**Go vs Python**

**Hello World**

**Python**

**print**("Hello world")

**Go**

**package** main

**import** "fmt"

**func** main() {

fmt.Println("Hello world")

}

**Print**

**Python**

**print**("Some string")

**print**("Some string", end="") *# no newline character printed*

**print**("Name: {}, Age: {}".format("Peter", 35))

**Go**

**package** main

**import** "fmt"

**func** main() {

fmt.Println("Some string")

fmt.Print("Some string")

fmt.Printf("Name: %s, Age: %d\n", "Peter", 35)

}

**Comments**

**Python**

*"""This is a doc string for the whole module"""*

*# This is a inline comment*

**class** **Class**(object):

*"""This is the doc string for the class"""*

**print**(\_\_doc\_\_)

**print**(Class.\_\_doc\_\_)

**Go**

**package** main

*// This is a general comment*

*/\* This is also a comment*

*but on multiple lines.*

*\*/*

*/\* This is the multi-line comment for the function main().*

*To get access to this from the command line, run:*

*godoc comments.go*

*\*/*

**func** main() {

}

**Multiline Strings**

**Python**

**print**(

*"""This is*

*a multi-line string.*

*"""*

)

**print**("O'word " 'Another "word" ' "Last word.")

**Go**

**package** main

**import** "fmt"

**func** main() {

fmt.Println(`This is

a multi-line string.

`)

fmt.Println(

"O'word " +

"Another \"word\" " +

"Last word.")

}

**Lists**

A slice is a segment of an array whose length can change.

The major difference between an array and a slice is that with the array you need to know the size up front. In Go, there is no way to equally easily add values to an existing sliceso if you want to easily add values, you can initialize a slice at a max length and incrementally add things to it.

**Python**

*# initialize list*

numbers = [0] \* 5

*# change one of them*

numbers[2] = 100

some\_numbers = numbers[1:3]

**print**(some\_numbers) *# [0, 100]*

*# length of it*

**print**(len(numbers)) *# 5*

*# initialize another*

scores = []

scores.append(1.1)

scores[0] = 2.2

**print**(scores) *# [2.2]*

**Go**

**package** main

**import** "fmt"

**func** main() {

*// initialized array*

**var** numbers [5]int *// becomes [0, 0, 0, 0, 0]*

*// change one of them*

numbers[2] = 100

*// create a new slice from an array*

some\_numbers := numbers[1:3]

fmt.Println(some\_numbers) *// [0, 100]*

*// length of it*

fmt.Println(len(numbers))

*// initialize a slice*

**var** scores []float64

scores = append(scores, 1.1) *// recreate to append*

scores[0] = 2.2 *// change your mind*

fmt.Println(scores) *// prints [2.2]*

*// when you don't know for sure how much you're going*

*// to put in it, one way is to*

**var** things [100]string

things[0] = "Peter"

things[1] = "Anders"

fmt.Println(len(things)) *// 100*

}

**Maps**

You can make a map of maps with:

elements : make(**map**[string]**map**[string]int)

elements["H"] = **map**[string]int{

"protons": 1,

"neutrons": 0,

}

But note, this is what you have struct for.

**Python**

elements = {}

elements["H"] = 1

**print**(elements["H"]) *# 1*

*# remove by key*

elements["O"] = 8

elements.pop("O")

*# do something depending on the being there*

**if** "O" **in** elements:

**print**(elements["O"])

**if** "H" **in** elements:

**print**(elements["H"])

**Go**

**package** main

**import** "fmt"

**func** main() {

elements := make(**map**[string]int)

elements["H"] = 1

fmt.Println(elements["H"])

*// remove by key*

elements["O"] = 8

delete(elements, "O")

*// only do something with a element if it's in the map*

**if** number, ok := elements["O"]; ok {

fmt.Println(number) *// won't be printed*

}

**if** number, ok := elements["H"]; ok {

fmt.Println(number) *// 1*

}

}

**Booleans**

Go doesn't have a quick way to evaluate if something is ["truthy"](http://en.wikipedia.org/wiki/Truthiness). In Python, for example, you can use an if statement on any type and most types have a way of automatically converting to True or False. For example you can do:

x = 1

**if** x:

**print** "Yes"

y = []

**if** y:

**print** "this won't be printed"

This is not possible in Go. You really need to do it explicitly for every type:

x := 1

**if** x != 0 {

fmt.Println("Yes")

}

**var** y []string

**if** len(y) != 0 {

fmt.Println("this won't be printed")

}

**Python**

**print**(True **and** False) *# False*

**print**(True **or** False) *# True*

**print**(**not** True) *# False*

**Go**

**package** main

**import** "fmt"

**func** main() {

fmt.Println(**true** && **false**) *// false*

fmt.Println(**true** || **false**) *// true*

fmt.Println(!**true**) *// false*

x := 1

**if** x != 0 {

fmt.Println("Yes")

}

**var** y []string

**if** len(y) != 0 {

fmt.Println("this won't be printed")

}

}

**Forloop**

Go has only one type of loop and that's the for loop.

**Python**

i = 1

**while** i <= 10:

**print**(i)

i += 1

*# ...or...*

**for** i **in** range(1, 11):

**print**(i)

**Go**

**package** main

**import** "fmt"

**func** main() {

i := 1

**for** i <= 10 {

fmt.Println(i)

i += 1

}

*// same thing more but more convenient*

**for** i := 1; i <= 10; i++ {

fmt.Println(i)

}

}

**Range**

**Python**

names = ["Peter", "Anders", "Bengt"]

**for** i, name **in** enumerate(names):

**print**("{}. {}".format(i + 1, name))

**Go**

**package** main

**import** "fmt"

**func** main() {

names := []string{

"Peter",

"Anders",

"Bengt",

}

*/\* This will print*

*1. Peter*

*2. Anders*

*3. Bengt*

*\*/*

**for** i, name := **range** names {

fmt.Printf("%d. %s\n", i+1, name)

}

}

**Switch**

**Python**

**def** input\_():

**return** int(input())

number = input\_()

**if** number == 8:

**print**("Oxygen")

**elif** number == 1:

**print**("Hydrogen")

**elif** number == 2:

**print**("Helium")

**elif** number == 11:

**print**("Sodium")

**else**:

**print**("I have no idea what **%d** is" % number)

*# Alternative solution*

number = input\_()

db = {1: "Hydrogen", 2: "Helium", 8: "Oxygen", 11: "Sodium"}

**print**(db.get(number, "I have no idea what **%d** is" % number))

**Go**

**package** main

**import** (

"fmt"

"strconv"

)

**func** str2int(s string) int {

i, err := strconv.Atoi(s)

**if** err != **nil** {

panic("Not a number")

}

**return** i

}

**func** main() {

**var** number\_string string

fmt.Scanln(&number\_string)

number := str2int(number\_string)

**switch** number {

**case** 8:

fmt.Println("Oxygen")

**case** 1:

fmt.Println("Hydrogen")

**case** 2:

fmt.Println("Helium")

**case** 11:

fmt.Println("Sodium")

**default**:

fmt.Printf("I have no idea what %d is\n", number)

}

*// Alternative solution*

fmt.Scanln(&number\_string)

db := **map**[int]string{

1: "Hydrogen",

2: "Helium",

8: "Oxygen",

11: "Sodium",

}

number = str2int(number\_string)

**if** name, exists := db[number]; exists {

fmt.Println(name)

} **else** {

fmt.Printf("I have no idea what %d is\n", number)

}

}

**Variadic Functions**

In Python you can accept varying types with somefunction(\*args) but this is not possible with Go. You can however, make the type an interface thus being able to get much more rich type structs.

**Python**

**def** average(\*numbers):

**return** sum(numbers) / len(numbers)

**print**(average(1, 2, 3, 4)) *# 10/4 = 2.5*

**Go**

**package** main

**import** "fmt"

**func** average(numbers ...float64) float64 {

total := 0.0

**for** \_, number := **range** numbers {

total += number

}

**return** total / float64(len(numbers))

}

**func** main() {

fmt.Println(average(1, 2, 3, 4)) *// 2.5*

}

**Time Elapsed**

**Python**

**import** **time**

t0 = time.time()

time.sleep(3.5) *# for example*

t1 = time.time()

**print**("Took {:.2f} seconds".format(t1 - t0))

**Go**

**package** main

**import** "fmt"

**import** "time"

**func** main() {

t0 := time.Now()

elapsed := time.Since(t0)

fmt.Printf("Took %s", elapsed)

}

**Closure Functions**

Note in the Python example you can access number in the inner function but you can't change it. Suppose you wanted to do this:

**def** increment(amount):

number += amount

increment(1)

increment(2)

Then you would get a UnboundLocalError error because the variable would be tied to the inner scope of the increment function.

Note: you can use the global statement, to get around that, example

**def** increment(amount):

**global** number

number += amount

increment(1)

increment(2)

**Python**

**def** run():

**def** increment(amount):

**return** number + amount

number = 0

number = increment(1)

number = increment(2)

**print**(number) *# 3*

run()

**Go**

**package** main

**import** "fmt"

**func** main() {

number := 0

*/\* It has to be a local variable like this.*

*You can't do `func increment(amount int) {` \*/*

increment := **func**(amount int) {

number += amount

}

increment(1)

increment(2)

fmt.Println(number) *// 3*

}

**Defer**

The cool thing about defer in Go is that you can type that near where it matters and it's then clear to the reader that it will do that later.

In Python you can sort of achive the same thing by keeping the content between the try: and the finally: block short.

**Python**

f = open("defer.py")

**try**:

f.read()

**finally**:

f.close()

**Go**

**package** main

**import** (

"os"

)

**func** main() {

f, \_ := os.Open("defer.py")

**defer** f.Close()

*// you can now read from this*

*// `f` thing and it'll be closed later*

}

**Panic Recover**

**Python**

**try**:

**raise** **Exception**("Shit")

**except** **Exception** **as** e:

**print**("error was:", e)

**Go**

**package** main

**import** "fmt"

**func** main() {

*// Running this will print out:*

*// error was: Shit!*

**defer** **func**() {

fmt.Println("error was:", recover())

}()

panic("Shit!")

}

**Mutables**

Python doesn't have the concept of pointers. Go does. But with Go you can send an array or a map into a function, have it modified there without being returned and it gets changed.

**Python**

**def** upone(mutable, index):

mutable[index] = mutable[index].upper()

list\_ = ["a", "b", "c"]

upone(list\_, 1)

**print**(list\_) *# ['a', 'B', 'c']*

dict\_ = {"a": "anders", "b": "bengt"}

upone(dict\_, "b")

**print**(dict\_) *# {'a': 'anders', 'b': 'BENGT'}*

**Go**

**package** main

**import** (

"fmt"

"strings"

)

**func** upone\_list(thing []string, index int) {

thing[index] = strings.ToUpper(thing[index])

}

**func** upone\_map(thing **map**[string]string, index string) {

thing[index] = strings.ToUpper(thing[index])

}

**func** main() {

*// mutable*

list := []string{"a", "b", "c"}

upone\_list(list, 1)

fmt.Println(list) *// [a B c]*

*// mutable*

dict := **map**[string]string{

"a": "anders",

"b": "bengt",

}

upone\_map(dict, "b")

fmt.Println(dict) *// map[a:anders b:BENGT]*

}

**Structs**

**Python**

**from** **math** **import** sqrt

**class** **Point**(object):

**def** \_\_init\_\_(self, x, y):

self.x = x

self.y = y

**def** distance(point1, point2):

**return** sqrt(point1.x \* point2.x + point1.y \* point2.y)

p1 = Point(1, 3)

p2 = Point(2, 4)

**print**(distance(p1, p2)) *# 3.74165738677*

**Go**

**package** main

**import** (

"fmt"

"math"

)

**type** Point **struct** {

x float64

y float64

}

**func** distance(point1 Point, point2 Point) float64 {

**return** math.Sqrt(point1.x\*point2.x + point1.y\*point2.y)

}

*// Since structs get automatically copied,*

*// it's better to pass it as pointer.*

**func** distance\_better(point1 \*Point, point2 \*Point) float64 {

**return** math.Sqrt(point1.x\*point2.x + point1.y\*point2.y)

}

**func** main() {

p1 := Point{1, 3}

p2 := Point{2, 4}

fmt.Println(distance(p1, p2)) *// 3.7416573867739413*

fmt.Println(distance\_better(&p1, &p2)) *// 3.7416573867739413*

}

**Methods**

**Python**

**from** **math** **import** sqrt

**class** **Point**(object):

**def** \_\_init\_\_(self, x, y):

self.x = x

self.y = y

**def** distance(self, other):

**return** sqrt(self.x \* other.x + self.y \* other.y)

p1 = Point(1, 3)

p2 = Point(2, 4)

**print**(p1.distance(p2)) *# 3.74165738677*

**print**(p2.distance(p1)) *# 3.74165738677*

**Go**

**package** main

**import** (

"fmt"

"math"

)

**type** Point **struct** {

x float64

y float64

}

**func** (this Point) distance(other Point) float64 {

**return** math.Sqrt(this.x\*other.x + this.y\*other.y)

}

*// Dince structs get automatically copied,*

*// it's better to pass it as pointer.*

**func** (this \*Point) distance\_better(other \*Point) float64 {

**return** math.Sqrt(this.x\*other.x + this.y\*other.y)

}

**func** main() {

p1 := Point{1, 3}

p2 := Point{2, 4}

fmt.Println(p1.distance(p2)) *// 3.7416573867739413*

fmt.Println(p1.distance\_better(&p2)) *// 3.7416573867739413*

}

**Goroutines**

Note that when you run these, the numbers come in in different order between runs.

In the Python example, it exits automatically when all requests have finished.

**Python**

**import** **urllib2**

**import** **multiprocessing**

**def** f(url):

req = urllib2.urlopen(url)

**try**:

**print**(len(req.read()))

**finally**:

req.close()

urls = ("https://www.peterbe.com", "https://python.org", "https://golang.org")

**if** \_\_name\_\_ == "\_\_main\_\_":

p = multiprocessing.Pool(3)

p.map(f, urls)

**Go**

**package** main

**import** (

"fmt"

"io/ioutil"

"net/http"

"sync"

)

**func** f(url string) {

response, err := http.Get(url)

**if** err != **nil** {

panic(err)

}

**defer** response.Body.Close()

body, err := ioutil.ReadAll(response.Body)

**if** err != **nil** {

panic(err)

}

fmt.Println(len(body))

}

*// See the example in https://golang.org/pkg/sync/#WaitGroup*

**func** main() {

**var** wg sync.WaitGroup

urls := []string{

"https://www.peterbe.com",

"https://python.org",

"https://golang.org",

}

**for** \_, url := **range** urls {

wg.Add(1)

**go** **func**(url string) {

**defer** wg.Done()

f(url)

}(url)

}

*// Wait for the goroutines to finish*

wg.Wait()

}

**Markdownserver**

Using simple ab (with concurrency):

$ ab -n 10000 -c 10 http://localhost:XXXX/markdown?body=THis+%2Ais%2A+a+string

Where XXXX is the port number depending on which server you're running.

Results:

Python (Flask) 2103.06 [#/sec] (mean)

Python (Tornado) 1834.48 [#/sec] (mean)

Node (Express) 4406.17 [#/sec] (mean)

Go 19539.61 [#/sec] (mean)

To run the Go version, first set your $GOPATH then:

$ go get github.com/russross/blackfriday

$ go run main.go

$ curl http://localhost:8080/markdown?body=THis+%2Ais%2A+a+string

To run the Tornado versions:

$ virtualenv venv

$ source venv/bin/activate

$ pip install tornado mistune markdown

$ python tornado\_.py

$ curl http://localhost:8888/markdown?body=THis+%2Ais%2A+a+string

To run the Flask version:

$ virtualenv venv

$ source venv/bin/activate

$ pip install Flask mistune markdown

$ python flask\_.py

$ curl http://localhost:5000/markdown?body=THis+%2Ais%2A+a+string

To run the NodeJS version:

$ npm install # picks up from package.json

$ node node\_.js

$ curl http://localhost:3000/markdown?body=THis+%2Ais%2A+a+string

**Python**

**try**:

**import** **mistune** **as** **markdown**

**except** **ImportError**:

**import** **markdown** *# py implementation*

**import** **tornado.ioloop**

**import** **tornado.web**

**class** **MarkdownHandler**(tornado.web.RequestHandler):

**def** get(self):

body = self.get\_argument("body")

self.write(markdown.markdown(body))

application = tornado.web.Application([(r"/markdown", MarkdownHandler)])

**if** \_\_name\_\_ == "\_\_main\_\_":

application.listen(8888)

tornado.ioloop.IOLoop.instance().start()

**import** **logging**

**try**:

**import** **mistune** **as** **markdown**

**except** **ImportError**:

**import** **markdown** *# py implementation*

**from** **flask** **import** Flask, request

app = Flask(\_\_name\_\_)

log = logging.getLogger("werkzeug")

log.setLevel(logging.ERROR)

@app.route("/markdown")

**def** markdown\_view():

**return** markdown.markdown(request.args["body"])

**if** \_\_name\_\_ == "\_\_main\_\_":

app.run()

**Go**

**package** main

**import** (

"net/http"

"os"

"github.com/russross/blackfriday"

)

**func** main() {

port := os.Getenv("PORT")

**if** port == "" {

port = "8080"

}

http.HandleFunc("/markdown", GenerateMarkdown)

http.ListenAndServe(":"+port, **nil**)

}

**func** GenerateMarkdown(rw http.ResponseWriter, r \*http.Request) {

markdown := blackfriday.MarkdownCommon(

[]byte(r.FormValue("body")))

rw.Write(markdown)

}

**ORM (Object Relational Mapper)**

This is a comparison between [gorp](https://github.com/coopernurse/gorp) and [sqlalchemy](http://www.sqlalchemy.org/).

Using pq and psycopg2 it creates a bunch of ORM instance objects, then edits them all one by one and then deletes them all. This example assumes PostgreSQL and that the table already exists.

It creates X number of "talks" which has the following column types:

id serial integer

topic varchar(200)

when timestamp

tags array of text

duration real

Then lastly it measures how long it takes to do all the inserts, all the updates and all the deletes.

When running these for **10,000 iterations** on my computer I get the following outputs:

$ python orm.py

insert 3.09894585609

edit 30.3197979927

delete 18.6974749565

TOTAL 52.1162188053

$ go run orm.go

insert 2.542336905s

edit 10.28062312s

delete 6.851942699s

TOTAL 19.674902724s

**Python**

*# \*- coding: utf-8 -\**

**import** **time**

**import** **random**

**import** **datetime**

**from** **sqlalchemy** **import** create\_engine

**from** **sqlalchemy.orm** **import** sessionmaker

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

**from** **sqlalchemy** **import** Column, Integer, String, Float, DateTime, Sequence

**from** **sqlalchemy.dialects** **import** postgresql

HOW\_MANY = 1000

*# import logging*

*# logging.basicConfig()*

*# logger = logging.getLogger('sqlalchemy.engine')*

*# logger.setLevel(logging.INFO)*

Base = declarative\_base()

**class** **Talk**(Base):

\_\_tablename\_\_ = "talks"

id = Column(Integer, Sequence("talks\_id\_seq"), primary\_key=True)

topic = Column(String)

when = Column(DateTime)

tags = Column(postgresql.ARRAY(String))

duration = Column(Float)

**def** \_random\_topic():

**return** random.choice(

(

"No talks added yet",

"I'm working on a branch of django-mongokit that I "

"thought you'd like to know about.",

"I want to learn Gaelic.",

"I'm well, thank you.",

" (Kaw uhn KEU-ra shin KAW-la root uh CHOO-nik mee uhn-royer?)",

"Chah beh shin KEU-ra, sheh shin moe CHYEH-luh uh vah EEN-tchuh!",

"STUH LUH-oom BRISS-kaht-chun goo MAWR",

"Suas Leis a' Ghàidhlig! Up with Gaelic!",

"Tha mi ag iarraidh briosgaid!",

)

)

**def** \_random\_when():

**return** datetime.datetime(

random.randint(2000, 2010),

random.randint(1, 12),

random.randint(1, 28),

0,

0,

0,

)

**def** \_random\_tags():

tags = [

"one",

"two",

"three",

"four",

"five",

"six",

"seven",

"eight",

"nine",

"ten",

]

random.shuffle(tags)

**return** tags[: random.randint(0, 3)]

**def** \_random\_duration():

**return** round(random.random() \* 10, 1)

**def** run():

engine = create\_engine(

"postgresql://peterbe:test123@localhost/fastestdb", echo=False

)

Session = sessionmaker(bind=engine)

session = Session()

session.query(Talk).delete()

t0 = time.time()

*# CREATE ALL*

talks = []

**for** i **in** range(HOW\_MANY):

talk = Talk(

topic=\_random\_topic(),

when=\_random\_when(),

duration=\_random\_duration(),

tags=\_random\_tags(),

)

session.add(talk)

talks.append(talk)

session.commit()

t1 = time.time()

*# EDIT ALL*

**for** talk **in** talks:

talk.topic += "extra"

talk.duration += 1.0

talk.when += datetime.timedelta(days=1)

talk.tags.append("extra")

session.merge(talk)

session.commit()

t2 = time.time()

*# DELETE EACH*

**for** talk **in** talks:

session.delete(talk)

session.commit()

t3 = time.time()

**print**("insert", t1 - t0)

**print**("edit", t2 - t1)

**print**("delete", t3 - t2)

**print**("TOTAL", t3 - t0)

**if** \_\_name\_\_ == "\_\_main\_\_":

run()

**Go**

**package** main

**import** (

"database/sql"

"errors"

"fmt"

"github.com/coopernurse/gorp"

\_ "github.com/lib/pq"

"log"

"math/rand"

*// "os"*

"regexp"

"strings"

"time"

)

**type** StringSlice []string

*// Implements sql.Scanner for the String slice type*

*// Scanners take the database value (in this case as a byte slice)*

*// and sets the value of the type. Here we cast to a string and*

*// do a regexp based parse*

**func** (s \*StringSlice) Scan(src **interface**{}) error {

asBytes, ok := src.([]byte)

**if** !ok {

**return** error(errors.New("Scan source was not []bytes"))

}

asString := string(asBytes)

parsed := parseArray(asString)

(\*s) = StringSlice(parsed)

**return** **nil**

}

**func** ToArray(str []string) string {

L := len(str)

out := "{"

**for** i, s := **range** str {

out += "\"" + s + "\""

**if** i+1 < L {

out += ","

}

}

out += "}"

**return** out

}

*// construct a regexp to extract values:*

**var** (

*// unquoted array values must not contain: (" , \ { } whitespace NULL)*

*// and must be at least one char*

unquotedChar = `[^",\\{}\s(NULL)]`

unquotedValue = fmt.Sprintf("(%s)+", unquotedChar)

*// quoted array values are surrounded by double quotes, can be any*

*// character except " or \, which must be backslash escaped:*

quotedChar = `[^"\\]|\\"|\\\\`

quotedValue = fmt.Sprintf("\"(%s)\*\"", quotedChar)

*// an array value may be either quoted or unquoted:*

arrayValue = fmt.Sprintf("(?P<value>(%s|%s))", unquotedValue, quotedValue)

*// Array values are separated with a comma IF there is more than one value:*

arrayExp = regexp.MustCompile(fmt.Sprintf("((%s)(,)?)", arrayValue))

valueIndex int

)

*// Find the index of the 'value' named expression*

**func** init() {

**for** i, subexp := **range** arrayExp.SubexpNames() {

**if** subexp == "value" {

valueIndex = i

**break**

}

}

}

*// Parse the output string from the array type.*

*// Regex used: (((?P<value>(([^",\\{}\s(NULL)])+|"([^"\\]|\\"|\\\\)\*")))(,)?)*

**func** parseArray(array string) []string {

results := make([]string, 0)

matches := arrayExp.FindAllStringSubmatch(array, -1)

**for** \_, match := **range** matches {

s := match[valueIndex]

*// the string \_might\_ be wrapped in quotes, so trim them:*

s = strings.Trim(s, "\"")

results = append(results, s)

}

**return** results

}

**const** HOW\_MANY = 1000

**func** random\_topic() string {

topics := []string{

"No talks added yet",

"I'm working on a branch of django-mongokit that I thought you'd like to know about.",

"I want to learn Gaelic.",

"I'm well, thank you.",

"(Kaw uhn KEU-ra shin KAW-la root uh CHOO-nik mee uhn-royer?)",

"Chah beh shin KEU-ra, sheh shin moe CHYEH-luh uh vah EEN-tchuh!",

"STUH LUH-oom BRISS-kaht-chun goo MAWR",

"Suas Leis a' Ghàidhlig! Up with Gaelic!",

"Tha mi ag iarraidh briosgaid!",

}

**return** topics[rand.Intn(len(topics))]

}

**func** random\_when() time.Time {

**return** time.Date(

2000+rand.Intn(10),

time.November,

rand.Intn(12),

rand.Intn(28),

0, 0, 0, time.UTC)

}

**func** random\_tags() []string {

tags := []string{

"one",

"two",

"three",

"four",

"five",

"six",

"seven",

"eight",

"nine",

"ten",

}

**return** tags[:rand.Intn(4)]

}

**func** random\_duration() float64 {

**return** rand.Float64() \* 10

}

**func** main() {

dbmap := initDb()

**defer** dbmap.Db.Close()

*// alter sequence talks\_id\_seq restart with 1;*

err := dbmap.TruncateTables()

checkErr(err, "TruncateTables failed")

*// dbmap.TraceOn("[gorp]", log.New(os.Stdout, "myapp:", log.Lmicroseconds))*

t0 := time.Now()

**var** talks [HOW\_MANY]Talk

trans, err := dbmap.Begin()

**if** err != **nil** {

panic(err)

}

*// CREATE*

**for** i := 0; i < HOW\_MANY; i++ {

topic := random\_topic()

when := random\_when()

tags := random\_tags()

duration := random\_duration()

talk := Talk{

Topic: topic,

When: when,

Tags: ToArray(tags),

Duration: duration,

}

err = dbmap.Insert(&talk)

checkErr(err, "Insert failed")

talks[i] = talk

}

trans.Commit()

t1 := time.Since(t0)

t0 = time.Now()

trans, err = dbmap.Begin()

**if** err != **nil** {

panic(err)

}

*// EDIT ALL*

**for** \_, talk := **range** talks {

talk.Topic += "extra"

talk.Duration += 1.0

talk.When = talk.When.Add(time.Hour \* 24)

tags := parseArray(talk.Tags)

talk.Tags = ToArray(append(tags, "extra"))

\_, err := dbmap.Update(&talk)

checkErr(err, "Update failed")

}

trans.Commit()

t2 := time.Since(t0)

t0 = time.Now()

trans, err = dbmap.Begin()

**if** err != **nil** {

panic(err)

}

*// DELETE ALL*

**for** \_, talk := **range** talks {

\_, err = dbmap.Exec("delete from talks where id=$1", talk.Id)

checkErr(err, "Delete failed")

}

trans.Commit()

t3 := time.Since(t0)

fmt.Println("insert", t1)

fmt.Println("edit", t2)

fmt.Println("delete", t3)

fmt.Println("TOTAL", t1+t2+t3)

}

**type** Talk **struct** {

*// db tag lets you specify the column name*

*// if it differs from the struct field*

Id int64 `db:"id"`

Topic string `db:"topic"`

When time.Time `db:"when"`

*// Tags StringSlice*

Tags string `db:"tags"`

Duration float64 `db:"duration"`

}

**func** initDb() \*gorp.DbMap {

*// connect to db using standard Go database/sql API*

*// use whatever database/sql driver you wish*

db, err := sql.Open("postgres", `

user=peterbe dbname=fastestdb

password=test123 sslmode=disable`)

checkErr(err, "sql.Open failed")

*// construct a gorp DbMap*

dbmap := &gorp.DbMap{Db: db, Dialect: gorp.PostgresDialect{}}

*// add a table, setting the table name to 'talks' and*

*// specifying that the Id property is an auto incrementing PK*

dbmap.AddTableWithName(Talk{}, "talks").SetKeys(**true**, "Id")

**return** dbmap

}

**func** checkErr(err error, msg string) {

**if** err != **nil** {

log.Fatalln(msg, err)

}

}

**Args**

To run this:

go run args.go peter anders bengt

And it should output:

PETER

ANDERS

BENGT

**Python**

**import** **sys**

**def** transform(\*args):

**for** arg **in** args:

**print**(arg.upper())

**if** \_\_name\_\_ == "\_\_main\_\_":

transform(\*sys.argv[1:])

**Go**

**package** main

**import** (

"fmt"

"os"

"strings"

)

**func** transform(args []string) {

**for** \_, arg := **range** args {

fmt.Println(strings.ToUpper(arg))

}

}

**func** main() {

args := os.Args[1:]

transform(args)

}

**Import Alias**

This example is a bit silly because you normally don't bother with an alias for short built-ins. It's import appropriate for long import nameslike:

**import** (

pb "github.com/golang/groupcache/groupcachepb"

)

You can also import packages that you won't actually use. E.g.

**import** (

\_ "image/png" *// import can do magic*

)

**Python**

**import** **string** **as** **s**

**print**(s.upper("world"))

**Go**

**package** main

**import** (

"fmt"

s "strings"

)

**func** main() {

fmt.Println(s.ToUpper("world"))

}

**Sprintf**

You might have seen things like fmt.Println("some string") and variations around it. But sometimes you might want to just generate a string using the formatting tools found under fmt without it necessarily going out on stdout. That's what [fmt.Sprintf](http://golang.org/pkg/fmt/" \l "Sprintf) is for.

**Python**

max = 10

**raise** **Exception**(f"The max. number is {max}")

**Go**

**package** main

**import** "fmt"

**func** main() {

max := 10

panic(fmt.Sprintf("The max. number is %d", max))

}

**Uniqify**

The Python version is neat in that it's entirely type agnostic as long as the value supports hashing. I'm sure it's possible to do an equivalent one in Go using interface{}. Patches welcome.

For faster variants in Python see [Fastest way to uniqify a list in Python](http://www.peterbe.com/plog/uniqifiers-benchmark).

For some more thoughts on this, and an example of a implementation that is not in-place check out [this mailing list thread](https://groups.google.com/d/topic/golang-nuts/-pqkICuokio/discussion).

**Python**

**def** uniqify(seq):

seen = {}

unique = []

**for** item **in** seq:

**if** item **not** **in** seen:

seen[item] = 1

unique.append(item)

**return** unique

items = ["B", "B", "E", "Q", "Q", "Q"]

**print**(uniqify(items)) *# prints ['B', 'E', 'Q']*

**Go**

**package** main

**import** "fmt"

**func** uniqify(items []string) []string {

uniq := make([]string, 0)

seen := make(**map**[string]bool)

*// For the highest memory efficiency, do:*

*// seen := make(map[string]struct{})*

*// see: https://stackoverflow.com/questions/37320287/maptstruct-and-maptbool-in-golang*

**for** \_, i := **range** items {

**if** \_, exists := seen[i]; !exists {

uniq = append(uniq, i)

seen[i] = **true**

}

}

**return** uniq

}

**func** main() {

items := []string{"B", "B", "E", "Q", "Q", "Q"}

items = uniqify(items)

fmt.Println(items) *// prints [B E Q]*

}

**Dotdict**

In the Python version you can alternatively be more explicit and use something like:

initials.setdefault(initial, 0)

instead of first checking if the key is there.

Note that in Go, when you set the type to be an int it automatically sets it to 0 upon initialization.

**Python**

initials = {}

**for** name **in** ("peter", "anders", "bengt", "bengtsson"):

initial = name[0]

*# if initial not in initials:*

*# initials[initial] = 0*

initials.setdefault(initial, 0)

initials[initial] += 1

**print**(initials)

*# outputs*

*# {'a': 1, 'p': 1, 'b': 2}*

**Go**

**package** main

**import** "fmt"

**func** main() {

names := []string{"peter", "anders", "bengt", "bengtsson"}

initials := make(**map**[string]int)

**for** \_, name := **range** names {

initial := string(name[0])

initials[initial]++

}

fmt.Println(initials)

*// outputs*

*// map[p:1 a:1 b:2]*

}