Go for SREs Using Python

...

Andrew Hamilton

What makes Go fun to work with?

Go is expressive, concise, clean, and efficient

It's a fast, statically typed, compiled language that feels like a dynamically typed, interpreted language

Relatively new but stable

Easy to build CLI tool or API quickly

Fast garbage collection

https://golang.org/doc/

Hello world

```
// Go
package main

import “fmt”

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

```
#!/usr/bin/env python3
# python3

def main():
    print(“Hello world”)

if __name__ == “__main__”:
    main()
```

```
$ go build hello_world.go
$ ./hello_world
Hello world
$
```

```
$ python3 hello_world.py
Hello world
$
```

Good default tool chain

Formatting (`gofmt` and `go import`) [`pylint`]

Testing (`go test`) [`pytest, nose`]

Race Detection (`go build -race`)

Source Code Checking (`go vet` and `go oracle`)

Help with refactoring (`go refactor`)

BUT...

There’s currently no official debugger

Debuggers can be very helpful

There are some unofficial debuggers

https://github.com/mailgun/godebug

Standard library

Does most of what you'll need already

Built for concurrency and parallel processing where useful

Quick to add and update features (HTTP2 support) but doesn't break your code

Expanding third-party libraries

Third-party libraries are just other repositories

Easy to build and propagate by pushing to Github or Bitbucket, etc

List of dependencies not kept in a separate file but pulled from imports

Versioning dependencies is done by vendoring within your project repository

No PyPI equivalent

Importing

```
import (
    “fmt”
    “net/http”

    “github.com/ahamilton55/some_lib/helpers”

    “github.com/gorilla/mux”
    “github.com/gorilla/session”
)
```

Getting dependencies

`go get`

Downloads dependencies under `$GOPATH`

Third-party dependencies are local to your `$GOPATH`

Vendoring

```
$GOPATH/src/github.com/ahamilton55/my_project/
                                              main.go
                                              my_lib/
                                                  lib_file.go
                                              vendor/
                                                  github.com/
                                                      ahamitlon55/
                                                          some_lib/
                                                              helpers/
                                                                  …
                                                      gorilla/
```

Dependency management only for the developer

Dependencies only required for compilation

Removes dependency management from the user

Simplified error checking

No catching exceptions

Common construct to return values and an error from most functions

Checking for an err

```
// Go
rtn, err := someFunction(val1, val2)
if err != nil {
     // do something here
     return
}

// continue on and believe in a the
// world
```

```
# Python
try:
    rtn = some_function(val1, val2)
except some_exception as e:
    # do something here hoping
    # you’re catching all raised
    # exceptions
    return

# move along and be happy
```

Documentation that I can quickly understand

Easy to figure out how to use a function

Interactive examples available inside of the documentation

Docs can be generated from code comments with “godoc”

All docs (including third-party) available offline inside of a browser (`godoc -http ":8123"`)

Documentation example

func **Serve**

```
func Serve(l net.Listener, handler Handler) error
```

Serve accepts incoming HTTP connections on the listener l, creating a new service goroutine for each. The service goroutines read requests and then call handler to reply to them. Handler is typically nil, in which case the DefaultServeMux is used.

https://golang.org/pkg/net/http/#Serve

Simple concurrency and parallelism

Will utilize all of the cores available on the host by default

Easily run functions in parallel by using goroutines (e.g. `go MyFunc()`)

Easily run 1000s of goroutines

Send data to goroutines using channels

Simple goroutine example

```
package main

import (
	"fmt"
	"time"
)

func output(name string, in chan int) {
	for {
		select {
		case val := <-in:
			fmt.Printf("%s: %d\n", name, val)
		default:
			time.Sleep(10 * time.Millisecond)
		}
	}
}
```

```
func main() {
	in := make(chan int)
	go output("1", in)
	go output("2", in)
	for i := 1; i <= 15; i++ {
		in <- i
	}
	time.Sleep(100 * time.Millisecond)
}
```

Example output

```
$ go run blah.go
2: 1
1: 2
2: 4
1: 3
1: 5
2: 6
1: 7
2: 9
1: 8
1: 10
1: 11
2: 12
2: 13
1: 14
1: 15
```

A single, self-contained binary is produced

Stupid errors caught by the compiler (misspelled/unused vars, missing import)

Allows for cross compilation for different platforms

Produces a single binary

Great for distribution

Why is a binary nice?

Paged at 2am

Look at the runbook and told to run a script

Looks like your host doesn't currently have all of the required dependencies

Run `pip install` on your system

Looks like it conflicts with another version

Create a virtualenv, activate it, run `pip install`

Try again and hope that it works this time.

Why is a binary nice?

Doing this at 2am sucks when you just want to fix the issue

Unfortunately users might not always test every script and tool on our system before it is needed

If you can build a binary and upload it to something like S3, operator can download a prebuilt and **verified** tool

But what about PEX?

PEP441 or magic zip files

PEX are cool and could help

PEX and pants/bazel/buck can be complex

Can require some rewriting

Deploying your code

// Go web app

Copy binary

Copy other files (config, templates, resources, etc)

Start up service

Place behind load balancer

Python web app

Copy over project files

Install required dependencies

Copy other files (config, templates, resources, etc)

Install nginx and gunicorn

Configure nginx and gunicorn

Start up nginx and gunicorn services

Place behind load balancer

But, but containers...

Can simplify this workflow

Added dependencies tend to bloat images and may require larger base images

Starting a docker container can be more than just “`docker run`”, also “`docker pull`”

A Go container can be about the size of the binary, Python will be a bit more than just the source

It’s already used in some cool tools

Kubernetes (CNCF, Google)

Docker CLI and tools such as Distribution (Docker)

Packer, Consul, Terraform, etc (HashiCorp)

etcd (CoreOS)

NATS (Apcera)

Hugo (Steve Francia, a.k.a. spf13)

Thanks

@ahamilton55