Idiomatic Python

--Bigger than Bigger

Zhang Yu@Nokia coach network

Agenda

Idioms
Data Manipulation
Control Flow
‘itertools’
Functional Python

Guido van Rossum
Shared publicly - Sep 19, 2013

Do **not** send me email like this:

"""
Hi Guido,

I came across your resume in a Google web search. You seem to have an awesome expertise on Python. I would be glad if you can reply my email and let me know your interest and availability.

.....
Our client immediately needs a PYTHON Developers at its location in *, NJ. Below are the job details. If interested and available, kindly fwd me your updated resume along with the expected rate and the availability.

[...]
"""

I might reply like this:

"""
I'm not interested and not available.
"""

The Zen of Python

```
In [1]: import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

美麗優於醜陋，明講好過暗諭。

簡潔者為上，複雜者次之，繁澀者為下。

平舖善於層疊，勻散勝過稠密；以致輕鬆易讀。

特例難免但不可打破原則，務求純淨卻不可不切實際。

斷勿使錯誤靜靜流逝，除非有意如此。

在模擬兩可之間，拒絕猜測的誘惑。

總會有一種明確的寫法，最好也只有一種，

但或須細想方可得。

凡事雖應三思後行，但坐而言不如起而行。

難以解釋的實作方式，必定是壞方法。

容易解釋的實作方式，可能是好主意。

命名空間讚，吾人多實用。

Idioms

- An unwritten rule
- A common use-case
- Usually make the code better in:
  - Readability
  - Speed
  - Resource usage

Data Manipulation

Unpacking

```
s = ('zhang', 'yu', 0106, 'feiyuw@gmail.com')

firstname = s[0]
lastname = s[1]
weight = s[2]
email = s[3]

# Idiomatic
firstname, lastname, weight, email = s
_, _, _, email = s # if only email is needed
```

Swap Values

```
temp = a
a = b
b = temp

# idiomatic way, using tuple packing & unpacking
a, b = b, a

# For those pathetic C programmers
a, b = (b, a)
```

Don’t Underestimate

```
# machine-oriented, error-prone
next_x = x + dx * t
next_y = y + dy * t
next_dx = influence(m, x, y, dx, dy, partial='x')
next_dy = influence(m, x, y, dx, dy, partial='y')

x = next_x
y = next_y
dx = next_dx
dy = next_dy
```

Don’t Underestimate (cont.)

```
# idiomatic, declarative
x, y, dx, dy = (x + dx * t,
                y + dy * t,
                influence(m, x, y, dx, dy, partial='x'),
                influence(m, x, y, dx, dy, partial='y'))
```

Concatenating Strings

```
fruits = ['cherry', 'coconut', 'blueberry', 'kiwi']

# bad
s = fruits[0]
for i in fruits[1:]:
    s += ', ' + f

# idiomatic
print ', '.join(fruits)
```

Looping over a collection

```
colors = ['red', 'green', 'blue', 'yellow']

# bad
for i in range(len(colors)):
    print colors[i]

# idiomatic
for color in colors:
    print color
```

Looping backwards

```
colors = ['red', 'green', 'blue', 'yellow']

for color in reversed(colors):
    print color

for color in colors[::-1]:
    print color
```

Looping with indices

```
colors = ['red', 'green', 'blue', 'yellow']

# bad
for i in range(len(colors)):
    print i, '-->', colors[i]

# idiomatic
for i, color in enumerate(colors):
    print i, '-->', color
```

Looping over a dictionary

```
codes = {'Xian': '29', 'Beijing':'10', 'Shanghai':'21'}
# bad
for k in codes:
    print k, '-->', codes[k]

# recommended
for k, v in codes.items():
    print k, '-->', v

for k, v in codes.iteritems():
    print k, '-->', v
```

‘defaultdict’

```
names = ['james', 'peter', 'simon', 'jack', 'john', 'lawrence']

# expected result
{8: ['lawrence'], 4: ['jack', 'john'], 5: ['james', 'peter', 'simon']}

# old way
groups = {}
for name in names:
    key = len(name)
    if key not in groups:
        groups[key] = []
    groups[key].append(name)
```

‘defaultdict’ (cont.)

```
# use ‘setdefault’ with default value prepared
groups = {}
for name in names:
    groups.setdefault(len(name), []).append(name)

# use ‘defaultdict’
from collections import defaultdict
groups = defaultdict(list)
for name in names:
    groups[len(name)].append(name)
```

Comprehensions

```
# bad
A, odd_or_even = [1, 1, 2, 3, 5, 8, 13, 21], []
for number in A:
    odd_or_even.append(isOdd(number))


# expected result
[True, True, False, True, True, False, True, True]
```

Comprehensions (2/3)

```
A = [1, 1, 2, 3, 5, 8, 13, 21]

# idiomatic way
[isOdd(a) for a in A]
[True, True, False, True, True, False, True, True]

[a for a in A if a%2 != 0]
[1, 1, 3, 5, 13, 21]
```

Comprehensions (3/3)

```
List: [a**2 for a in A]
[1, 1, 4, 9, 25, 64, 169, 441]

Set: {int(sqrt(a)) for a in A}
set([1, 2, 3, 4])

Dict: {a:a%3 for a in A if a%3}
{8: 2, 1: 1, 2: 2, 5: 2, 13: 1}
```

Control Flow

Truthiness

```
# Avoid comparing directly to True, False, or None
if names != []:
...
if foo == True:
...

# idiomatic way
if names:
...
if foo:
```

Truthiness (cont.)

All of the following are considered ‘False’

- None
- False
- zero for numeric types
- empty sequence, e.g. [], tuple()
- empty dictionaries
- a value of 0 or False returned when either __len__ or __non_zero__ is called

‘if-in’

```
# ugly, repeating variables
is_generic_color = False
if color == 'red' or color == 'green' or color == 'blue':
    is_generic_color = True


# idiomatic way
is_generic_color = color in ('red', 'green', 'blue')
```

‘for-else’

```
ages = [42, 21, 18, 33, 19]

# old way
are_all_adult = True
for age in ages:
    if age < 18:
        are_all_adult = False
        break

if are_all_adult:
    print 'All are adults!'
```

‘for-else’ (cont.)

```
ages = [42, 21, 18, 33, 19]

# idiomatic way
for age in ages:
    if age < 18:
        break
else: # go through without break
    print 'All are adults!'
```

Context Manager

```
# old way
f = open('data.csv')
try:
    data = f.read()
finally:
    f.close()


# idiomatic way
with open('data.csv') as f:
    data = f.read()
```

import itertools

Looping with two collections

```
colors = ['red', 'blue', 'green', 'yellow']
fruits = ['cherry', 'blueberry', 'kiwi']

# old way
min_len = min(len(colors), len(fruits))
for i in range(min_len):
    print fruits[i], '-->', colors[i]

# idiomatic way
from itertools import izip
for fruit, color in izip(fruits, colors):
    print fruit, '-->', color
```

Building Dictionaries

```
fruits = ['cherry', 'blueberry', 'kiwi', 'mango']
colors = ['red', 'blue', 'green', 'yellow']

# expected
{'kiwi': 'green', 'cherry': 'red', 'mango': 'yellow',
'blueberry': 'blue'}

# old way
pairs = {}
for fruit, color in izip(fruits, colors):
    pairs[fruit] = color
```

Building Dictionaries (cont.)

```
fruits = ['cherry', 'blueberry', 'kiwi', 'mango']
colors = ['red', 'blue', 'green', 'yellow']


# idiomatic way
from itertools import izip

pairs = dict(izip(fruits, colors))
```

‘groupby’

```
names = ['james', 'peter', 'simon', 'jack', 'john', 'lawrence']

{8: ['lawrence'], 4: ['jack', 'john'], 5: ['james', 'peter',
'simon']}


# use itertools
{k:list(v) for k, v in groupby(names, len)}
```

More

- chain([1,2,3], ['a','b'], [4]) ==> 1,2,3,'a','b',4
- repeat('A', 3) ==> 'A' 'A' 'A'
- cycle('ABCD') ==> A B C D A B C D ...
- compress('ABCDEF', [1,0,1,0,1,1]) ==> A C E F
- combinations/permutations/product
- ...

Functional Python

What is functional

- Imperative programming (C/C++, Java)
- Declarative programming
  - Functional programming (Lisp, Haskell, OCaml)
  - Logic programming (Prolog, Clojure)

“Functions are data, too. Can be passed through and manipulated like data.”

partial

```
# old way
def log(level, message):
    print "[{level}]: {msg}".format(level=level, msg=message)

def log_debug(message):
    log('debug', message)

def log_warn(message):
    log('warn', message)
```

partial (2/3)

```
# old way
def create_log_with_level(level):
    def log_with_level(message):
        log(level, message)
    return log_with_level


# construct functions like data
log_debug = create_log_with_level('debug')
log_warn = create_log_with_level('warn')
```

partial (3/3)

```
# use functools
from functools import partial

log_debug = partial(log, 'debug')
log_warn = partial(log, 'warn')
```

Decorator

```
# old way, mixing administrative logic with domain logic
def web_lookup(url, cache={}):
    if url not in cache:
        cache[url] = urllib.urlopen(url).read()
    return cache[url]


# use decorator, as in AOP
@cache
def web_lookup(url):
    return urllib.urlopen(url).read()
```

Decorator (cont.)

```
# implementation of the ‘cache’ decorator
from functools import wraps

def cache(func):
    saved = {}
    @wraps
    def new_func(*args):
        if args not in saved:
            saved[args] = func(*args)
        return saved[args]
    return new_func
```

Combine

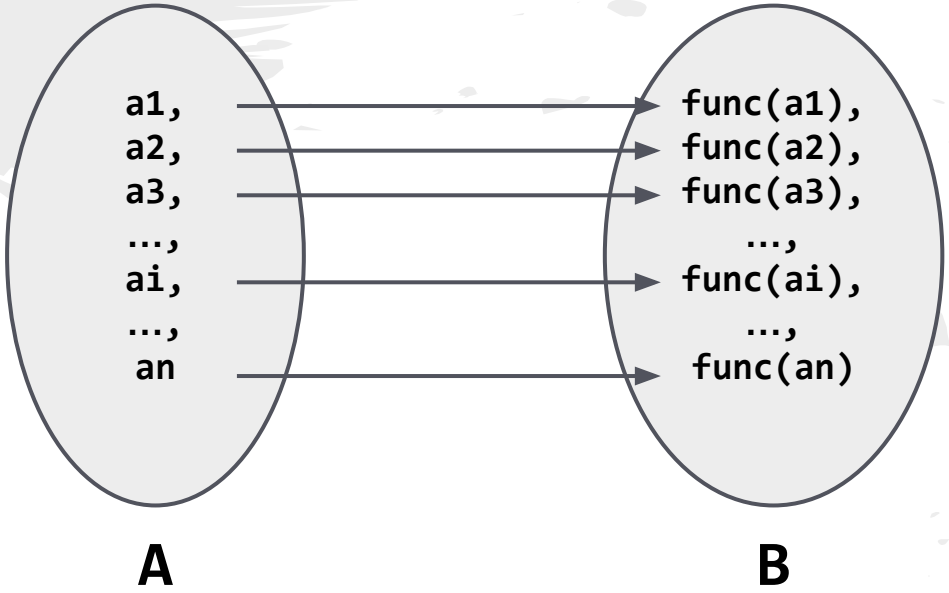
```
# imperative way
expr, res = '28++32+++32+39', 0
for token in expr.split('+'):
    if token:
        res += int(token)


# result of split
["28", "", "32", "", "", "32", "39"]
```

Combine (cont.)

```
# functional way
res = sum(map(int, filter(bool, expr.split('+'))))

# step by step
["28", "", "32", "", "", "32", "39"]

filter(pred, seq) => [t for t in seq if pred(t)]
["28", "32", "32", "39"]

map(func, seq) => [func(t) for t in seq]
[28, 32, 32, 39]
```

‘map’

B = map(func, A)



‘all’

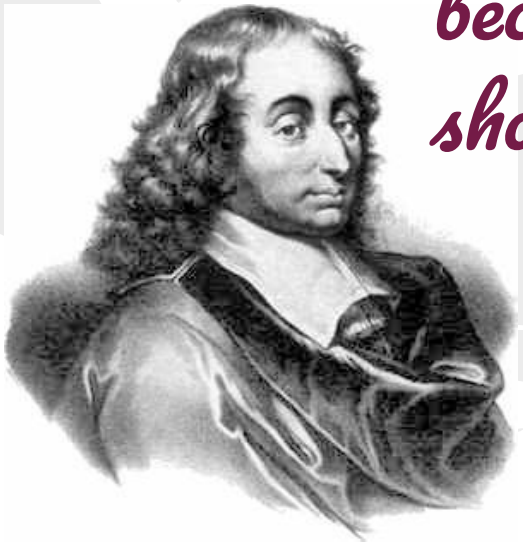
```
ages = [42, 21, 18, 33, 19]


# more expressive than using ‘for-else’
if all(map(lambda a:a>=18, ages)):
    print 'All are adults!'
```

Fluent Interface

```
expr = '28++32+++32+39'

IterHelper(expr.split('+')).filter(bool).map(int).sum()


ages = [42, 21, 18, 33, 19]

IterHelper(ages).map(lambda x:x>=18).all()
```

Fluent Interface (cont.)

```
class IterHelper(object):
    def __init__(self, iterable = []):
        self.iterable = iterable

    def dump(self):
        return list(self.iterable)

    def map(self, func):
        return IterHelper(itertools.imap(func, self.iterable))

    def filter(self, predicate):
        return IterHelper(itertools.ifilter(predicate, self.iterable))

    def sum(self):
        return sum(self.iterable)

    def all(self):
        return all(self.iterable)
```



I have made this longer than usual because I have not had time to make it shorter.

-- Blaise Pascal (1623-1662)

Terse Code Is Not A Free Lunch!

```
' '.join('{0:08b}'.format(ord(x)) for x in 'Bigger Than Bigger!')
```

01000010 01101001 01100111 01100111 01100101
01110010 00100000 01010100 01101000 01100001
01101110 00100000 01000010 01101001 01100111
01100111 01100101 01110010 00100001

Bigger Than Bigger!

Thank You!

References:

- code like a pythonista: http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html
- itertools: https://docs.python.org/2/library/itertools.html
- functional python: http://ua.pycon.org/static/talks/kachayev
- functools: https://docs.python.org/2/library/functools.html
- pydash: http://pydash.readthedocs.org/en/latest/