(PYTHON) ()

Functional Programming in Python

TODAY'S SLIDES

http://goo.gl/iRXD0Y

ABOUT ME

+ **Colin Su** a.k.a **LittleQ**
+ http://about.me/littleq
+ Production Experience
  - Erlang
  - Python
  - Java{,Script}

(PYTHON)

PROGRAMMING IN PYTHON

+ Imperative Programming
  - Shell Scripting
+ Procedural Programming
  - Languages with functions
+ Declarative Programming
  - Functional Programming

(PYTHON) 

FUNCTIONAL PROGRAMMING

+ No state
+ Immutable data
+ Function as first-class citizen
+ Higher-order function
+ Purity
+ Recursion, tail recursion
+ ...the list is too long to name here

(PYTHON) 

IP VS FP

Your Terminal

```
$ ./program1
$ ./program2 --arg=1
$ ./program3
```

Imperative

```
$ ./program1 | ./program2 --arg=1 | ./program3
```

Functional

(PYTHON) 

FUNCTION AS FIRST CLASS OBJ

```python
def add(a, b):
    return a + b

add2 = add

add2(1,2) # 3

def giveMeAdd():
    def add(a, b):
        return a + b
    return add
```

(PYTHON) 

LAMBDA SUPPORT

```
add = lambda a, b : a + b

minus = lambda a, b : a - b

multiply = lambda a, b : a * b

divide = lambda a, b : a / b
```

(PYTHON) 

(POOR) LAMBDA SUPPORT

```
add = lambda a, b:
    c = a + b
    return c
```

(PYTHON) 

(POOR) LAMBDA SUPPORT

```
add = lambda a, b:
    c = a + b
    return c
```

```
  File "test.py", line 29
    add = lambda a, b:
                      ^
SyntaxError: invalid syntax
```

(PYTHON)

HIGHER-ORDER FUNCTION

+ Function eats functions, and returns functions

```
def food():
    return "food got eaten"
drink = lambda: "drink got drunk"

def me(*args):
    def eat():
        return map(lambda x: x(), args)
    return eat

act = me(food, drink)
print act()


>>> ['food got eaten', 'drink got drunk']
```

(PYTHON) 

MAP/REDUCE/FILTER

+ A war between readability and conciseness
+ Made your code pluggable

(PYTHON) 

CALCULATE "1++2+3+++4+5"

(PYTHON) 

IMPERATIVE.PY

```
INPUT = "1+2++3+++4++5+6+7++8+9++10"
result = 0

for num in INPUT.split('+'):
    if num:
        result += int(num)

print result
```

Iterate on the same variable to perform task

(PYTHON) 

FUNCTIONAL.PY

```
from operator import add
INPUT = "1+2++3+++4++5+6+7++8+9++10"

print reduce(add, map(int, filter(bool, INPUT.split('+'))))
```

(PYTHON) 

FUNCTIONAL PYTHON MADE SIMPLE

(PYTHON) 

BUT NOT ALL LANGUAGES DO...

(PYTHON) 

IMPERATIVE.JAVA

```
Multiset<Integer> lengths = HashMultiset.create();

for (String string : strings) {
  if (CharMatcher.JAVA_UPPER_CASE.matchesAllOf(string)) {
    lengths.add(string.length());
  }
}
```

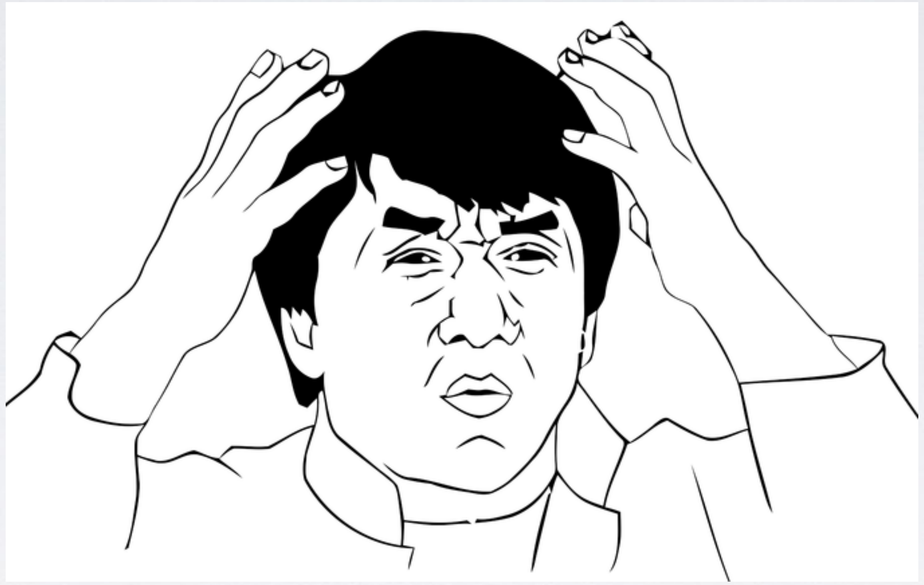
Looks making sense

(PYTHON) 

FUNCTIONAL.JAVA

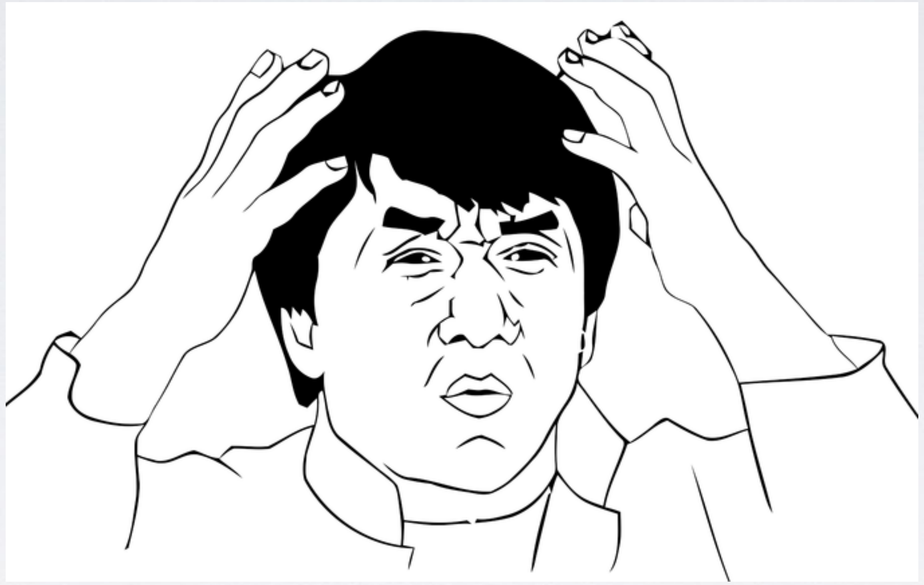
```
Function<String, Integer> lengthFunction = new Function<String, Integer>() {
  public Integer apply(String string) {
    return string.length();
  }
};

Predicate<String> allCaps = new Predicate<String>() {
  public boolean apply(String string) {
    return CharMatcher.JAVA_UPPER_CASE.matchesAllOf(string);
  }
};

Multiset<Integer> lengths = HashMultiset.create(
  Iterables.transform(Iterables.filter(strings, allCaps), lengthFunction));
```

(PYTHON) 



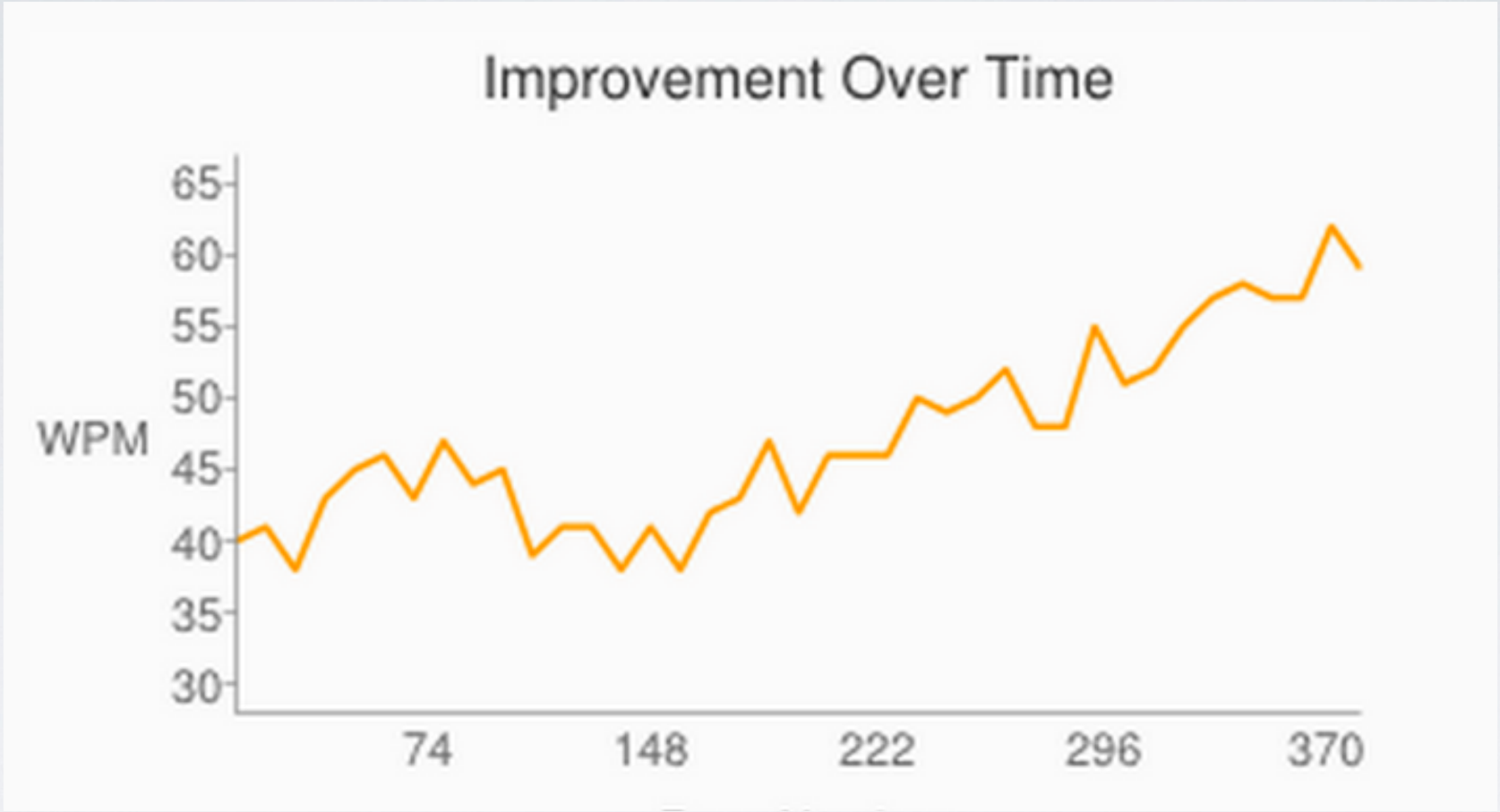
WTF

(PYTHON)



Welcome **T**o **F**unctional

(PYTHON) 



But my typing speed got increased since using Java.

(PYTHON)

PARTIAL FUNCTION APPLICATION

$(((a \times b) \rightarrow c) \times a) \rightarrow (b \rightarrow c)$

PARTIAL FUNCTION APPLICATION

```python
from functools import partial
def f(x, y, z):
    return x + y + z

f2 = partial(f, 100, 10)
print f2(5) #115
```

(PYTHON) 

PARTIAL FUNCTION APPLICATION

+ Keyword-based partial application

```
from functools import partial
from datetime import datetime

def log(message, prefix="", postfix=""):
    print prefix, message, postfix

error = partial(log, postfix=datetime.now(), prefix="[ERROR]")
error("something goes wrong")

# [ERROR] something goes wrong 2014-04-10 01:37:07.250509
```

(PYTHON) 

CURRYING

$((a \times b \times c) \rightarrow d) \rightarrow (((a \rightarrow b) \rightarrow c) \rightarrow d)$

CURRYING

+ Simple sum

```
def simple_sum(a, b):
    return sum(range(a, b+1))

>>> simple_sum(1, 10)
55
```

(PYTHON) 

CURRYING

+ Squared sum

```
def square_sum(a, b):
    return sum(map(lambda x: x**2, range(a,b+1)))

>>> square_sum(1,10)
385
```

(PYTHON) 

CURRYING

+ Square root sum

```
def sqrt_sum(a, b):
    return sum(map(math.sqrt, range(a,b+1)))

>>> sqrt_sum(1,10)
22.4682781862041
```

(PYTHON) 

CURRYING

+ Curried sum()

```python
import math

def fsum(f):
    def apply(a, b):
        return sum(map(f, range(a,b+1)))
    return apply

simple_sum = fsum(int)
square_sum = fsum(lambda x: x**2)
sqrt_sum = fsum(math.sqrt)

print simple_sum(1,10) # 55
print square_sum(1,10) # 385
print sqrt_sum(1,10) # 22.4682781862
```

(PYTHON) 

CURRYING

+ Combined with partial

```
def fsum(f):
    def apply(a, b):
        return sum(map(f, range(a,b+1)))
    return apply

from functools import partial
from operator import mul

mul_sum = fsum(partial(mul, 2))

print mul_sum(1,10) # 110
```

(PYTHON) 

DECORATOR

+ Another way to curry

```
import math

def fsum(f):
    def apply(a, b):
        return sum(map(f, range(a,b+1)))
    return apply

@fsum
def sqrt_sum(x):
    return math.sqrt(x)

print sqrt_sum(1,10) # 22.4682781862
```

(PYTHON) 

GENERIC CURRY IN PYTHON

```python
import functools

def curry(func):
    def curried(*args, **kwargs):
        if not args and not kwargs:
            return func()
        return curry(functools.partial(func, *args, **kwargs))
    return curried

@curry
def add(a, b, c, d, e, f, g):
    return a + b + c + d + e + f + g

add12 = add(1)(2)
add1234 = add12(3)(4)
add1234567 = add1234(5)(6)(7)
print add1234567() # 28
```

(PYTHON) 

TAIL RECURSION

+ No grammer-level support, just simulation

```python
def printAll(strings):
    if strings:
        print strings[0]
        return printAll(strings[1:])

printAll([1,2,3,4,5])
```

Python 2

```python
def printAll(strings):
    if strings:
        head, *tail = strings
        print(head)
        return printAll(tail)

printAll([1,2,3,4,5])
```

Python 3

(PYTHON) 

SO?

(PYTHON) 

FUNCTIONAL PYTHON?

Pros

+ First-class function
+ lambda
+ built-in map/filter/reduce
+ functools
+ generators as lazy-evaluation

Cons

+ non-pure
+ lambda (?)
+ memory-cost operations
+ No optimization for tail recursion
+ No pattern matching

(PYTHON) 

CONCLUSION

+ Python is not for replacing any functional language
+ But a good way to begin functional programming

THINK FUNCTIONAL.

(PYTHON) 

END

No Q&A, thanks (run away)

(PYTHON)