Go for Python Programmers

by Shahriar Tajbakhsh at EuroPython 2016

Shahriar Tajbakhsh

Software Engineer @ Osper

github.com/s16h

twitter.com/STajbakhsh

shahriar.svbtle.com

Opposite of P.S.

As I prepared this talk, I realised that it was probably a bad idea…

Why is this talk a bad idea?

It kind of implies writing/using Go as you would write Python; which is bad because it leads to un-idiomatic Go code.

Is it really that bad?

I’m fairly sure it is.

Anyhow…

Talk Structure

1. Quick overview of **history.**

2. Comparison of general **syntax and semantics**.

3. **Ecosystem and tools** of Go and Python.

History

First appeared in 2009.

Influenced by ALGOL 60, Pascal, C, CSP, Modula-2, Squeak, Oberon-2, Alef…

First appeared in 1991.

Influenced by ABC, ALGOL 68, C, C++, Dylan, Haskell, Icon, Java, Lisp, Modula-3, Perl…

Syntax and Semantics

```go
package main

import "fmt"

func main() {
   text := "Hello, world!"
   fmt.Println(text)
}
```

```python
def main():
    text = 'Hello, world!'
    print(text)


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

Package

```go
package main

import "fmt"

func main() {
    text := "Hello, world!"
    fmt.Println(text)
}
```

Every .go file **has** to have a package declaration.

Package

```go
package main

import "fmt"

func main() {
    text := "Hello, world!"
    fmt.Println(text)
}
```

All .go files in the same directory **must** have the same package name.

Import

```go
package main

import "fmt"

func main() {
    text := "Hello, world!"
    fmt.Println(text)
}
```

Usage is very similar to Python.

Import

```go
package main

import "fmt"

func main() {
   text := "Hello, world!"
   fmt.Println(text)
}
```

Each package to be imported is listed on a separate line, inside quotation marks.

Functions

```go
package main

import "fmt"

func main() {
    text := "Hello, world!"
    fmt.Println(text)
}
```

😬

We’ll talk about them later.

Variable Deceleration

```go
package main

import "fmt"

func main() {
    text := "Hello, world!"
    fmt.Println(text)
}
```

`text` is a of type string. That’s inferred by the compiler, in this case.

Types

Four categories:

basic, aggregate, reference and interface

Not quite *categorised* in the same way as Go.

Go-style interfaces don’t really exist Python.

Basic Data Types

int, int8, int16, int32, int64	long
uint, uint8, uint16, uint32, uint64	long
float, float32, float64	float
complex64, complex128	complex
bool	bool
string	str

Aggregate Types

`array`	`array`
`struct`	`~class` (maybe more of a `namedtuple`)

Reference Types

slices	list
maps	dict
channels	🤔

Interface Types

Used to express generalisation or abstractions about the behaviour of other types.

We’ll talk a bit more about them later.

Deceleration and Usage

```
var text string
text = "Some string!"

var count uint = 2

pi := 3.14
```

Storage location, with specific type and an associated name.

Zero Values

```
var text string
text = "Some string!"

var count uint = 2

pi := 3.14
```

`text` is `""` at this point.

Variables declared without an explicit initial value are given their zero value.

Fun with Zero Values

```
counts := make(map[string]int)
input := bufio.NewScanner(os.stdin)
for input.Scan() {
    counts[input.Text()]++
}
```

We would use `Counter` but Go’s zero value results in behaviour that we would get with `defaultdict`.

Functions

```go
func name(parameter-list) (result-list) {
    body
}
```

```python
def name(*args, **kwargs):
    body
```

Functions

```
func Adder(a int, b int) int {
    return a + b
}
```

Example of a useless function.

Functions

```
func Adder(a int, b int) (c int) {
    c = a + b
    return c
}
```

You can also have named results.

Functions

```
func Adder(a int, b int) (c int) {
    c = a + b
    return a + b
}
```

Type of a function is called its *signature*.

It is defined by sequence of parameter types and sequence of result types.

Functions

Like in Python, functions in Go are first-class values. They can be passed around.

They're zero value is `nil`.

Functions

```
func Size() (int, int) {
    return 1, 2
}

width, height := Size()
```

Just like Python, functions can return more than one result.

These functions return a tuple of values.

Errors and Error Handling

```go
result, err = Foo()
if err != nil {
    // It's all good
} else {
    // An error occurred.
}
```

```python
try:
    something...
except:
    handle…
else:
    success...
finally:
    whatever...
```

Errors and Error Handling

```
func main() {
    f := createFile("/tmp/foo.txt")
    defer closeFile(f)
    .
    .
    .
}
```

`Defer` is used to ensure that a function call is performed later in a program’s execution, usually for purposes of cleanup.

Errors and Error Handling

But sometimes, there are genuinely exceptional circumstances. For example, when running out of memory or out-of-bounds array access.

Errors and Error Handling

In these exceptional cases, Go *panics*.

Errors and Error Handling

When Go panics:

1. Normal execution stops.
2. All deferred function (in that goroutine) calls are executed.
3. Program crashes with a log message.

Errors and Error Handling

Although giving up is usually the right response to a panic, it might sometimes make sense to try and recover from it; at least for clean-up.

Errors and Error Handling

```
func Parse(input string) (s *Syntax, err error) {
    defer func() {
        if p := recover(); p != nil {
            err = fmt.Errorf("internal error: %v", p)
        }
    }()
    // ... parser...
}
```

What about OOP?

As we know, Python is object oriented. It has all the fancy stuff: classes, inheritance etc.

Go can also be considered object oriented but not in the same way as Python.

OOP in Go

Go says an object is simply a value or variable that has methods, and a method is a function associated with a particular type.

OOP in Go

There is no support for inheritance in Go.

✌🏼

Composition it is.

OOP

```go
type Point struct {
    X float64
    Y float64
}
```

```python
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

OOP

```go
type Point struct {
    X float64
    Y float64
}

func (p Point) Distance(q Point) float64 {
    return math.Hypot(q.X-p.X, q.Y-p.Y)
}
```

```python
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, other):
        return math.sqrt(
            (other.x - self.x) ** 2 +
            (other.y - self.y) ** 2
        )
```

OOP

As mentioned, Go doesn’t have inheritance. But it composes types by struct embedding.

Composes *what* by *what whatting*!?

Struct Embedding

```go
type Point struct {
    X float64
    Y float64
}

type NamedPoint struct {
    Point
    Name string
}
```

Struct Embedding

```go
point := Point{1, 2}
namedPoint := NamedPoint(point, "Osper")

fmt.Println(namedPoint.X) // 1.0
fmt.Println(namedPoint.Distance(point)) // 0.0
fmt.Println(namedPoint.Name) // Osper
```

Anything else OOP-esque?

🤔

Anything else OOP-esque?

I mentioned Go interfaces earlier.

Conceptually, they are in fact very similar to duck-typing in Python.

Interfaces

A type *satisfies* an interface if it posses all the methods the interface requires.

Interfaces

```go
type Writer interface {
    Write(p []byte) (n int, err error)
}

type Reader interface {
    Read(p []byte) (n int, err error)
}

type ReadWriter interface {
    Reader
    Writer
}
```

Concurrency

Go’s support for concurrency is considered one of its strengths.

In Python…LOL (I joke!)

Concurrency

1. goroutines (Communicating Sequential Processes)

2. Traditional shared memory.

threading (ROFL), multiprocessing, asyncio…

Goroutines

Light-weight threads managed by the go runtime.

To start a new goroutine, just prepend **go** to a function call.

Goroutines

Light-weight threads managed by the go runtime.

To start a new goroutine, just prepend go to a function call.

Goroutines

```go
package main

import (
    "fmt"
    "time"
)

func WasteTime(delay time.Duration) {
    time.Sleep(delay)
    fmt.Println("Time wasted!")
}

func main() {
    go WasteTime(2000 * time.Millisecond)
    fmt.Println("End of main()")
    time.Sleep(4000 * time.Millisecond)
}
```

Channels

Channels are a typed “buffer” through which you can send and receive values between goroutines.

Channels

```go
package main

import "fmt"

func main() {
    // create new channel of type int
    ch := make(chan int)

    // start new anonymous goroutine
    go func() {
        // send 42 to channel
        ch <- 42
    }()

    // read from channel
    fmt.Println(<-ch)
}
```

Ecosystem and Tools

Testing

```
$ go test …
```

unittest is pretty good.
py.test is sweet.

Lots of really good and mature tools.

Testing

```
$ go test …
```

By convention, files whose name ends in `_test.go` are test files.

Code Formatting

```
$ go fmt source.go
```

PEP 8

Use tools such as flake8

Package Management

```
$ go get package
```

Will fetch a remote packages, compile it and install it.

Quite a few different tools one can use (e.g. pip).

Some think it’s a mess.

Package Management

`$GOPATH` environment variable used to specify the location of your workspace.

`virtualenv` is widely used for managing per-project dependencies.

Documentation Generation

```
$ go doc …
```

Godoc extracts and generates documentation for Go programs.

Different tools for automatic and manual doc generation (e.g. Sphinx, autodoc, PyDoc etc.).

Conclusion

😁

That’s all, Thanks!

Shahriar Tajbakhsh

Software Engineer @ Osper

github.com/s16h

twitter.com/STajbakhsh

shahriar.svbtle.com

Q&PA

Questions and Possible Answers