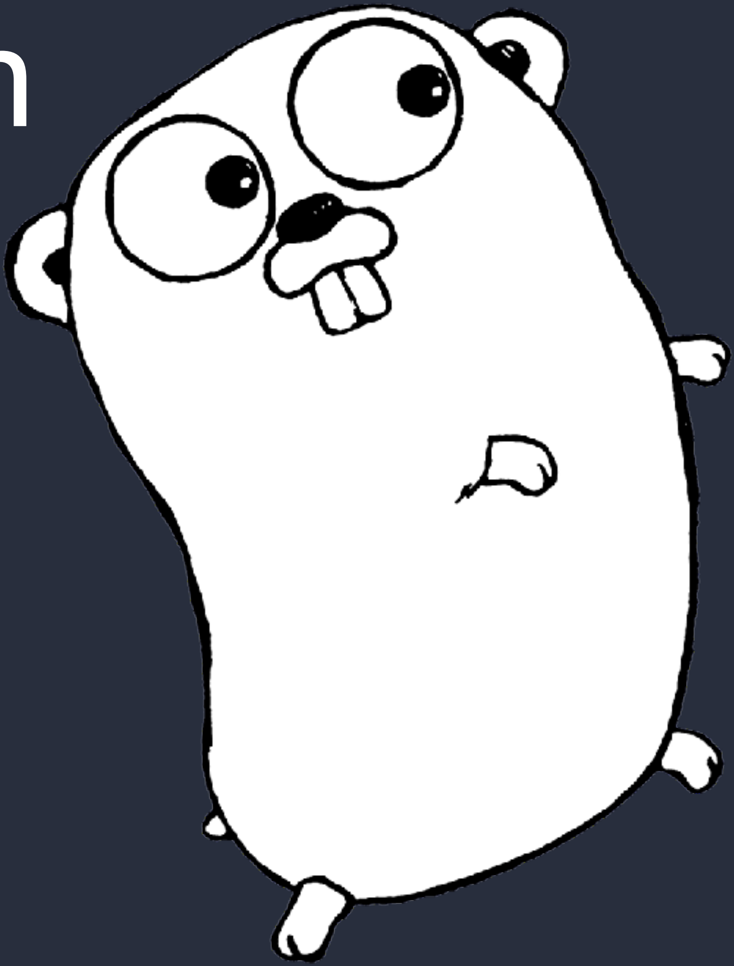
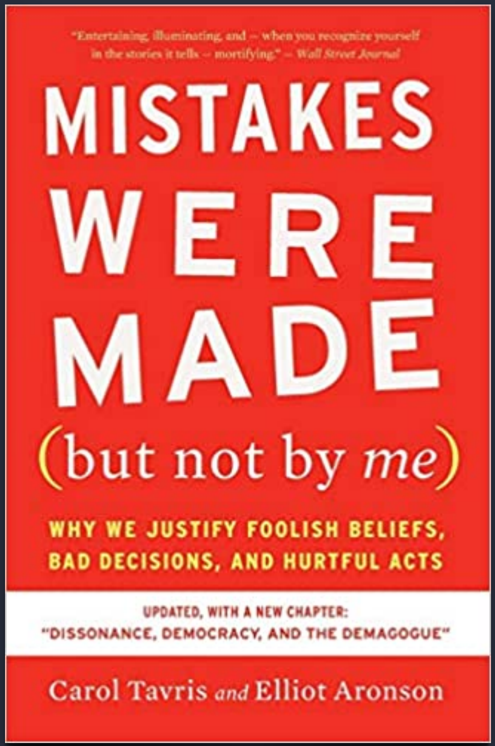
Go Version





"Entertaining, illuminating, and – when you recognize yourself in the stories it tells – mortifying." – *Wall Street Journal*

MISTAKES WERE MADE (but not by *me*)

WHY WE JUSTIFY FOOLISH BELIEFS, BAD DECISIONS, AND HURTFUL ACTS

UPDATED, WITH A NEW CHAPTER: "DISSONANCE, DEMOCRACY, AND THE DEMAGOGUE"

Carol Tavris *and* Elliot Aronson

MESOSPHERE

D2IQ

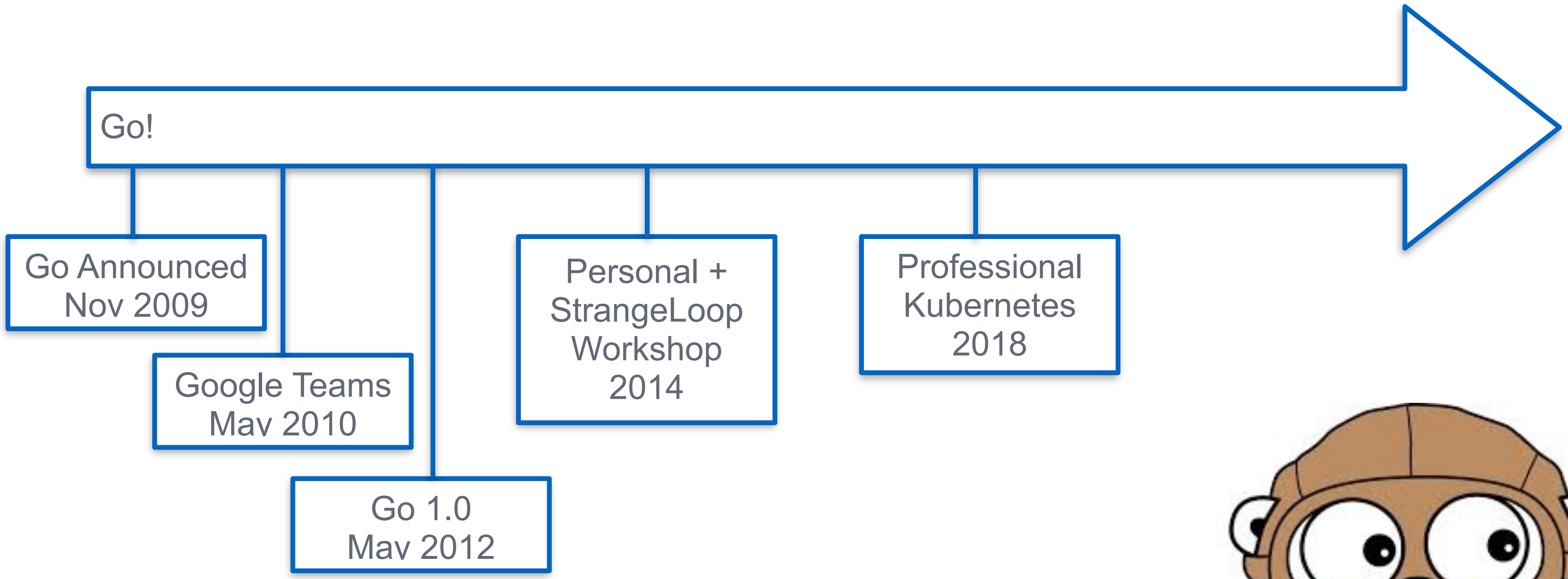
Ken Sipe

Distribute Application Engineer

Apache Mesos Contributor, Kubernetes Commiter
Apache Committer Myriad, Open DCOS
Developer: Embedded, C++, Java, Groovy, Grails, C#, GoLang

@KenSipe

Experience with Go

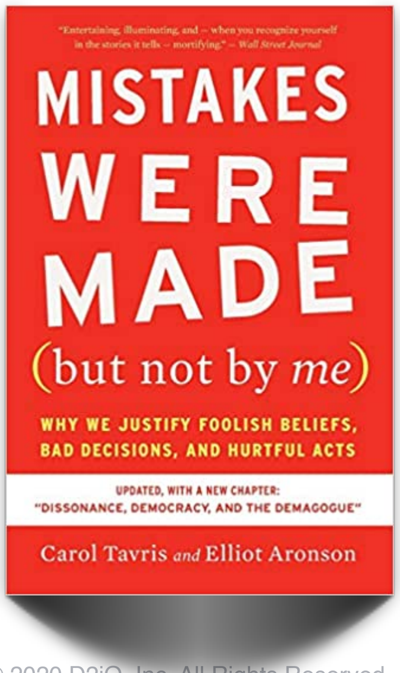

Go!
Go Announced Nov 2009
Google Teams May 2010
Go 1.0 May 2012
Personal + StrangeLoop Workshop 2014
Professional Kubernetes 2018

Agenda

- The Go Way
- Navel Gazing for Lint
- Common Mistakes, Go!

Mistakes were made

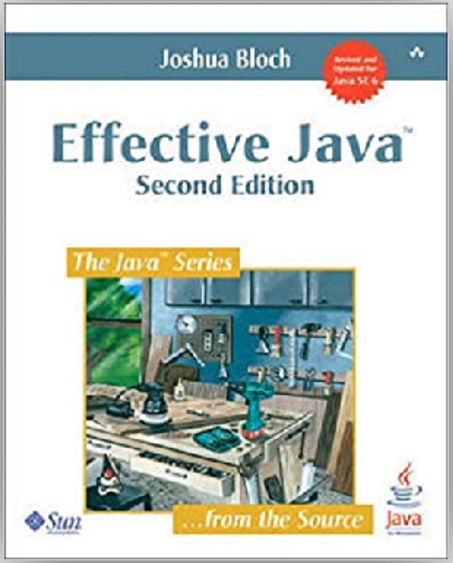
"**Mistakes were made**" is an expression that is commonly used as a rhetorical device, whereby a speaker acknowledges that a situation was handled poorly or inappropriately but seeks to evade any direct admission or accusation of responsibility by not specifying the person who made the mistakes.


"Entertaining, illuminating, and – when you recognize yourself in the stories it tells – mortifying." – Wall Street Journal
MISTAKES WERE MADE
(but not by me)
WHY WE JUSTIFY FOOLISH BELIEFS, BAD DECISIONS, AND HURTFUL ACTS
UPDATED, WITH A NEW CHAPTER: "DISSONANCE, DEMOCRACY, AND THE DEMAGOGUE"
Carol Tavris and Elliot Aronson

Mistakes Were Made (But Not by Me) Key Idea #1:
Instead of admitting our mistakes, we tend to justify them.

Each Language Has Challenges

- Reflection (primitive vs object)
- Equals contract (reflexive, symmetric, transitive, consistent)
- Exceptions (runtime, checked, abstractions, handling)
- Serialization (basically write your own)

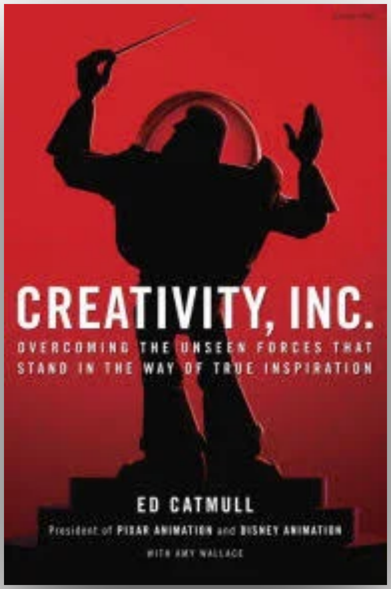


Mistakes have Value

“**Mistakes aren’t a necessary evil**. They aren’t evil at all.

They are an inevitable consequence of doing something new (and, as such, **should be seen as valuable**; without them, we’d have no originality)

— Ed Catmull


CREATIVITY, INC.
OVERCOMING THE UNSEEN FORCES THAT STAND IN THE WAY OF TRUE INSPIRATION
ED CATMULL
President of PIXAR ANIMATION and DISNEY ANIMATION
WITH AMY WALLACE

Golang is opinionated

- format is defined

  go fmt <src>
- no semi-colons
- brackets are one way {

}

  the way God intended them :)

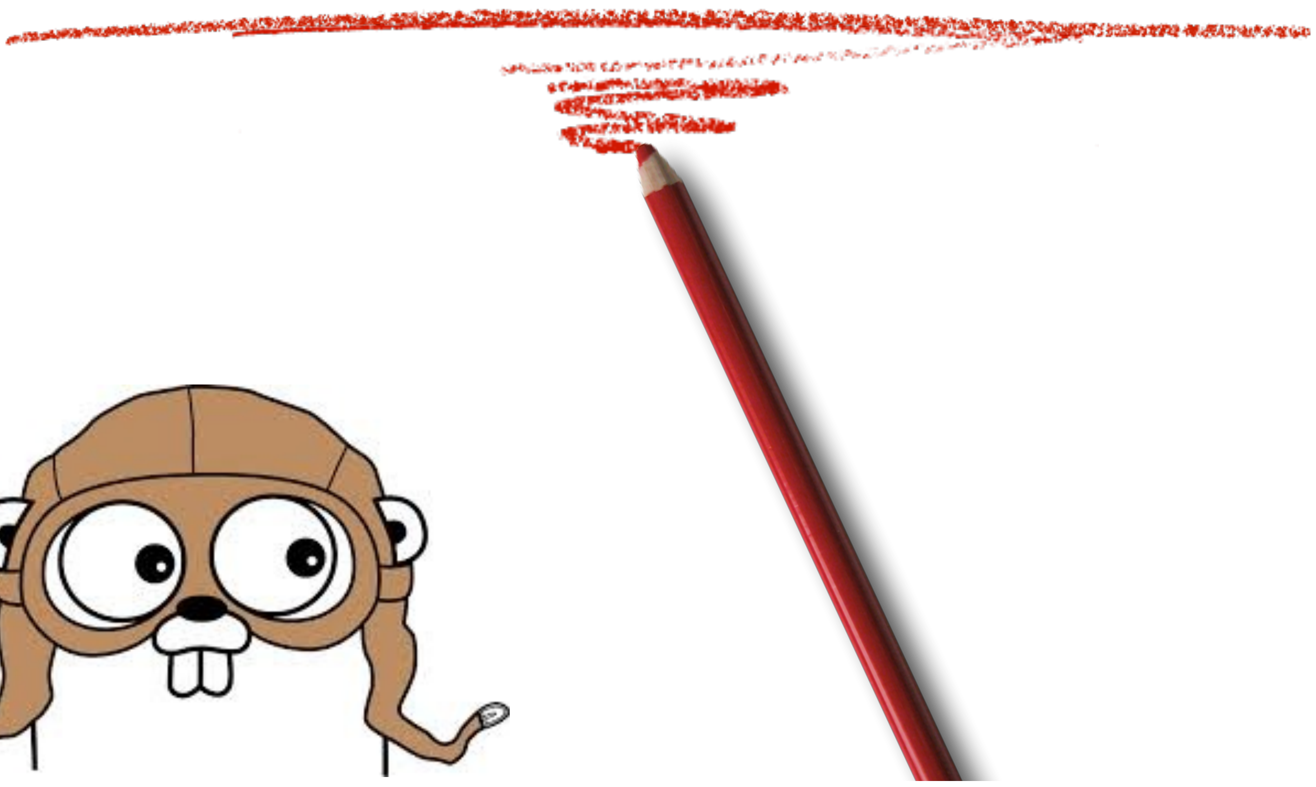
Beyond Effective Go

https://golang.org/doc/effective_go.html

- Too Many Yaml encoder/decoder
- Poorly named packages
- Poorly named interfaces, structs
- Poor combination of packages + exports
- Poor management of error handling (usually ignored)
- Versioning

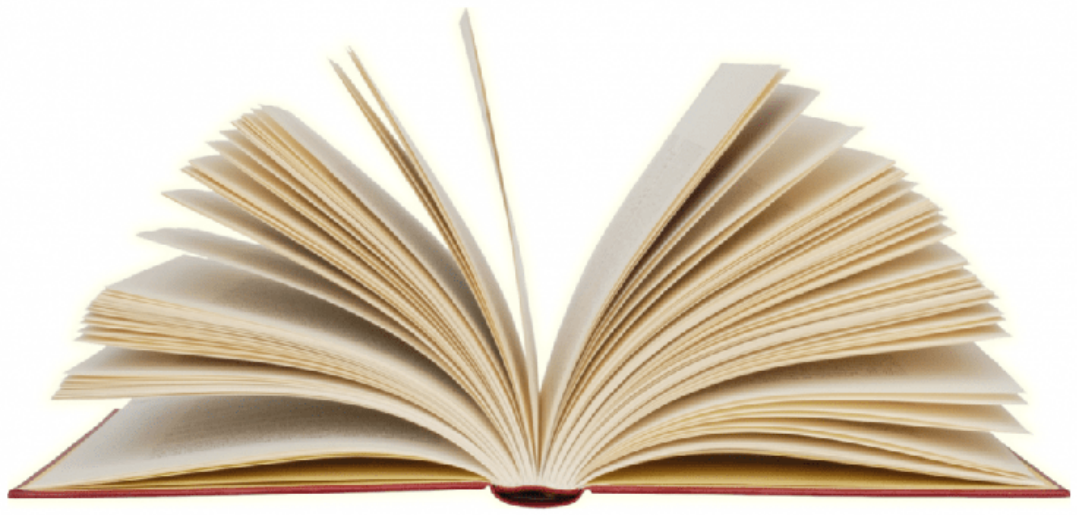
Most of my common issues are not with the language

The Go Way

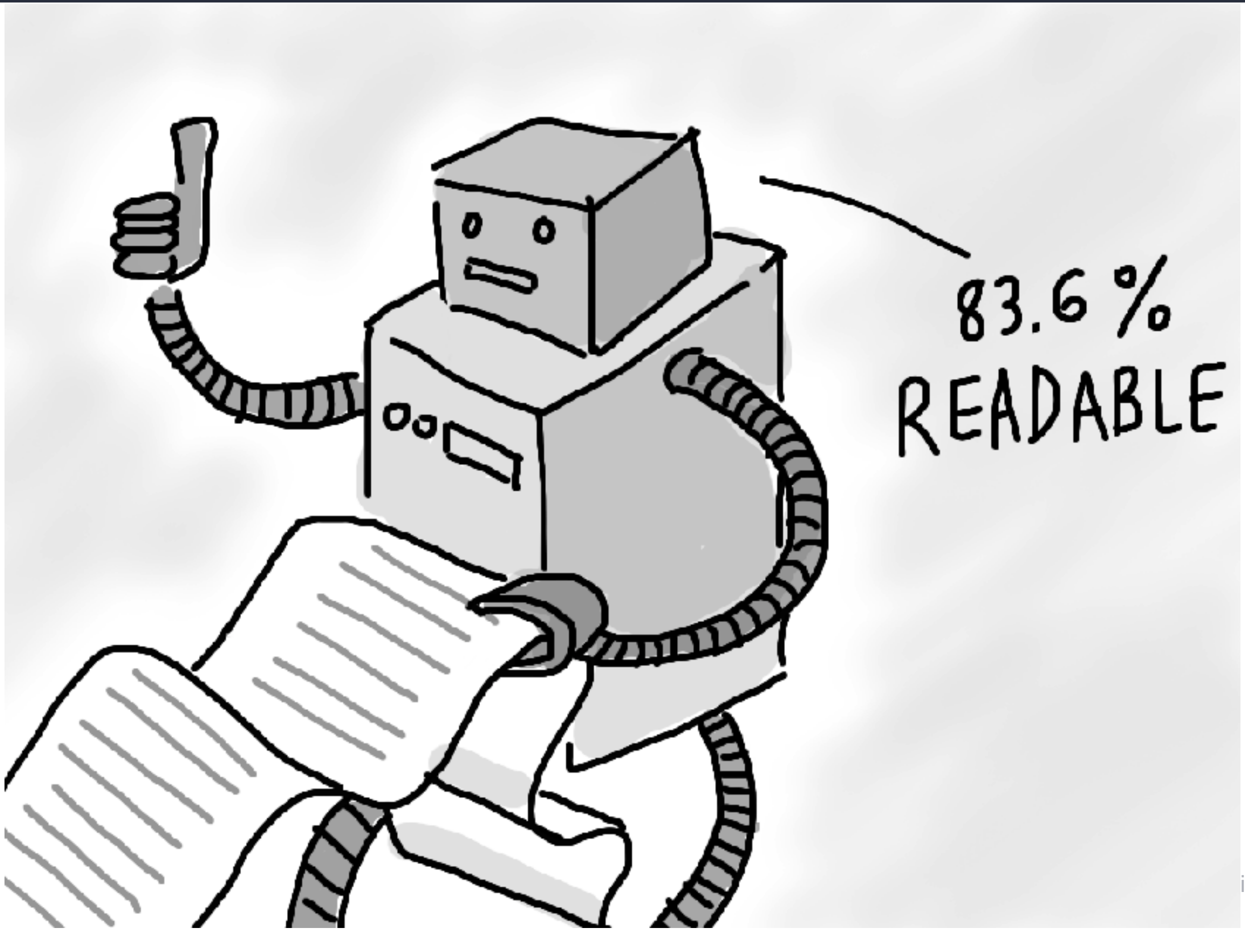


Favorite Part of Go

READABILITY



Favorite Part of Go



Good Names

- Consistent (easy to guess)
- Short
- Accurate (easy to understand)

https://talks.golang.org/2014/names.slide#1

Rule of Thumb

- The **greater the distance** between a name's declaration and its use, the **longer the name** should be.

- The package name should be used as meaningful part of understanding the name. (use of package for use of declaration uses it has the potential for greater "distance".

Not Winning

```go
func RuneCount(buffer []byte) int {
    runeCount := 0
    for index := 0; index < len(buffer); {
        if buffer[index] < RuneSelf {
            index++
        } else {
            _, size := DecodeRune(buffer[index:])
            index += size
        }
        runeCount++
    }
    return runeCount
}
```

Winning

```go
func RuneCount(b []byte) int {
    count := 0
    for i := 0; i < len(b); {
        if b[i] < RuneSelf {
            i++
        } else {
            _, n := DecodeRune(b[i:])
            i += n
        }
        count++
    }
    return count
}
```

Parameters

```go
func AfterFunc(d Duration, f func()) *Timer

func Escape(w io.Writer, s []byte)
```

Where the types are more ambiguous, the names may provide documentation

```go
func Unix(sec, nsec int64) Time

func HasPrefix(s, prefix []byte) bool
```

Return Values

Return values on exported functions mainly be named for documentation purposes.

```
func Copy(dst Writer, src Reader) (written int64, err error)

func ScanBytes(data []byte, atEOF bool) (advance int, token []byte, err error)
```

Receivers

- By convention, one or two characters that reflect the receiver type
- Receiver names should be consistent across a type's methods
- No receiver name if not used

```go
func (b *Buffer) Read(p []byte) (n int, err error)

func (sh serverHandler) ServeHTTP(rw ResponseWriter, req *Request)

func (r Rectangle) Size() Point

func (Rectangle) Execute()
```

Exported Package-Level Names

- **Exported names are qualified by their package names**

```
bytes.Buffer and strings.Reader

not

bytes.ByteBuffer and strings.StringReader
```

```
// new what?  client
client.New()
```

Choose package names that lend meaning to the names they export.
Steer clear of `util`, `common`, and the like.

https://blog.golang.org/package-names

Errors

- Error **types** should be form of FooError
- Error **values** should be form ErrFoo

```go
// error type
type ExitError struct {
    ...
}

// Error value
var ErrFormat = errors.New("image: unknown format")
```

Unspoken Rules for Go

- “Happy Path” flow is de-dented to the left
- Handle unexpected cases and errors early and return often
- Refactor until true :)

Unspoken Rules for Go

```go
// not winning
func things(x int) someType {
    if x > 2 {
        return 100
    } else {
        return 200
    }
}

// winning
func things(x int) someType {
    if x > 2 {
        return 100
    }
    return 200
}
```

Unspoken Rules for Go - Example 2

```go
// Handling Happy case first –
// leading to nested if checks...
func example() error {
     err := somethingThatReturnsError()
     if err == nil {
        //Happy processing
        err = somethingElseThatReturnsError()
        if err == nil {
           //More Happy processing
        } else {
           return err
        }
     } else {
        return err
     }
}
```

```go
func ABetterExample() error {
     err := somethingThatReturnsError()
     if err != nil {
        return err
     }

     // Happy processing
     err = somethingElseThatReturnsError()
     if err != nil {
        return err
     }
     // More Happy processing
}
```

Project Setup

/
/cmd
/internal
/pkg
/test

/api
/web
/hack
/tools

- Think of your Go project as being “in” the src
- No “/src”

$GOPATH/src/{repo_namespace}/{project}

{repo_namespace}/{project} == module

$GOPATH/src/sigs.k8s.io/kind

https://github.com/golang-standards/project-layout

Organizing Imports

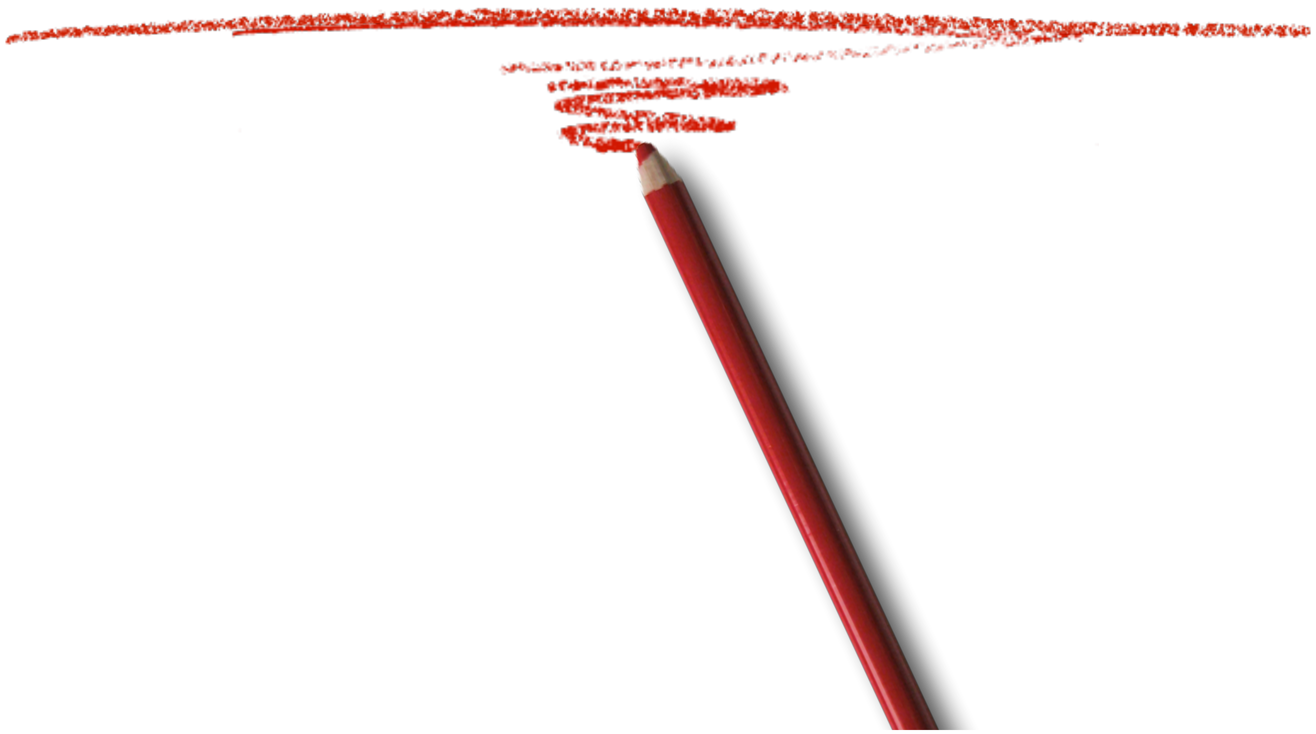
```go
import (
    "errors"
    "fmt"
    "sort"

    "github.com/spf13/cobra"
    "github.com/thoas/go-funk"
    "github.com/xlab/treeprint"

    "github.com/kudobuilder/kudo/pkg/kudoctl/clog"
    "github.com/kudobuilder/kudo/pkg/kudoctl/env"
)
```

Demo: ex2-string-array

https://github.com/codementor/go-mistakes

1. Project Setup
2. Run go run cmd/wman/main.go meetup
3. Run string_test.go::TestReverse

Navel Gazing for Lint



Lint

linter is a tool that analyzes source code to flag programming errors, bugs, stylistic errors, and suspicious constructs.

- Code Formatting
- Code Complexity
- Style and Patterns Checking
- Bugs
- Unused Code
- Performance
- Reports
- Misc

https://github.com/golangci/awesome-go-linters

Lots of Great Go Linters

staticcheck	go vet on steroids, applying a ton of static analysis checks
gocyclo	Computes and checks the cyclomatic complexity of functions
lll	Line length linter, used to enforce line length in files
misspell	Finds commonly misspelled English words
structcheck	Find unused global variables and constants
unparam	Report unused function parameters
errcheck	Errcheck is a program for checking for unchecked errors in Go programs

golangci-lint

“One linter to rule them all, One linter to find them, One linter to bring them all, and in the code base bind them”

golangci-lint

```
go get github.com/golangci/golangci-lint/cmd/golangci-lint@v1.31.0

brew install golangci/tap/golangci-lint
```

Configuration

{project}/.golangci.yml

```yaml
linters:
  auto-fix: false
  enable:
    - errcheck
    - goimports
    - golint
    - gosec
    - misspell
    - scopelint
    - unconvert
    - unparam
    - nakedret
    - gocyclo
    - dupl
    - goconst
    - lll
    - stylecheck
    - varcheck
    - deadcode
    - structcheck
    - infecting
    - goconst
    - staticcheck
    - unused
```

https://golangci-lint.run/usage/configuration/

deadcode

```
pkg/string/cmd/lint.go:16:6: `newLintCmd` is unused (deadcode)
func newLintCmd() *cobra.Command {
     ^
pkg/string/cmd/lint.go:42:6: `check` is unused (deadcode)
func check(test bool) bool {
```

- Use or delete
- Or Export (`check` -> `Check`)

Demo: ex3-lint-1 AND ex4-lint-2

https://github.com/codementor/go-mistakes

1. `make lint`

errcheck

```
pkg/string/cmd/dogyears.go:45:7: Error return value of `strconv.Atoi` is not checked (errcheck)
        age, _ := strconv.Atoi(a)
           ^
```

```yaml
linters-settings:
  errcheck:
    check-type-assertions: true
    check-blank: true
```

```go
age, _ := strconv.Atoi(a)
```

↓

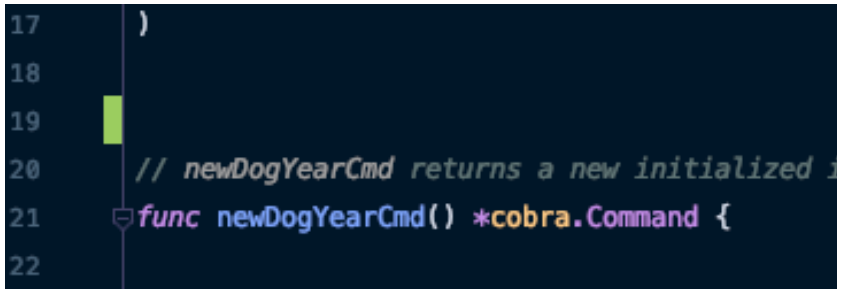
```go
age, err := strconv.Atoi(a)
if err != nil {
    return err
}
```

- Check errors!

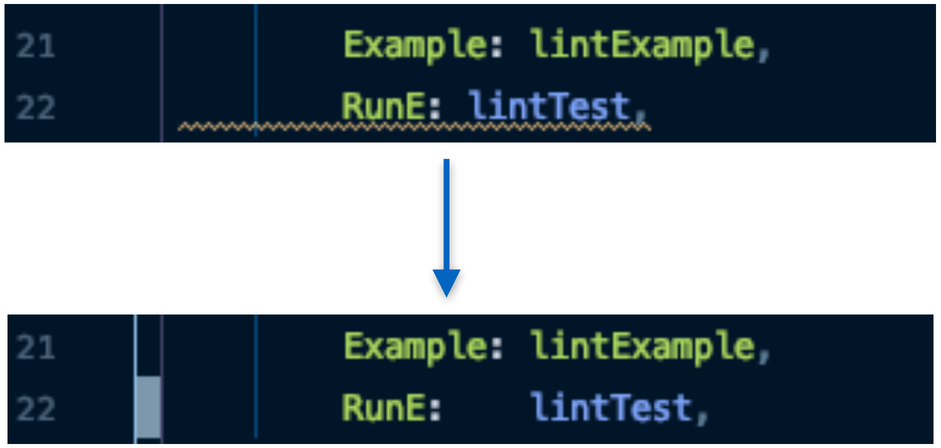
goimports

golangci-lint run
pkg/string/cmd/dogyears.go:19: File is not `goimports`-ed with -local github.com/codementor (goimports)

pkg/string/cmd/lint.go:22: File is not `goimports`-ed with -local github.com/codementor (goimports)



```
17    )
18
19
20    // newDogYearCmd returns a new initialized 
21    func newDogYearCmd() *cobra.Command {
22
```

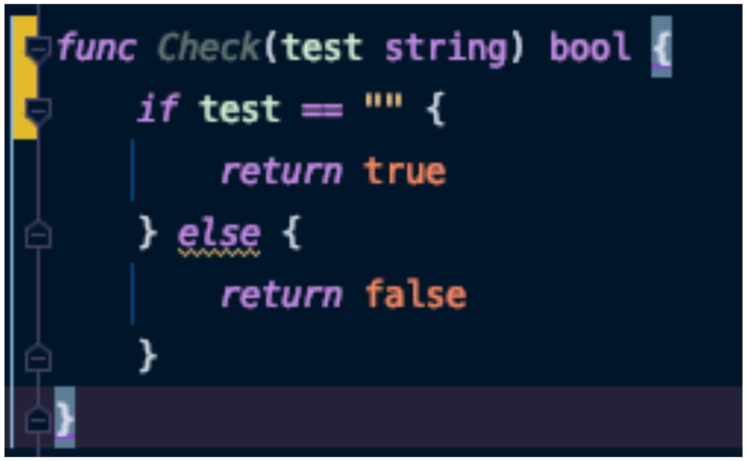


```
21          Example: lintExample,
22          RunE: lintTest,
```

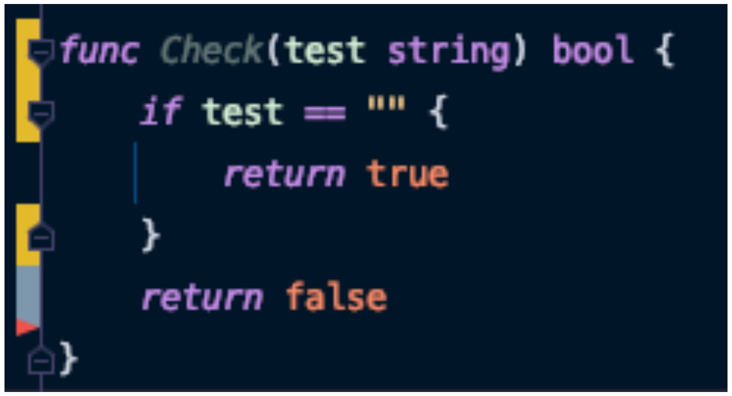
```
21          Example: lintExample,
22          RunE:    lintTest,
```

golint

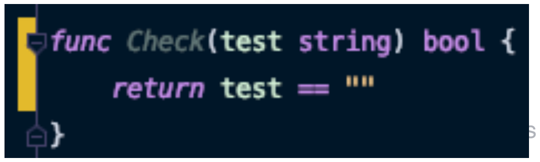
```
golangci-lint run
pkg/string/cmd/lint.go:45:9: `if` block ends with a `return` statement, so drop this `else` and outdent its block (golint)
            } else {
```



```go
func Check(test string) bool {
	if test == "" {
		return true
	} else {
		return false
	}
}
```



```go
func Check(test string) bool {
	if test == "" {
		return true
	}
	return false
}
```



```go
func Check(test string) bool {
	return test == ""
}
```

golint

```
golangci-lint run
pkg/string/cmd/lint.go:77:11: should omit 2nd value from range; this loop is equivalent to `for key := range ...`
(golint)
        for key, _ := range set {
                 ^
```

```go
    for key, _ := range set {
        fmt.Println(key)
    }
```

→

```go
    for key := range set {
        fmt.Println(key)
    }
```

gosimple

```
golangci-lint run
pkg/string/cmd/lint.go:49:7: S1016: should convert x (type T) to T2 instead of using struct literal (gosimple)
            y := T2{
              ^
```

```go
type T struct {
    Field1 string
    Field2 string
}
type T2 struct {
    Field1 string
    Field2 string
}
func lintTest(cmd *cobra.Command, args []string) error {
    var x T
    y := T2{
        Field1: x.Field1,
        Field2: x.Field2,
    }
```

- Two structs with identical fields can be converted between each other (Go 1.8+)

→

```go
    var x T
    y := T2(x)
```

gosimple

```
golangci-lint run
pkg/string/cmd/lint.go:57:5: S1009: should omit nil check; len() for nil slices is defined as zero (gosimple)
        if strs != nil && len(strs) != 0 {
           ^
```

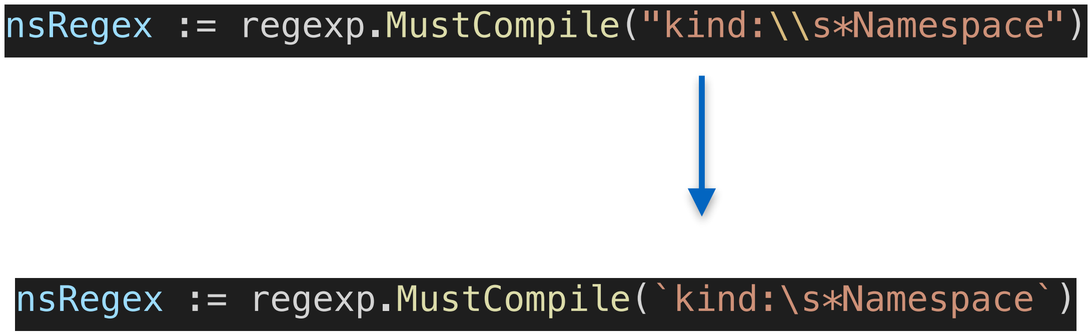
```go
    if strs != nil && len(strs) != 0 {
        fmt.Println("strs is not empty")
    }
```

$\longrightarrow$

```go
    if len(strs) != 0 {
        fmt.Println("strs is not empty")
    }
```

gosimple

```
golangci-lint run
pkg/string/cmd/lint.go:65:13: S1007: should use raw string (`...`) with regexp.MustCompile to avoid having to esca
                nsRegex := regexp.MustCompile("kind:\\s*Namespace")
                           ^
```



```go
nsRegex := regexp.MustCompile("kind:\\s*Namespace")
```

```go
nsRegex := regexp.MustCompile(`kind:\s*Namespace`)
```

gosimple

```
golangci-lint run
pkg/string/cmd/lint.go:61:2: S1011: should replace loop with `newStrs = append(newStrs, strs...)` (gosimple)
        for _, str := range strs {
        ^
```

```go
for _, str := range strs {
    newStrs = append(newStrs, str)
}
```

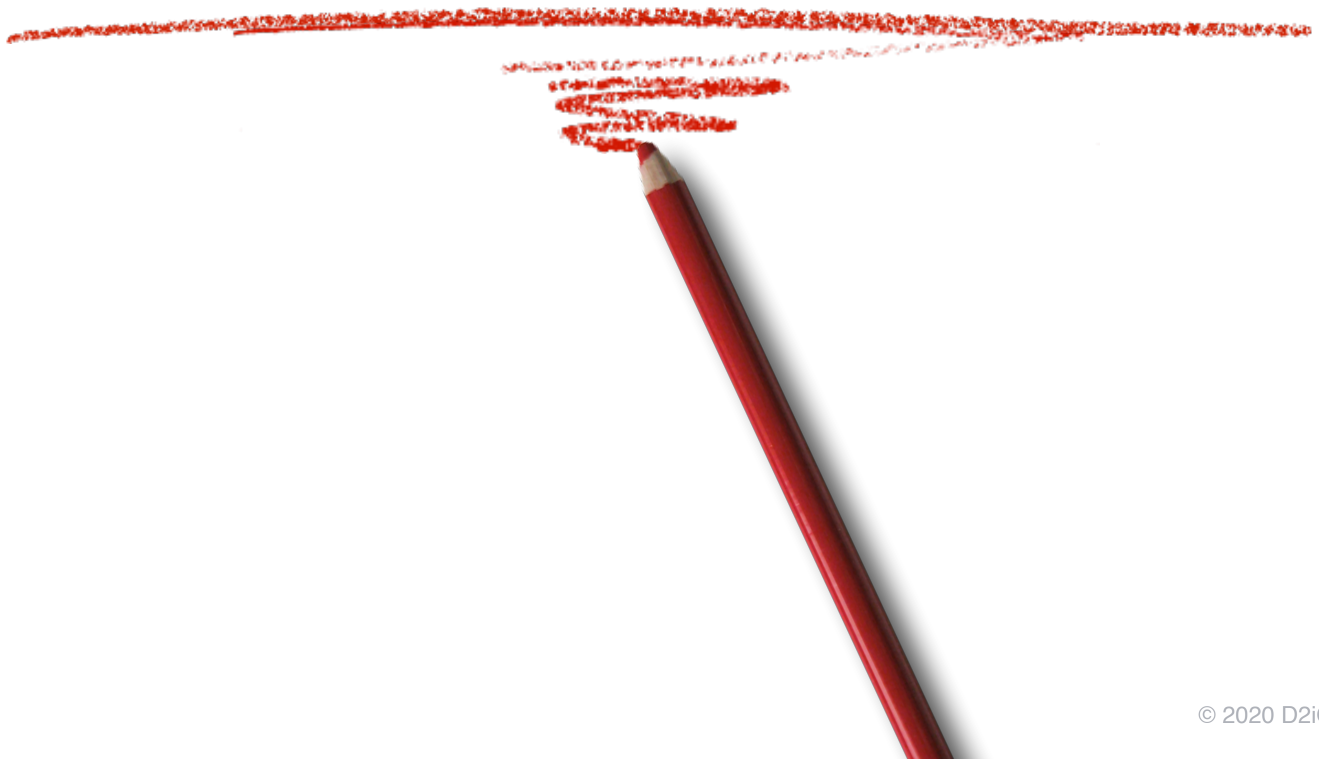
```go
newStrs = append(newStrs, strs...)
```

Demo: ex5-lint-3

https://github.com/codementor/go-mistakes

1. `make lint`

Common Mistakes, Go!



String Chans vs String Arrays

https://github.com/kubernetes-sigs/kind/blob/f9db81c462abccb2c3a527cdf54160e681579554/pkg/build/nodeimage/build_impl.go#L266-L291

```go
pulledImages := make(chan string, len(requiredImages))
```

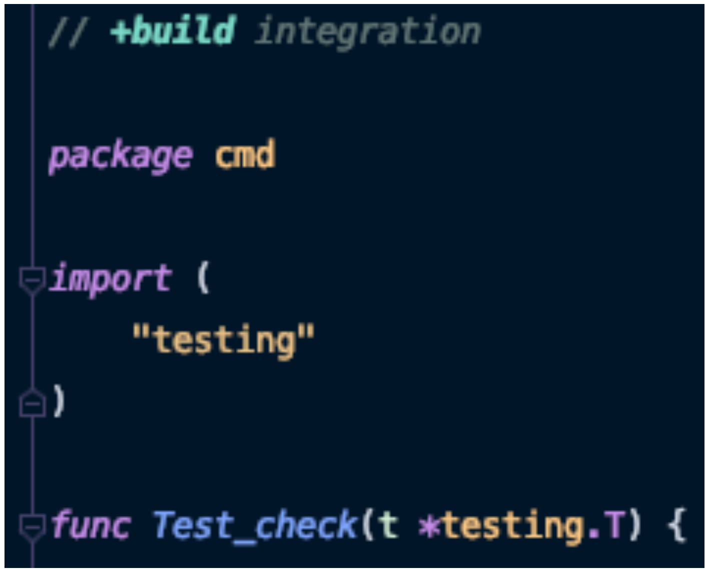
Vs

```go
var pulledImages []string
```

What is communicated?

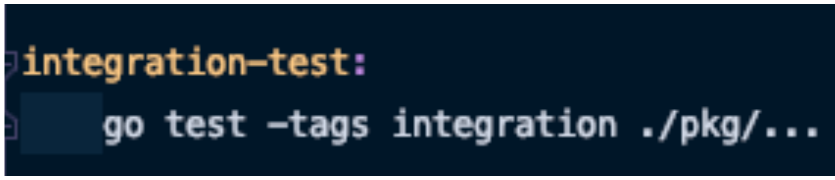
Build tags

- Commonly miss in Editor that code does NOT compile



```go
// +build integration

package cmd

import (
	"testing"
)

func Test_check(t *testing.T) {
```

```
pkg/string/cmd/lint_test.go:21:14: undefined: check
FAIL	github.com/codementor/go-mistakes/pkg/string/cmd [build failed]
FAIL
make: *** [integration-test] Error 2
```



```make
integration-test:
	go test -tags integration ./pkg/...
```

Demo: ex6-build-tags

https://github.com/codementor/go-mistakes

1. Open in Goland
2. Run `make integration-test`

Changing Values in a Range

```go
s := []int{1, 1, 1}
for _, n := range s {
    n += 1
}
fmt.Println(s)
// Output: [1 1 1]
```

```go
s := []int{1, 1, 1}
for i := range s {
    s[i] += 1
}
fmt.Println(s)
// Output: [2 2 2]
```

Looping it Wrong*

```go
package main

import (
    "fmt"
)

func main() {
    var out []*int
    for i := 0; i < 3; i++ {
        out = append(out, &i)
    }
    fmt.Println("Values:", *out[0], *out[1], *out[2])
    fmt.Println("Addresses:", out[0], out[1], out[2])
}
// output: Values: 3 3 3
// expecting: Values: 0 1 2
```

- The address `i` is used to hold values for the for loop. That address is the same for all iterations. We are storing pointers, which is the same each iteration. The value is the value at the end of the last iteration.

Looping over closures

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    data := []string{"one","two","three"}
    for _,v := range data {
        go func() {
            fmt.Println(v)
        }()
    }
    time.Sleep(3 * time.Second)
    //goroutines print: three, three, three
}
```

- v pointer is reused over data

```go
for _,v := range data {
        v := v //
        go func() {
            fmt.Println(v)
        }()
    }
```

Demo: ex7-loops

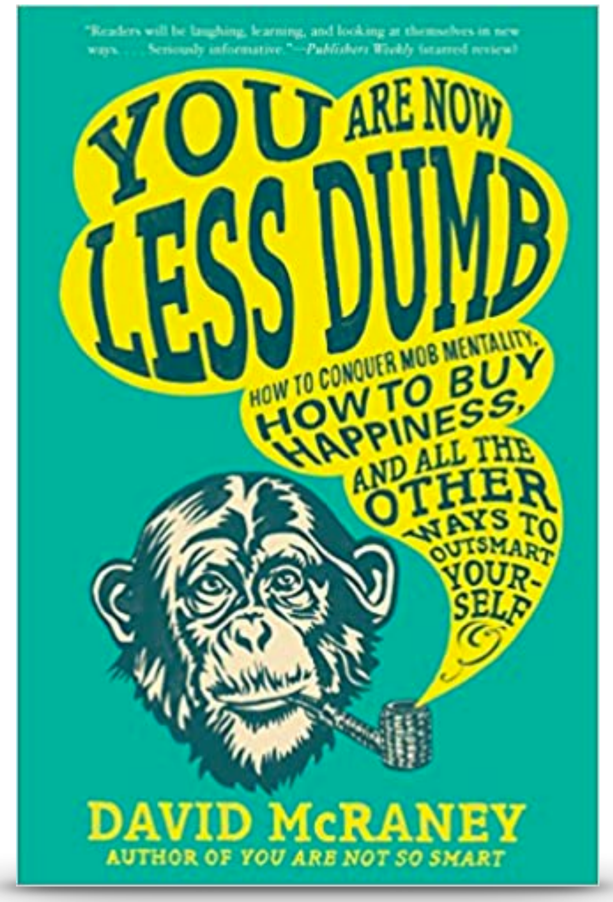
https://github.com/codementor/go-mistakes

1. `make lint`

So Many More!

Check out:

- [50 Shades of Go]
- [Do you make these Go coding mistakes]


"Readers will be laughing, learning, and looking at themselves in new ways. . . . Seriously informative." —Publishers Weekly (starred review)
YOU ARE NOW LESS DUMB
HOW TO CONQUER MOB MENTALITY, HOW TO BUY HAPPINESS, AND ALL THE OTHER WAYS TO OUTSMART YOURSELF
DAVID McRANEY
AUTHOR OF YOU ARE NOT SO SMART

Thank You!

@kensipe
kensipe@gmail.com
https://k8s.slack.com/#/