GO

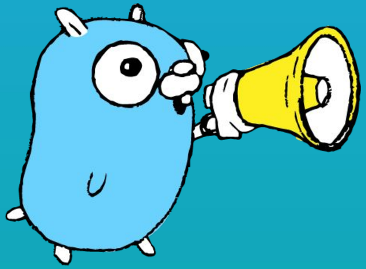
An introduction to Go for Python engineers

Go Prymer

J Gregory

Cardiff Go

@n3crophonic



Let's go!

A quick comparison

Let’s say hello to the world!

Let’s say hello to the world!

```python
from datetime import datetime


def greet(n):
    """Returns a nice greeting including
        the current date and time"""
    return "Hello! The time is " + n


def _now():
    return datetime.now().isoformat("T")


def main():
    print(greet(_now()))


if __name__ == "__main__":
    main()
```

```go
package main

import (
  "fmt"
  "time"
)

// Greet returns a nice greeting including
// the current date and time
func Greet(n string) string {
  return "Hello! The time is " + n
}

func now() string {
  return time.Now().Format(time.RFC3339)
}

func main() {
  fmt.Println(Greet(now()))
}
```

Let’s get running!

For python we invoke the interpreter

```
~/python $ python hello.py

Hello! The time is 2020-07-24T15:01:45.393178
```

Go compiles to a native executable binary using **go build**

```
~/go $ go build main.go
~/go $ ./main

Hello! The time is 2020-07-24T14:00:42+01:00
```

We can compile and run a Go program in one step using the **go run** command

```
~/go $ go run main.go

Hello! The time is 2020-07-24T14:00:42+01:00
```

The basic run experience should be fairly familiar

“

Q. What about virtual envs?

A. You don’t need them

- Go’s **modules** takes care of dependencies
- Dependencies are locked using **go.mod** & **go.sum** files
- All built into the standard toolchain!

Who goes where?

Comparing layouts

```
myproject/
├── Pipfile
├── Pipfile.lock
├── test/
│   ├── test_users.py
│   └── test_routes.py
└── app/
    ├── __init__.py
    ├── app.py
    ├── api/
    │   ├── __init__.py
    │   └── api.py
    └── data/
        ├── __init__.py
        └── data.py
```

```
myproject/
├── main.go
├── api/
│   ├── api.go
│   └── api_test.go
├── data/
│   ├── data.go
│   └── data_test.go
├── go.mod
└── go.sum
```

Colour coding:

- Dependencies
- Tests
- **Entrypoints**
- Python specific

The (almost) simplest Go project

Everything lives in a single folder and package

```
github.com/
└── user/
    └── myproject/
        ├── main.go
        ├── api.go
        ├── api_test.go
        ├── data.go
        ├── data_test.go
        ├── go.mod
        └── go.sum
```

All code is in **package main**

A common approach

Libraries in pkg/ and internal/ folders

```
github.com/user/myproject/
├── cmd/
│   └── service/
│       └── main.go
├── internal/
│   └── data/
│       ├── data.go
│       └── data_test.go
├── pkg/
│   └── api/
│       ├── api.go
│       └── api_test.go
├── go.mod
└── go.sum
```

Allows building of multiple apps sharing library code in one repo

“

Q. How many packages?

A. As few as possible!

- Start with everything in a single package
- Split into new packages only when you need to
- Splitting too early is a recipe for circular dependencies!

Good package names

Are short and concise	`time`, `list`, `http` are all good names
Should compliment their public functions and types	eg. `time.Now()`, `list.Add()`, `http.Client` read smoothly
Should not stutter	eg. prefer `client.New()` to `client.NewClient()`
Should avoid meaningless names	Names like `util`, `common` or `misc` provide clients with no sense of what the package contains

Take time to choose good packages names!

What no while?

The fundamentals

```python
# Dynamic typing
s = "cat"
i = True
s = i # This is fine in python!

# Call and assign
total = add(2,10)

# Functions declared with def
# Blocks defined by indent
def add(x,y):
  return x + y


# Nothing!
nowt = None
```

```go
// Strong typing
var s string = "cat"
i := true // bools are lowercase!
s = i     // Compiler error!

// Type of "total" is inferred
total := add(2,10) // int

// Functions declared with func
// Blocks defined by braces
func add(x int, y int) int {
  return x + y
}

// Also nothing!
nowt := nil
```

It’s all ***for*** one, and one ***for*** all!

```python
# Get index and value
for i, v in enumerate(z):
  print(i,v)


# A c-style while
i=0
while(i<len(z)):
  print(i,z[i])
  i+=1


# Infinite loop
while True:
  print("on and on!")
```

```go
// Go range acts like "enumerate" - it's
// not the same as python's range!
for i, v := range(z) {
  fmt.Println(i,v)

// No while, just a cut down "for"!
i := 0
for i < len(z) {
  fmt.Println(i, z[i])
  i++
}

// Infinite? Just omit the params!
for {
  fmt.Println("_for_ever")
}
```

You’re just my type - dictionaries == maps

```python
# Declare and assign
d = {"a":"x", "b":"y"}


# Declare, then assign
d2 = {}
d2["a"] = "x"
d2["b"] = "y"


# Assigning to an undeclared
# dict is a runtime error
d3["a"] = "z"
```

```go
// Declare and assign
d := map[string]string{"a": "x", "b": "y"}


// Declare, then assign
d2 := make(map[string]string)
d2["a"] = "x"
d2["b"] = "y"


// Declared but not initialised
var d3 map[string]string
d3["a"] = "x" // Compile error!

// Not declared or initialised
d4["a"] = "x" // Compile error!
```

You're just my type too - lists == arrays/slices

```python
# Declare and assign
s = ["a","b"]


# Declare, then assign
s2 = []
s2.append("a")
s2.append("b")
s2[1] = "z"

# Assign to undeclared is a
# runtime error
s3[0] = "a"
```

```go
// Declare and assign
e := [2]string{"a","b"} // array
s := []string{"a","b"}  // slice

// Declare, then assign
s2 := make([]string,0,2)
s2 = append(s2, "a")
s2 = append(s2, "b")
s2[1] = "z"

// Declared but not initialised
var s3 []string
s3[0] = "a" // Compile error!

// Not declared or initialised
s4[0] = "a" // Compile error!
```

Structural anatomy

```go
// Basket is a customer shopping basket
type Basket struct {
  ID    string  `json:"basket_id"`
  Total float64 `json:"total"`

  paid  bool

  Items []LineItem{
    SKU          string
    LinePrice    float64
    Amount       int
  }
}
```

Struct tags make cool stuff like marshaling to and from JSON easy

Although struct tags aren’t required!

Structs can contain any valid type, including other structs! (Or here, a slice of `LineItem` structs)

Don’t be classy

```python
class Dog:

  def __init__(self, name):
    self.name = name

  def bark(self):
    print(self.name + " says wuff")

  def _fleas(self):
    print("shh don't tell")


pet = Dog("fido")
pet.bark()
```

```go
type Dog struct {
  Name  string
}

func (d Dog) Bark() {
  fmt.Println(d.Name + "says wuff")
}

func (d Dog) fleas() {
  fmt.Println("shh don't tell")
}

func main() {
  pet := Dog{Name: "fido"}
  pet.Bark()
}
```

Structural composition

```go
type animal struct {
  Name string `json:"name"`
}

type dog struct {
  animal
  Walkies time.Time
}

func (d *dog) Pat() {
  fmt.Println("who's a good doggy?")
}

type cat struct {
  animal
  lives int // that's private!
}
```

```go
type petter interface {
  Pat()
}


var fido *dog
fido = &dog{} // It's a dog!

var butch interface{}
butch = &dog{} // All good!

var scooby petter
scooby = &dog{} // Good too!

var felix petter
felix = &cat{} // Compiler error!
```

“

Q. Where are my getters?

A. Go doesn’t use them (mostly)!

- Remember, structs aren’t classes
- To set or read a public field you access it directly
- Though, that doesn’t mean you *can’t* use them sometimes!

Don’t be exceptional

```
try:
  id = getID()
  name, age = loadByID(id)
  ratio = age/len(name)
  # Other logic...
  # ...
except ZeroDivisionError:
  # Something did a bad
thing!
  # ...
except KeyError as e:
  # Something read badly
  # ...
except:
  # Other things went awry
```

```
id, err := getID()
if err != nil {
  // handle it!
}

name, age, err := loadByID(id)
if err != nil {
  // handle it!
}

if name == "" {
  // handle it!
}
ratio := age/len(name)
```

Things that are specific to Go

The special stuff

Marshaling

```go
type Feline struct {
   Name  string `json:"name"`
   Age   int
   Furry bool   `json:"fluffy"`
   lives int
}

f := &Feline{
   Name: "Bagpuss",
   Age: 10,
   Furry: true,
   lives: 9,
}

b, err := json.Marshal(&f)
// Handle error and print!
//...
```

```json
{
  "name":"Bagpuss",
  "Age":10,
  "fluffy":true
}
```

In the JSON result:

Name becomes **name**
Age stays as **Age**
Furry becomes **fluffy**
lives is not marshaled!

Cross compilation

All platforms, one compiler

```
// Set the OS and architecture to
// build for as environment vars
// passed to go build
//
// Some examples:

$> GOOS=linux GOARCH=amd64 go build

$> GOOS=plan9 GOARCH=386 go build

$> GOOS=darwin GOARCH=amd64 go build

$> GOOS=windows GOARCH=amd64 go build
```

Build anywhere, for anywhere!

Ensure stuff happens with defer()

Let Go remember for you

```
func readFile(name string) {

  f, err := os.Create(name)
  if err != nil {
    panic(err)
  }
  defer f.Close()

  // .. do cool stuff with the file

  // .. don’t need to remember to close file
  //    defer will take care of it as it
  //    drops out of scope!
  return
}
```

Never forget to clean up!

Concurrency with Goroutines

Go is concurrent by design

```
func main() {
  worker()
  worker()

  // … more awesomeness
}
```

```
func main() {
  go worker()
  go worker()

  // … more awesomeness
}
```

Easy and safe scalability

Channels

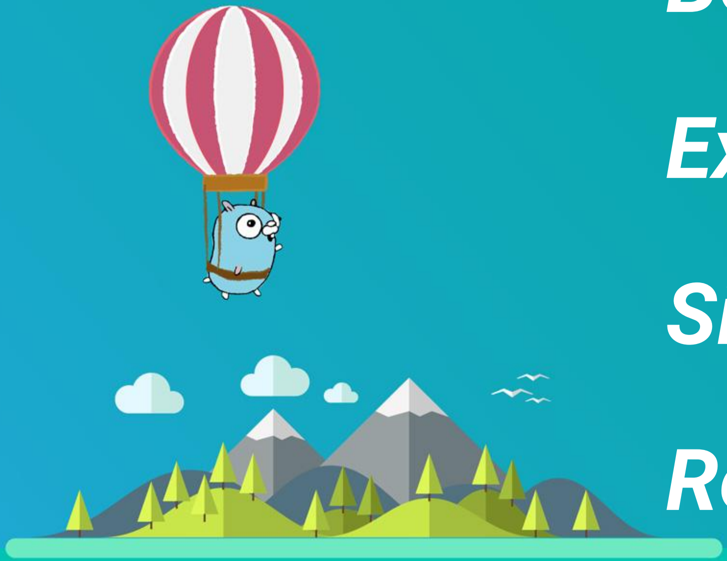
Safe concurrent data with channels

```go
func main() {
  c := make(chan int, 1)
  go addOne(41, c)
  result := <- c
  fmt.Println("Result was:", result)
}

func addOne(i int, c chan int) {
  c <- i + 1
}
```

```
$> go run main.go

Result was 42
```

No more mutexes!

Coding with style



“

***Beautiful** is better than ugly*

***Explicit** is better than implicit*

***Simple** is better than complex*

***Readability** counts*

”

Extract from “The Zen of Python”
by Tim Peters

Let's print the elements of a list (z) and their index

We could use a **while** loop

```
i = 0
while i < len(z):
    print i, z[i]
    i += 1
```

Or maybe it's cleaner to use a **for** loop with a **range**

```
for i in range(0, len(z)):
    print i, z[i]
```

Though following the idiomatic principals, in python you'd probably more expect to see this

```
for i, item in enumerate(z):
    print i, item
```

Of course, these aren't the only ways!

What about something similar in Go?

Go doesn't have a **while** loop, but as we've seen we can use a **for** in its place

```go
i := 0
for i < len(z) {
    fmt.Println(i, z[i])
    i += 1
}
```

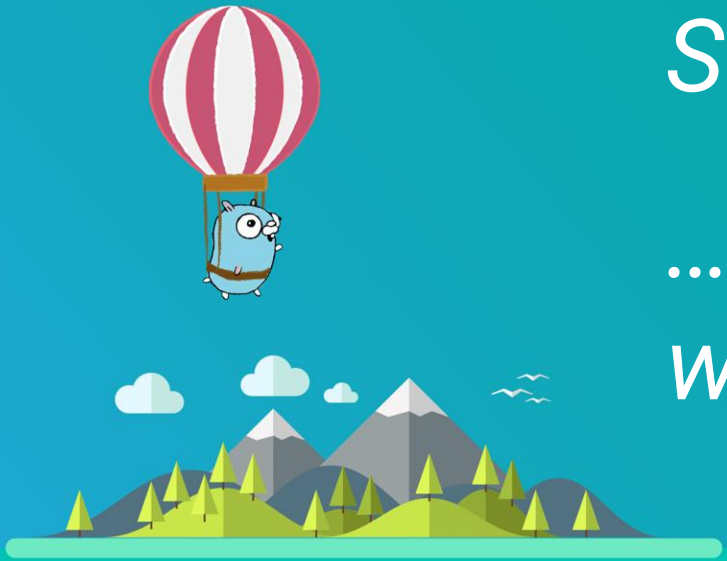
Or we could use a c-style **for**

```go
for i:=0 ; i<len(z) ; i++ {
    fmt.Println(i, z[i])
}
```

Though in idiomatic Go, you'd more often expect the use of **range**

```go
for i, v := range(z) {
    fmt.Println(i, v)
}
```

Remember, Go's **range** is not the same as python's!



“

Start with simplicity …

… invoke complexity only when you need it

”

A core principle of Go

“

Q. *Any final thoughts?*

A. Yup!

- Don’t web search “go”, use “golang!”
- Always run ***go fmt***!
- Don’t overthink it!

Over to you!

Any Questions?

Thanks for listening!

J Gregory

Twitter: @n3crophonic

Github: github.com/necrophonic

Instagram: cafpanda

See also:

Idiomatic Code
https://intermediate-and-advanced-software-carpentry.readthedocs.io/en/latest/idiomatic-python.html
https://golang.org/doc/effective_go.html
https://blog.golang.org/package-names

Getting started with Go
https://tour.golang.org

Concurrency
https://blog.golang.org/codelab-share

Python references
https://deepsource.io/blog/python-walrus-operator/

Awesome list of training
https://github.com/ardanlabs/gotraining/blob/master/reading/README.md

...and there's loads more great resources out there!