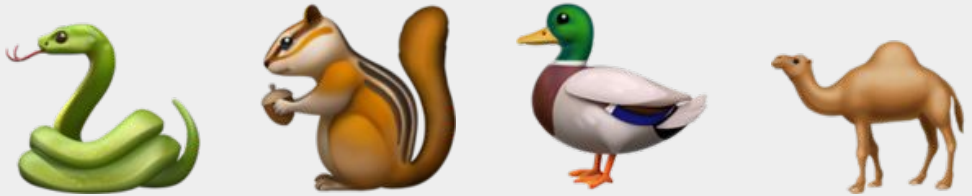
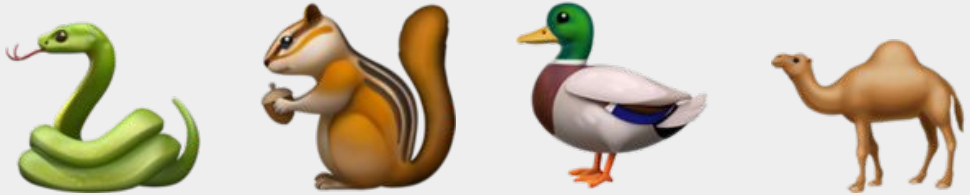




An introduction to **Go** for **Python** developers

and the significance of **codebase diversity**

An introduction to **Go** for **Python** developers

and the significance of **codebase diversity**

About me

Electrical Engineering

Computer Science

since 2015
Build- and Infrastructure-Automation Engineer
at **Demonware**

Drummer

Cook

Craft Beer Enthusiast

@__FrontSide__

@frontside

__Frontside__

The evolution of Python

The evolution of Python

1989 **Start of Implementation**

The evolution of Python

1989 **Start of Implementation**

Taylor Swift was born

The evolution of Python

1989 **Start of Implementation**

1991 **First Release**

Inheritance, Exceptions, Core Data types

The evolution of Python

1989 **Start of Implementation**

1991 **First Release**
Inheritance, Exceptions, Core Data types

1994 **Version 1**
Functional tools (Guido van Rossum doesn’t approve)

The evolution of Python

1989 **Start of Implementation**

1991 **First Release**
Inheritance, Exceptions, Core Data types

1994 **Version 1**
Functional tools (Guido van Rossum doesn’t approve)

2000 **Version 2**
List comprehensions, full garbage collection

The evolution of Python

1989 **Start of Implementation**

1991 **First Release**
Inheritance, Exceptions, Core Data types

1994 **Version 1**
Functional tools (Guido van Rossum doesn’t approve)

2000 **Version 2**
List comprehensions, full garbage collection

2001 **Version 2.2**
Became fully object oriented through
unification of types and classes

The evolution of Python

1989 **Start of Implementation**

1991 **First Release**
Inheritance, Exceptions, Core Data types

1994 **Version 1**
Functional tools

2000 **Version 2**
List comprehensions, full garbage collection

2001 **Version 2.2**
Became fully object oriented through unification of types and classes

Taylor Swift was born

2006 **Taylor Swift releases first album**

The evolution of Python

1989 **Start of Implementation**

1991 **First Release**
Inheritance, Exceptions, Core Data types

1994 **Version 1**
Functional tools

2000 **Version 2**
List comprehensions, full garbage collection

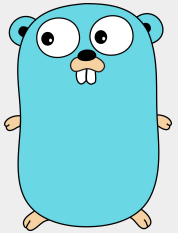
2001 **Version 2.2**
Became fully object oriented through unification of types and classes

2008 **Version 3**

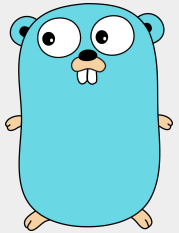
Taylor Swift was born

2006 **Taylor Swift releases first album**

The evolution of Go



The evolution of Go

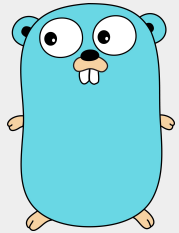


2009

First public appearance

Used in some Google prod systems

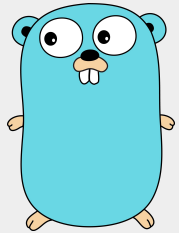
The evolution of Go



2009 **First public appearance**
Used in some Google prod systems

2012 **Version 1**
Most of the things we know today are already here

The evolution of Go



2009 **First public appearance**
Used in some Google prod systems

2012 **Version 1**
Most of the things we know today are already here

2015 **Version 1.5**
No more C (Go, Assembler)

The evolution of Go

2009 **First public appearance**
Used in some Google prod systems

2012 **Version 1**
Most of the things we know today are already here

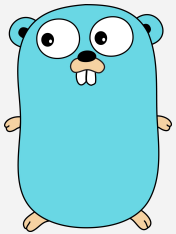
2015 **Version 1.5**
No more C (Go, Assembler)

2017 **Version 1.9**

Slide Style

Python

Shell



Go

Shell

Object-oriented

(amongst other paradigms)

Concurrency-Oriented

(but offers tools to implement object-oriented concepts)

Hello World

```
print(“Hello World”)
```

```
$ python3 hello.py
Hello World
```

```
package main

import “fmt”

func main() {
    fmt.Println(“Hello World”)
}
```

```
$ go run hello.go
Hello World
```

Compilation

- Compile into python bytecode
- Needs python interpreter
- Small files

```
$ python3 -m py_compile hello.py

$ ls
__pycache__    hello.py

$ ls -lh __pycache__/*
116B hello.cpython-36.pyc
```

Compilation

- Compile into python bytecode
- Needs python interpreter
- Small files

```
$ python3 -m py_compile hello.py

$ ls
__pycache__    hello.py

$ ls -lh __pycache__/*
116B hello.cpython-36.pyc
```

- Compile into machine code
- Large Binaries
- Build for target architecture

```
$ go build hello.go

$ ls -lh
72B  hello.go
1.6M hello
```

Compilation

- Compile into python bytecode
- Needs python interpreter
- Small files

```
$ python3 -m py_compile hello.py

$ ls
__pycache__    hello.py

$ ls -lh __pycache__/*
116B hello.cpython-36.pyc
```

- Compile into machine code
- Large Binaries
- Build for target architecture

```
$ go build hello.go

$ ls -lh
72B  hello.go
1.6M hello
```

Question: Why are go binaries so large?

Compilation

- Compile into python bytecode
- Needs python interpreter
- Small files

```
$ python3 -m py_compile hello.py

$ ls
__pycache__    hello.py

$ ls -lh __pycache__/*
116B hello.cpython-36.pyc
```

- Compile into machine code
- Large Binaries
- Build for target architecture

```
$ go build hello.go

$ ls -lh
72B  hello.go
1.6M hello
```

Question: Why are go binaries so large?

- Fully self-contained
- Includes runtime with garbage collector, reflection and other things
- Includes dependency code (here: fmt)

Manipulating an array

```
def increase(nums):
    for idx in range(nums):
        nums[idx] += 1

nums = [1, 2, 3]
increase(nums)
print(nums)
```

```
$ python3 array.py
[2, 3, 4]
```

Manipulating an array

```python
def increase(nums):
    for idx in range(nums):
        nums[idx] += 1

nums = [1, 2, 3]
increase(nums)
print(nums)
```

```
$ python3 array.py
[2, 3, 4]
```

```go
func Increase(nums []int) {
    for id, n in := nums {
        nums[id] = n + 1
    }
}

func main() {
    nums := []int{1, 2, 3}
    Increase(nums)
    fmt.Println(nums)
}
```

```
$ go run array.go
[2 3 4]
```

Manipulating an array

```
func Increase(nums []int) {
    for id, n in := nums {
        nums[id] = n + 1
    }
}

func main() {
    nums := []int{1, 2, 3}
    Increase(nums)
    fmt.Println(nums)
}
```

```
$ go run array.go
[2 3 4]
```

```
func Increase(nums [3]int) {
    for id, n in := nums {
        nums[id] = n + 1
    }
}

func main() {
    nums := [3]int{1, 2, 3}
    Increase(nums)
    fmt.Println(nums)
}
```

```
$ go run array.go
[1 2 3]
```

Slice

```
[]int{1, 2 ,3}
```

pass-by-reference
Extendable

Array

```
[3]int{1, 2 ,3}
```

pass-by-value
fixed-size

Slice

```
[]int{1, 2 ,3}
```

pass-by-reference
Extendable

Array

```
[3]int{1, 2 ,3}
```

pass-by-value
fixed-size

Identifying semantic inconsistencies

```
def get_item_name(iid):

  try:
    return get_item_by_id(iid).name
  except:
    raise("No item with id %d" % id)

print(get_item_name(123))
```

```
func GetItemName(iid int) (int, error) {

  item, err := getItemById(iid)

  if err != nil {
    fmt.Errorf("No item w. id %d", id)
    return "", err
  }

  return item.name, nil

}

func main() {
    name, _ := GetItemName(123)
    fmt.Println(name)
}
```

Identifying semantic inconsistencies

```
def get_item_name(iid):

  try:
    return get_item_by_id(iid).name
  except:
    raise("No item with id %d" % id)

print(get_item_name(123))
```

```
$ python3 name.py
Rashers
```

```
func GetItemName(iid int) (int, error) {

  item, err := getItemById(iid)

  if err != nil {
    fmt.Errorf("No item w. id %d", id)
    return "", err
  }

  return item.name, nil

}

func main() {
    name, _ := GetItemName(123)
    fmt.Println(name)
}
```

```
$ go run name.go
./name.go:22: undefined: id
```

Identifying semantic inconsistencies

I have to work towards 100% code coverage to **ensure semantic correctness of all flow paths.**

I still need to write tests. But I can **focus them on my business logic's semantic correctness.**

“But we should always strive for high code coverage. So what’s the problem ?”

*“Program testing can be used to show the presence of bugs, but **never to show their absence!**”*

- Edsger W. Dijkstra
Notes On Structured Programming

$$p(\textit{fly in room} \mid \textit{no evidence for fly in room}) > 0$$

Tests can show presence of bugs, and not their absence?
2010, Blog Post

tonyxzt.blogspot.ie/2010/01/tests-can-show-presence-of-bugs-not.html

Encapsulation

```
class Person:

    age = 41
    _ppsn = “1234567AB”

print(Person.age)
print(Person._ppsn)
```

```
$ python3 age.py
41
‘1234567AB’
```

Official convention; doesn’t actually do anything.

Encapsulation

```
class Person:

    age = 41
    __ppsn = “1234567AB”

print(Person.age)
print(Person.__ppsn)
```

```
$ python3 age.py
41
AttributeError: type object 'Person'
has no attribute '__ppsn'
```

Not the official convention for encapsulation,
but it works.

Encapsulation

```
class Person:

    age = 41
    __ppsn = “1234567AB”

print(Person.age)
print(Person.__ppsn)
```

```
$ python3 age.py
41
AttributeError: type object 'Person'
has no attribute '__ppsn'
```

Not the official convention for encapsulation, but it works.

package person

```
type Person struct {
    Age int,
    ppsn string,
}
```

package main

```
func main() {

    p := Person{41, “12345678AB”}
    fmt.Println(p.Age)
    fmt.Println(p.ppsn)

}
```

```
$ go run age.go
.ppsn undefined (cannot refer to
unexported field or method ppsn)
```

Interfaces

```
interface Storage {
    AddItem()
}


type MySQLStorage struct {}

func (m MySQLStorage) AddItem() {
    // algorithm to add Item to MySQL DB
}
```

Structs implement interfaces implicitly if they implement all methods defined in the interface

Here: **`MySQLStorage implements Storage`**

Error handling

```
class NotOfAgeException(Exception):
    pass
```

Every class inheriting from “Exception”
can be raised

Error handling

```
class NotOfAgeException(Exception):
    pass
```

Every class inheriting from “Exception” can be raised

builtin

```
type error interface {
    Error() string
}
```

Every struct with an Error() method is an error type

package person

```
type NotOfAgeError struct {
    msg string
}

func (e *NotOfAgeError) Error() string {
    return e.msg
}
```

Error handling

```
def add_to_cart(customer, product):

  if product.restr_age > customer.age:
    raise NotOfAgeException()

  # ... add to cart logic ...
```

```
def main():

  # ... some shopping logic ...

  try:
    add_to_cart(customer, product)
  except NotOfAgeException:
    # handle in some way
  except OtherException:
    # handle in a different way
```

Error handling

```
def add_to_cart(customer, product):

  if product.restr_age > customer.age:
    raise NotOfAgeException()

  # ... add to cart logic ...
```

```
def main():

  # ... some shopping logic ...

  try:
    add_to_cart(customer, product)
  except NotOfAgeException:
    # handle some way
  except OtherException:
    # handle in a different way
```

```
func AddToCart(c Custom, p Prod) error {

    if p.restricted_age > c.age {
        return NotOfAgeError{}
    }

    // ... add to cart logic ...

    return nil
}
```

```
func main() {

  // ... some shopping logic ...

  err := AddToCart(customer, product)

  if e,ok := err.(*NotOfAgeError); ok {
    // handle some way
  }

}
```

Concurrency

```
import threading

class Compute(threading.Thread):

    def __init__(self):
        threading.Thread.__init__(self)

    def run(self):
        // some algorithm


Compute().start()
Compute().start()
```

Concurrency

```
import threading

class Compute(threading.Thread):

    def __init__(self):
        threading.Thread.__init__(self)

    def run(self):
        // some algorithm


Compute().start()
Compute().start()
```

```
func Compute() {
    // some algorithm
}

func main() {
    go Compute()
    Compute()
}
```

Cross-routine communication

```
func main() {

    numberChannel := make(chan int)
    go Producer(numberChannel)
    Printer(numberChannel)

}

func Producer(ch chan int) {
    for {
        ch <- SlowAlgorithmToGetNumber()
    }
}

func Printer(ch chan int) {
    for {
        Print(<-ch)
    }
}
```

Cross-routine communication

```
func main() {

    numberChannel := make(chan int)
    go Producer(numberChannel)
    Printer(numberChannel)

}

func Producer(ch chan int) {
    for {
        ch <- SlowAlgorithmToGetNumber()
    }
}

func Printer(ch chan int) {
    for {
        Print(<-ch)
    }
}
```

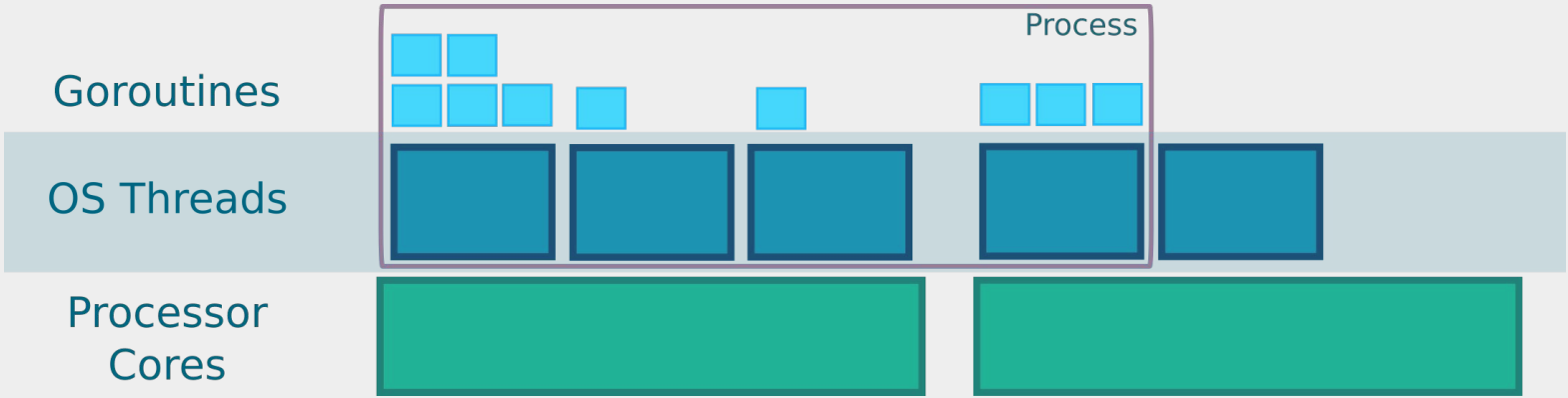
Blocks if channel occupied

Blocks until value available

You do not need to worry about synchronization.

No explicit locking required.

If GOMAXPROCS > 1
goroutines may be executed at the same time i.e. in parallel



As of Go 1.5 GOMAXPROCS defaults to your machine's number of processor cores

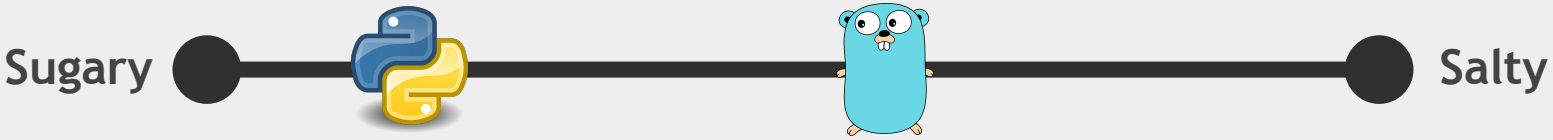
“Parallelism is not the goal of concurrency. Concurrency’s goal is structure.”

- Rob Pike

2013, Concurrency is not parallelism

https://www.youtube.com/watch?v=cN_DpYBzKso

Syntactic Flavour



Syntactic Salt

Enforced indentation

```
if 1 > 2:
print(“Wat?”)
```

```
$ python3 indentation.py
IndentationError: expected an
indented block
```

Enforced use of imports and variables

```
package main

import “fmt”

func main() {

    a := 1

}
```

```
$ go run unused.go
Imported and not used: “fmt”
```

Syntactic Salt

Enforced indentation

```
if 1 > 2:
print(“Wat?”)
```

```
$ python3 indentation.py
IndentationError: expected an
indented block
```

Enforced use of imports and variables

```
package main

func main() {

    a := 1

}
```

```
$ go run unused.go
a declared and not used
```

Formatting

PEP 8

- spaces
- 79 chars per line
- CamelCase for Classes
- snake_case for functions
- etc …

go fmt

- tabs
- no max. line length
- CamelCase everywhere
- etc …

Non negotiable (if used)

Tabs vs Spaces:
It’s <u>not</u> up to you

Dependency Management

pypi repository; e.g.

```
$ pip install docker
```

```
import docker
```

No pypi equivalent

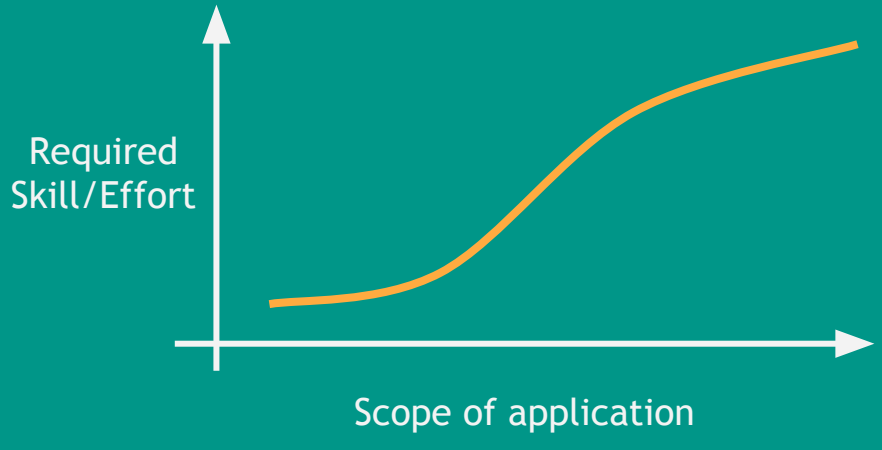
By default go pulls dependencies from the tip of the repository's master branch

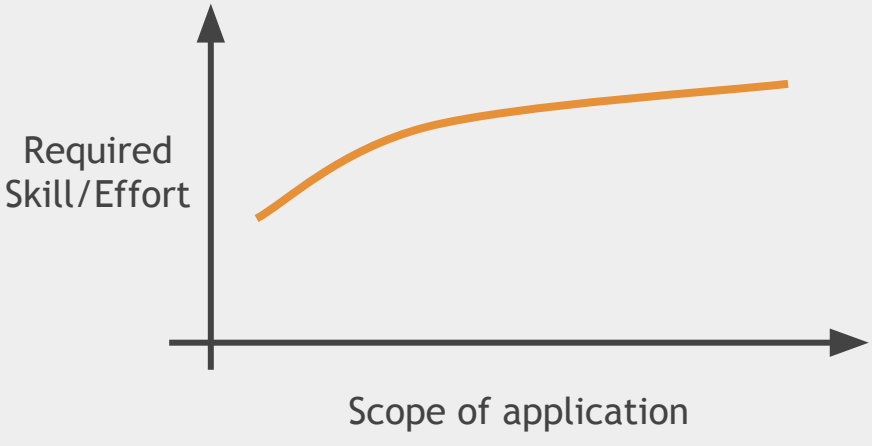
e.g.

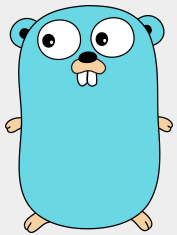
```
$ go get
```

```
import "github.com/moby/moby/client"
```

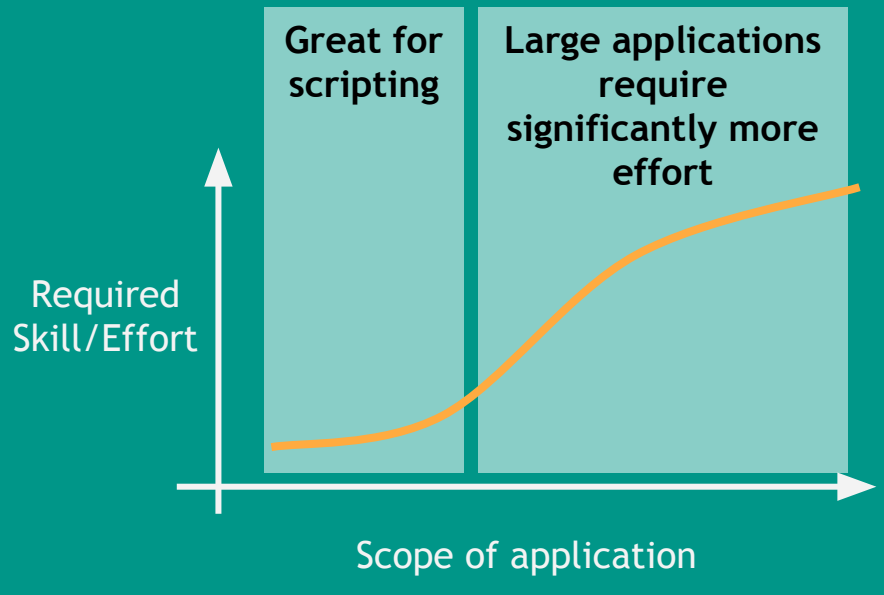
Effort = f(Scope)
from my experience

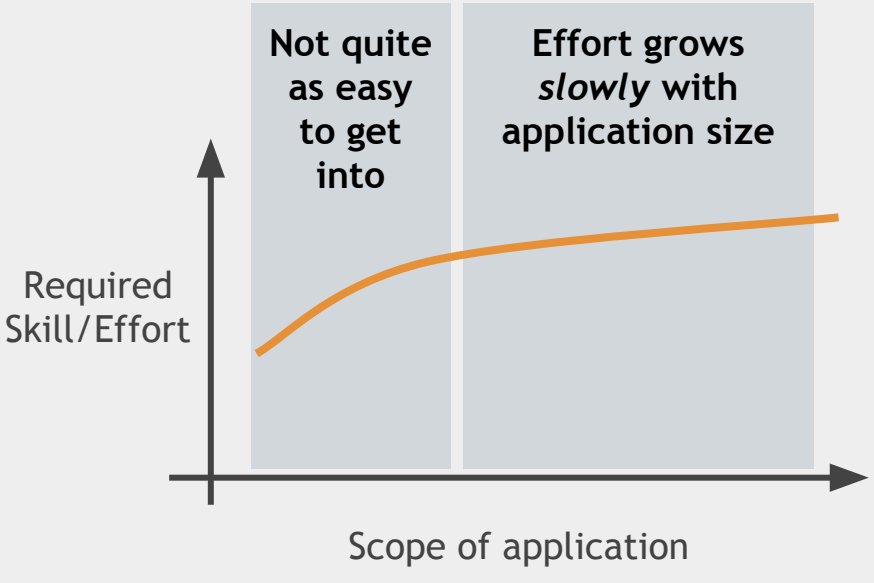


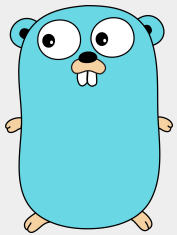




Effort = f(Scope)







“Ok great, but why bother learning a new language?”

“Ok great, but why bother learning a new language?”

“Because, Science!”

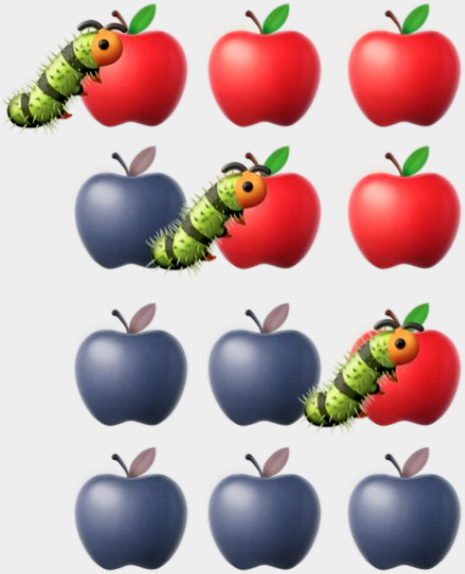
“Both theory and observation indicate that genetic heterogeneity [amongst crop] provides greater disease suppression…”

Genetic diversity and disease control in rice

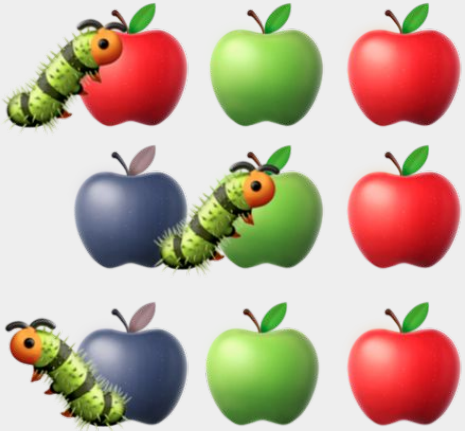
2000, Academic Article in Nature Science Journal

http://www.nature.com/nature/journal/v406/n6797/abs/406718a0.html

Monoculture



Polyculture



Emoji: Apple Inc.

Using the same language everywhere leads to repeating the same mistakes.

A bad coding practice can easily spread across all your applications.

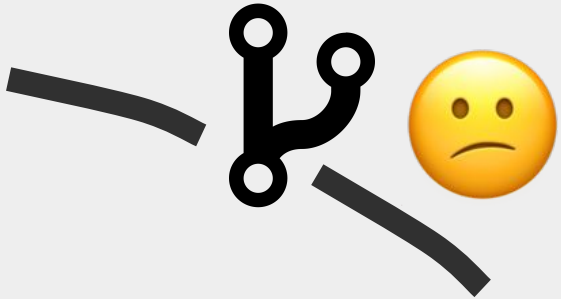
OPINION

“What if this isn’t true?”

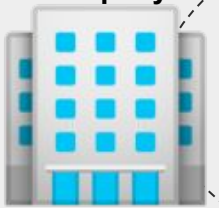
“What if this isn’t true?”

“Then there are plenty of other reasons”

useful
3rd-party tool



Company



our software

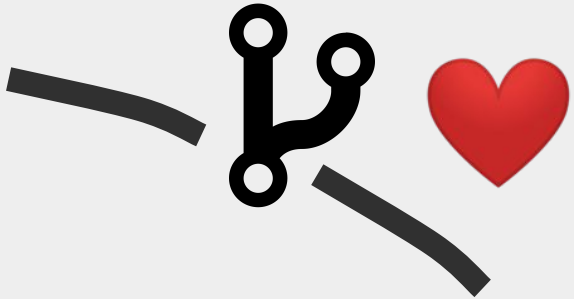
our software

our software

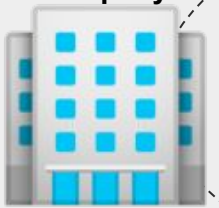
our software

Why use not just use one language everywhere?

useful
3rd-party tool



Company



our software

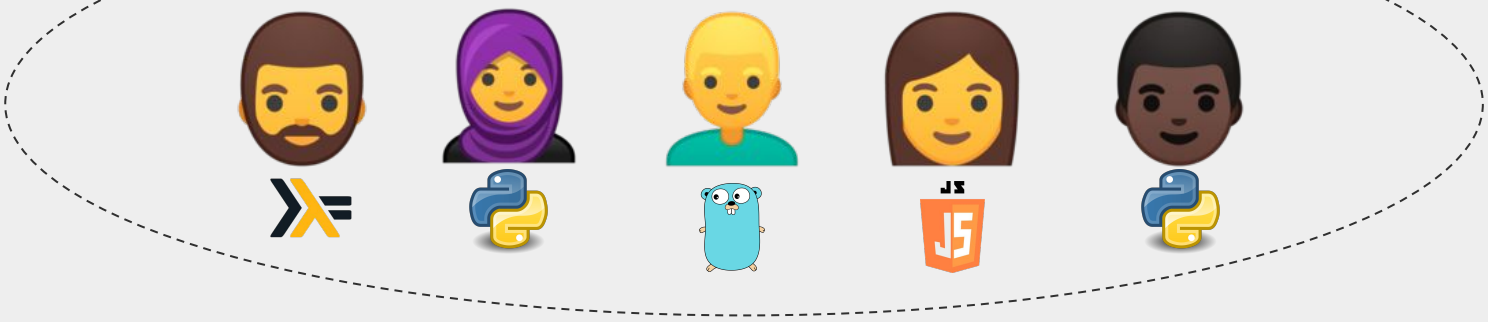
our software

our software

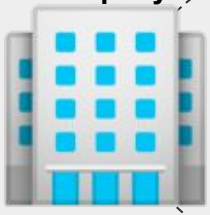
our software

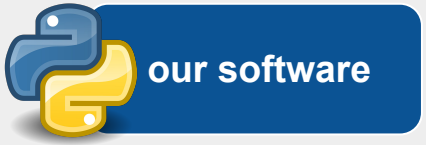
Why use not just use one language everywhere?

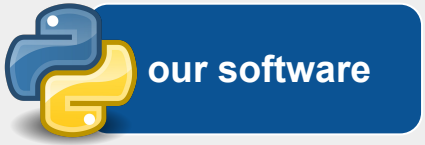
Employee Market

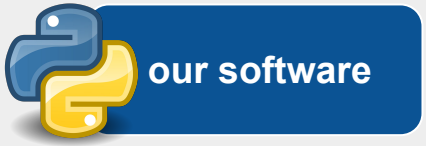


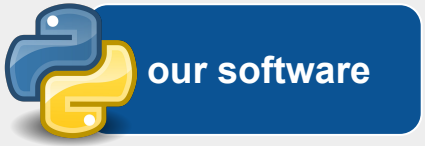
Company





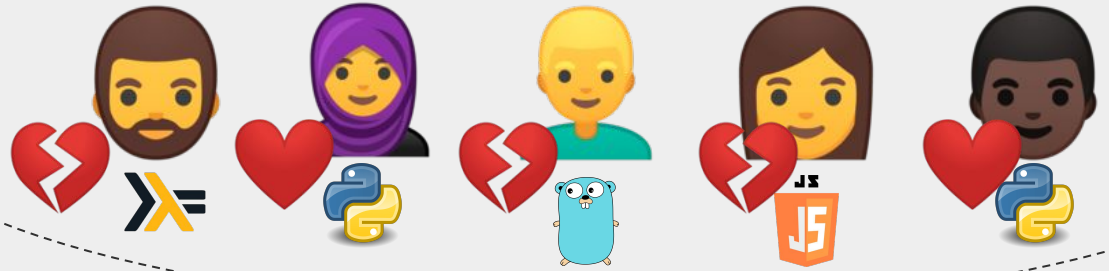




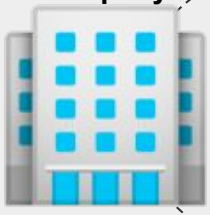


Why use not just use one language everywhere?

Employee Market



Company



our software

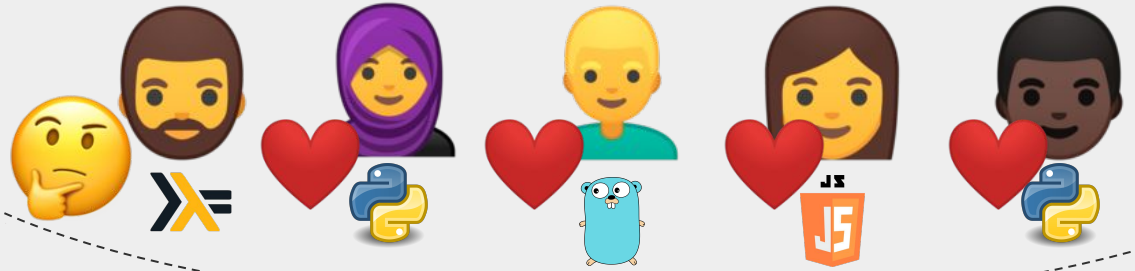
our software

our software

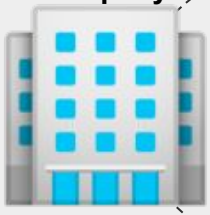
our software

Why use not just use one language everywhere?

Employee Market



Company



Perspective

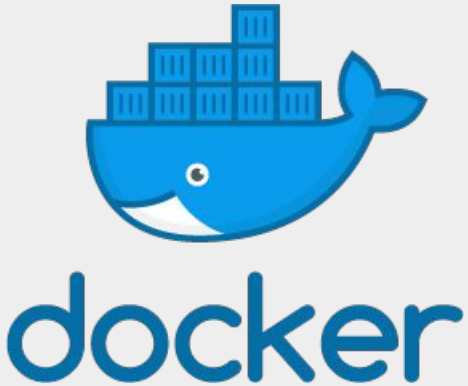
New challenges
for employees
and the company

Drives innovation

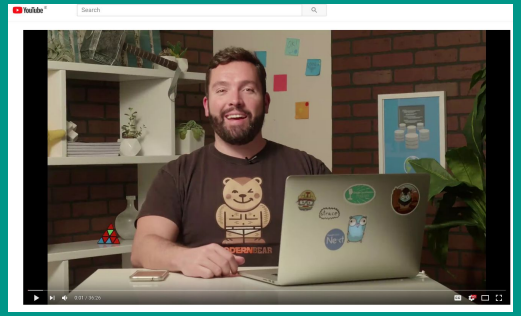
It all comes down to using the right tools

“Each language may require different environments.
This makes deployment harder.”

Containers provide an abstraction layer and help with deployment



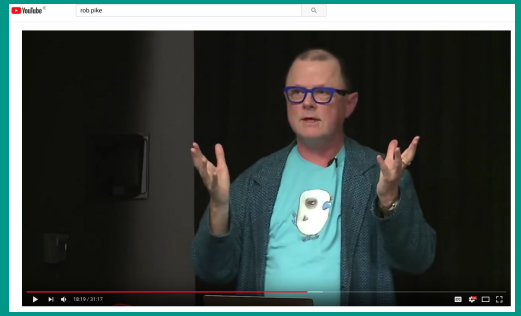
More...



Francesc Campoy

Just for func

YouTube



Rob Pike

Talk on **Concurrency** and others

YouTube

Hahicorp

Talks and Code

YouTube GitHub