Go for Python Programmers
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...


Influenced by ABC, ALGOL 68, C, C++, Dylan, Haskell, Icon, Java, Lisp, Modula-3, Perl...
Syntax and Semantics
package main

import "fmt"

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

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

    if __name__ == '__main__':
        main()
Every .go file has to have a package declaration.
All .go files in the same directory must have the same package name.
package main

import "fmt"

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

import "fmt"

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

We’ll talk about them later.

```go
package main

import "fmt"

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

import "fmt"

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

text is a 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

<table>
<thead>
<tr>
<th>Python Types</th>
<th>Python Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>int, int8, int16, int32, int64</td>
<td>long</td>
</tr>
<tr>
<td>uint, uint8, uint16, uint32, uint64</td>
<td>long</td>
</tr>
<tr>
<td>float, float32, float64</td>
<td>float</td>
</tr>
<tr>
<td>complex64, complex128</td>
<td>complex</td>
</tr>
<tr>
<td>bool</td>
<td>bool</td>
</tr>
<tr>
<td>string</td>
<td>str</td>
</tr>
</tbody>
</table>
## Aggregate Types

<table>
<thead>
<tr>
<th>Aggregate Type</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>array</td>
</tr>
<tr>
<td>struct</td>
<td>~class (maybe more of a namedtuple)</td>
</tr>
</tbody>
</table>
## Reference Types

<table>
<thead>
<tr>
<th>Python Logo</th>
<th>Python 3 Logo</th>
</tr>
</thead>
<tbody>
<tr>
<td>slices</td>
<td>list</td>
</tr>
<tr>
<td>maps</td>
<td>dict</td>
</tr>
<tr>
<td>channels</td>
<td>😞</td>
</tr>
</tbody>
</table>
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

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

var count uint = 2

pi := 3.14
```

Storage location, with specific type and an associated name.
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

```
func name(parameter-list) (result-list) {
  body
}
def name(*args, **kwargs):
  body
```
Functions

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

Example of a useless function.
Functions

You can also have named results.
Functions

Type of a function is called its *signature*.

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

```go
def Adder(a int, b int) (c int) {
    c = a + b
    return a + b
}
```
Functions

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

They’re zero value is `nil`.
functions can return more than one result.

These functions return a tuple of values.
Errors and Error Handling

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

```
try:
    something...
except:
    handle...
else:
    success...
finally:
    whatever...
```
Errors and Error Handling

```go
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

```go
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.
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

type Point struct {
    X float64
    Y float64
}

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

```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)
```

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

func (p Point) Distance(q Point) float64 {
}
```
As mentioned, Go doesn’t have inheritance. But it composes types by struct embedding.

Composes *what* by *what whatting!*?
Struct Embedding

type Point struct {
    X float64
    Y float64
}

type NamedPoint struct {
    Point
    Name string
}
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 possesses all the methods the interface requires.
Interfaces

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
}
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

package main

import {
  "fmt"
  "time"
}

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

func main() {
  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.
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
$ 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
$ 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.
$ 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!
Questions and Possible Answers