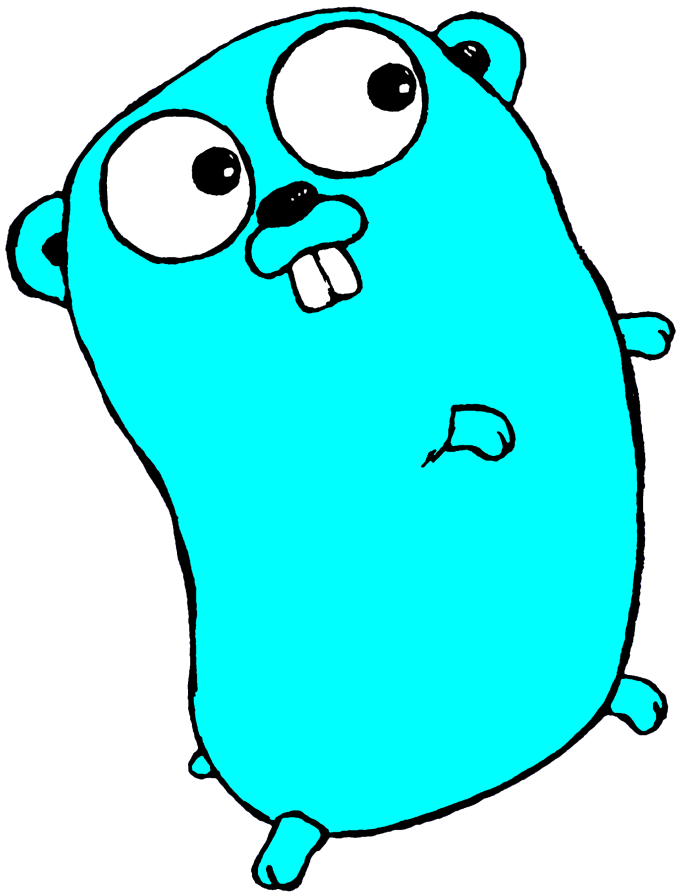
Playground

- You don’t even have to install Go to try it!
- Edit and run Go code in your browser



Exercises for this training

- The Go Playground lets you save your code and share it at a URL
- Others can edit that code
- We'll use that ability for most of our exercises
- We'll post a link
- You visit it and follow the instructions there (add code, fill in blanks, etc.)



Syntax

Go file layout

- Package clause
- Imports
- Code

```
package main

import "fmt"

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

Calling Functions

```
package main

import "fmt"

func main() {
    fmt.Println() // No arguments
    fmt.Println("argument 1")
    fmt.Println("argument 1", "argument 2")
}
```

Calling Functions

- `fmt.Println` can take any number and type of arguments.
- Most functions require a *specific* number and type of arguments.

```
package main

import "math"

func main() {
    math.Floor(3.1415)          // valid...
    math.Floor()                // not enough arguments
    math.Floor(3.1415, 12.34)   // too many arguments
    math.Floor("a string")      // wrong type
}
```

Imports

```
package main

func main() {
    fmt.Println(math.Floor(2.75))
    fmt.Println(strings.Title("head first go"))
}
```

Compile errors:

```
prog.go:4:2: undefined: fmt
prog.go:4:14: undefined: math
prog.go:5:2: undefined: fmt
prog.go:5:14: undefined: strings
```

Imports

Need to add `import` statement:

```
package main

import (
    "fmt"
    "math"
    "strings"
)

func main() {
    fmt.Println(math.Floor(2.75)) // => 2
    fmt.Println(strings.Title("head first go")) // => Head First Go
}
```

Unused imports not allowed

```
package main

import (
    "fmt"
    "os"
)

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

Compile error:

```
temp.go:5:5: imported and not used: "os"
```

“goimports”

- Wrapper for `go fmt`
- Automatically adds/removes imports

Install:

```
$ go get golang.org/x/tools/cmd/goimports
```

Do a web search for “goimports” for directions on integrating with your editor.

“goimports”

Before saving

```
package main

func main() {
    fmt.Println(math.Floor(2.75))
    fmt.Println(strings.Title("head first go"))
}
```

“goimports”

After saving

```
package main

import (
        "fmt"
        "math"
        "strings"
)

func main() {
        fmt.Println(math.Floor(2.75))
        fmt.Println(strings.Title("head first go"))
}
```

Variables

```
var myInteger int
myInteger = 1
var myFloat float64
myFloat = 3.1415
fmt.Println(myInteger)                  // => 1
fmt.Println(myFloat)                    // => 3.1415
fmt.Println(reflect.TypeOf(myInteger)) // => int
fmt.Println(reflect.TypeOf(myFloat))   // => float64
```

Short Variable Declarations

```
myInteger := 1
myFloat := 3.1415
fmt.Println(myInteger)                  // => 1
fmt.Println(myFloat)                    // => 3.1415
fmt.Println(reflect.TypeOf(myInteger)) // => int
fmt.Println(reflect.TypeOf(myFloat))   // => float64
```

Must use every variable you declare

```
subtotal := 24.70
tax := 1.89
fmt.Println(subtotal)
```

Compile error:

```
prog.go:9:2: tax declared and not used
```

Type conversions

```
var length float64 = 1.2
var width int = 2
// Can't assign an `int` value
// to a `float64` variable:
length = width
fmt.Println(length)
```

Compile error:

```
cannot use width (type int) as type float64 in assignment
```

Type conversions

```
var length float64 = 1.2
var width int = 2
// But you can if you do a type
// conversion!
length = float64(width)
fmt.Println(length) // => 2
```

Type conversions

```
var length float64 = 1.2
var width int = 2
// Can't do a math operation with a float64 and an int:
fmt.Println("Area is", length*width)
// Or a comparison:
fmt.Println("length > width?", length > width)
```

Type conversions

```
var length float64 = 1.2
var width int = 2
// But you can if you do type conversions!
fmt.Println("Area is", length*float64(width))
fmt.Println("length > width?", length > float64(width))
```

Output:

```
Area is 2.4
length > width? false
```

“if”

```
if 1 < 2 {
    fmt.Println("It's true!")
}
```

Output:

```
It's true!
```

“if”

Parentheses discouraged. `go fmt` will remove these:

```
if (1 < 2) {
    fmt.Println("It's true!")
}
```

Opening curly brace *must* be on same line as `if`. This is a syntax error:

```
if (1 < 2)
{
    fmt.Println("It's true!")
}
```

“for”

```
for x := 4; x <= 6; x++ {
    fmt.Println("x is now", x)
}
```

Output:

```
x is now 4
x is now 5
x is now 6
```

Exercise: Go syntax

https://is.gd/goex_syntax

Exercise: Go syntax

```
// Replace the blanks ("____") in the below code so that it
// compiles, runs, and prints the message "Hello, O'Reilly!".
____ main

____ "fmt"

____ main() {
    myString ____ "Hello, O'Reilly!"
    fmt.Println(____)
}
```

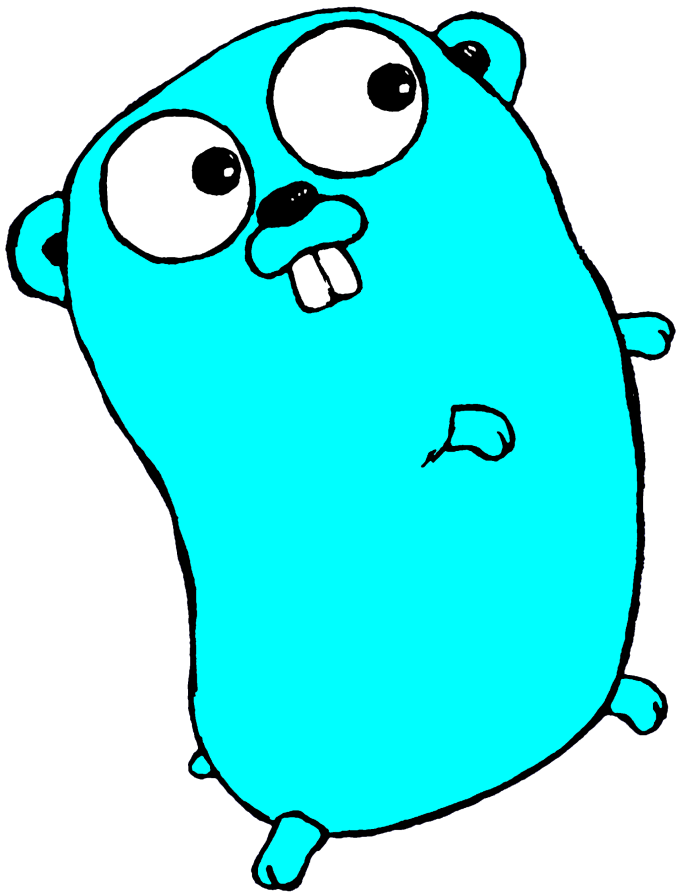
Exercise: Go syntax cheat sheet

- Every Go source file is part of a **package**.
- To use code from other packages, you have to **import** them.
- The `main` function is called when a program first starts.
- Functions are declared using the `func` keyword.
- A function call needs parentheses following the function name: `mypackage.MyFunction("my argument")`

`https://is.gd/goex_syntax`

Exercise: Go syntax solution

```
package main

import "fmt"

func main() {
    myString := "Hello, O'Reilly!"
    fmt.Println(myString)
}
```



Declaring Functions

Declaring functions

```
func sayHi() {
    fmt.Println("Hi!")
}

func main() {
    sayHi()
}
```

Function names

- Use `CamelCase`: capitalize each word after the first.
- If the first letter of a function name is `Capitalized`, it's considered **exported**: it can be used from other packages.
- If the first letter of a function name is `uncapitalized`, it's considered **unexported**: it can only be used *within* its package.
- This is why all the names of standard library functions we've been calling are capitalized. (E.g. `fmt.Println`, `math.Floor`, etc.)

More on exported/unexported later.

Parameters

```
// In parentheses, list parameter name(s)
// followed by type(s).
func say(phrase string, times int) {
    for i := 0; i < times; i++ {
        fmt.Print(phrase)
    }
    fmt.Print("\n")
}

func main() {
    // Provide argument(s) when calling.
    say("Hi", 4)  // => HiHiHiHi
    say("Bye", 2) // => ByeBye
}
```

Variable scope

Variable scope limited to function where it's declared.

```
func myFunction() {
    myVariable := 10
}

func main() {
    myFunction()
    fmt.Println(myVariable) // out of scope!
}
```

Compile error:

```
prog.go:11:14: undefined: myVariable
```

Variable scope

By the way, variable scope also limited by “if” blocks:

```
if grade >= 60 {
    status := "passing"
} else {
    status := "failing"
}
fmt.Println(status) // out of scope!
```

Variable scope

And by “for” blocks:

```
for x := 1; x <= 3; x++ {
    y := x + 1
    fmt.Println(y)
}
fmt.Println(y) // out of scope!
```

Variable scope

Solution is to declare variable *before* block:

```
var status string // declare up here
if grade >= 60 {
    status = "passing" // still in scope
} else {
    status = "failing" // still in scope
}
fmt.Println(status) // still in scope
```

Variable scope

Same with loops:

```
var y int // declare up here
for x := 1; x <= 3; x++ {
    y = x + 1 // still in scope
    fmt.Println(y)
}
fmt.Println(y) // still in scope
```

Function return values

So how do we get a value from a function to its caller?

```
func myFunction() {
    myVariable := 10
}

func main() {
    myFunction()
    fmt.Println(myVariable) // out of scope!
}
```

Function return values

Add a return value!

```
// Add return value type after parentheses
func myFunction() int {
    // Use "return" keyword
    return 10
}

func main() {
    // Assign returned value to variable
    myVariable := myFunction()
    fmt.Println(myVariable)
}
```

Multiple return values

```
func main() {
    flag := strconv.ParseBool("true")
    flag = strconv.ParseBool("foobar")
    fmt.Println(flag)
}
```

Compile error:

```
prog.go:9:7: assignment mismatch: 1 variable but strconv.ParseBool returns
2 values
prog.go:10:7: assignment mismatch: 1 variable but strconv.ParseBool
returns 2 values
```

Multiple return values

```
func main() {
    flag, err := strconv.ParseBool("true")
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(flag)
    flag, err = strconv.ParseBool("foobar")
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(flag)
}
```

Output:

```
true
2009/11/10 23:00:00 strconv.ParseBool: parsing "foobar": invalid syntax
```

Error handling

“In Go, error handling is important. The language’s design and conventions encourage you to explicitly check for errors where they occur (as distinct from the convention in other languages of throwing exceptions and **sometimes** catching them).” (Emphasis mine)

-Andrew Gerrand, https://blog.golang.org/error-handling-and-go

Writing functions with multiple return values

```
func myParseBool(myString string) (bool, error) {
    if myString == "true" {
        return true, nil
    } else if myString == "false" {
        return false, nil
    } else {
        return false, fmt.Errorf("bad string %s", myString)
    }
}

func main() {
    bool, err := myParseBool("false")
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(bool) // => false
}
```

Writing functions with multiple return values

```
func myParseBool(myString string) (bool, error) {
    if myString == "true" {
        return true, nil
    } else if myString == "false" {
        return false, nil
    } else {
        return false, fmt.Errorf("bad string %s", myString)
    }
}

func main() {
    bool, err := myParseBool("foobar")
    if err != nil {
        log.Fatal(err) // => 2020-03-13 19:34:00 bad string foobar
    }
    fmt.Println(bool)
}
```

Exercise: Declaring functions

https://is.gd/goex_define_functions

Exercise: Declaring functions

```
package main

import (
    "fmt"
)

// YOUR CODE HERE:
// Declare a "divide" function such that the call in the
// "main" function will compile and return 2.8.
// "divide" should accept two float64 values as parameters,
// and return a single float64 value that represents the
// first parameter divided by the second.
// EXTRA CREDIT:
// Have "divide" return TWO values, a float64 and an error.
// If the second parameter is 0, return an error value
// with the message "can't divide by 0". Otherwise, return
// nil for the error value. You can use the fmt.Errorf
// function to generate an error value. You'll also need
// to update the code in "main" to handle the error value.

func main() {
    quotient := divide(5.6, 2)
    fmt.Printf("%0.2f\n", quotient) // => 2.80
}
```

Exercise: Declaring functions cheat sheet

```
func oneReturnValue(param1 param1Type, param2 param2Type) returnType {
    return valueToReturn
}

func twoReturnValues(param1 param1Type, param2 param2Type) (returnType1,
returnType2) {
    if thereIsAProblem {
        return aMeaninglessValue, fmt.Errorf("an error message")
    }
    return valueToReturn, nil
}
```

https://is.gd/goex_define_functions

Exercise: Declaring functions solution

```
package main

import (
    "fmt"
)

func divide(dividend float64, divisor float64) float64 {
    return dividend / divisor
}

func main() {
    quotient := divide(5.6, 2)
    fmt.Printf("%0.2f\n", quotient)
}
```

Exercise: Declaring functions extra credit

```
package main

import (
    "fmt"
)

func divide(dividend float64, divisor float64) (float64, error) {
    if divisor == 0.0 {
        return 0, fmt.Errorf("can't divide by 0")
    }
    return dividend / divisor, nil
}

func main() {
    quotient, err := divide(5.6, 0.0)
    if err != nil {
        fmt.Println(err)
    } else {
        fmt.Printf("%0.2f\n", quotient)
    }
}
```

Pass-by-value

- Go is a “pass-by-value” language (as opposed to “pass-by-reference”).
- This means Go functions receive a copy of whatever values you pass to them.
- That’s fine, until you want a function to alter a value…

Pass-by-value

```
func main() {
    amount := 6
    // We want to set "amount" to 12
    double(amount)
    fmt.Println(amount) // But this prints "6"!
}

// double is SUPPOSED to take a value and double it
func double(number int) {
    // But this doubles the COPY, not the original
    number *= 2
}
```

Pointers

The `&` (“address of”) operator gets a pointer to a value.

```
amount := 6
fmt.Println(amount)  // => 6
fmt.Println(&amount) // => 0x1040a124
```

Pointers

We can get pointers to values of any type.

```
var myInt int
fmt.Println(&myInt)   // => 0x1040a128
var myFloat float64
fmt.Println(&myFloat) // => 0x1040a140
var myBool bool
fmt.Println(&myBool)  // => 0x1040a148
```

Pointers

A pointer to an `int` is written `*int`, a pointer to a `bool` as `*bool`, and so on.

```
func main() {
    var myInt int
    fmt.Println(reflect.TypeOf(&myInt))   // => *int
    var myFloat float64
    fmt.Println(reflect.TypeOf(&myFloat)) // => *float64
    var myBool bool
    fmt.Println(reflect.TypeOf(&myBool))  // => *bool
}
```

Pointers

You can declare variables that hold pointers:

```
var myInt int
var myIntPointer *int
myIntPointer = &myInt
fmt.Println(myIntPointer) // => 0x1040a128

var myFloat float64
var myFloatPointer *float64
myFloatPointer = &myFloat
fmt.Println(myFloatPointer) // => 0x1040a140
```

Pointers

The `*` *operator* gets the value a pointer refers to.

```
myInt := 4
myIntPointer := &myInt
fmt.Println(myIntPointer)    // => 0x1040a124
fmt.Println(*myIntPointer)   // => 4

myFloat := 98.6
myFloatPointer := &myFloat
fmt.Println(myFloatPointer)  // => 0x1040a140
fmt.Println(*myFloatPointer) // => 98.6
```

Pointers

The `*` operator can also be used to update the value at a pointer:

```
myInt := 4
fmt.Println(myInt)          // => 4
myIntPointer := &myInt
// Update the value at the pointer.
*myIntPointer = 8
fmt.Println(*myIntPointer) // => 8
fmt.Println(myInt)          // => 8
```

Pointers

We can use pointers to fix our `double` function:

```
func main() {
    amount := 6
    // Pass pointer instead of value
    double(&amount)
    fmt.Println(amount) // => 12
}

// Accept pointer instead of value
func double(number *int) {
    // Update value at pointer
    *number *= 2
}
```

Exercise: Passing pointers

https://is.gd/goex_pointers

Exercise: Passing pointers

```
// Update this program as described below.

package main

import "fmt"

// negate takes a boolean value and returns its
// opposite. E.g.: negate(false) returns true.
// But we WANT this function to accept a POINTER
// to a boolean value, and update the value at
// the pointer to its opposite. Once this change
// is made, the function doesn't need to return
// anything.
func negate(myBoolean bool) bool {
    return !myBoolean
}

func main() {
    truth := true
    // Change this to pass a pointer.
    negate(truth)
    // Prints "true", but we want "false".
    fmt.Println(truth)
    lies := false
    // Change this to pass a pointer.
    negate(lies)
    // Prints "false", but we want "true".
    fmt.Println(lies)
}
```

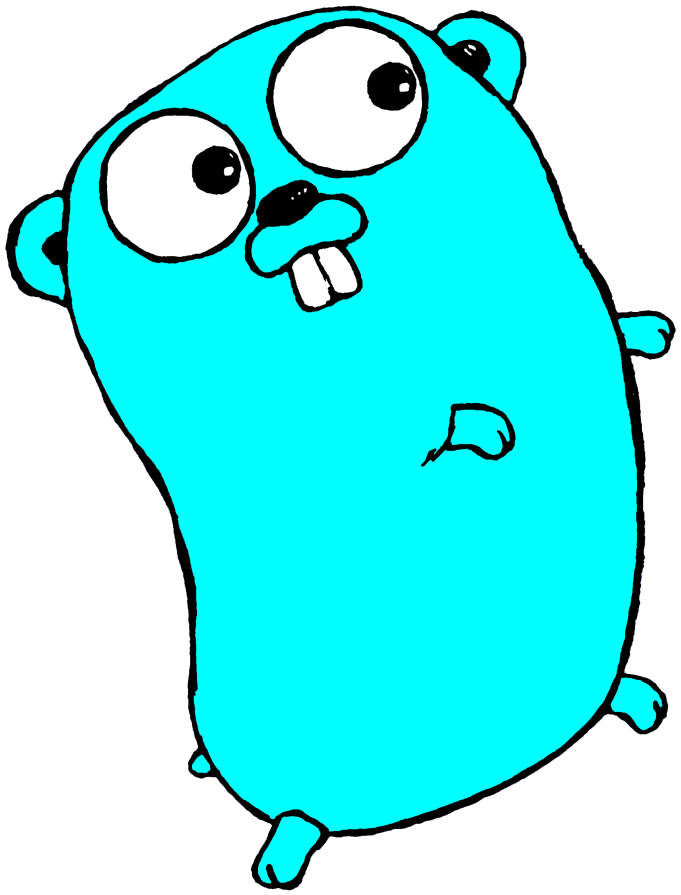
Exercise: Passing pointers cheat sheet

- `!true` is `false`, `!false` is `true`.
- Pointer types are written as `*myType`.
- Get a pointer to a variable's value with `&myVariable`.

`https://is.gd/goex_pointers`

Exercise: Passing pointers solution

```go
package main

import "fmt"

func negate(myBoolean *bool) {
    *myBoolean = !*myBoolean
}

func main() {
    truth := true
    negate(&truth)
    fmt.Println(truth) // => false
    lies := false
    negate(&lies)
    fmt.Println(lies) // => true
}
```



Declaring Packages

The “main” package

- Code intended for direct execution goes in the `main` package.
- Go looks for a `main` function and calls that first.

```
package main

import "fmt"

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

The “main” package

But sticking everything in one package will only get you so far…

The Go workspace

- A directory to hold package code.
- `~/go` by default.
- Or set `$GOPATH` environment variable to a different directory.

Workspace subdirectories

- `bin`: holds binary executables.
  - Add it to your `$PATH` and you can run them from anywhere.
- `pkg`: holds compiled package files.
  - You generally don't need to touch this.
- `src`: holds source code.
  - Including your code!

Setting up a package

Let’s move our functions to another package.

~/go/src/greeting/greeting.go

```
package greeting

import "fmt"

func Hello() {
        fmt.Println("Hello!")
}

func Hi() {
        fmt.Println("Hi!")
}
```

Importing our package

random_directory/hi.go

```
package main

import "greeting"

func main() {
        greeting.Hello()
        greeting.Hi()
}
```

“go run”

```
$ go run hi.go
Hello!
Hi!
```

Moving “main” to the workspace

~/go/src/hi/main.go

```
package main

import "greeting"

func main() {
        greeting.Hello()
        greeting.Hi()
}
```

“go install”

```
$ go install hi
$ tree ~/go
go
|-- bin
|   `-- hi
`-- src
    |-- greeting
    |   `-- greeting.go
    `-- hi
        `-- main.go
```

“go install”

```
$ export PATH=$PATH:$HOME/go/bin
```

(Go installer does this for you.)

“go install”

```
$ hi
Hello!
Hi!
```

Exported

We ensured our function names were capitalized so they were exported:

~/go/src/greeting/greeting.go

```
package greeting

import "fmt"

func Hello() {
        fmt.Println("Hello!")
}

func Hi() {
        fmt.Println("Hi!")
}
```

Unexported

What if we made them unexported?

~/go/src/greeting/greeting.go

```
package greeting

import "fmt"

func hello() {
        fmt.Println("Hello!")
}

func hi() {
        fmt.Println("Hi!")
}
```

Unexported

Even if we update the function calls in `main` to match…

`~/go/src/hi/main.go`

```
package main

import "greeting"

func main() {
        greeting.hello()
        greeting.hi()
}
```

Unexported

We’re not allowed to call unexported functions from another package.

```
$ go install hi
# hi
go/src/hi/main.go:6:9: cannot refer to unexported name greeting.hello
go/src/hi/main.go:6:9: undefined: greeting.hello
go/src/hi/main.go:7:9: cannot refer to unexported name greeting.hi
go/src/hi/main.go:7:9: undefined: greeting.hi
```

Unexported

So why would you ever make functions unexported?

- Unexported methods are Go's equivalent to Java's `private` methods.
- Use for helper functions that other packages shouldn't call.
- Once you export a function, you shouldn't change it any more.
  - You can change how it works *internally*…
  - But you shouldn't change its parameters, return value, etc.
  - If you do, you risk breaking others' code!
- But you can change unexported functions all you want!

Import paths

Suppose we want to add support for other languages…

We can nest them under the `greeting` directory.

```
$ tree ~/go/
go
`-- src
    `-- greeting
        |-- dansk
        |   `-- dansk.go
        |-- deutsch
        |   `-- deutsch.go
        `-- greeting.go
```

Import paths

~/go/src/greeting/deutsch/deutsch.go

```
// Notice it's not "greeting/deutsch",
// it's just "deutsch".
package deutsch

import "fmt"

func Hallo() {
    fmt.Println("Hallo!")
}

func GutenTag() {
    fmt.Println("Guten Tag!")
}
```

Import paths

~/go/src/greeting/dansk/dansk.go

```
// Notice it's not "greeting/dansk",
// it's just "dansk".
package dansk

import "fmt"

func Hej() {
    fmt.Println("Hej!")
}

func GodMorgen() {
    fmt.Println("God morgen!")
}
```

Import paths

Now we can import and use these packages as well.

~/go/src/hi/main.go`

```
package main

import (
    "greeting"
    "greeting/dansk"
    "greeting/deutsch"
)

func main() {
    greeting.Hello()   // => Hello!
    greeting.Hi()      // => Hi!
    dansk.Hej()        // => Hej!
    dansk.GodMorgen()  // => God morgen!
    deutsch.Hallo()    // => Hallo!
    deutsch.GutenTag() // => Guten Tag!
}
```

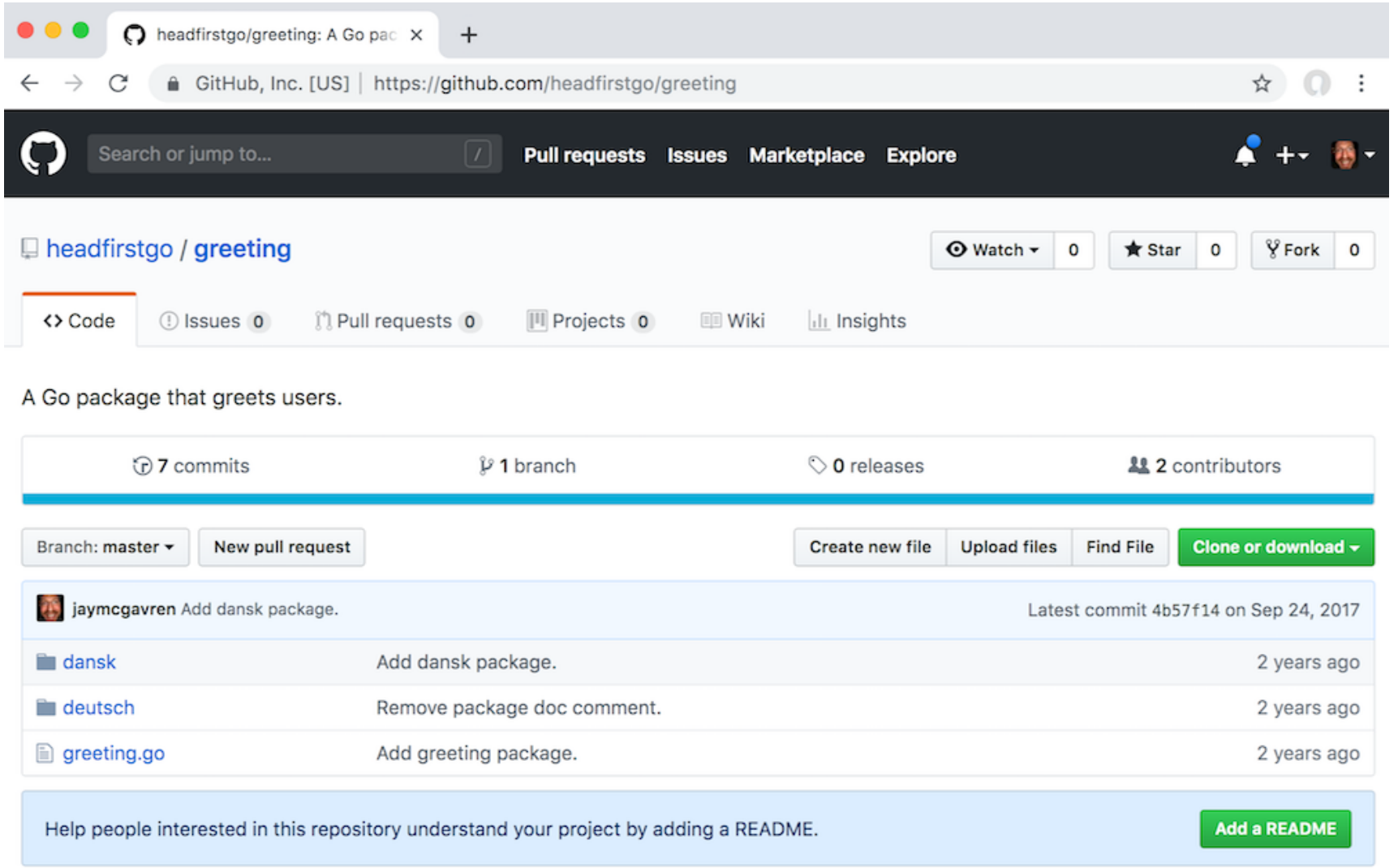
Import paths

- Notice the import paths are not the same as the package names!
- Package name is whatever is used in `package` clause in files: `package dansk`
- By convention, last segment of import path is used as package name.

Import Path	Package Name
greeting	*greeting*
greeting/*dansk*	*dansk*
greeting/*deutsch*	*deutsch*

“go get”

I set up a repo at `https://github.com/headfirstgo/greeting`, and pushed the package code there…



“go get”

Now anyone can retrieve the package with `go get github.com/headfirstgo/greeting`:

```
$ go get github.com/headfirstgo/greeting
$ tree ~/go
go
`-- src
    `-- github.com
        `-- headfirstgo
            `-- greeting
                |-- dansk
                |   `-- dansk.go
                |-- deutsch
                |   `-- deutsch.go
                `-- greeting.go
```

“go get”

~/go/src/hi/main.go

```
package main

import (
        "github.com/headfirstgo/greeting"
        "github.com/headfirstgo/greeting/dansk"
        "github.com/headfirstgo/greeting/deutsch"
)

func main() {
        greeting.Hello()   // => Hello!
        dansk.Hej()        // => Hej!
        deutsch.GutenTag() // => Guten Tag!
}
```

“go doc”

- To document a package, just add an ordinary comment before its `package` clause.
- Comments can span as many lines as you need.

```
// Package greeting greets the user in English.
package greeting

import "fmt"

func Hello() {
    fmt.Println("Hello!")
}

func Hi() {
    fmt.Println("Hi!")
}
```

“go doc”

- To document functions, add an ordinary comment before each.

```
// Package greeting greets the user in English.
package greeting

import "fmt"

// Hello prints the string "Hello!".
func Hello() {
    fmt.Println("Hello!")
}

// Hi prints the string "Hi!".
func Hi() {
    fmt.Println("Hi!")
}
```

“go doc”

Get documentation on a package:

```
$ go doc github.com/headfirstgo/greeting
package greeting // import "github.com/headfirstgo/greeting"

Package greeting greets the user.

func Hello()
func Hi()
```

“go doc”

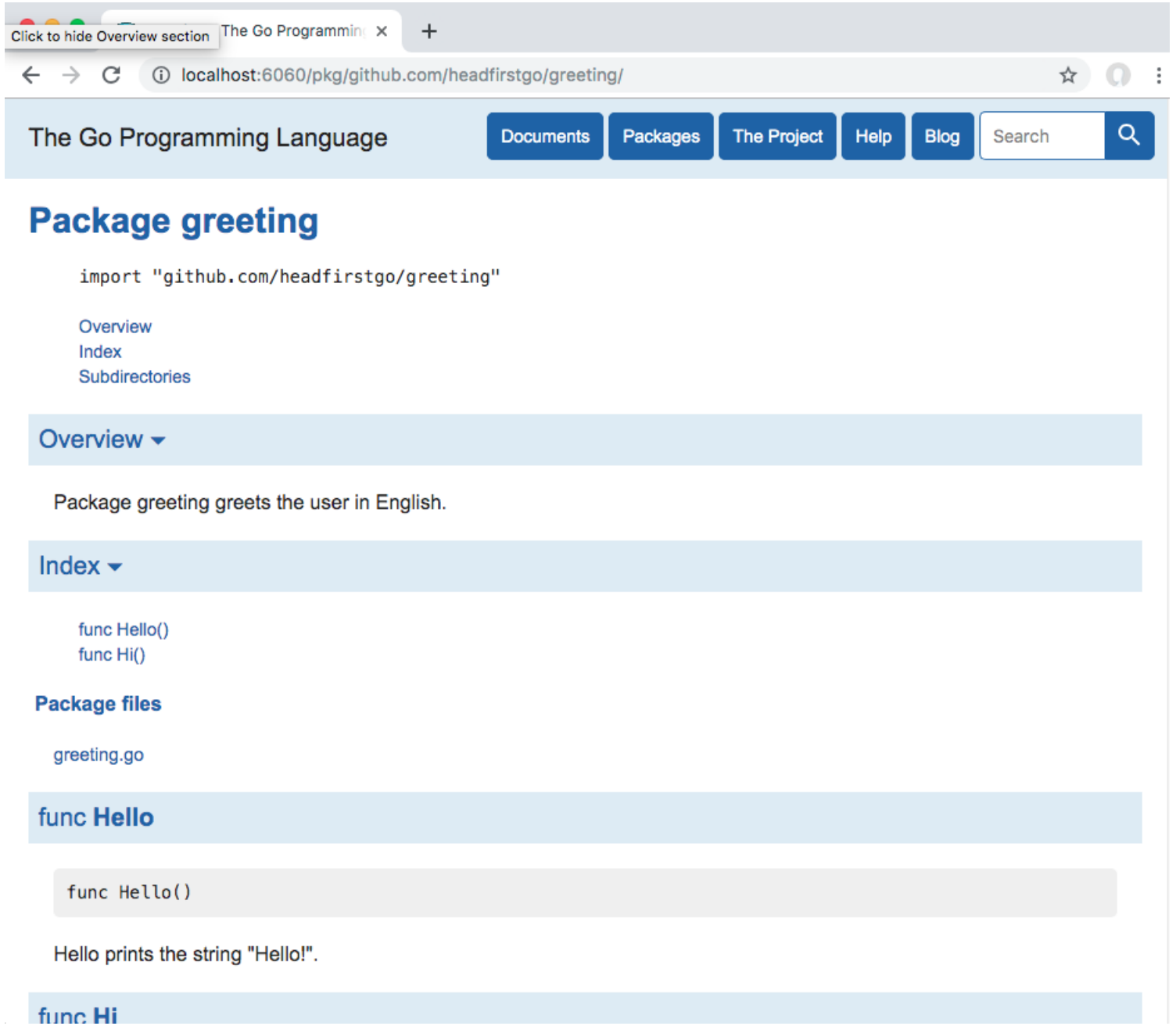
Get documentation on a function:

```
$ go doc github.com/headfirstgo/greeting Hello
func Hello()
    Hello prints the string "Hello!".
```

Web documentation

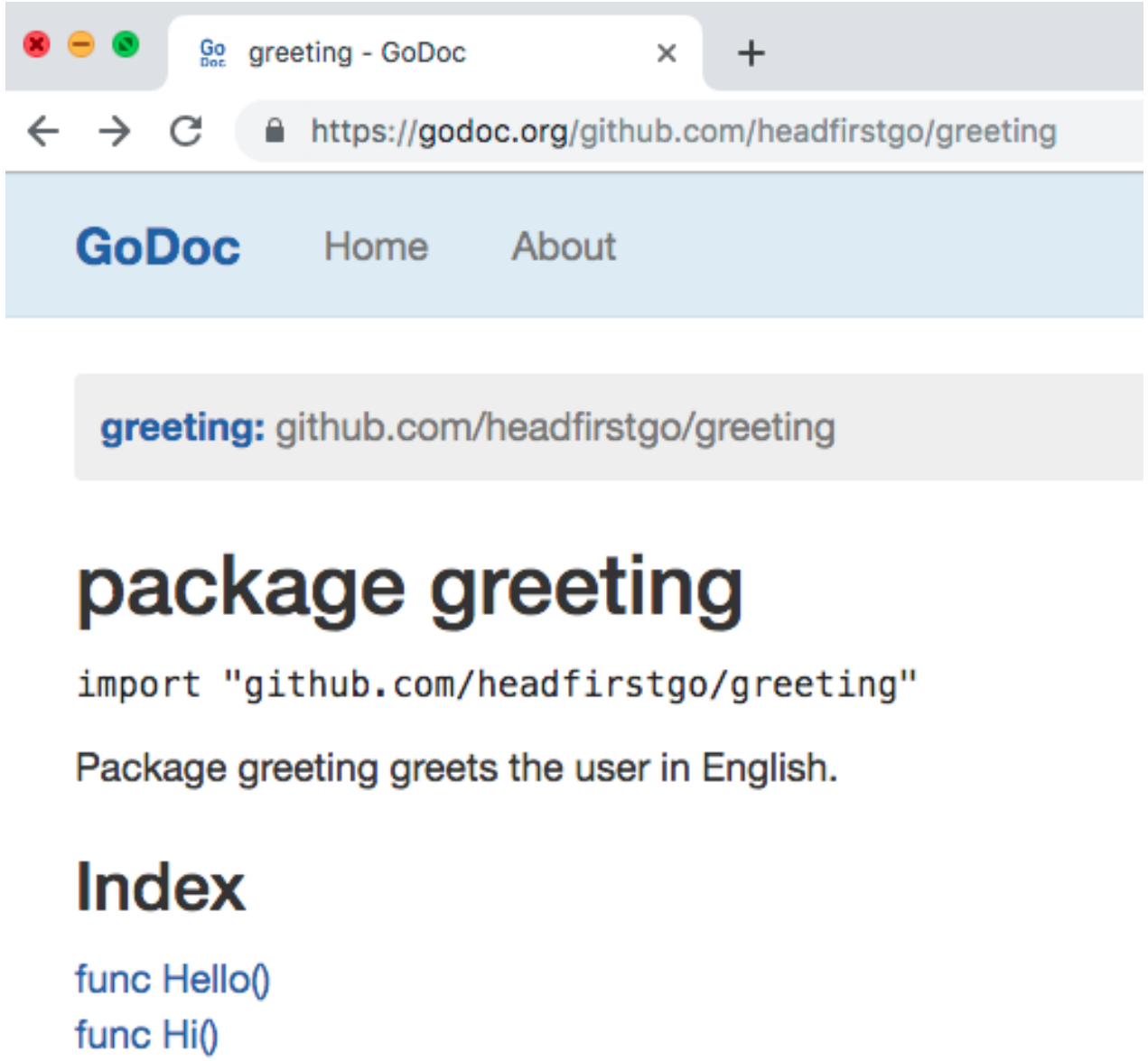
```
$ godoc -http=:6060
```

Then visit `http://localhost:6060/pkg/...`



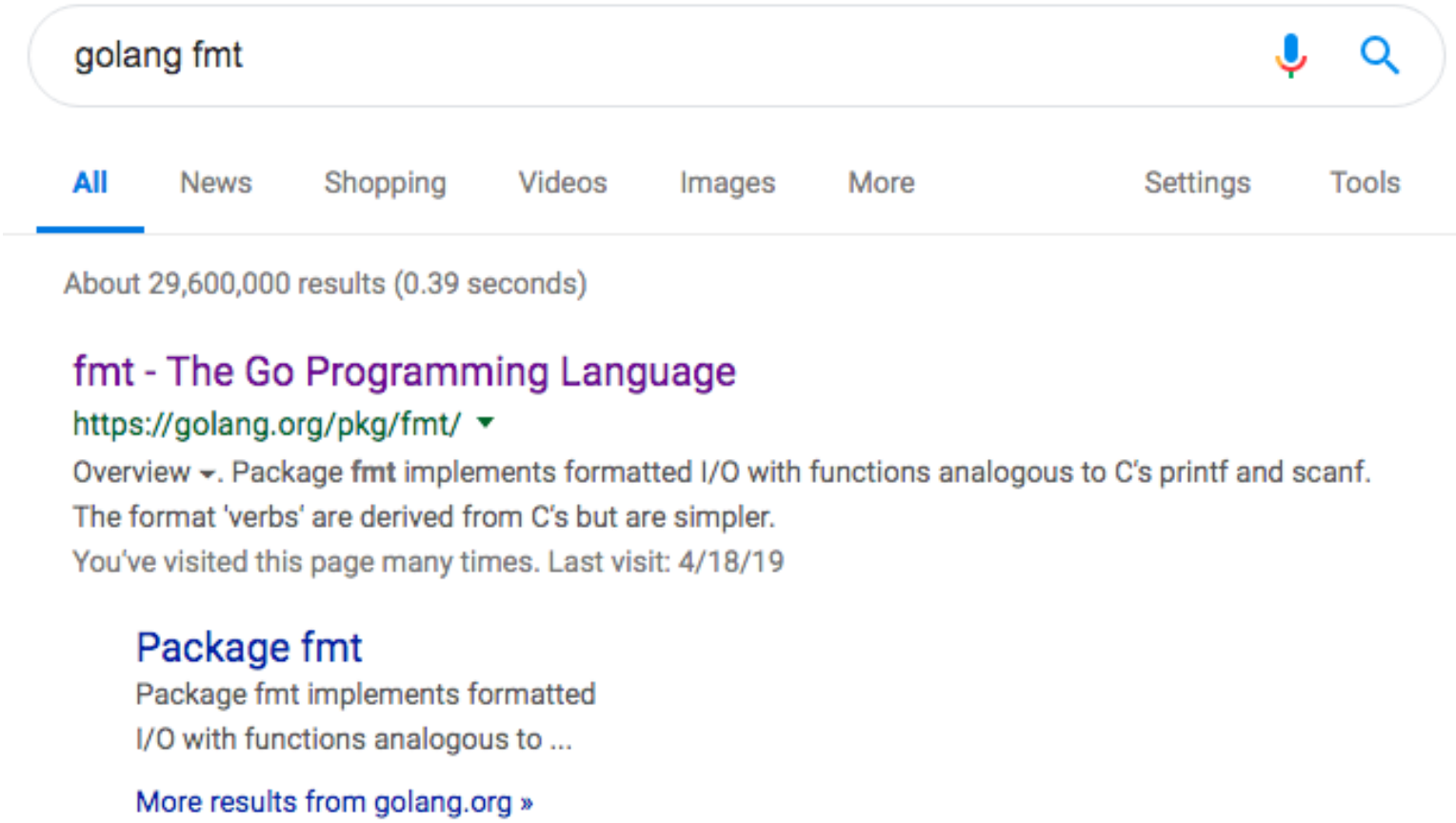
Web documentation

- Other servers like `godoc.org` make package documentation available on the web.
- For example, `https://godoc.org/github.com/headfirstgo/greeting` got created automatically for the `greeting` package.

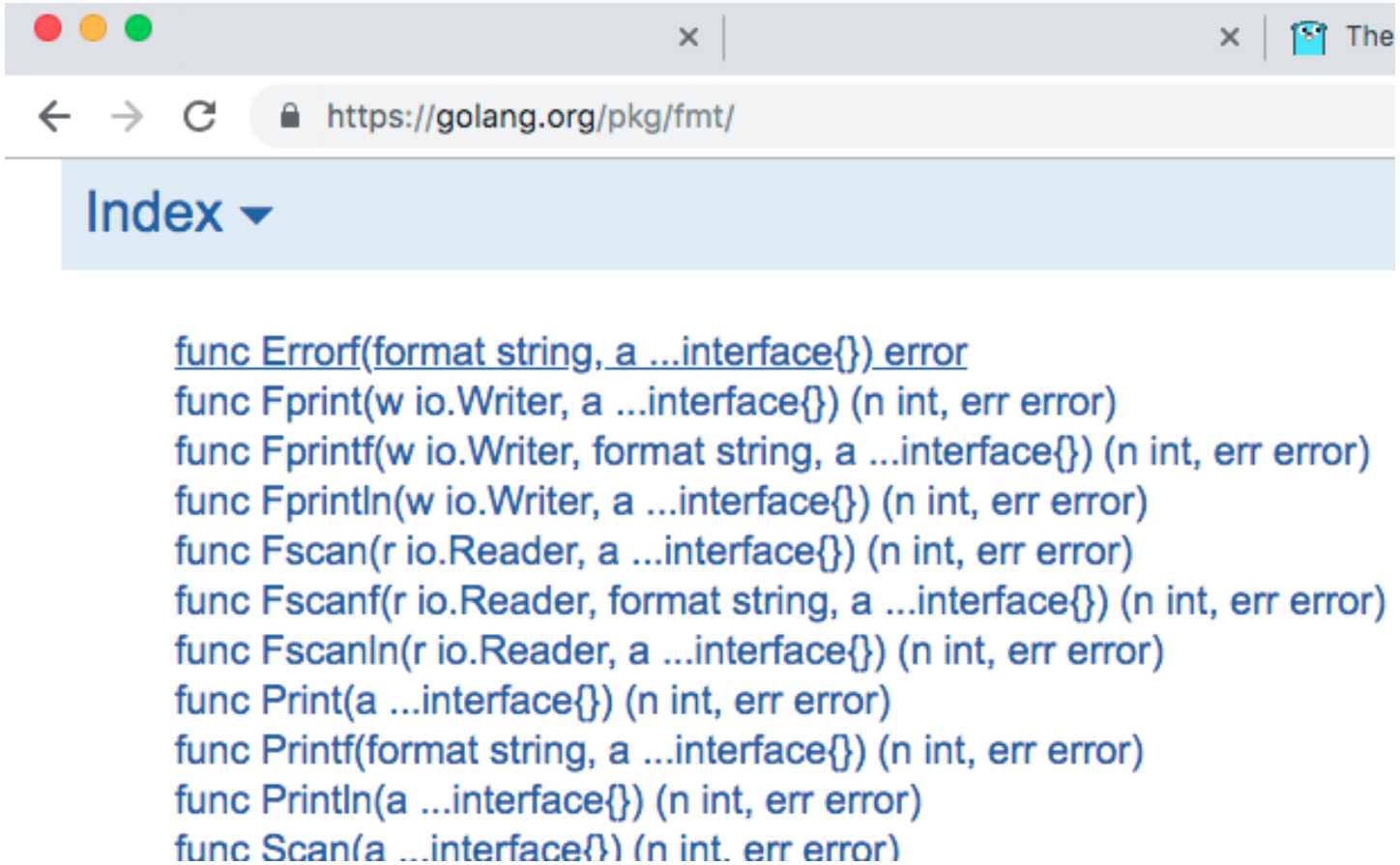


Web documentation

- Want to know more about the `fmt` package?
- Just Google “golang fmt”!



Web documentation



Exercise: Using package documentation

https://is.gd/goex_documentation

Exercise: Using package documentation

```go
package main

import (
    "fmt"
    "log"
)

func main() {
    string1 := "12.345"
    string2 := "1.234"

    // YOUR CODE HERE:
    // Look up documentation for the "strconv" package's
    // ParseFloat function. (You can use either "go doc"
    // or a search engine.) Use ParseFloat to convert
    // string1 to a float64 value. Assign the converted
    // number to the variable number1, and any error value
    // to the variable err. Use the integer 64 for
    // ParseFloat's bitSize argument.

    if err != nil {
        log.Fatal("Could not parse string")
    }

    // YOUR CODE HERE:
    // Use ParseFloat to convert string2 to a float64
    // value. Assign the converted number to the variable
    // number2, and any error value to the variable err.

    if err != nil {
        log.Fatal("Could not parse string")
    }

    fmt.Println(number1 - number2)
}
```

Exercise: Using package documentation cheat sheet

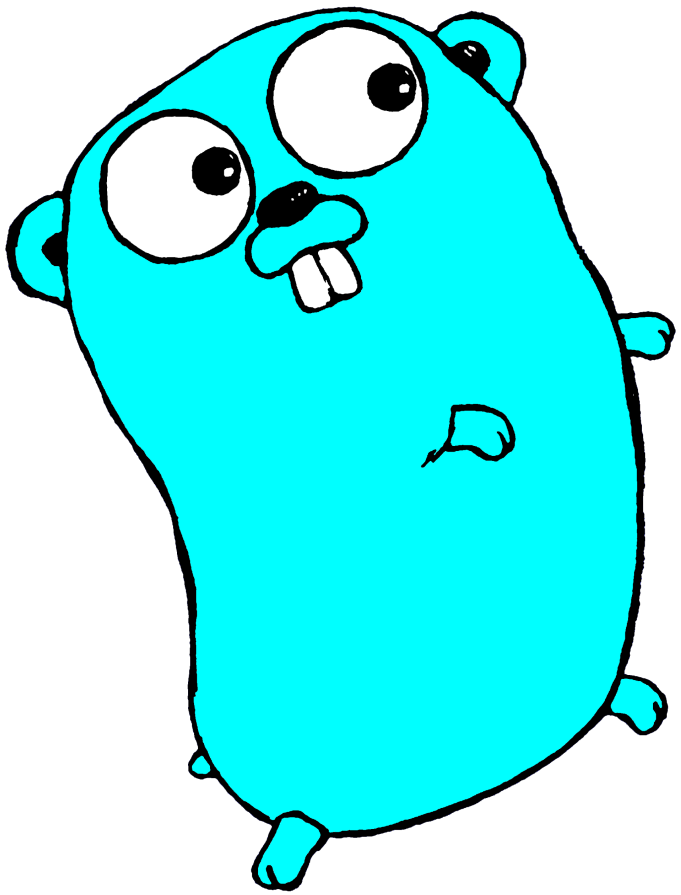
```
$ go doc strconv Parsefloat
func ParseFloat(s string, bitSize int) (float64, error)
    ParseFloat converts the string s to a floating-point number...
```

- Don't forget to import the `"strconv"` package.
- Use the package name before the function name: `strconv.ParseFloat(...)`
- The first argument to `ParseFloat` is the string you want to convert to a `float64`.
- Use the integer `64` as the second argument.
- Provide variables for both the `float64` return value and the `error` return value: `number1, err := strconv.ParseFloat(string1, 64)`

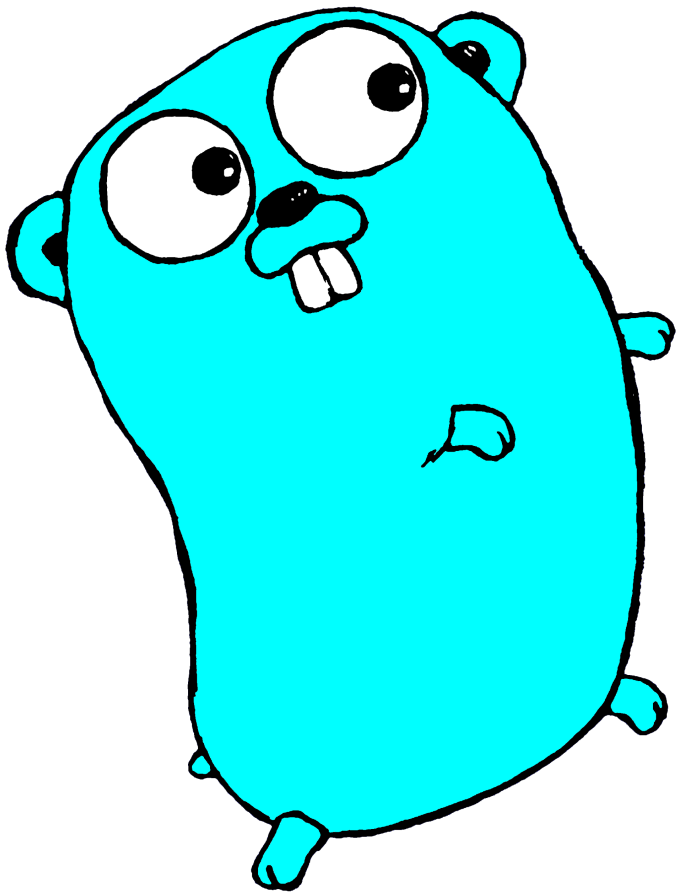
`https://is.gd/goex_documentation`

Exercise: Using package documentation solution

```
package main

import (
    "fmt"
    "log"
    "strconv"
)

func main() {
    string1 := "12.345"
    string2 := "1.234"
    number1, err := strconv.ParseFloat(string1, 64)
    if err != nil {
        log.Fatal("Could not parse string")
    }
    number2, err := strconv.ParseFloat(string2, 64)
    if err != nil {
        log.Fatal("Could not parse string")
    }
    fmt.Println(number1 - number2)
}
```



Data Structures



Arrays

Arrays

```
// Array type written as [size]ContainedType
var primes [3]int
// Array indices start at 0
primes[0] = 2
primes[1] = 3
fmt.Println(primes[0]) // => 2
fmt.Println(primes[1]) // => 3
```

Arrays

Arrays can hold any type

```
var flags [2]bool
flags[0] = true
flags[1] = false
fmt.Println(flags) // => [true false]
var names [5]string
names[0] = "Tracy"
names[2] = "Gerard"
fmt.Println(names) // => [Tracy  Gerard  ]
```

Arrays and “fmt.Printf”

Hmm, spacing on that last one is weird…

```
var names [5]string
names[0] = "Tracy"
names[2] = "Gerard"
fmt.Println(names) // => [Tracy  Gerard  ]
```

Let’s look at it with `fmt.Printf("%#v")`:

```
fmt.Printf("%#v\n", names) // => [5]string{"Tracy", "", "Gerard", "", ""}
```

Arrays and zero values

Unless otherwise assigned, array elements hold the zero values for their type.

```
var primes [3]int
fmt.Printf("%#v\n", primes) // => [3]int{0, 0, 0}
var flags [2]bool
fmt.Printf("%#v\n", flags)  // => [2]bool{false, false}
var names [5]string
fmt.Printf("%#v\n", names)  // => [5]string{"", "", "", "", ""}
```

Array literals

- Remember `fmt.Printf("%#v")` prints values as they would appear in Go code.

```
var names [5]string
names[0] = "Tracy"
names[2] = "Gerard"
fmt.Printf("%#v\n", names) // => [5]string{"Tracy", "", "Gerard", "", ""}
```

- That syntax is an array **literal**.

Array literals

Use array literals to create an array and initialize its elements all at once.

```
names := [5]string{"Tracy", "", "Gerard", "", ""}
fmt.Println(names[0]) // => Tracy
fmt.Println(names[1]) // =>
fmt.Println(names[2]) // => Gerard
fmt.Println(names[3]) // =>
fmt.Println(names[4]) // =>
```

Array literals

Slices and maps have a similar literal syntax; we'll get to those in a bit.

```
myArray := [3]string{"Amy", "Jose", "Ben"}
mySlice := []string{"Amy", "Jose", "Ben"}
myMap := map[string]int{"Amy": 84, "Jose": 96, "Ben": 78}
```

Arrays and loops

Array elements can be accessed using a loop.

```
names := [3]string{"Amy", "Jose", "Ben"}
for i := 0; i < len(names); i++ {
    fmt.Println(names[i])
}
```

Output:

```
Amy
Jose
Ben
```

Arrays and loops

Don’t access/assign outside array bounds; program will panic (a runtime error that crashes the program).

```
names := [3]string{"Amy", "Jose", "Ben"}
for i := 0; i <= len(names); i++ {
    fmt.Println("index", i, names[i])
}
```

```
index 0 Amy
index 1 Jose
index 2 Ben
panic: runtime error: index out of range

goroutine 1 [running]:
main.main()
    /tmp/sandbox741567581/prog.go:10 +0x180
```

“for … range” loops

It’s safer to use a `for ... range` loop:

```
names := [3]string{"Amy", "Jose", "Ben"}
for index, name := range names {
    fmt.Println(index, name)
}
```

“for … range” loops and blank identifier

Don’t want the index, or don’t want the element? Assign it to the blank identifier.

```
names := [3]string{"Amy", "Jose", "Ben"}
for _, name := range names {
    fmt.Println(name)
}
for index, _ := range names {
    fmt.Println(index)
}
```

Output:

```
Amy
Jose
Ben
0
1
2
```

“for … range” loops and other types

`for ... range` also available with slices…

```
mySlice := []string{"Amy", "Jose", "Ben"}
for index, name := range mySlice {
    fmt.Println(index, name)
}
```

Output:

```
0 Amy
1 Jose
2 Ben
```

“for … range” loops and other types

`for ... range` also available with maps.

```
myMap := map[string]int{"Amy": 84, "Jose": 96, "Ben": 78}
for name, score := range myMap {
    fmt.Println(name, score)
}
```

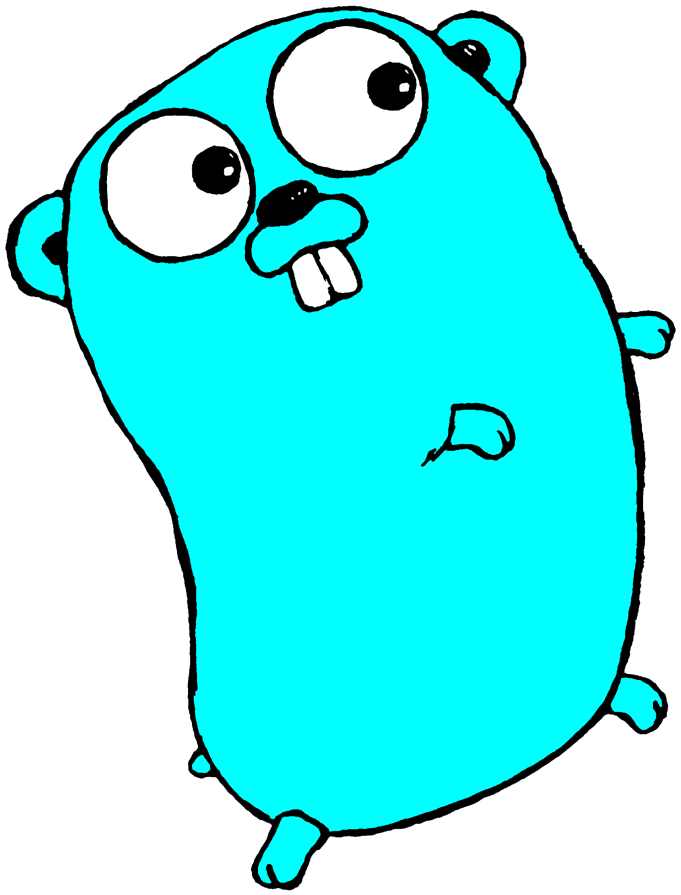
Output:

```
Amy 84
Jose 96
Ben 78
```

More on these later.

Limitations of arrays

- Arrays are fixed size - can’t grow when needed.
- To me, they’re only useful as a basis for slices.



Slices

Slices

Slice type is written just like array type, but with nothing between the `[]`.

```
var myArray [3]int
var mySlice []int
```

Zero value for slices

Unlike with arrays, the zero value for a slice is `nil`.

```
var mySlice []int
if mySlice == nil {
    fmt.Println("mySlice is nil") // => mySlice is nil
}
```

Slices and “make()”

Make a new slice with the built-in `make` function.

```
var mySlice []int
mySlice = make([]int, 3)
if mySlice == nil {
    fmt.Println("mySlice is nil") // doesn't run
}
fmt.Println(len(mySlice))  // => 3
fmt.Printf("%#v\n", mySlice) // => []int{0, 0, 0}
```

Slices and “make()”

Could rewrite that using a short variable declaration:

```
mySlice := make([]int, 3)
if mySlice == nil {
    fmt.Println("mySlice is nil") // doesn't run
}
fmt.Println(len(mySlice))  // => 3
fmt.Printf("%#v\n", mySlice) // => []int{0, 0, 0}
```

Zero values of slice elements

Just like with arrays, the zero value of each slice element is the zero value for the type the slice holds.

```
myFloats := make([]float64, 2)
fmt.Printf("%#v\n", myFloats) // => []float64{0, 0}
myStrings := make([]string, 4)
fmt.Printf("%#v\n", myStrings) // => []string{"", "", "", ""}
```

Slices

Use slices just like an array!

```
primes := make([]int, 2)
primes[0] = 2
primes[1] = 3
fmt.Println(primes[0])      // => 2
fmt.Println(primes[1])      // => 3
fmt.Printf("%#v\n", primes) // => []int{2, 3}
```

Slices and “append”

Need to add more items? Use the built-in `append` function!

```
primes := make([]int, 2)
primes[0] = 2
primes[1] = 3
fmt.Println(len(primes))    // => 2
fmt.Printf("%#v\n", primes) // => []int{2, 3}
primes = append(primes, 5)
fmt.Println(len(primes))    // => 3
fmt.Printf("%#v\n", primes) // => []int{2, 3, 5}
```

Slice literals

A slice literal looks just like an array literal, except the `[]` is empty:

```
primes := []int{2, 3, 5}
fmt.Println(primes[0]) // => 2
fmt.Println(primes[1]) // => 3
fmt.Println(primes[2]) // => 5
```

Slices and “for … range”

Use `for ... range` with a slice just like you do with an array.

```
names := []string{"Amy", "Jose", "Ben"}
for index, name := range names {
    fmt.Println(index, name)
}
```

Output:

```
0 Amy
1 Jose
2 Ben
```

Exercise: Slices

https://is.gd/goex_slices

Exercise: Slices

```
// Fill in the blanks so this program compiles and produces
// the output shown.
package main

import "fmt"

func main() {
    // Declare a variable that holds a slice of ints.
    var wholeNumbers ____
    // Create a new slice of 5 elements and assign it to the variable.
    wholeNumbers = ____
    // Assign 42 to the second element.
    wholeNumbers[____] = 42
    fmt.Printf("%#v\n", wholeNumbers) // => []int{0, 42, 0, 0, 0}

    // Make a 1-element slice of booleans.
    flags := ____
    // Set the first and only element to true.
    flags[0] = true
    // Append the value false to the slice.
    flags = ____
    // Append the value true to the slice.
    flags = ____
    fmt.Printf("%#v\n", flags) // => []bool{true, false, true}

    // Create a slice with the strings "cat", "bat", and "rat".
    words := ____
    // For each element of the slice, print its index and value.
    for ____ := ____ words {
        fmt.Println(i, word)
    }
    // => 0 cat
    // => 1 bat
    // => 2 rat
}
```

Exercise: Slices cheat sheet

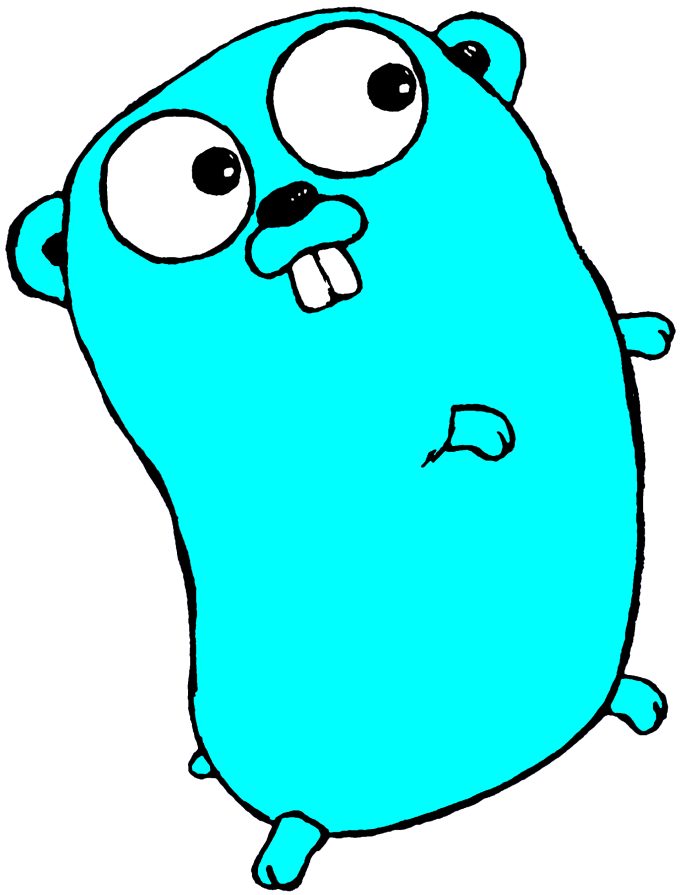
- To declare a variable that holds a slice of `myType` values, write `var myVar []myType`.
- Slices can be created using the `make` function.
  - The first argument to `make` should be the type of the slice.
  - The second argument should be the number of elements the slice holds.
- Slice element indexes start at `0`, so `mySlice[0]` refers to the first element, `mySlice[1]` to the second, and so on.
- To append an element to `mySlice`, use `mySlice = append(mySlice, newElement)`.
- A slice literal looks like this: `[]int{1, 2, 3}`

```
for index, element := range mySlice {
    fmt.Println(index, element)
}
```

https://is.gd/goex_slices

Exercise: Slices solution

```go
func main() {
    // Declare a variable that holds a slice of ints.
    var wholeNumbers []int
    // Create a new slice of 5 elements and assign it to the variable.
    wholeNumbers = make([]int, 5)
    // Assign 42 to the second element.
    wholeNumbers[1] = 42
    fmt.Printf("%#v\n", wholeNumbers) // => []int{0, 42, 0, 0, 0}

    // Make a 1-element slice of booleans.
    flags := make([]bool, 1)
    // Set the first and only element to true.
    flags[0] = true
    // Append the value false to the slice.
    flags = append(flags, false)
    // Append the value true to the slice.
    flags = append(flags, true)
    fmt.Printf("%#v\n", flags) // => []bool{true, false, true}

    // Create a slice with the strings "cat", "bat", and "rat".
    words := []string{"cat", "bat", "rat"}
    // For each element of the slice, print its index and value.
    for i, word := range words {
        fmt.Println(i, word)
    }
}
```



Maps

Maps

- Heard of “dictionaries” or “hashes”? Maps are Go’s equivalent.
- Like arrays and slices, maps store a collection of values.
- But arrays and slices use indexes to access their elements. Indexes can only be sequential integers.
- Maps use keys to store values under. A map’s keys can be *any* type you want!

```
ranks := make(map[string]int)
ranks["gold"] = 1
ranks["silver"] = 2
ranks["bronze"] = 3
fmt.Println(ranks["gold"], ranks["bronze"]) // => 1 3
```

Maps

- Though maps work differently than slices, the syntax is similar.
- You’re going to get some deja vu from the slices section.

Map types

Map type is written `map[keyType]valueType`.

```
var myArray [3]bool
var mySlice []bool
var myMap   map[string]bool
```

Zero value for maps

- The zero value for a map is `nil`.
- Unlike arrays.
- Just like slices.

```
var myMap map[int]string
if myMap == nil {
    fmt.Println("myMap is nil") // => myMap is nil
}
```

Maps and “make()”

Make a new map with the built-in `make` function.

```
var atomicNumbers map[int]string
atomicNumbers = make(map[int]string)
if atomicNumbers == nil {
    fmt.Println("atomicNumbers is nil") // doesn't run
}
fmt.Printf("%#v\n", atomicNumbers) // => map[int]string{}
```

Maps and “make()”

Could rewrite that using a short variable declaration:

```
atomicNumbers := make(map[int]string)
if atomicNumbers == nil {
    fmt.Println("atomicNumbers is nil") // doesn't run
}
fmt.Printf("%#v\n", atomicNumbers) // => map[int]string{}
```

Accessing map values

- Syntax to assign to set/get a map value is similar to that for arrays/slices.
- Just use a value of your specified key type instead of an integer index.

```
ranks := make(map[string]int)
ranks["gold"] = 1
ranks["silver"] = 2
ranks["bronze"] = 3
fmt.Println(ranks["gold"], ranks["bronze"]) // => 1 3
```

Map literals

- `fmt.Printf("%#v")` can be used to quickly view an entire map's contents.
- `Printf` outputs the values in **map literal** syntax.

```
ranks := make(map[string]int)
ranks["gold"] = 1
ranks["silver"] = 2
ranks["bronze"] = 3
fmt.Printf("%#v\n", ranks)
// => map[string]int{"bronze":3, "gold":1, "silver":2}
```

Map literals

- Map literals can be used to quickly initialize a map with several keys/values.
- No need for `make` when using a literal.

```
ranks := map[string]int{"gold": 1, "silver": 2, "bronze": 3}
fmt.Println(ranks["gold"])   // => 1
fmt.Println(ranks["silver"]) // => 2
fmt.Println(ranks["bronze"]) // => 3
```

Maps and “for … range”

- `for ... range` works with a map in much the same way as an array or slice.
- Provide two variables:
  - First will be assigned a key (instead of an integer index).
  - Second will be assigned the corresponding value.

```
scores := map[string]int{"Amy": 84, "Jose": 96, "Ben": 78}
for name, score := range scores {
    fmt.Println(name, score)
}
```

Output:

```
Amy 84
Jose 96
Ben 78
```

Maps and “for … range”

Want to process only the keys? Provide only one variable.

```
scores := map[string]int{"Amy": 84, "Jose": 96, "Ben": 78}
for name := range scores {
    fmt.Println(name)
}
```

Output:

```
Jose
Ben
Amy
```

Maps and “for … range”

Want to process only the values? Use the blank identifier for the first variable.

```
scores := map[string]int{"Amy": 84, "Jose": 96, "Ben": 78}
for _, score := range scores {
    fmt.Println(score)
}
```

Output:

```
78
84
96
```

Exercise: Maps

https://is.gd/goex_maps

Exercise: Maps

```
// Fill in the blanks so this program compiles and produces
// the output shown.
func main() {
    // Declare a variable that holds a map with strings for keys
    // and booleans for values.
    var isVowel ____
    // Call make to create the map.
    isVowel = ____
    // Assign true to the key "a", and false to the keys "b" and "c".
    ____ = true
    ____ = false
    ____ = false
    fmt.Printf("%#v\n", isVowel)
    // => map[string]bool{"a":true, "b":false, "c":false}


    // Make a map with ints as keys and strings as values. The 1 key
    // should have the value "H", 2 should have the value "He", and
    // 3 should have the value "Li".
    elements := ____
    // Print all the keys and corresponding values in the slice.
    // Order doesn't matter.
    for ____ := range elements {
        fmt.Println(atomicNumber, symbol)
    }
}
```

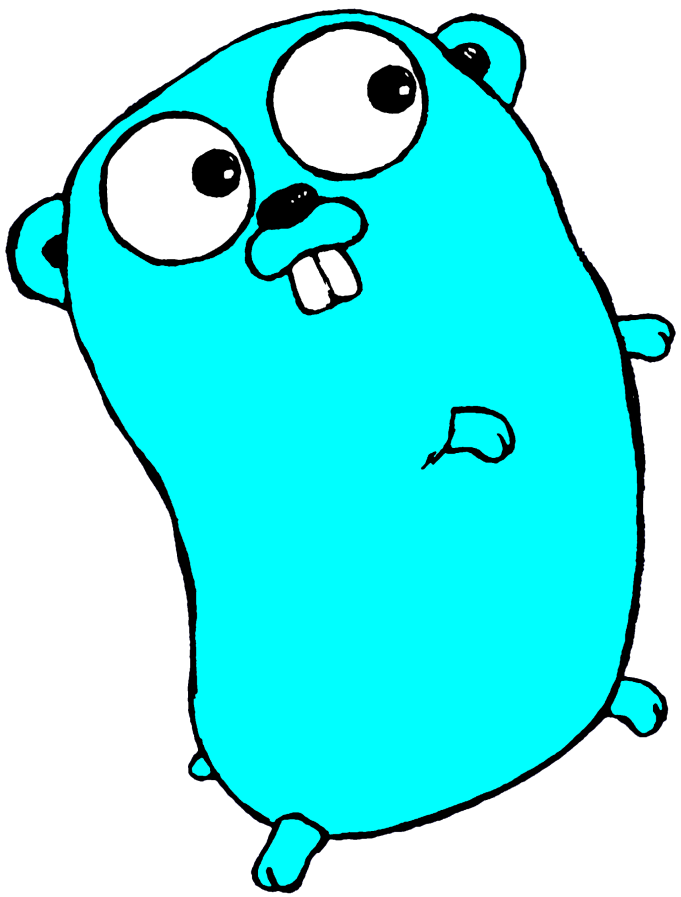
Exercise: Maps cheat sheet

```
var myMap map[keyType]valueType
myMap = make(map[keyType]valueType)
otherMap := map[int]string{1: "foo", 2: "bar"}
otherMap[3] = "baz"
for key, value := range otherMap {
    fmt.Println(key, value)
}
```

https://is.gd/goex_maps

Exercise: Maps solution

```
func main() {
    var isVowel map[string]bool
    isVowel = make(map[string]bool)
    isVowel["a"] = true
    isVowel["b"] = false
    isVowel["c"] = false
    fmt.Printf("%#v\n", isVowel)
    // => map[string]bool{"a":true, "b":false, "c":false}

    elements := map[int]string{1: "H", 2: "He", 3: "Li"}
    for atomicNumber, symbol := range elements {
        fmt.Println(atomicNumber, symbol)
    }
    // => 3 Li
    // => 1 H
    // => 2 He
}
```



Structs

Structs

Anonymous struct types…

```
var bucket struct {
    number float64
    word   string
    toggle bool
}
bucket.number = 3.14
bucket.word = "pie"
bucket.toggle = true
fmt.Println(bucket.number) // => 3.14
fmt.Println(bucket.word)   // => pie
fmt.Println(bucket.toggle) // => true
```

Custom types

`type myType` followed by an underlying type declares a new type.

```
type myType struct {
    number float64
    word   string
    toggle bool
}
```

Custom types

```
func main() {
    var bucket myType
    bucket.number = 3.14
    bucket.word = "pie"
    bucket.toggle = true
    fmt.Println(bucket.number) // => 3.14
    fmt.Println(bucket.word)   // => pie
    fmt.Println(bucket.toggle) // => true
}
```

Embedding structs is like inheriting fields

```
type Coordinates struct {
    Latitude  float64
    Longitude float64
}

type Landmark struct {
    Name string
    // An "anonymous field"
    // Has no name of its own, just a type
    Coordinates
}

func main() {
    var l Landmark
    l.Name = "The Googleplex"
    // Fields for "embedded struct" are "promoted"
    l.Latitude = 37.42
    l.Longitude = -122.08
    fmt.Println(l.Name, l.Latitude, l.Longitude)
    // => The Googleplex 37.42 -122.08
}
```

Exercise: Struct types

https://is.gd/goex_structs

Exercise: Struct types

```
type Subscriber struct {
    Name   string
    Rate   float64
    Active bool
}

type Employee struct {
    Name   string
    Salary float64
}

// YOUR CODE HERE:
// Define a struct type named Address that has Street, City, State,
// and PostalCode fields, each with a type of "string".
// Then embed the Address type within the Subscriber and Employee
// types using anonymous fields, so that the code in "main" will
// compile, run, and produce the output shown.

func main() {
    var subscriber Subscriber
    subscriber.Name = "Aman Singh"
    subscriber.Street = "123 Oak St"
    subscriber.City = "Omaha"
    subscriber.State = "NE"
    subscriber.PostalCode = "68111"
    fmt.Println("Name:", subscriber.Name)                // => Name: Aman Singh
    fmt.Println("Street:", subscriber.Street)            // => Street: 123 Oak St
    fmt.Println("City:", subscriber.City)                // => City: Omaha
    fmt.Println("State:", subscriber.State)              // => State: NE
    fmt.Println("Postal Code:", subscriber.PostalCode) // => Postal Code: 68111

    var employee Employee
    employee.Name = "Joy Carr"
    employee.Street = "456 Elm St"
    employee.City = "Portland"
    employee.State = "OR"
    employee.PostalCode = "97222"
    fmt.Println("Name:", employee.Name)                // => Name: Joy Carr
    fmt.Println("Street:", employee.Street)            // => Street: 456 Elm St
    fmt.Println("City:", employee.City)                // => City: Portland
    fmt.Println("State:", employee.State)              // => State: OR
    fmt.Println("Postal Code:", employee.PostalCode) // => Postal Code: 97222
}
```

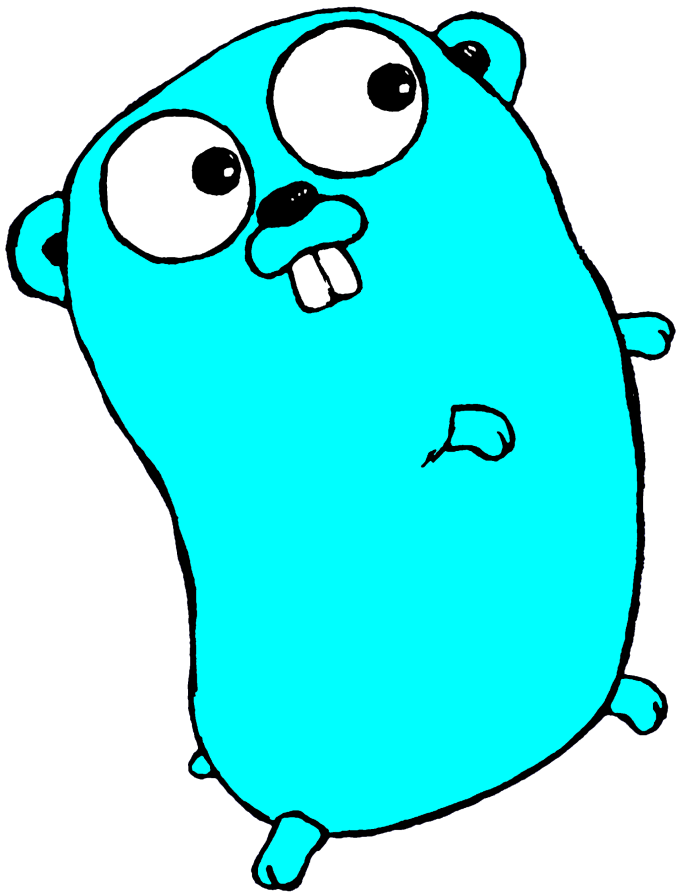
Exercise: Struct types cheat sheet

```
type Profile struct {
    Bio string
}
type User struct {
    Profile
}

var user User
user.Bio = "I teach Go workshops."
```

https://is.gd/goex_structs

Exercise: Struct types solution

```
type Subscriber struct {
    Name   string
    Rate   float64
    Active bool
    Address
}

type Employee struct {
    Name   string
    Salary float64
    Address
}

type Address struct {
    Street     string
    City       string
    State      string
    PostalCode string
}
```



Defined Types

Underlying basic types

A custom type can have an underlying basic type

```
type Liters float64
type Gallons float64

func main() {
    var carFuel Gallons
    var busFuel Liters
    // Defining a type defines a conversion
    // from the underlying type to the new type
    carFuel = Gallons(10.0)
    busFuel = Liters(240.0)
    fmt.Println(carFuel) // => 10
    fmt.Println(busFuel) // => 240
}
```

Methods

```
type MyType string

// Specify a "receiver parameter" within a function
// definition to make it a method. The receiver
// parameter's type will be the type the method
// gets defined on.
func (m MyType) sayHi() {
    fmt.Println("Hi")
}
```

Methods

```
type MyType string

func (m MyType) sayHi() {
    fmt.Println("Hi")
}

func main() {
    value := MyType("a MyType value")
    value.sayHi() // => Hi
    anotherValue := MyType("another value")
    anotherValue.sayHi() // => Hi
}
```

Receiver parameter acts like just another parameter

```
type MyType string

func (m MyType) sayHi() {
    fmt.Println("Hi from", m)
}

func main() {
    value := MyType("a MyType value")
    value.sayHi() // => Hi from a MyType value
    anotherValue := MyType("another value")
    anotherValue.sayHi() // => Hi from another value
}
```

Underlying type is *not* a superclass

The underlying type specifies how a type's data will be stored, so this is OK…

```
type MyType string

func (m MyType) sayHi() {
    fmt.Println("Hi from", m)
}

type MyType2 MyType

func main() {
    value2 := MyType2("a MyType2 value")
    fmt.Println(value2)
}
```

Underlying type is *not* a superclass

But a type does *not* inherit methods from its underlying type.

```
type MyType string

func (m MyType) sayHi() {
    fmt.Println("Hi from", m)
}

type MyType2 MyType

func main() {
    value2 := MyType2("a MyType2 value")
    value2.sayHi()
}
```

Compile error:

```
prog.go:15:8: value2.sayHi undefined (type MyType2 has no field or method
sayHi)
```

Underlying type is *not* a superclass

“Although Go has types and methods and allows an object-oriented style of programming, there is no type hierarchy.”

—https://golang.org/doc/faq

- There is no method inheritance!
- But there’s another way to get the same benefits…

Embedding structs is like inheriting fields

Remember how fields for an embedded struct get promoted to the outer struct?

```
type Coordinates struct {
    Latitude  float64
    Longitude float64
}

type Landmark struct {
    Name string
    Coordinates
}

func main() {
    var l Landmark
    l.Name = "The Googleplex"
    // Fields for "embedded struct" are "promoted"
    l.Latitude = 37.42
    l.Longitude = -122.08
    fmt.Println(l.Name, l.Latitude, l.Longitude)
    // => The Googleplex 37.42 -122.08
}
```

Promotion of embedded types' methods

Methods for an embedded type get promoted too!

```
func (c Coordinates) Location() string {
    return fmt.Sprintf("(%0.2f, %0.2f)",
        c.Latitude, c.Longitude)
}

func main() {
    var l Landmark
    l.Name = "The Googleplex"
    l.Latitude = 37.42
    l.Longitude = -122.08
    // Methods from embedded type are
    // promoted to outer type
    fmt.Println(l.Location())
    // => (37.42, -122.08)
}
```

Promotion of embedded types' methods

- Embed additional types to gain additional methods.
- You've heard "favor composition over inheritance"…
- Go implements that principle at the language level.

```
type Coordinates struct {
    Latitude  float64
    Longitude float64
}
func (c Coordinates) Location() string {
    return fmt.Sprintf("(%0.2f, %0.2f)",
        c.Latitude, c.Longitude)
}
type Landmark struct {
    Name string
    Coordinates
}
```

Exercise: Defined types

https://is.gd/goex_defined_types

Exercise: Defined types

```
// YOUR CODE HERE:
// Define a Rectangle struct type with Length and Width
// fields, each of which has a type of float64.

// YOUR CODE HERE:
// Define an Area method on the Rectangle type. It should
// accept no parameters (other than the receiver parameter).
// It should return a float64 value calculated by multiplying
// the receiver's Length by its Width.

// YOUR CODE HERE:
// Define a Perimeter method on the Rectangle type. It should
// accept no parameters. It should return a float64 value
// representing the receiver's perimeter (2 times its Length
// plus 2 times its Width).

func main() {
    // Once you've defined the above code correctly,
    // this code should compile and run.
    var myRectangle Rectangle
    myRectangle.Length = 2
    myRectangle.Width = 3
    fmt.Println("Area:", myRectangle.Area())            // => Area: 6
    fmt.Println("Perimeter:", myRectangle.Perimeter()) // => Perimeter: 10
}
```

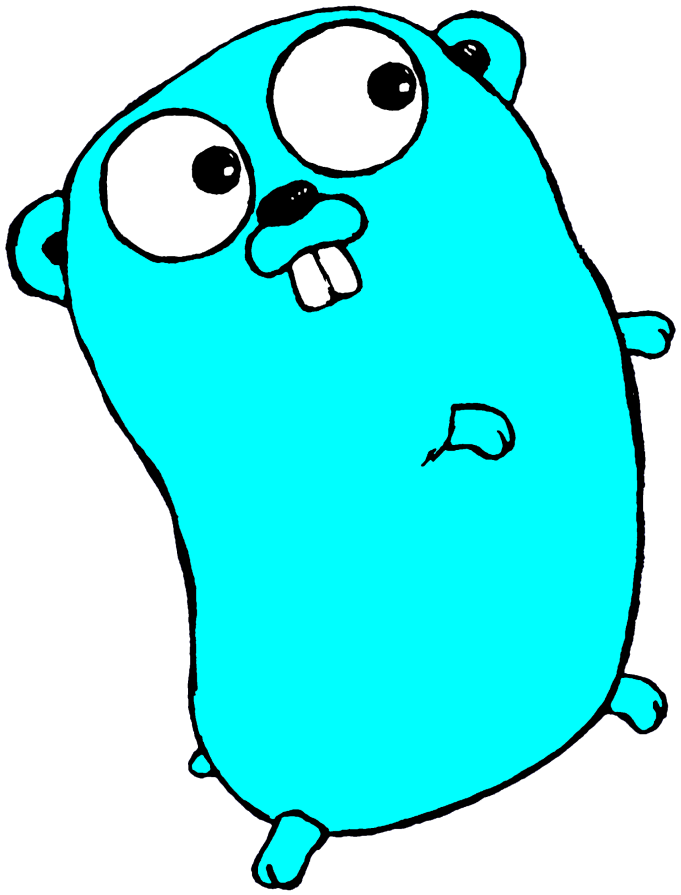
Exercise: Defined types cheat sheet

```
type User struct {
    Name string
}
func (u User) ShoutName() string {
    return strings.ToUpper(u.Name)
}
```

https://is.gd/goex_defined_types

Exercise: Defined types solution

```go
package main

import (
    "fmt"
)

type Rectangle struct {
    Length float64
    Width  float64
}

func (r Rectangle) Area() float64 {
    return r.Length * r.Width
}

func (r Rectangle) Perimeter() float64 {
    return (2 * r.Length) + (2 * r.Width)
}

func main() {
    var myRectangle Rectangle
    myRectangle.Length = 2
    myRectangle.Width = 3
    fmt.Println("Area:", myRectangle.Area())
    fmt.Println("Perimeter:", myRectangle.Perimeter())
}
```



Interfaces

Interfaces

A type with `Play` and `Stop` methods…

```
type TapePlayer struct {
    Batteries string
}
func (t TapePlayer) Play(song string) {
    fmt.Println("Playing", song)
}
func (t TapePlayer) Stop() {
    fmt.Println("Stopped!")
}
```

Interfaces

Another type with `Play` and `Stop` methods…

```
type TapeRecorder struct {
    Microphones int
}
func (t TapeRecorder) Play(song string) {
    fmt.Println("Playing", song)
}
func (t TapeRecorder) Record() {
    fmt.Println("Recording")
}
func (t TapeRecorder) Stop() {
    fmt.Println("Stopped!")
}
```

Interfaces

A function that accepts a `TapePlayer`...

```
func playList(device TapePlayer, songs []string) {
    for _, song := range songs {
        device.Play(song)
    }
    device.Stop()
}
```

Interfaces

```
func main() {
    mixtape := []string{"Jessie's Girl", "Whip It", "9 to 5"}
    var player TapePlayer
    playList(player, mixtape)
}
```

Output:

```
Playing Jessie's Girl
Playing Whip It
Playing 9 to 5
Stopped!
```

Interfaces

But don’t try to pass a `TapeRecorder` to `playList`!

```
func main() {
    mixtape := []string{"Jessie's Girl", "Whip It", "9 to 5"}
    var recorder TapeRecorder
    playList(recorder, mixtape)
}
```

Compile error:

```
prog.go:40:10: cannot use recorder (type TapeRecorder) as type TapePlayer
in argument to playList
```

Interfaces

Define a `Player` interface with the methods you want:

```
type Player interface {
    // Must have a Play method with
    // a single string parameter
    Play(string)
    // Must have a Stop method with
    // no parameters
    Stop()
}
```

- Notice we don't have to modify the `TapePlayer` or `TapeRecorder` type definitions!
- Any type with `Play(string)` and `Stop()` methods automatically *satisfies* this interface.

Interfaces

Modify the `playList` function to accept a value of the `Player` (interface) type:

```
func playList(device Player, songs []string) {
    for _, song := range songs {
        device.Play(song)
    }
    device.Stop()
}
```

Interfaces

Now, you can pass in a `TapePlayer` *or* a `TapeRecorder` (or any other type with `Play` and `Stop` methods)!

```
func main() {
    mixtape := []string{"Jessie's Girl", "Whip It", "9 to 5"}
    var player TapePlayer
    playList(player, mixtape)
    var recorder TapeRecorder
    playList(recorder, mixtape)
}
```

Interfaces

Output:

```
Playing Jessie's Girl
Playing Whip It
Playing 9 to 5
Stopped!
Playing Jessie's Girl
Playing Whip It
Playing 9 to 5
Stopped!
```

Type assertions

- If you have a value with an interface type, you can only call methods included in that interface.
- This is true even if the concrete value has that method!

Type assertions

```
func main() {
    // Even though we're passing in a TapeRecorder...
    TryOut(TapeRecorder{})
}

func TryOut(player Player) {
    player.Play("Test Track")
    player.Stop()
    // Player interface doesn't include this method!
    player.Record()
}
```

Compile error:

```
player.Record undefined (type Player has no field or method Record)
```

Type assertions

```
func main() {
    TryOut(TapeRecorder{})
}

func TryOut(player Player) {
    player.Play("Test Track")
    player.Stop()
    // Do a type assertion to get the concrete value back...
    recorder := player.(TapeRecorder)
    // Then you can call Record on that.
    recorder.Record()
}
```

Type assertions

Output:

```
Playing Test Track
Stopped!
Recording
```

Exercise: Interfaces

https://is.gd/goex_interfaces

Exercise: Interfaces

```
type Whistle string
func (w Whistle) MakeSound() {
    fmt.Println("Tweet!")
}

type Horn string
func (h Horn) MakeSound() {
    fmt.Println("Honk!")
}

type Robot string
func (r Robot) MakeSound() {
    fmt.Println("Beep Boop")
}
func (r Robot) Walk() {
    fmt.Println("Powering legs")
}

// YOUR CODE HERE:
// Define a NoiseMaker interface type, which the above
// Whistle, Horn, and Robot types will all satisfy.
// It should require one method, MakeSound, which has
// no parameters and no return values.

// YOUR CODE HERE:
// Define a Play function that accepts a parameter with
// the NoiseMaker interface. Play should call MakeSound
// on the parameter it receives.

func main() {
    // When the above code has been implemented
    // correctly, this code should run and produce
    // the output shown.
    Play(Whistle("Toyco Canary")) // => Tweet!
    Play(Horn("Toyco Blaster"))   // => Honk!
    Play(Robot("Botco Ambler"))   // => Beep Boop
}
```

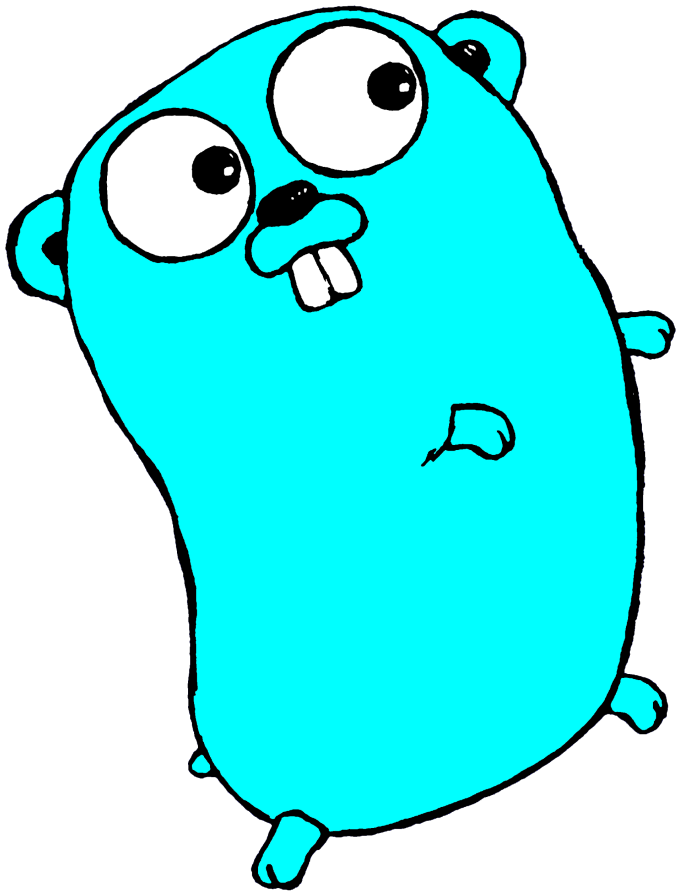
Exercise: Interfaces cheat sheet

```
type Pen string
func (p Pen) Write() {
    fmt.Println("writing stuff")
}

type WritingInstrument interface {
    Write()
}

func Test(w WritingInstrument) {
    w.Write()
}
```

https://is.gd/goex_interfaces

Exercise: Interfaces solution

```go
package main

import "fmt"

type Whistle string

func (w Whistle) MakeSound() {
    fmt.Println("Tweet!")
}

type Horn string

func (h Horn) MakeSound() {
    fmt.Println("Honk!")
}

type Robot string

func (r Robot) MakeSound() {
    fmt.Println("Beep Boop")
}
func (r Robot) Walk() {
    fmt.Println("Powering legs")
}

type NoiseMaker interface {
    MakeSound()
}

func play(n NoiseMaker) {
    n.MakeSound()
}

func main() {
    play(Whistle("Toyco Canary")) // => Tweet!
    play(Horn("Toyco Blaster"))   // => Honk!
    play(Robot("Botco Ambler"))   // => Beep Boop
}
```



Handling Errors

“defer”

It’s usually polite to end conversations with “goodbye”:

```
func Socialize() {
    fmt.Println("Hello!")
    fmt.Println("Nice weather, eh?")
    fmt.Println("Goodbye!")
}

func main() {
    Socialize()
}
```

Output:

```
Hello!
Nice weather, eh?
Goodbye!
```

“defer”

Write `defer` before a function call, and it will be “deferred” until enclosing function ends.

```
func Socialize() {
    // This call will be made when Socialize ends.
    defer fmt.Println("Goodbye!")
    fmt.Println("Hello!")
    fmt.Println("Nice weather, eh?")
}
```

Output:

```
Hello!
Nice weather, eh?
Goodbye!
```

“defer” calls made no matter what

```
func Socialize() error {
    // Deferred call is made even if Socialize
    // exits early (say, due to an error).
    defer fmt.Println("Goodbye!")
    fmt.Println("Hello!")
    return fmt.Errorf("I don't want to talk.")
    // The below code won't be run!
    fmt.Println("Nice weather, eh?")
    return nil
}

func main() {
    err := Socialize()
    if err != nil {
        log.Fatal(err)
    }
}
```

“defer” calls made no matter what

Output:

```
Hello!
Goodbye!
2019/04/22 11:22:29 I don't want to talk.
exit status 1
```

A (somewhat) more realistic example

- We need a function that prints the contents of a file, then closes it.
- If there's an error, it should be returned.
- But the file should be closed even if there's an error!

```
func main() {
    err := PrintLines("lorem_ipsum.txt")
    if err != nil {
        log.Fatal(err)
    }
}
```

A (somewhat) more realistic example

```
func PrintLines(fileName string) error {
    file, err := os.Open(fileName)
    if err != nil {
        return err
    }
    defer file.Close()
    scanner := bufio.NewScanner(file)
    for scanner.Scan() {
        fmt.Println(scanner.Text())
    }
    if scanner.Err() != nil {
        return scanner.Err()
    }
    return nil
}
```

“panic”

- `panic` usually signals an *unanticipated* error.
- This example is just to show its mechanics.

```
func Socialize() {
    fmt.Println("Hello!")
    panic("I need to get out of here!")
    // The below code won't be run!
    fmt.Println("Nice weather, eh?")
    fmt.Println("Goodbye!")
}
```

“panic”

Output:

```
Hello!
panic: I need to get out of here.

goroutine 1 [running]:
main.Socialize()
        /Users/jay/socialize4_panic.go:9 +0x79
main.main()
        /Users/jay/socialize4_panic.go:16 +0x20
exit status 2
```

“panic” and “defer”

```
func Socialize() {
    defer fmt.Println("Goodbye!")
    fmt.Println("Hello!")
    panic("I need to get out of here!")
    // The below code won't be run!
    fmt.Println("Nice weather, eh?")
}
```

“panic” and “defer”

Output:

```
Hello!
Goodbye!
panic: I need to get out of here!

goroutine 1 [running]:
main.Socialize()
        /Users/jay/socialize5_panic_defer.go:10 +0xd5
main.main()
        /Users/jay/socialize5_panic_defer.go:16 +0x20
exit status 2
```

“recover”

```
func CalmDown() {
    // Halt the panic.
    panicValue := recover()
    // Print value passed to panic().
    fmt.Println(panicValue)
}

func Socialize() {
    defer fmt.Println("Goodbye!")
    defer CalmDown()
    fmt.Println("Hello!")
    panic("I need to get out of here!")
    // The below code won't be run!
    fmt.Println("Nice weather, eh?")
}
```

“recover”

Output:

```
Hello!
I need to get out of here!
Goodbye!
```

- Deferred `CalmDown` prints the `panic` value.
- Deferred `Println` prints “Goodbye!”.

“panic” should not be used like an exception

I know of one place in the standard library that `panic` is used in normal program flow: in a recursive parsing function that panics to unwind the call stack after a parsing error. (The function then recovers and handles the error normally.)

“panic” should not be used like an exception

Generally, `panic` should be used only to indicate “impossible” situations:

```
func awardPrize() {
    doorNumber := rand.Intn(3) + 1
    if doorNumber == 1 {
        fmt.Println("You win a cruise!")
    } else if doorNumber == 2 {
        fmt.Println("You win a car!")
    } else if doorNumber == 3 {
        fmt.Println("You win a goat!")
    } else {
        // This should never happen.
        panic("invalid door number")
    }
}
```

“panic” should not be used like an exception

- If you know an error could happen, use normal control flow statements to handle it.
- Google “golang errors are values” (which should take you to `https://blog.golang.org/errors-are-values`) for some tips on making error handling more pleasant.

Exercise: Handling errors

`https://is.gd/goex_recovery`

Exercise: Handling errors

```go
type Refrigerator struct {
    Brand string
}

type Food string

func (r Refrigerator) Open() {
    fmt.Println("Opening refrigerator")
}
func (r Refrigerator) Close() {
    fmt.Println("Closing refrigerator")
}
func (r Refrigerator) FindFood(food string) (Food, error) {
    // Food storage not implemented yet; always return error!
    // Note: don't change FindFood as part of this exercise!
    return Food(""), fmt.Errorf("%s not found", food)
}

// YOUR CODE HERE:
// Modify the code in the Eat function so that fridge.Close will
// always be called at the end, even if fridge.FindFood returns
// an error. Once you've figured the solution out, your changes
// will actually be quite small! Note: it wouldn't be appropriate
// to use either "panic" or "recover" in this exercise; we won't
// be using either one.
func Eat(fridge Refrigerator) error {
    fridge.Open()
    food, err := fridge.FindFood("bananas")
    if err != nil {
        return err
    }
    fmt.Println("Eating", food)
    fridge.Close()
    return nil
}

// CURRENT OUTPUT:
// Opening refrigerator
// bananas not found
// DESIRED OUTPUT:
// Opening refrigerator
// Closing refrigerator
// bananas not found
func main() {
    var fridge Refrigerator
    err := Eat(fridge)
    if err != nil {
        fmt.Println(err)
    }
}
```

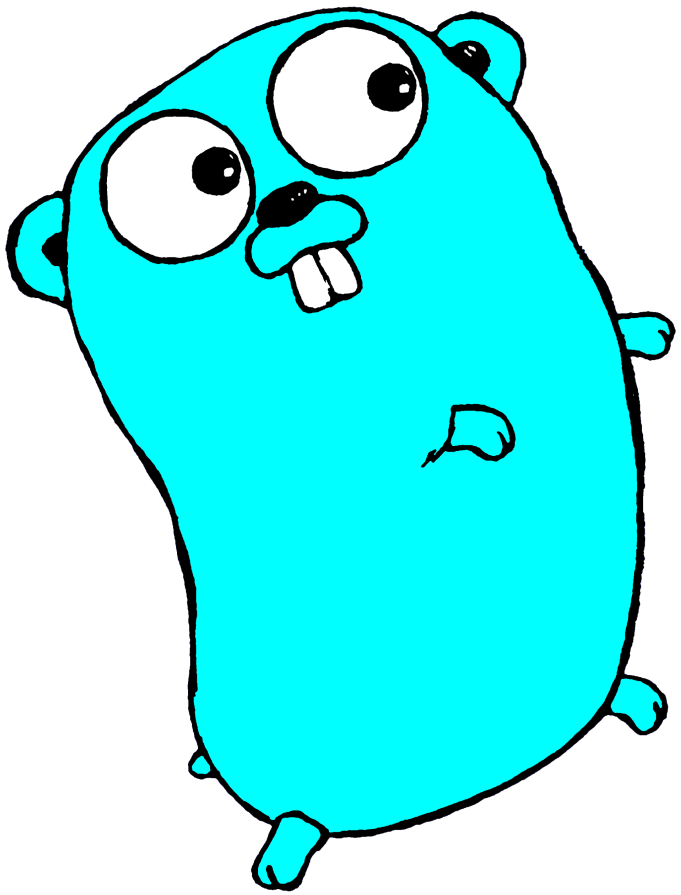
Exercise: Handling errors cheat sheet

```
func Camp() error {
    var fire Fire
    fire.Light()
    defer fire.Extinguish()
    return fmt.Errorf("spotted a bear")
    fmt.Println("Toasting marshmallows")
    return nil
}
```

https://is.gd/goex_recovery

Exercise: Handling errors solution

```
type Refrigerator struct {
    Brand string
}
type Food string
func (r Refrigerator) Open() {
    fmt.Println("Opening refrigerator")
}
func (r Refrigerator) Close() {
    fmt.Println("Closing refrigerator")
}
func (r Refrigerator) FindFood(food string) (Food, error) {
    // Food storage not implemented yet; always return error!
    return Food(""), fmt.Errorf("%s not found", food)
}
func Eat(fridge Refrigerator) error {
    fridge.Open()
    defer fridge.Close()
    food, err := fridge.FindFood("bananas")
    if err != nil {
        return err
    }
    fmt.Println("Eating", food)
    return nil
}
func main() {
    var fridge Refrigerator
    err := Eat(fridge)
    if err != nil {
        fmt.Println(err)
    }
}
```



Concurrency

A non-concurrent program

```
// responseSize retrieves "url" and prints
// the response length in bytes.
func responseSize(url string) {
    fmt.Println("Getting", url)
    // Note: errors ignored with _!
    response, _ := http.Get(url)
    defer response.Body.Close()
    body, _ := ioutil.ReadAll(response.Body)
    fmt.Println(len(body))
}
```

A non-concurrent program

```
func main() {
    // Note the time we started.
    start := time.Now()
    responseSize("https://example.com/")
    responseSize("https://golang.org/")
    responseSize("https://golang.org/doc")
    // Print how long everything took.
    fmt.Println(time.Since(start).Seconds(), "seconds")
}
```

A non-concurrent program

Output:

```
Getting https://example.com/
1270
Getting https://golang.org/
8158
Getting https://golang.org/doc
12558
1.5341211000000001 seconds
```

If only we could make additional calls to `responseSize` while we were waiting for HTTP responses…

Goroutines

- `responseSize` function unchanged.
- Just add `go` keyword before each call to it.

```
func main() {
    start := time.Now()
    go responseSize("https://example.com/")
    go responseSize("https://golang.org/")
    go responseSize("https://golang.org/doc")
    fmt.Println(time.Since(start).Seconds())
}
```

Goroutines

Output:

```
3.1e-06
```

- Run time so brief the duration is printed in scientific notation.
- None of the `responseSize` goroutines get to even request their URL.
- Problem is, `main` goroutine exits, ending the program, without waiting for `responseSize` goroutines.

Channels

- Modify `responseSize` to accept a “channel” as a parameter.

```
// A channel type is written as "chan" followed by data type.
func responseSize(url string, channel chan int) {
    fmt.Println("Getting", url)              // Unchanged
    response, _ := http.Get(url)             // Unchanged
    defer response.Body.Close()              // Unchanged
    body, _ := ioutil.ReadAll(response.Body) // Unchanged
    // Send body length value via channel.
    channel <- len(body)
}
```

Channels

```
func main() {
    start := time.Now() // Unchanged
    // Make a channel to carry ints.
    sizes := make(chan int)
    // Pass channel to each call to responseSize.
    go responseSize("https://example.com/", sizes)
    go responseSize("https://golang.org/", sizes)
    go responseSize("https://golang.org/doc", sizes)
    // Read and print values from channel.
    fmt.Println(<-sizes)
    fmt.Println(<-sizes)
    fmt.Println(<-sizes)
    fmt.Println(time.Since(start).Seconds()) // Unchanged
}
```

Channels

```
Getting https://golang.org/doc
Getting https://golang.org/
Getting https://example.com/
1270
8158
12558
0.695384291
```

- Finishes in half the time of the original! (YMMV.)
- The channel accomplishes two things:
  - Channel reads cause `main` goroutine to block until `responseSize` goroutines send, so they have time to finish before program ends.
  - The channel transmits data from the `responseSize` goroutines back to the `main` goroutine.

Exercise: Goroutines and channels

https://is.gd/goex_goroutines

Exercise: Goroutines and channels

```
// This program should call the "repeat" function twice, using two
// separate goroutines. The first goroutine should print the string
// "x" repeatedly, and the second goroutine should print "y"
// repeatedly. You'll also need to create a channel that carries
// boolean values to pass to "repeat", so the goroutine can signal
// when it's done.
//
// Output will vary, but here's one possible result:
// yyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyxxxxxxxxxx
//
// Replace the blanks ("____") in the code so the program will
// compile and run.
package main

import (
    "fmt"
)

// repeat prints a string multiple times, then writes "true" to the
// provided channel to signal it's done.
func repeat(s string, channel ____ bool) {
    for i := 0; i < 30; i++ {
        fmt.Print(s)
    }
    channel ____ true
}

func main() {
    channel := ____(chan bool)
    ____ repeat("x", channel)
    ____ repeat("y", channel)
    <-channel
    <-channel
}
```

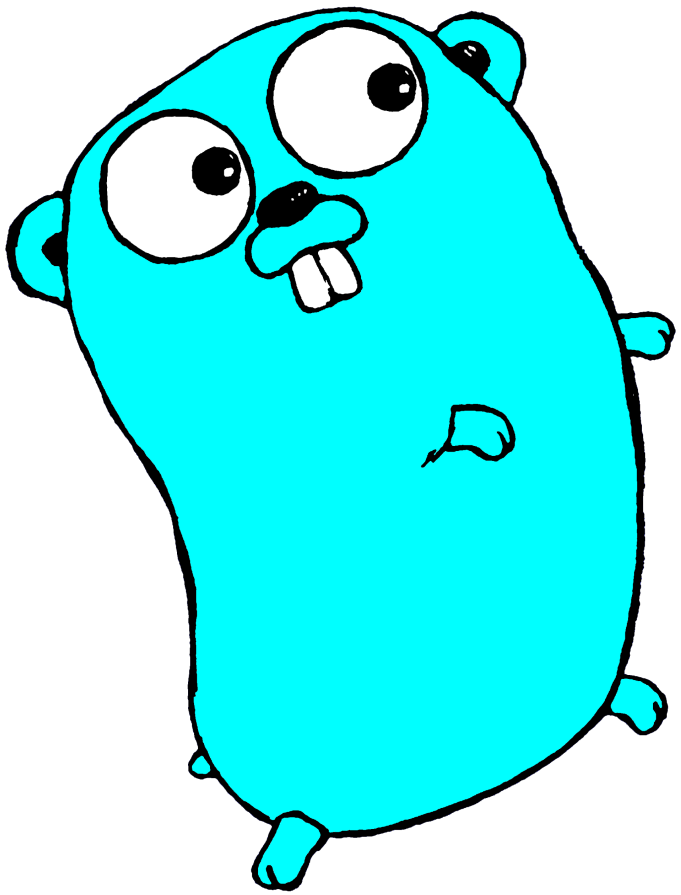
Exercise: Goroutines and channels cheat sheet

```
func myFunc(channel chan int) {
    channel <- 42
}

func main() {
    channel := make(chan int)
    go myFunc(channel)
    go myFunc(channel)
    fmt.Println(<-channel)
    fmt.Println(<-channel)
}
```

https://is.gd/goex_goroutines

Exercise: Goroutines and channels solution

```
package main

import "fmt"

func repeat(s string, channel chan bool) {
    for i := 0; i < 30; i++ {
        fmt.Print(s)
    }
    channel <- true
}

func main() {
    channel := make(chan bool)
    go repeat("x", channel)
    go repeat("y", channel)
    <-channel
    <-channel
}
```



Where to Go Next

“go test”

Suppose we’re not certain this function works correctly…

```
func JoinWithCommas(phrases []string) string {
    if len(phrases) == 2 {
        return phrases[0] + " and " + phrases[1]
    } else {
        result := strings.Join(phrases[:len(phrases)-1], ", ")
        result += ", and "
        result += phrases[len(phrases)-1]
        return result
    }
}
```

“go test”

We can use the `testing` package to write a test function…

```
import (
    "fmt"
    "testing"
)

func TestTwoElements(t *testing.T) {
    list := []string{"apple", "orange"}
    want := "apple and orange"
    got := JoinWithCommas(list)
    if got != want {
        t.Error(errorString(list, got, want))
    }
}
```

“go test”

Add tests for all the functionality we want…

```
func TestOneElement(t *testing.T) {
    list := []string{"apple"}
    want := "apple"
    got := JoinWithCommas(list)
    if got != want {
        t.Error(errorString(list, got, want))
    }
}
func TestThreeElements(t *testing.T) {
    list := []string{"apple", "orange", "pear"}
    want := "apple, orange, and pear"
    got := JoinWithCommas(list)
    if got != want {
        t.Error(errorString(list, got, want))
    }
}

func errorString(list []string, got string, want string) string {
    return fmt.Sprintf("JoinWithCommas(%#v) = \"%s\", want \"%s\"",
         list, got, want)
}
```

“go test”

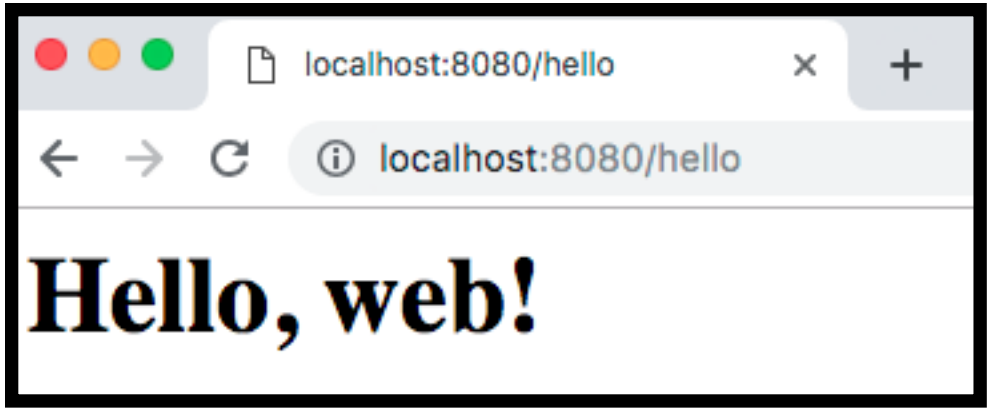
Then just run `go test` and get a summary of the tests that failed!

```
$ go test github.com/headfirstgo/prose
--- FAIL: TestOneElement (0.00s)
lists_test.go:13: JoinWithCommas([]string{"apple"}) = ", and apple", want
"apple"
FAIL
FAIL    github.com/headfirstgo/prose    0.006s
```

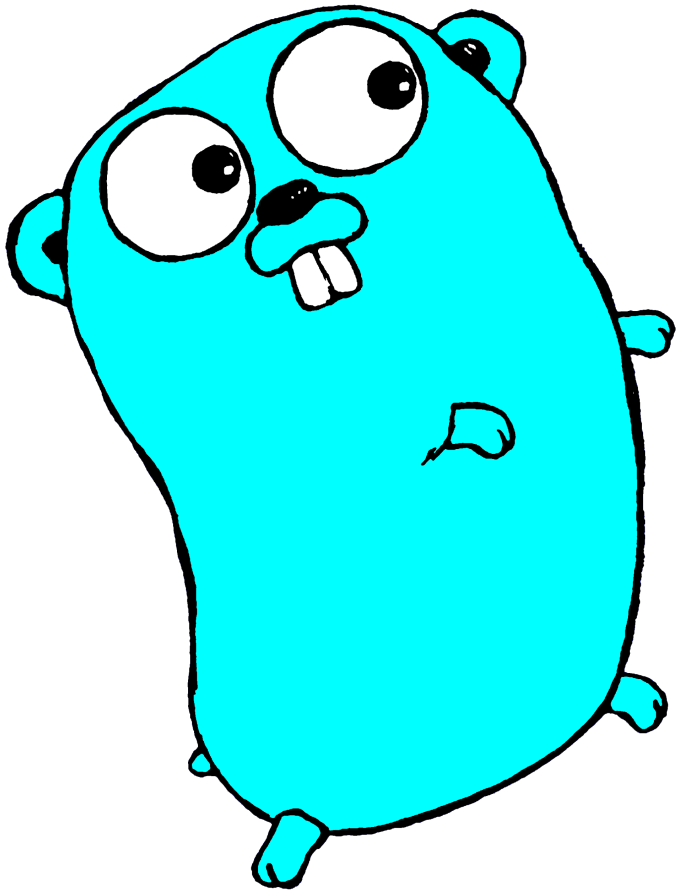
Web development

The standard library includes the `net/http` package:

```
package main

import "net/http"

func helloHandler(writer http.ResponseWriter, request *http.Request) {
    // Note: unhandled error!
    writer.Write([]byte("<h1>Hello, web!</h1>"))
}

func main() {
    http.HandleFunc("/hello", helloHandler)
    // Note: unhandled error!
    http.ListenAndServe("localhost:8080",
        nil)
}
```



Go Gopher



Other resources

- Go Tour: `https://tour.golang.org`
- Go Playground: `https://play.golang.org`
- Head First Go: `https://headfirstgo.com`