FUNCTIONAL PROGRAMMING WITH PYTHON

BY ALEXEY KACHAYEV, 2012

ABOUT ME

- Position: CTO at Kitapps Inc.
- Production experience: Python, Java, JS, Go, Scala, Clojure, Erlang
- Looked at: Haskell, Lisp, Scheme

GOALS

- Short functional paradigm presentation
- Dispel popular myths about FP
- Characterize Python & FP relations
- ~~Why should I care?~~ How can I make my code better?

MORE **HOW**, LESS **WHY**

ABOUT FUNCTIONAL

- Imperative programming (C/C++, Java)
- Declarative programming
  - Functional programming (Haskell, Scheme, OCaml)
  - Logic programming (Prolog, Clojure core.logic)

- IP = computation in terms of statements that change a program state
- FP = computation as the evaluation of mathematical functions and avoids state and mutable data

ABOUT FUNCTIONAL

- Avoid state
- Immutable data
- First-class functions
- Higher-order functions
- Pure functions
- Recursion, tail recursion
- Iterators, sequences, lazy evaluation, pattern matching, monads....

ABOUT FUNCTIONAL

"IMPERATIVE TERMINAL"

```
$ ./program1
$ ./program2 --param1=1
$ ./program3
```

"FUNCTIONAL TERMINAL"

```
$ ./program1 | ./program2 --param1=1 | ./pro
gram3
```

PROGRAMMING TASK

Calculate partially invalid string with operations: "28+32+++32++39"

PROGRAMMING TASK

Imperative style = actions that change state from initial state to result

```
expr, res = "28+32+++32++39", 0
for t in expr.split("+"):
    if t != "":
        res += int(t)

print res
```

```
"28+32+++32++39", 0
"28", 0
"32", 28
"", 60
"", 60
"32", 60
"", 92
"39", 92
131
```

PROGRAMMING TASK

Functional style = apply transformation (and compositions)

```
from operator import add
expr = "28+32+++32++39"
print reduce(add, map(int, filter(bool, expr
.split("+"))))
```

```
"28+32+++32++39"
["28","32","","","32","","39"]
["28","32","32","39"]
[28,32,32,39]
131
```

MAP/REDUCE/FILTER

- Readability VS. conciseness
- Technical aspects VS. operation substance
- Code reusage ("pluggability")

PYTHON HINTS

MODULE "OPERATOR"

```
>>> operator.add(1,2)
3
>>> operator.mul(3,10)
30
>>> operator.pow(2,3)
8
>>> operator.itemgetter(1)([1,2,3])
2
```

PYTHON HINTS

MODULE "ITERTOOLS"

```
>>> list(itertools.chain([1,2,3], [10,20,30]
))
[1, 2, 3, 10, 20, 30]
>>> list(itertools.chain(*(map(xrange, range
(5)))))
[0, 0, 1, 0, 1, 2, 0, 1, 2, 3]
>>> list(itertools.starmap(lambda k,v: "%s =
> %s" % (k,v),
...                         {"a": 1, "b": 2}.
items()))
['a => 1', 'b => 2']
>>> list(itertools.imap(pow, (2,3,10), (5,2,
3)))
[32, 9, 1000]
>>> dict(itertools.izip("ABCD", [1,2,3,4]))
{'A': 1, 'C': 3, 'B': 2, 'D': 4}
```

CAN WE AVOID LOOPS?

```
>>> name = None
>>> while name is None:
...     name = raw_input()
...     if len(name) < 2:
...         name = None
```

RECURSION

```
def get_name():
    name = raw_input()
    return name if len(name) >= 2 else get_n
ame()
```

TAIL RECURSION?

ERLANG

```
fib(N) -> fib(0,1,N).
fib(_,S,1) -> S;
fib(F,S,N) -> fib(S,F+S,N-1).
```

PYTHON

sorry...

FUNCTIONS

FIRST-CLASS

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

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

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

    return a, b, add
```

FUNCTIONS

HIGH-ORDERED FUNCTION

Pass function as argument

```
map(lambda x: x^2, [1,2,3,4,5])
```

FUNCTIONS

HIGH-ORDERED FUNCTION

Function returns function as result

```
def speak(topic):
    print "My speach is " + topic

def timer(fn):
    def inner(*args, **kwargs):
        t = time()
        fn(*args, **kwargs)
        print "took {time}".format(time=time
()-t)

    return inner

speaker = timer(speak)
speaker("FP with Python")
```

FUNCTIONS

HIGH-ORDERED FUNCTION

I've already saw this...

```
@timer
def speak(topic):
    print "My speach is " + topic

speak("FP with Python")
```

FUNCTIONS

HOW TO WRITE GOOD CODE DEALING WITH FUNCTIONS?

PARTIAL FUNCTION APPLICATION

> *“ The process of fixing a number of arguments to a function, producing another function of smaller arity ”*

SIMPLY-TYPED LAMBDA CALCULUS:

```
papply : (((a × b) → c) × a) → (b → c) = λ(f
, x). λy. f (x, y)
```

Oh, never mind...

PARTIAL FUNCTION APPLICATION

```
def log(level, message):
    print "[{level}]: {msg}".format(level=le
vel, msg=message)


log("debug", "Start doing something")
log("debug", "Continue with something else")
log("debug", "Finished. Profit?")

def debug(message):
    log("debug", message)
```

PARTIAL FUNCTION APPLICATION

Simplify...

```
def log(level, message):
    print "[{level}]: {msg}".format(level=le
vel, msg=message)


from functools import partial
debug = partial(log, "debug")

debug("Start doing something")
debug("Continue with something else")
debug("Finished. Profit?")
```

CURRYING

> *“ The technique of transforming a function that takes multiple arguments in such a way that it can be called as a chain of functions each with a single argument ”*

SIMPLY-TYPED LAMBDA CALCULUS:

```
curry: ((a × b) → c) → (a → (b → c)) = λf. λx. λy. f (x, y)
```

Oh, never mind...

CURRYING

Segment sum

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

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

CURRYING

Squares

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

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

CURRYING

Logarithms

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

>>> log_sum(1,10)
15.104412573075518
```

CURRYING

In general...

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

log_sum = fsum(math.log)
square_sum = fsum(lambda x: x**2)
simple_sum = fsum(int) ## fsum(lambda x: x)
```

```
>>> fsum(lambda x: x*2)(1, 10)
110
>>> import functools
>>> fsum(functools.partial(operator.mul, 2))
(1, 10)
110
```

CURRYING (STANDARD LIBRARY)

```
>>> from operator import itemgetter
>>> itemgetter(3)([1,2,3,4,5])
4

>>> from operator import attrgetter as attr
>>> class Speaker(object):
...     def __init__(self, name):
...         self.name = "[name] " + name
...
>>> alexey = Speaker("Alexey")
>>> attr("name")(alexey)
'[name] Alexey'
```

CURRYING (STANDARD LIBRARY)

```
>>> from operator import methodcaller

>>> methodcaller("__str__")([1,2,3,4,5])
'[1, 2, 3, 4, 5]'
>>> methodcaller("keys")(dict(name="Alexey",
 topic="FP"))
['topic', 'name']

>>> values_extractor = methodcaller("values"
)
>>> values_extractor(dict(name="Alexey", top
ic="FP"))
['FP', 'Alexey']

>>> methodcaller("count", 1)([1,1,1,2,2])
>>> # same as [1,1,1,2,2].count(1)
3
```

GOOD FUNCTION IS SMALL FUNCTION

BAD

```
>>> ss = ["UA", "PyCon", "2012"]
>>> reduce(lambda acc, s: acc + len(s), ss,
0)
11
```

NOT BAD...

```
>>> ss = ["UA", "PyCon", "2012"]
>>> reduce(lambda l,r: l+r, map(lambda s: le
n(s), ss))
11
```

GOOD

```
>>> ss = ["UA", "PyCon", "2012"]
>>> reduce(operator.add, map(len, ss))
11
```

PYTHON HINTS

TYPES ARE CALLABLE

```
>>> map(str, range(5))
['0', '1', '2', '3', '4']
```

CLASSES ARE CALLABLE

```
>>> class Speaker(object):
...     def __init__(self, name):
...         self.name = name
>>> map(Speaker, ["Alexey", "Andrey", "Vsevo
lod"])
[<__main__.Speaker>, <__main__.Speaker>, <__
main__.Speaker>]
```

INSTANCE METHODS ARE CALLABLE

```
>>> map([1,2,3,4,5].count, [1,2,3,10,11])
[1, 1, 1, 0, 0]
```

FUNCTIONS

PURE

```
def is_interesting(topic):
    return topic.contains("FP")
```

NOT PURE

```
def speak(topic):
    print topic
```

PURE ??

```
def set_talk(speaker, topic):
    speaker["talk"] = topic
    return speaker
```

MUTABLE DATA

GLOBAL VARIABLE IS EVIL

and everybody knows why

```
current_speaker = {name: "Alexey Kachayev",
talk: "FP with Python"}

def ask_question(question):
    print "{name}, I have question {question
} about your {talk}"

def quit(reason):
    current_speaker = {name: "Andrey Svetlov
"} # <-- this will fail
    current_speaker["talk"] = "Oh, boring...
" # <-- mutable state
```

MUTABLE DATA

AND WHAT ABOUT LOCAL?

```
def run_conf():
    speaker = {name: "Alexey Kachayev", talk
: "FP with Python"}

    def ask_question(question):
        print "{name}, I have question {ques
tion} about {talk}"

    def quit(reason):
        current_speaker["talk"] = "Oh, borin
g..." # <- same

    name = lambda: speaker["name"]
```

PURITY IS COOL

- map can be pmap
- Less bugs
- Easier to test
- More ways to reuse

CLASSES AND OOP

MUTABILITY...

```
class Speaker(object):
    def __init__(self, name, topic):
        self.name = name
        self.topic = topic

    def ask(self, question):
        print "{name}, {q}".format(name=self
.name, q=question)

    def talk(self):
        print "I'm starting {topic}".format(
topic=self.topic)

me = Speaker("Alexey", "FP with Python")
me.name = "Andrey" # <- WTF???

# or ....
```

THINK FUNCTIONAL: CLASSES AND OOP

MUTABLE DICT + PARTIAL BINDING

```
def ask(self, question):
    print "{name}, {q}?".format(name=self["n
ame"], q=question)

def talk(self):
    print "I'm starting {topic}".format(topi
c=self["topic"])

from functools import partial
def cls(**methods):
    def bind(self):
        return lambda (name, method): (name,
 partial(method, self))
    return lambda **attrs: dict(
        attrs.items() + map(bind(attrs.copy(
)), methods.items())
    )
```

THINK FUNCTIONAL: CLASSES AND OOP

```
>>> me = Speaker(name="Alexey", topic="FP wi
th Python")

>>> me["name"]
'Alexey'
>>> me["topic"]
'FP with Python'
>>> me["talk"]
<functools.partial object at 0x109798d60>
>>> me["talk"]()
I'm starting FP with Python
>>> me["ask"]
<functools.partial object at 0x109798c58>
>>> me["ask"]("WTF")
Alexey, WTF?
```

bindings are immutable

THINK FUNCTIONAL: CLASSES AND OOP

IF YOU NEED MUTABLE IMPLEMENTATION...

```
def cls(**methods):
    def bind(**attrs):
        attrs = attrs.copy()
        attrs.update(dict(map(lambda (n,m):
n, partial(m, attrs),
                              methods.items(
))))
        return attrs
    return bind
```

STOP WRITING CLASSES

- Jack Diederich
- PyCon US 2012
- https://www.youtube.com/watch?v=o9pEzgHorH0

SIMPLIFY YOUR CODE, AVOIDING CLASSES

STOP WRITING CLASSES

```
class Greeting(object):
    def __init__(self, greeting="hello"):
        self.greeting = greeting

    def greet(self, name):
        return "{greet}! {name}".format(gree
t=self.greeting, name)

hola = Greeting("hola")
print hola.greet("bob")

>>> "hola! bob"
```

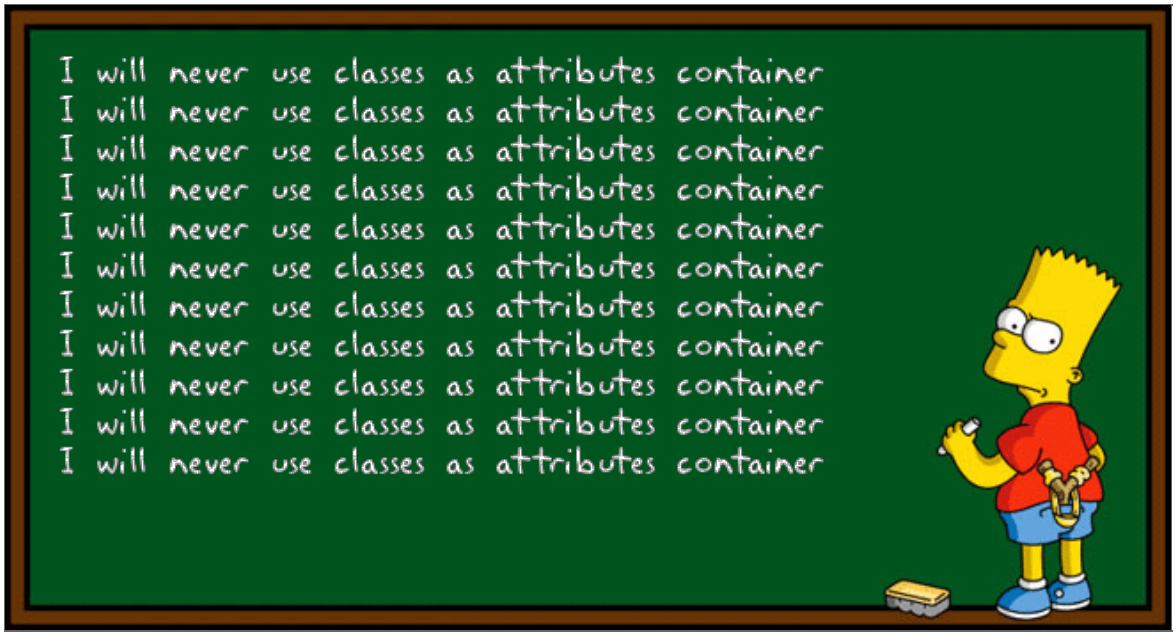
ARE YOU KIDDING ME???

STOP WRITING CLASSES

```
def greet(greeting, target):
    return "{greet}! {name}".format(greet=gr
eeting, name)

hola = functools.partial(greet, "hola")
```

STOP WRITING CLASSES



THINK FUNCTIONAL: DATA

ASSUME THAT WE DON'T HAVE DICT

```
def dct(*items):
    def pair((key, value)):
        return lambda k: value if k == key e
lse None

    def merge(l, r):
        return lambda k: l(k) or r(k)

    return reduce(merge, map(pair, items), p
air((None, None)))
```

```
>>> me = dct(("name", "Alexey"), ("topic", "
FP with Python"))
>>> me("name")
'Alexey'
>>> me("topic")
'FP with Python'
## use this for cls function
>>> me("ask")("WTF")
Alexey, WTF?
```

Sure, I know about complexity...

PYTHON & FP

PRO

- Functions as first-class citizens
- lambda
- Standard library: map/filter/reduce, itertools, operator
- Generators can be used for lazy-evaluation (in some cases)

PYTHON & FP

CON

- Impossible to separate pure / non-pure
- Mutable variables
- Costly mem copy operations

PYTHON & FP

CON

- Imperative style for cycles
- No optimization for tail recursion

PYTHON & FP

CON

- No pattern matching syntax
- Classes-based only type system
- No functions overloading mechanism
- Functions composition is not implemented in stdlib
- Imperative errors handling based on exceptions

PYTHON & FP

CON

PYTHON

```
map(lambda x: x*2, [1,2,3])
```

SCALA

```
List(1,2,3).map(_*2)
```

CLOJURE

```
(map #(* % 2) '(1 2 3))
```

HASKELL

```
map (2*) [1,2,3]
```

WHAT DID WE MISS?

ALMOST EVERYTHING

- Errors handling without exceptions
- Pattern matching
- Message passing
- Functional data structures
- Custom data types
- Lazy evaluation

WHERE CAN I FIND MORE?

- SICP (http://deptinfo.unice.fr/~roy/sicp.pdf)
- Book "Purely Functional Data Structures"
- Book "The Functional Approach to Programming"
- Coursera "Functional Programming Principles in Scala"
- Real World Haskell (http://book.realworldhaskell.org/read/)
- Learn You a Haskell for Great Good! (http://learnyouahaskell.com/)
- Learn You Some Erlang for Great Good! (http://learnyousomeerlang.com/)

THE END

THANK YOU FOR ATTENTION!

- Alexey Kachayev
- Email: kachayev@gmail.com
- Twitter: @kachayev
- Github: kachayev

THIS PRESENTATION:

https://github.com/kachayev/talks