Stop coding Pascal

...emotional sketch about past, present and future of programming languages, Python, compilers, developers, Life, Universe and Everything

About me

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

Einstein problem solving principle

So, next 55 minutes we will talk about the core of the problems

What are we going to talk about?

- difference between syntax and semantic
- imperative code: when and WHY?
- machine VS language: problems and solutions
- where did Python features come from
- what is the current problem
- why should I care?

Simple task

count of unique substring

Pascal

```
begin
  readln(s);
  for i:=1 to 26 do
   begin
    read(ch);
    if ch='0' then ok[chr(ord('a')+i-1)]:=1;
   end;
  readln;
  readln(k);
  n:=length(s);
  for i:=1 to n do
   begin
    j:=i-1;
    tmp:=0;
    while (j+1<=n) and (tmp+ok[s[j+1]]<=k) do begin tmp:=tmp+ok[s[j+1]]; inc(j); end;
    if i<=j then
     begin
      inc(size);
      a[size]:=copy(s,i,j-i+1);
     end;
   end;
  n:=0;
  if size=0 then begin writeln(0); halt; end;
  sort(1,size);
  n:=1;
  b[1]:=a[1];
  for i:=2 to size do if a[i]<>a[i-1] then
```

* just sample...

it's really hard to show Pascal version in slide

Python v. I

```
1  # i'm not sure if this code works...
2  def uniq_substrings(origin):
3      suffixes = [""]
4      for k in range(len(origin)):
5          suffixes.append(origin[k:])
6      suffixes.sort()
7
8      result = 0
9      for j in range(len(suffixes)-1):
10         pre, post = suffixes[j], suffixes[j+1]
11         lpost = len(post)
12         diff = 0
13         for i in range(max(len(pre), len(post))):
14             if pre[i] == post[i]:
15                 diff += 1
16             else:
17                 break
18         result += (lpost - diff)
19     return result
```

Python v.2

```python
# python 2+
from itertools import takewhile, ifilter, izip, tee, starmap, chain

def uniq_substrings(origin):
    suffixes = chain([""], sorted(tails(origin)))
    return sum(starmap(suffix, pairwise(suffixes)))

def pairwise(iterable):
    a, b = tee(iterable)
    next(b, None)
    return izip(a, b)

def common_prefix(pre, post):
    return takewhile(lambda (k1, k2): k1==k2, izip(pre, post))

def suffix(pre, post):
    return len(post) - ilen(common_prefix(pre, post))

def tails(origin):
    return (origin[i:] for i in xrange(ilen(origin)))

def ilen(iterable):
    return sum(1 for it in iterable)
```

Haskell

```haskell
import Data.List

uniqSubstr :: String -> Int
uniqSubstr = sum . (map pair) . pairwise . sort . tails
    where
        pairwise l@(_:ht) = zip l ht
        prefix pre post = length $ takeWhile (uncurry (==)) $ zip pre post
        pair (pre, post) = length $ drop (prefix pre post) post
```

What the difference is?

- “pascal” VS “python.v1” - syntax (mostly)
- “python.v1” VS “python.v2” - semantic
- “python.v2” VS “haskell” - (mostly) syntax (*)

* iterators VS. lazy-evaluation is a different story

Let’s dig deeper

Why is Haskell code so compact?

- transformations
- compositions
- * unix way, BTW

Where did all these `i`, `j`, `k` come from?

Instructions

vs.

Transformations

… and deeper

Turing machine

vs.

λ-calculus

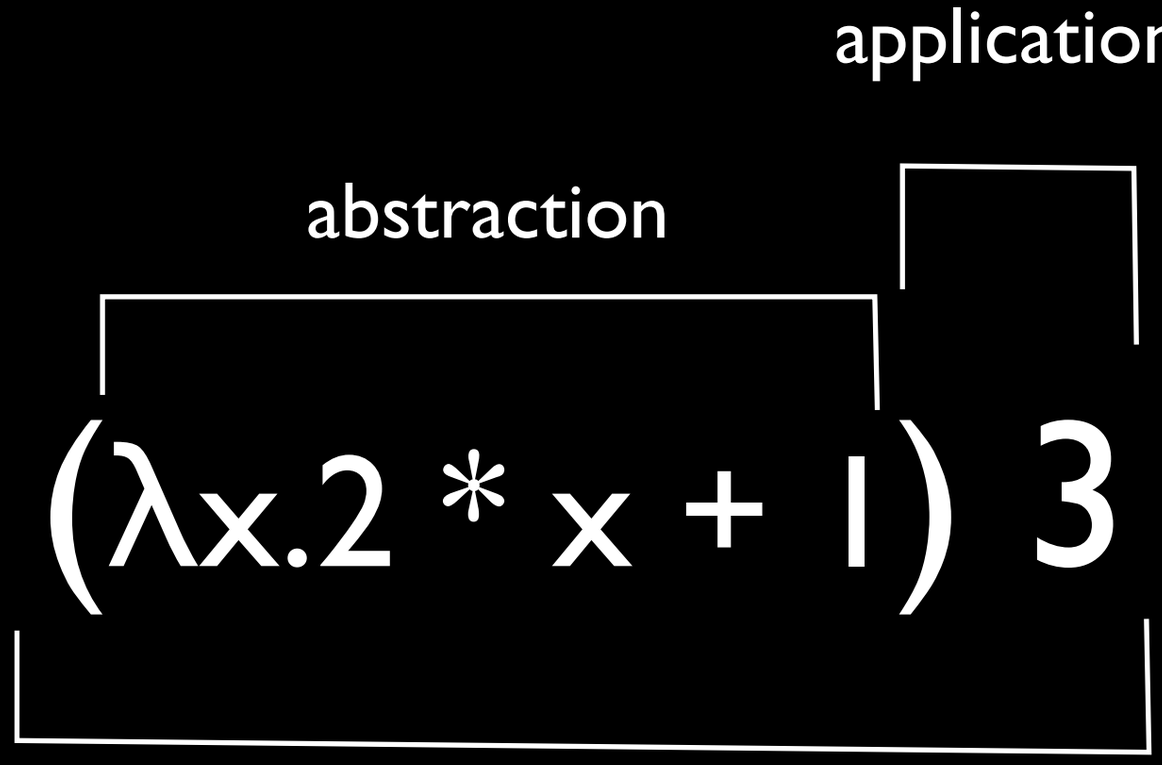
Turing machine

- infinite memory
- instructions (finite)

λ - calculus

- terms
- abstraction
- application
- β-reduction

The are many application operators in Haskell, ML



Q: “How it’s possible that everything is a transformation?”

A: “Have you ever thought about how (4+5-2*9) works?”

Hardware & compiler

vs.

Programming language

What the problem is?

"reusability" && "composability"

... oh, off course modularity matters, but we created many ways to split our programs since goto

Imperative

- hard to reuse/compose (context)
- hard to test (context)
- “interactive style” is hard
- it’s not the language that I want to talk
- parallelism is impossible
- ... but it’s widespread
- ... but it’s common
- ... but it’s hard to see the root of the problems (“Vietnam”)

Imperative advantages(?)

- algorithms O(*) - the same
- low level optimization? - oh, not in Python
- manual memory control? - oh, not in Python

Logic combinators

Set theory

Type theory

λ-notation

λ-calculus

λ-calculus as PL

AST-base

Lisp (family)

ML (family)

Haskell

OCaml

Python

ABC

Pascal

SETL

Algol 60/8

C

Fortran

Speedcoding

Assembly

Turing machine

When somebody tells you that each language is sugar over Turing machine...

do not believe

Python

- mostly imperative, but...
- higher-ordered functions (*)
- lambdas (*)
- no for(i=0;i<10;i++) (**)
- iterators (**)
- map/filter/zip/itertools (**)
- generators (**)
- futures (concurrency, tulip)

* ast/abt

** sequence-based semantic

Move on to more practical questions

* starting from easiest: looking for high-level patterns

checkio.py (~/Dropbox/code/generic_python2) – VIM

[1:ffind.py]*[4:checkio.py]*![6:narc.py]*

-MiniBufExplorer-

```python
import os
import sys
import argparse
import re

# Define colors
RED_CHARACTER = '\x1b[31m'
GREEN_CHARACTER = '\x1b[32m'
YELLOW_CHARACTER = '\x1b[33m'
BLUE_CHARACTER = '\x1b[36m'
PURPLE_CHARACTER = '\x1b[35m'
NO_COLOR = '\x1b[0m'


def search(directory, file_pattern, path_match,
           follow_symlinks=True, output=True, colored=True):
    ''' Search the files matching the pattern.
        The files will be returned, and can be optionally printed '''

    pattern = re.compile(file_pattern)

    results = []

    for root, sub_folders, files in os.walk(directory,
                                            followlinks=follow_symlinks):
        # Ignore hidden directories
        if '/.' in root:
            continue

        # Search in files and subfolders
        for filename in files + sub_folders:
            full_filename = os.path.join(root, filename)
            to_match = full_filename if path_match else filename
            match = re.search(pattern, to_match)
            if match:
                # Split the match to be able to colorize it
                # prefix, matched_pattern, sufix
                smatch = [to_match[:match.start()],
                          to_match[match.start(): match.end()],
                          to_match[match.end():]]
                if not path_match:
                    # Add the fullpath to the prefix
                    smatch[0] = os.path.join(root, smatch[0])

                if output:
                    print_match(smatch, colored)

                results.append(full_filename)

    return results


def print_match(splitted_match, colored, color=RED_CHARACTER):
    ''' Output a match on the console '''
    if colored:
        a, b, c = splitted_match
        colored_output = (a, color, b, NO_COLOR, c)
    else:
```

ffind/ffind/ffind.py

```python
from itertools import chain, combinations, permutations
import operator
# Store found primes to increase performance through memoization
# Also, store first primes
PRIMES = [2, 3, 5, 7]


def prime(number):
    ''' Find recursively if the number is a prime. Returns True or False'''

    # Check on memoized results
    if number in PRIMES:
        return True

    # By definition, 1 is not prime
    if number == 1:
        return False

    # Divide the number between all their lower prime numbers (excluding 2)
    # Use this function recursively
    lower_primes = (p for p in PRIMES if number > p)
    if any(p for p in lower_primes if number % p == 0):
        return False

    # The number is not divisible, it's a prime number
    # Store to memoize
    PRIMES.append(number)
    return True


def partition(iterable, chain=chain, map=map):
    s = iterable if hasattr(iterable, '__getslice__') else tuple(iterable)
    n = len(s)
    first, middle, last = [0], range(1, n), [n]
    getslice = s.__getslice__
    return [map(getslice, chain(first, div), chain(div, last))
            for i in range(n) for div in combinations(middle, i)]


def group_factors(factors):
    all_groups = []
    for perm in permutations(factors, len(factors)):
        for groups in partition(perm):
            groups = [list(g) for g in groups]
            [g.sort() for g in groups]
            groups.sort()
            if groups not in all_groups:
                all_groups.append(groups)

    return all_groups


def factor(number):
    # Be sure to generate all primes
    if PRIMES[-1] < number:
        [prime(i) for i in xrange(PRIMES[-1], number + 1)]

    factors = []
```

generic_python2/checkio.py

```python
NUM_DIGITS = 10


powers = []
for i in xrange(NUM_DIGITS + 1):
    powers.append([j ** i for j in xrange(10)])


def get_digits(number):
    while number:
        yield number % 10
        number /= 10


def number_is_narcissistic(number):
    num_digits = int(math.log10(number)) + 1
    power = powers[num_digits]
    acc = 0
    for i in get_digits(number):
        acc += power[i]
        if acc > number:
            return False

    return acc == number


def number_is_narcissistic_no_str(number):
    acc = 0
    original_number = number
    num_digits = int(math.log10(number)) + 1
    power = powers[num_digits]
    for digit in get_digits(number):
        acc += power[digit]
        if acc > original_number:
            return False

    return acc == original_number


def comb_is_candidate(comb, num_digits):
    acc = 0
    limit = 10 ** num_digits
    min_limit = 10 ** (num_digits - 1)
    power = powers[num_digits]
    for n in comb:
        acc += power[n]
        if acc > limit:
            return False

    return acc > min_limit


def number_power(comb, power):
    return sum(c ** power for c in comb)


def all_candidate_numbers(num_digits):
    comb = combinations_with_replacement(xrange(10), num_digits)
```

generic_python2/narcissistic/narc.py

--No lines in buffer--

http://wrongsideofmemphis.files.wordpress.com/2013/03/screen-shot-2013-03-23-at-12-32-17.png

Do you see the patterns?

```
    results = []

    for root, sub_folders, files in os.walk(directory,
                                            followlinks=follow_symlinks):
        # Ignore hidden directories
        if '/.' in root:
            continue

        # Search in files and subfolders
        for filename in files + sub_folders:
            full_filename = os.path.join(root, filename)
            to_match = full_filename if path_match else filename
            match = re.search(pattern, to_match)
            if match:
                # Split the match to be able to colorize it
                # prefix, matched_pattern, sufix
                smatch = [to_match[:match.start()],
                          to_match[match.start(): match.end()],
                          to_match[match.end():]]
                if not path_match:
                    # Add the fullpath to the prefix
                    smatch[0] = os.path.join(root, smatch[0])

                if output:
                    print_match(smatch, colored)

                results.append(full_filename)

    return results
```

Do you see the patterns?

```
def partition(iterable, chain=chain, map=map):
    s = iterable if hasattr(iterable, '__getslice__') else tuple(iterable)
    n = len(s)
    first, middle, last = [0], range(1, n), [n]
    getslice = s.__getslice__
    return [map(getslice, chain(first, div), chain(div, last))
            for i in range(n) for div in combinations(middle, i)]


def group_factors(factors):
    all_groups = []
    for perm in permutations(factors, len(factors)):
        for groups in partition(perm):
            groups = [list(g) for g in groups]
            [g.sort() for g in groups]
            groups.sort()
            if groups not in all_groups:
                all_groups.append(groups)

    return all_groups
```

Do you see the patterns?

```python
def number_is_narcissistic(number):
    num_digits = int(math.log10(number)) + 1
    power = powers[num_digits]
    acc = 0
    for i in get_digits(number):
        acc += power[i]
        if acc > number:
            return False

    return acc == number


def number_is_narcissistic_no_str(number):
    acc = 0
    original_number = number
    num_digits = int(math.log10(number)) + 1
    power = powers[num_digits]
    for digit in get_digits(number):
        acc += power[digit]
        if acc > original_number:
            return False

    return acc == original_number


def comb_is_candidate(comb, num_digits):
    acc = 0
    limit = 10 ** num_digits
    min_limit = 10 ** (num_digits - 1)
    power = powers[num_digits]
    for n in comb:
        acc += power[n]
        if acc > limit:
            return False

    return acc > min_limit
```

Do you see the patterns?

```
# imperative cycles do not compose at all
def friend_phones(friends):
    to_call = []
    for friend in friends:
        phones = ""
        for phone in friend.phones:
            phones += "; " + str(phone)
        to_call.append(phones)
```

```python
# move to separated function
def phones(friend):
    phones = ""
    for phone in friend.phones:
        phones += "; " + str(phone)
    return phones
```

```python
def friend_phones(friends):
    to_call = []
    for friend in friends:
        to_call.append(phones(friend))
    return to_call
```

```python
# high level pattern "map"
def map(f, iterable):
    # never do this in your code,
    # use builtin map
    result = []
    for it in iterable:
        result.append(f(it))
    return result

def phones(friend):
    return ";".join(map(str, friend.phones))

def friend_phones(friends):
    return map(phones, friends)
```

```python
# we can do better!
def map(f, iterable):
    # never do this in your code,
    # use builtin map
    for it in iterable:
        yield f(it)


def phones(friend):
    return ";".join(map(str, friend.phones))

def friend_phones(friends):
    return map(phones, friends)
```

```
# more patterns!
def partial(f, *binds):
    # never do this in your code,
    # use functools.partial
    def _applyied(*args):
        return f(*(binds + args))
    return _applyied

def phones(friend):
    return ";".join(map(str, friend.phones))

friend_phones = partial(map, phones)
```

Do you see the patterns?

```
# ok, and what about filter?
def friend_phones(friends):
    to_call = []
    for friend in friends:
        phones = ""
        for phone in friend.phones:
            if phone.startswith("8-000"):
                phones += "; " + str(phone)
        to_call.append(phones)
```

* we already talked that loops do not compose

Do you see the patterns?

```
def sum(iterable):
    result = 0
    for it in iterable:
        result += it
    return result

def product(iterable):
    result = 1
    for it in iterable:
        result *= it
    return result
```

```
def maximum(iterable):
    result = float("-inf")
    for it in iterable:
        result = max(result, it)
    return result

def all(iterable):
    result = True
    for it in iterable:
        result = result and it
    return result
```

```
def max_position(origin):
    pos, value, i = float("-inf"), -1, 0
    for k in range(len(origin)):
        if origin[k] > value:
            pos, value = i, origin[k]
        i += 1
    return pos, value

def max_position(iterable):
    pos, value = float("-inf"), -1
    for (i, val) in enumerate(iterable):
        if val > value:
            pos, value = i, val
    return pos, value

def max_position(iterable):
    def check((pos, local_max), (i, value)):
        return (pos, local_max) if value < local_max else (i,value)
    return reduce(check, enumerate(iterable), (float("-inf"), -1))
```

Not only syntax...

- transformations instead of instructions
- reduction declarations without dealing with application
- reuse pure function in some context (functor)
- high(er) level of composability

When syntax sucks...

```
# more patterns! compose, currying
from functools import partial
from operator import attrgetter as prop

def comp(f1, f2, f3):
    # we can do this in general case,
    # but that will be other pattern (further)!
    def _composition(*args, **kwargs):
        return f1(f2(f3(*args, **kwargs)))
    return _composition

phones = comp(";".join, partial(map, str), prop("phones"))
friend_phones = partial(map, phones)
```

Iterators is not only about lists

... this is the semantic way to think about possible solutions

```
from itertools import dropwhile, tee, izip

def repeatfunc(f, zero):
    curr = zero
    while 1:
        yield curr
        curr = f(curr)

def pairwise(origin):
    a, b = tee(origin)
    next(b, None)
    return izip(a,b)

def snd(origin):
    it = origin if hasattr(origin, "next") else iter(origin)
    next(it, None)
    return next(it, None)

def square_root(n, eps=0.01):
    def f(x): return (x + float(n)/x)/2
    def outside((left, right)): return abs(left-right) > eps
    return snd(next(dropwhile(outside, pairwise(repeatfunc(f, n)))))
```

Only last function matters

... other functions are common and you can find them in Python documentation or implemented in Fn.py library

More examples

Lazy evaluation and declarative approach:
http://kachayev.github.com/talks/

What stuff do you know about?

- iterators
- generators
- lazy-evaluation
- undelimited continuations
- delimited continuations
- coroutines
- macros
- monads
- “staging”
- “deref scope”

What stuff do you use in code?

- iterators
- generators
- lazy-evaluation
- undelimited continuations
- delimited continuations
- coroutines
- macros
- monads
- “staging”
- “deref scope”

What stuff do you want to use?

- iterators
- generators
- lazy-evaluation
- undelimited continuations
- delimited continuations
- coroutines
- macros
- monads
- “staging”
- “deref scope”

I saw many coroutines

during conference talks

I never saw coroutines

in real-life projects

Can you describe* coroutine advantages?

* using one word

Can you describe coroutine disadvantages?

What the problem is?

- easy to start with simplest stuff (it's cool, but don't stop!)
- habits, traditions (???)
- mutable variables and assignments dictate (*)
- syntax doesn't support non-imperative semantic ("for" is only one good example of support, "yield from" is also cool) (**)
- internal contradictions (***)

Can you see semantic under the syntax?

```
(lambda flask:
    (lambda app:
        (app,
         app.route('/')(
             lambda: 'Hello World!'
         )
        )[0]
    )(flask.Flask('__name__')).run()
)(__import__('flask'))
```

https://gist.github.com/e000/1023982

Are you getting on a bit?

Can you see ABT under your AST?

I don't want you to write code this way

I just want you to understand how it works and why it's possible

BTW, it's common

pattern in JS code...

`yield from`

is not only the new syntax!

```python
# python 2.7+
class Node(object):

    def __init__(self, value, left, right):
        self.value = value
        self.left = left
        self.right = right

    def __iter__(self):
        yield self.value
        for n in self.left:
            yield n
        for n in self.right:
            yield n
```

```python
# python 3.3+
class Node:

    def __init__(self, value, left, right):
        self.value = value
        self.left = left
        self.right = right

def __iter__(self):
    yield self.value
    yield from self.left
    yield from self.right
```

```python
# fn.py
from fn import Stream

class Node:

    def __init__(self, value, left, right):
        self.value = value
        self.left = left
        self.right = right

def __iter__(self):
    # compare to:
    # return (self.value #:: self.left #:: self.right)
    return iter(Stream(self.value) << self.left << self.right)
```

It's all about composition.

You can (*) write point-free (**) code

* you just don't have readable syntax to do this

** applyTwice is good example to show

Isn't this wonderful?

```
1  allPhones = for {
2      friend <- user.getFriends()
3      phone <- friend.getPhones()
4      if phone.startsWith("8-001")
5  } yield (friend.name, phone)
```

Contra

- no composition syntax!!!
- okay... def new function (not readable enough)
- no recursion!!!
- okay... the list is...? iterators is...?
- oh, I know! recursion = fold + unfold :)
- no fold!!! we have list comprehensions
- but... LC = map&filter... okay...

Code vs. Ideas

So now reduce(). This is actually the one I've always hated most, ... almost every time I see a reduce() call with a non-trivial function argument, I need to grab pen and paper to diagram... it's better to write out the accumulation loop explicitly.

(c) Guido Van Rossum

fold/unfold

... dig deeper

```haskell
# foldl _ acc [] = acc
# foldl f acc x:xs = foldl f (f acc x) xs
def foldl(f, zero):
    def _folder(origin):
        left = zero
        for el in origin:
            left = f(left, el)
        return left
    return _folder

import operator

sum = foldl(operator.add, 0)
all = foldl(operator.and_, True)
any = foldl(operator.or_, False)
product = foldl(operator.mul, 1)
maximum = foldl(max, float("-inf"))
```

Everybody knows this examples...

```
def map(f, origin):
    return foldl(
        lambda to, el: to + [f(el)],
        []
    )(origin)

def filter(pred, origin):
    return foldl(
        lambda to, el: to + [el] if pred(el) else [],
        []
    )(origin)

def accumulate(f, origin):
    return foldl(
        lambda to, el: to + [f(to[-1], el)],
        []
    )(origin)
```

Dig deeper.

```
def unfold(f):
    def _unfolder(start):
        prev, curr = None, start
        while 1:
            prev, curr = f(curr)
            yield prev
            if curr is None: break
    return _unfolder

>>> doubler = unfold(lambda x: (x*2, x*2))
>>> list(islice(doubler(10), 0, 10))
[20, 40, 80, 160, 320, 640, 1280, 2560, 5120, 10240]
```

I want you to think about semantic

In 5 years Python will solve other problems

In 5 years Python should (*) solve other problems

* technologies are changing very fast

Is Python ready to be a cutting-edge language?

Are you ready?

Questions?

* other talks: https://kachayev.github.com/talks
** fn.py: https://github.com/kachayev/fn.py