Fn.py

ideas and internals

Fn.py

https://github.com/kachayev/fn.py

About me

- Alexey Kachayev
- CTO at KitApps Inc.
- Open source activist
- Functional programming advocate
- Erlang, Python, Scala, Clojure, Go, Haskell
- @kachayev
- github.com/kachayev
- kachayev at gmail.com

About Python

- minimalistic syntax
- dynamically typed
- imperative (by design)
- “multi paradigm”
- functions are variables
- lambdas
- iterators, generators, decorators, ...
- mutable data types (mostly)

About Fn.py

- enjoy FP in Python
- developed on my own talks materials
- not only proof of concept
- tremendous support from community

Overview

- Scala-style lambdas
- Stream and infinite sequences declaration
- TCO / trampolines
- Itertools recipes
- `Option` monad
- Python 2/3 unification for functional stuff

Scala-style lambdas

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

-- Clojure
(map #(* % 2) [1 2 3])

-- Haskell
map (2*) [1,2,3]

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

Scala-style lambdas

```python
from fn import _
from fn.op import zipwith
from itertools import repeat

assert list(map(_ * 2, range(5))) == [0,2,4,6,8]
assert list(filter(_ < 10, [9,10,11])) == [9]
assert list(zipwith(_ + _)([0,1,2], repeat(10))) == [10,11,12]
```

Scala-style lambdas

```
from fn import _

print (_ + 2)
# "(x1) => (x1 + 2)"
print (_ + _ * _)
# "(x1, x2, x3) => (x1 + (x2 * x3))"
```

Why?

- Code readability
- Improve you vision
- Keyboard will work longer

How?

- `_Callable` class
- `shortcut` instance
- overloaded operators
- each instance - unary or double-side operations tree

Streams and infinite sequences

What?

- Lazy evaluated list
- Composable producers
- Any number of consumers
- All consumers share one iterable origin
- Infinite (potentially) sequence

```python
from fn import Stream

s = Stream() << [1,2,3,4,5]
assert list(s) == [1,2,3,4,5]
assert s[1] == 2
assert list(s[0:2]) == [1,2]

s = Stream() << range(6) << [6,7]
assert list(s) == [0,1,2,3,4,5,6,7]

def gen():
    yield 1
    yield 2
    yield 3

s = Stream() << gen << (4,5)
assert list(s) == [1,2,3,4,5]
```

So what?

Haskell

```
Prelude> let fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
Prelude> take 10 fibs
[0,1,1,2,3,5,8,13,21,34]
Prelude> take 20 fibs
[0,1,1,2,3,5,8,...,610,987,1597,2584,4181]
```

Clojure

```
user> (def fib (lazy-cat [0 1] (map + fib (rest fib))))
user> (take 10 fib)
(0 1 1 2 3 5 8 13 21 34)
user> (take 20 fib)
(0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181)
```

Scala

```
scala> def fibs: Stream[Int] =
...      0 #:: 1 #:: fibs.zip(fibs.tail).map{case (a,b) => a + b}
scala> fibs(10)
res1: Int = 55
scala> fibs.take(10).toArray
res2: Array[Int] = Array(0, 1, 1, 2, 3, 5, 8, 13, 21, 34)
scala> fibs.take(20).toArray
res3: Array[Int] = Array(0, 1, 1, 2, 3, 5, 8, ..., 1597, 2584, 4181)
```

Python

```python
from fn import Stream
from fn.iters import take, drop, map
from operator import add

f = Stream()
fib = f << [0, 1] << map(add, f, drop(1, f))

assert list(take(10, fib)) == [0,1,1,2,3,5,8,13,21,34]
assert fib[20] == 6765
assert fib[30:35] == [832040,1346269,2178309,3524578,5702887]
```

Tail call optimization

“I don't believe in recursion as the basis of all programming. This is a fundamental belief of certain computer scientists, especially those who love Scheme and like to teach programming by starting with a "cons" cell and recursion.” Guido van Rossum

```python
# heavy stack consuming
# this will fail with RuntimeError
def fact(n):
    if n == 0: return 1
    return n * fact(n-1)

# tail call
# will fail the same way
def fact(n, acc=1):
    if n == 0: return acc
    return fact(n-1, acc*n)
```

Okay....

Trampolines

unwrap tail calls to while loop

Recursion = Induction

recursion = induction basis + step

while loop = induction basis + step

Trampoline

```python
from fn import recur

@recur.tco
def fact(n, acc=1):
    # induction basis
    if n == 0: return False, acc
    # next induction step
    # you can pass callable instead of True
    return True, (n-1, acc*n)
```

Control flow patterns

```python
class Request(dict):
    def parameter(self, name):
        return self.get(name, None)

r = Request(testing="Fixed", empty="   ")
param = r.parameter("testing")
if param is None:
    fixed = ""
else:
    param = param.strip()
    if len(param) == 0:
        fixed = ""
    else:
        fixed = param.upper()
```

Scala-style `Option`

```python
from operator import methodcaller
from fn.monad import optionable

class Request(dict):
    @optionable
    def parameter(self, name):
        return self.get(name, None)

r = Request(testing="Fixed", empty="   ")
fixed = r.parameter("testing")
           .map(methodcaller("strip"))
           .filter(len)
           .map(methodcaller("upper"))
           .get_or("")
```

```python
from fn.monad import Option

request = dict(url="face.png", mimetype="PNG")
tp = Option \
        .from_value(request.get("type", None)) \
        .or_call(from_mimetype, request) \
        .or_call(from_extension, request) \
        .get_or("application/undefined")
```

What?

- analog = list with 1 or 0 elements
- functor + fmap
- monadic flatMap operation
- “retry” computations

When?

- computations that may fail
- heavy execution branching (if/else/try/etc)

Where to find more?

https://github.com/kachayev/fn.py

Questions?

http://kachayev.github.com/talks