LAZY EVALUATION AND DECLARATIVE APPROACH IN PYTHON

BY ALEXEY KACHAYEV, 2012

ABOUT ME

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

GOALS

- Short functional paradigm presentation
- Other point of view on algorithmic problems
- Python iterators/generators: advanced usage
- Lazy sequences to work with infinite data structures

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

(read this presentation only with **code samples**!)

ABOUT DECLARATIVE PROGRAMMING

- 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

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", 1
for t in expr.split("*"):
    if t != "":
        res *= int(t)

print res
```

```
"28*32***32**39", 1
"28", 1
"32", 28
"", 896
"", 896
"32", 896
"", 28672
"39", 28672
1118208
```

PROGRAMMING TASK

Functional style = apply transformation (and compositions)

```
from operator import mul
expr = "28*32***32**39"
print reduce(mul, map(int, filter(bool, expr.split("*"))), 1
)
```

```
"28*32***32**39"
["28","32","","","32","","39"]
["28","32","32","39"]
[28,32,32,39]
1118208
```

FUNCTIONAL IS ABOUT...

- Avoid state
- **Immutable data**
- First-class, higher-order functions
- Purity
- Recursion, tail recursion
- **Iterators, sequences**
- **Lazy evaluation**
- Pattern matching, monads....

(our focus **in red**)

WIKIPEDIA: LAZY EVALUATION

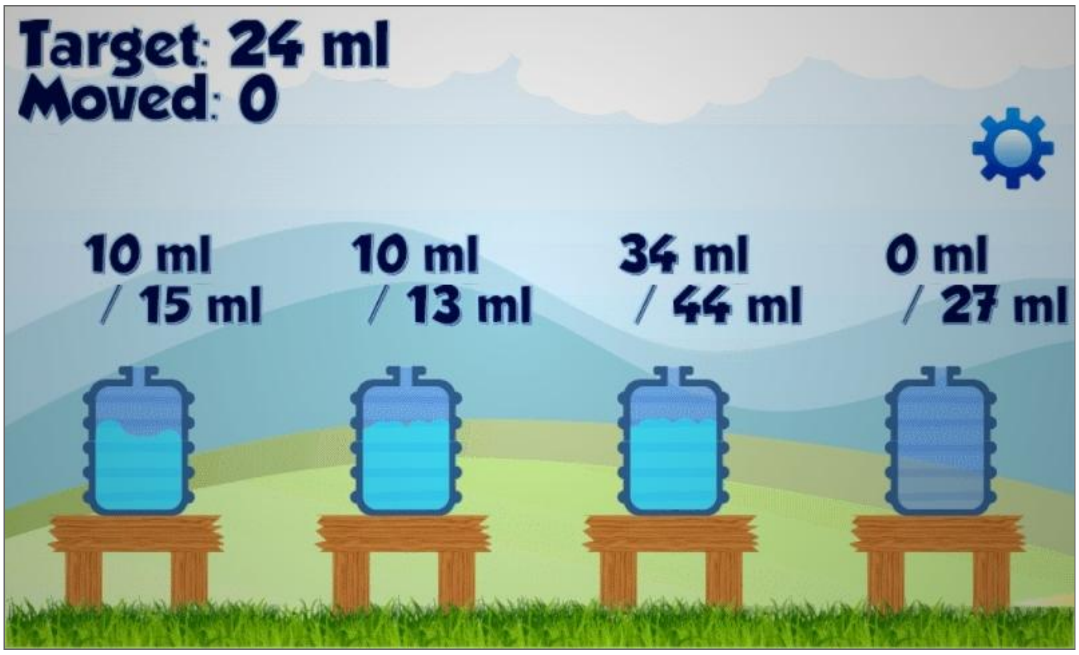
“ In programming language theory, lazy evaluation or call-by-need is an evaluation strategy which delays the evaluation of an expression until its value is needed (non-strict evaluation) and which also avoids repeated evaluations (sharing)
”

WIKIPEDIA: LAZY EVALUATION

Benefits:

- Performance increases
- The ability to construct potentially infinite data structures
- The ability to define control flow

"POURING WATER PROBLEM"



CODE

- declare initial state
- declare moves and path
- declare all possible moves

lazy_evaluation.py (https://github.com/kachayev/talks/blob/master/kharkiv

CODE: IMMUTABLE OPERATIONS ON LIST

We need: create new list, change element at position

```
In [1]: update
Out[1]: function __main__.update

In [2]: update([1,2,3], (0, 10))
Out[2]: [10, 2, 3]

In [3]: update([1,2,3], (0, 10), (1, 20), (3, 30))
Out[3]: [10, 20, 3]
```

CODE: IMMUTABLE OPERATIONS ON LIST

Implementation. Think twice!

```
def update(container, *pairs):
  """
  Create copy of given container, chagning one (or more) ele
ments.

  Creating of new list is necessary here, cause we don't wan
t to
  deal with mutable data structure in current situation - it

  will be too problematic to keep everything working with ma
ny
  map/reduce operations if data will be mutable.
  """
  def pair(left, right):
      pos, el = right
      # todo: use takewhile/dropwhile or write "split-at" he
lper
      return [(e if i != pos else el) for i, e in enumerate(
```

CODE: "LAZY" WRAPPER

We need: wrap generator to "reuse" yields

```
In [4]: def gen():
   ...:     yield 1
   ...:     yield 2
   ...:     yield 3
   ...:

In [5]: g = gen()
In [6]: list(g)
Out[6]: [1, 2, 3]
In [7]: list(g)
Out[7]: []

In [11]: glazy = lazy(gen)
In [12]: list(glazy)
Out[12]: [1, 2, 3]
In [13]: list(glazy)
Out[13]: [1, 2, 3]
```

CODE: "LAZY" WRAPPER

Implementation

```
class lazy:
    def __init__(self, origin, state=None):
        self._origin = origin() if callable(origin) else ori
gin
        self._state = state or []
        self._finished = False
    def __iter__(self):
        return self if not self._finished else iter(self._st
ate)
    def __next__(self):
        try:
            n = next(self._origin)
        except StopIteration as e:
            self._finished = True
            raise e
        else:
            self._state.append(n)
```

CODE

- end state declaration and folding
- solution declaration
- hint: Path class

lazy_evaluation.py (https://github.com/kachayev/talks/blob/master/kharkivp

WHAT'S FOR PYTHON 2.*?

- `itertools.imap` instead of `map`
- `itertools.ifilter` instead of `filter`
- `reduce` instead of `functools.reduce`
- `def pair(left, (pos, el))` is possible
- `for _ in _: yield _` instead of `yield from`

FIBONACCI IN HASKELL

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

Usage:

```
Prelude> take 10 fibs
[0,1,1,2,3,5,8,13,21,34]
Prelude> take 20 fibs
[0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,
4181]
```

FIBONACCI IN 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)
```

FIBONACCI IN SCALA

```
def fibs: Stream[Int] =
    0 #:: 1 #:: fibs.zip(fibs.tail).map{case (a,b) => a + b}
```

Usage:

```
scala> fibs(10)
res0: Int = 55
scala> fibs.take(10).toArray
res1: Array[Int] = Array(0, 1, 1, 2, 3, 5, 8, 13, 21, 34)
scala> fibs.take(20).toArray
res2: Array[Int] = Array(0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55
, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181)
```

FIBONACCI IN PYTHON

```
def fib():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a+b
```

Usage:

```
>>> from itertools import islice
>>> print list(islice(fib(), 10))
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
>>> print list(islice(fib(), 20))
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610
, 987, 1597, 2584, 4181]
```

Lazy, imperative

ADVANCED: DECLARATIVE FIBONACCI

Our goal:

```
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
]
```

Lazy, declarative

Stream implementation:

stream.py (https://github.com/kachayev/talks/blob/master/kharkiv

CONCLUSIONS

PRO

- conciseness
- possible parallel execution

CON

- immutability doesn't supported

THE END

THANK YOU FOR ATTENTION!

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

THIS PRESENTATION:

https://github.com/kachayev/talks/kharkivpy#6