Productivity through consistency

PYTHONIC APIs

How to leverage the Python Data Model to build idiomatic and easy to use APIs
LUCIANO RAMALHO

Technical Principal

@ramalhoorg
luciano.ramalho@thoughtworks.com
Fluent Python (O’Reilly, 2015)
Python Fluente (Novatec, 2015)
Python к вершинам мастерства* (DMK, 2015)
流暢的 Python† (Gotop, 2016)
also in Polish, Korean...

* Python. To the heights of excellence
† Smooth Python
CONSISTENCY

A relative concept
CONSISTENCY BY DESIGN

P = 8.0 mm
= 5/6 × H
= 2.5 × h

4.8 mm
1.7 mm

h = 3.2 mm
= 1/3 × H
= 0.4 × P

3.2 mm

2 × P − 0.2 mm
= 15.8 mm

H = 9.6 mm
= 3 × h
= 1.2 × P

P − 0.2 mm
= 7.8 mm
**CONSISTENT WITH...?**

Is Python consistent?

```python
len(text)  # string
len(rates) # array of floats
len(names) # list
```

How about Java?

```java
text.length() // String
rates.length  // array of floats
names.size()  // ArrayList
```
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!
Java designers implemented `array.length` as an attribute for performance reasons; array isn’t even a class, but a `magical` construct.

```java
rates.length // array of floats
```
Guido implemented `len()` as a *built-in* function for the same reason — better performance by avoiding attribute look-up and method invocation:

```
len(text)  # string
len(rates) # array of floats
len(names) # list
```

For built-in types (and types implemented as extensions in C), `len(x)` returns the value of a field from a struct describing `x`.

For user-defined types coded in Python, the interpreter invokes `x.__len__()`. Only then we pay the cost of a method invocation.
Classes implemented in Python can be consistent with *built-in* types by implementing special methods to support many operations:

```python
>>> v1 = Vector([3, 4])
>>> len(v1)
2
>>> abs(v1)
5.0
>>> v1 == Vector((3.0, 4.0))
True
>>> list(v1)  # iteration
[3.0, 4.0]
>>> x, y = v1  # iteration!
>>> x, y
(3.0, 4.0)
```
The standard library is not the only source of inspiration
Requests: HTTP for Humans by Kenneth Reitz.

Famous for its Pythonic elegance:

```python
import requests

r = requests.get('https://api.github.com', auth=('user', 'pass'))

print(r.status_code)
print(r.headers['content-type'])
```

# output:
200
'application/json'
THE REQUESTS LIBRARY

Without **requests**:

```python
import urllib2

gh_url = 'https://api.github.com'

req = urllib2.Request(gh_url)

password_manager = urllib2.HTTPPasswordMgrWithDefaultRealm()
password_manager.add_password(None, gh_url, 'user', 'pass')

auth_manager = urllib2.HTTPBasicAuthHandler(password_manager)

opener = urllib2.build_opener(auth_manager)

urllib2.install_opener(opener)

handler = urllib2.urlopen(req)

print handler.getcode()
print handler.headers.getheader('content-type')
```

# output:
200
'application/json'

*(possibly biased example from https://gist.github.com/kennethreitz/973705)*
WHY REQUESTS FEELS PYTHONIC

Idiomatic traits of `requests`:

- no need to instantiate multiple objects to configure a request
- batteries included: default configuration handles authentication
- result object has a `status_code` attribute (instead of a `getcode()` method)
- result object has a mapping of header fields (instead of a `getheader(key)` method)

```python
import requests

r = requests.get('https://api.github.com', auth=('user', 'pass'))

print r.status_code
print r.headers['content-type']
```
from vector import Vector

def test_vector_unary_minus():
    assert -Vector([1, 2, 3]) == Vector([-1, -2, -3])

$ py.test
========================= test session starts =============================
platform darwin -- Python 3.5.1, pytest-2.9.1, py-1.4.31, pluggy-0.3.1
rootdir: /Users/lramalho/prj/oscon/pythonic-api/exercises/vector, inifile:
collected 6 items

test_vector_add.py ....
test_vector_neg.py .
test_xunit_vector_neg.py .

======================== 6 passed in 0.06 seconds =========================
import unittest

from vector import Vector

class TestStringMethods(unittest.TestCase):
    def test_vector_unary_minus(self):
        self.assertAlmostEqual(Vector([-1, -2, -3]), Vector([1, 2, 3]))

$ python3 -m unittest
.
 Ran 1 test in 0.000s

OK
from vector import Vector

def test_vector_unary_minus():
    assert -Vector([1, 2, 3]) == Vector([-1, -2, -3])

import unittest

from vector import Vector

class TestStringMethods(unittest.TestCase):
    def test_vector_unary_minus(self):
        self.assertEqual(-Vector([1, 2, 3]), Vector([-1, -2, -3]))
Why Py.Test Feels Pythonic

Idiomatic traits of `py.test`:

- no need to subclass anything to create a test (but test classes are supported)
- test cases are just functions
- no need to import `pytest` module for simple tests — the test runner script (`py.test`) knows how to inspect modules to find test functions and classes
- use of the `assert` keyword (instead of `TestCase.assertEqual` etc.)
- `py.test` uses advanced metaprogramming to build report from stack traces

```python
def test_vector_unary_minus():
    assert -Vector([1, 2, 3]) == Vector([-1, -2, -3])
```
import pytest

from vector import Vector

@ pytest.fixture
def v2d():
    return Vector([1, 2])

def test_2d_vector_addition(v2d):
v = Vector([3, 4])
assert v + v2d == Vector([4, 6])

def test_not_implemented_exception(v2d):
    with pytest.raises(TypeError) as exc:
        v2d + 1
assert 'unsupported operand' in str(exc.value)
TYPICAL CHARACTERISTICS OF IDIOMATIC LIBRARIES

Pythonic APIs usually:

• don’t force you to write boilerplate code
• provide ready to use functions and objects
• don’t force you to subclass unless there’s a very good reason
• include the batteries: make easy tasks easy
• are simple to use but not simplistic: make hard tasks possible
• leverage the Python Data model to:
  • provide objects that behave as you expect
  • avoid boilerplate through introspection (reflection) and metaprogramming

```python
from vector import Vector

def test_vector_unary_minus():
    assert Vector([1, 2, 3]) == Vector([-1, -2, -3])
```
THE PYTHON DATA MODEL

Python's object model
WHAT IS AN OBJECT MODEL

A.k.a. “metaobject protocol”

Standard interfaces/protocols for objects that represent language constructs:

- functions
- classes
- instances
- modules
- etc...
“Data Model” is how the Python docs call its object model. APIs defined with special methods named in __dunder__ form.

```python
class Vector:
    typecode = 'd'

    def __init__(self, components):
        self._components = array(self.typecode, components)

    def __len__(self):
        return len(self._components)

    def __iter__(self):
        return iter(self._components)

    def __abs__(self):
        return math.sqrt(sum(x * x for x in self))

    def __eq__(self, other):
        return all(a == b for a, b in zip(self, other))
```
HOW SPECIAL METHODS ARE USED

Special methods are invoked by the Python interpreter — rarely by user code.

Similar to how we work with frameworks by implementing methods or functions that the framework invokes.

THE HOLLYWOOD PRINCIPLE: DON’T CALL US, WE’LL CALL YOU!
WHEN PYTHON INVOKES SPECIAL METHODS

The interpreter invokes special methods on user defined types to handle:

- arithmetic and boolean expressions — i.e. operator overloading
- implicit conversion to `str` (ex: `print(x)`)
- conversion to `bool` in boolean contexts: `if, while, and, or, not`
- attribute access (`o.x`), including dynamic or virtual attributes
- emulating collections: `o[k], k in o, len(o)`
- iteration: `for`, tuple unpacking, star arguments — `f(*x)`
- context managers — `with` blocks
- metaprogramming: attribute descriptors, metaclasses
FIRST EXAMPLE

A vector class for applied math
EUCLIDEAN VECTOR

This is only a didactic example. To work with vectors and matrices, you need **NumPy**!

```python
>>> v1 = Vector([2, 4])
>>> v2 = Vector([2, 1])
>>> v1 + v2
Vector([4.0, 5.0])
```

http://x.co/pythonic

https://github.com/fluentpython/pythonic-api
from array import array
import math

class Vector:
    typecode = 'd'

    def __init__(self, components):
        self._components = array(self.typecode, components)

    def __len__(self):
        return len(self._components)

    def __iter__(self):
        return iter(self._components)

    def __abs__(self):
        return math.sqrt(sum(x * x for x in self))

    def __eq__(self, other):
        return (len(self) == len(other) and all(a == b for a, b in zip(self, other)))
from array import array
import math

class Vector:
    typecode = 'd'

    def __init__(self, components):
        self._components = array(self.typecode, components)

    def __len__(self):
        return len(self._components)

    def __iter__(self):
        return iter(self._components)

    def __abs__(self):
        return math.sqrt(sum(x * x for x in self))

    def __eq__(self, other):
        return (len(self) == len(other) and all(a == b for a, b in zip(self, other)))
SOME SPECIAL METHODS

__len__
Number of items in a collection

>>> len('abc')
3
>>> v1 = Vector([3, 4])
>>> len(v1)
2

__iter__
Implements the Iterable interface, returns an Iterator

>>> x, y = v1
>>> x, y
(3.0, 4.0)
>>> list(v1)
[3.0, 4.0]

__abs__
Implements abs(): the absolute value or modulus

>>> abs(3+4j)
5.0
>>> abs(v1)
5.0

__eq__
Overloaded equality operator ==

>>> v1 == Vector((3.0, 4.0))
True
Beyond `toString()`
Objects should produce two standard text representations.

**str(o)**
Display to users, via `print()` or UI
Implemented via `__str__`
`object.__str__` delegates to `__repr__`

**repr(o)**
Text for debugging, logging etc.
Implement via `__repr__`; if possible, emulate syntax to rebuild object.

---

**How to display an object as a string: `printString` and `displayString`**

Bobby Woolf

---

When I talk about how to use different sorts of objects, people often ask me what these objects look like. I draw a bunch of bubbles and arrows, underline things while I’m talking, and (hopefully) people nod knowingly. The bubbles are the objects I’m talking about, and the arrows are the pertinent relationships between them. But of course the diagram is not just circles and lines; everything has labels to identify them. The labels for the arrows are easy: The name of the method in the source that returns the target. But the labels for the bubbles are not so obvious. It’s a label that somehow describes the object and tells you which one it is. We all know how to label objects in this way, but what is it that we’re doing?

This is a Smalltalk programmer’s first brush with a bigger issue: How do you display an object as a string? Turns out this is not a very simple issue. VisualWorks gives you four different ways to display an object as a string: `printString`, `displayString`, `TypeConverter`, and `PrintConverter`. Why do these need to be more than one way? Which option do you use when?

This article is in two parts. This month, I’ll talk about `printString` and `displayString`. In September, I’ll talk about `TypeConverter` and `PrintConverter`.

`printString` AND `displayString`:
There are two messages you can send to an object to display it as a string:
- `printString:`—Displays the object the way the developer wants to see it.
- `displayString:`—Displays the object the way the user wants to see it.

`printString` is an old as Smalltalk itself. It was part of the original Smalltalk-80 standard and was probably in Smalltalk long before that. It is an essential part of how `Inspector` is implemented, an inspector being a development tool that can open a window to display any object. An inspector shows all of an object’s slots (its named and indexed instance variables); when you select one, it shows that slot’s value as a string by sending the slot’s value the message `printString`. The inspector also shows another slot, the `performable` self. When you select that slot, the inspector displays the object’s inspecting by sending it `printString`.

`displayString` was introduced in VisualWorks 1.0, more than 10 years after `printString`. `displayString` is an essential part of how `SequenceView` (VisualWorks’ List widget) is implemented. The list widget displays its items by displaying a string for each item. The purpose of this display-string is very similar to that of the `print-string`, but the results are often different.

`printString` describes an object to a Smalltalk programmer.
To a programmer, one of an object’s most important properties is its class. Thus, a `printString` either names the object’s class explicitly (a `VisualLauncher`, `OrderedCollection`, `#(#, #)`, etc.) or the class is implied (`printString` is a `Symbol`, `1.2` is a `Float`, etc.). The user, on the other hand, couldn’t care less what an object’s class is. Because most users don’t know OO, telling them that this is an object and what its class is would just confuse them. The user wants to know the name of the object. `displayString` describes the object to the user by printing the object’s name (although what constitutes an object’s “name” is open to interpretation).

STANDARD IMPLEMENTATION:
The first thing to understand about `printString` is that it doesn’t do much: its companion method, `printOn`, does all of the work. This makes `printString` more efficient because it uses a stream for concatenation.1 Here are the basic implementations in VisualWorks:

```
Object>>printString:
| aStream |
aStream => WriteStream on: (String new: 16).
self printOn: aStream.
aStream contents.

Object>>printOn:
| aStream |
title |
title => self class name.
```

1 http://www.ags.com The Smalltalk Report
from array import array
import math
import reprlib

class Vector:
    typecode = 'd'

    # ...

    def __str__(self):
        return str(tuple(self))

    def __repr__(self):
        components = reprlib.repr(self._components)
        components = components[components.find('['):-1]
        return 'Vector({})'.format(components)
from array import array
import math
import reprlib

class Vector:
    typecode = 'd'

    # ...

    def __str__(self):
        return str(tuple(self))

    def __repr__(self):
        components = reprlib.repr(self._components)
        components = components[:components.find('['):-1]
        return 'Vector({})'.format(components)
INDEXING & SLICING

Supporting the [ ] operator
THE [ ] OPERATOR

To support [], implement __getitem__

The same method is used to get items by index/key and to get slices. Implementing slicing is not mandatory (and may not make sense).

After self, __getitem__ gets an argument which can be:

- An integer index
- An arbitrary key (ex. a tuple)
- An instance of the slice type

```python
>>> class Foo:
...     def __getitem__(self, x):
...         return 'x -> ' + repr(x)
...     
...
>>> o = Foo()
>>> o[42]
'x -> 42'
>>> o[1:3]
'x -> slice(1, 3, None)'
>>> o[10:100:3]
'x -> slice(10, 100, 3)'
```
from array import array
import math
import reprlib
import numbers

class Vector:
    typeid = 'd'

    # ...

def __getitem__(self, index):
    cls = type(self)
    if isinstance(index, slice):
        return cls(self._components[index])
    elif isinstance(index, numbers.Integral):
        return self._components[index]
    else:
        msg = '{cls.__name__} indices must be integers'
        raise TypeError(msg.format(cls=cls))
from array import array
import math
import reprlib
import numbers

class Vector:
    typecode = 'd'

    # ...

def __getitem__(self, index):
    cls = type(self)
    if isinstance(index, slice):
        return cls(self._components[index])
    elif isinstance(index, numbers.Integral):
        return self._components[index]
    else:
        msg = '{cls.__name__} indices must be integers'
        raise TypeError(msg.format(cls=cls))

>>> v = Vector([10, 20, 30, 40, 50])
>>> v[0]
10.0
>>> v[-1]
50.0
>>> v[:3]
Vector([10.0, 20.0, 30.0])
Applying the double dispatch pattern
Compound interest formula, compatible with all numeric types in the standard library, including **Fraction** and **Decimal**:

\[
\text{interest} = \text{principal} \times ((1 + \text{rate})^{\text{periods}} - 1)
\]

Java version of the same formula, coded for **BigDecimal**:

```java
interest = principal.multiply(BigDecimal.ONE.add(rate).pow(periods).subtract(BigDecimal.ONE));
```
OPERATOR OVERLOADING

Special methods such as __add__, __eq__, __xor__ etc. implement arithmetic, comparison and bitwise.

Page 13 of Fluent Python:

```
>>> a = 2
>>> b = 3
>>> a + b
5
>>> a.__add__(b)
5
```

<table>
<thead>
<tr>
<th>Category</th>
<th>Method names and related operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unary numeric operators</td>
<td><strong>neg</strong>, <strong>pos</strong>, <strong>abs</strong> abs()</td>
</tr>
<tr>
<td>Rich comparison operators</td>
<td><strong>lt</strong>, <strong>le</strong>, <strong>eq</strong>, <strong>ne</strong>, <strong>gt</strong>, <strong>ge</strong></td>
</tr>
<tr>
<td>Arithmetic operators</td>
<td><strong>add</strong>, <strong>sub</strong>, <strong>mul</strong>, <strong>truediv</strong>, <strong>floordiv</strong>, <strong>mod</strong>, <strong>divmod</strong>, <strong>pow</strong>, <strong>round</strong>, __round()</td>
</tr>
<tr>
<td>Bitwise operators</td>
<td><strong>invert</strong>, <strong>lshift</strong>, <strong>rshift</strong>, <strong>and</strong>, <strong>or</strong>, <strong>xor</strong></td>
</tr>
</tbody>
</table>

Table 1-2. Special method names for operators
from array import array
import math
import reprlib
import numbers

class Vector:
    typecode = 'd'

    # ...

def __mul__(self, scalar):
    if isinstance(scalar, numbers.Real):
        return Vector(n * scalar for n in self)
    else:
        return NotImplemented

>>> v1 = Vector([1, 2, 3])
>>> v1 * 10
Vector([10.0, 20.0, 30.0])
>>> from fractions import Fraction
>>> v1 * Fraction(1, 3)
Vector([0.3333333333333333, 0.6666666666666667, 1.0])
A PROBLEM...

The expression `a * b` should call `a.__mul__(b)`.

But if `a` is an `int`, the `__mul__` method `int` can’t handle a `Vector` operand.
DOUBLE-DISPATCH

\[ a \ast b \]

- \( a \) has \_mul\_?
  - yes: call \( a._\text{mul}_(b) \)
  - no: result is \text{NotImplemented}?
    - yes: raise \text{TypeError}
    - no: return result

- \( b \) has \_rmul\_?
  - yes: call \( b._\text{rmul}_(a) \)
  - no: result is \text{NotImplemented}?
    - yes: raise \text{TypeError}
    - no: return result
from array import array
import math
import reprlib
import numbers

class Vector:
    typecode = 'd'

    # ...

def __mul__(self, scalar):
    if isinstance(scalar, numbers.Real):
        return Vector(n * scalar for n in self)
    else:
        return NotImplemented

def __rmul__(self, scalar):
    return self * scalar

>>> v1 = Vector([1, 2, 3])
>>> v1 * 10
Vector([10.0, 20.0, 30.0])
>>> 10 * v1
Vector([10.0, 20.0, 30.0])
One of the new features in Python 3.5
@ OPERATOR

Intended for matrix multiplication or vector dot product.

\[ A \cdot B = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n \]

No implementation in the standard library, but NumPy uses it.
@ OPERATOR

Intended for matrix multiplication or vector dot product. No implementation in the standard library, but NumPy uses it.

\[ A \cdot B = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n \]

```python
>>> va = Vector([1, 2, 3])
>>> vz = Vector([5, 6, 7])
>>> va @ vz  # 1*5 + 2*6 + 3*7
38
>>> [10, 20, 30] @ vz
380.0
>>> va @ 3
Traceback (most recent call last):
  ...
TypeError: unsupported operand type(s) for @: 'Vector' and 'int'
```
from array import array
import math
import reprlib
import numbers

class Vector:
    typecode = 'd'

    # ...

    def __matmul__(self, other):
        try:
            return sum(a * b for a, b in zip(self, other))
        except TypeError:
            return NotImplemented

    def __rmatmul__(self, other):
        return self @ other  # only valid in Python 3.5

>>> va = Vector([1, 2, 3])
>>> vz = Vector([5, 6, 7])
>>> va @ vz  # 1*5 + 2*6 + 3*7
38
>>> [10, 20, 30] @ vz
380.0
from array import array
import math
import reprlib
import numbers

class Vector:
    typecode = 'd'

    # ...

def __matmul__(self, other):
    try:
        return sum(a * b for a, b in zip(self, other))
    except TypeError:
        return NotImplemented

def __rmatmul__(self, other):
    return self @ other  # only valid in Python 3.5
EXERCISE 1
Implementing more Vector operators
UNARY - AND BINARY + FOR VECTORS

In this exercise you’ll implement the \_\_neg\_\_ and \_\_add\_\_ methods.

Clone or download repository from:

https://github.com/fluentpython/pythonic-api

Short link to exercise instructions online:

http://x.co/pythonic1
BONUS: HASHING

Using Vector instances in sets and as dict keys
An object is *hashable* if and only if:

- Its value is immutable
- It implements `__hash__`
- It implements `__eq__`
- When `a == b`, then `hash(a) == hash(b)`
COMPUTING THE HASH OF AN OBJECT

Basic algorithm: compute the hash of each object attribute or item and aggregate recursively with xor.

```python
>>> v1 = Vector([3, 4])
>>> hash(v1) == 3 ^ 4
True
>>> v3 = Vector([3, 4, 5])
>>> hash(v3) == 3 ^ 4 ^ 5
True
>>> v6 = Vector(range(6))
>>> hash(v6) == 0 ^ 1 ^ 2 ^ 3 ^ 4 ^ 5
True
>>> v2 = Vector([3.1, 4.2])
>>> hash(v2) == hash(3.1) ^ hash(4.2)
True
```
MAP-REDUCE
An example of map-reduce:

```python
def __hash__(self):
    hashes = (hash(x) for x in self)
    return functools.reduce(operator.xor, hashes, 0)
```

```python
>>> {v1, v2, v3, v6}
{Vector([0.0, 1.0, 2.0, 3.0, 4.0, ...]), Vector([3.0, 4.0, 5.0]), Vector([3.0, 4.0]), Vector([3.1, 4.2])}
```
RETHINKING STRATEGY

Handling language building blocks programmatically
The Classic Strategy Pattern

**Strategy** in the *Design Patterns* book by Gamma et.al.:

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

![Diagram showing the Strategy pattern with an Order context and concrete strategies for FidelityPromo, BulkItemPromo, and LargeOrderPromo]
CLASSIC STRATEGY: WHAT IT DOES

The API for a classic **Strategy** implementation looks like this:

```python
>>> joe = Customer('John Doe', 0)   # <1>
>>> ann = Customer('Ann Smith', 1100)
>>> cart = [LineItem('banana', 4, .5),  # <2>
...         LineItem('apple', 10, 1.5),
...         LineItem('watermellon', 5, 5.0)]
>>> Order(joe, cart, FidelityPromo())  # <3>
<Order total: 42.00 due: 42.00>
>>> Order(ann, cart, FidelityPromo())  # <4>
<Order total: 42.00 due: 39.90>
>>> banana_cart = [LineItem('banana', 30, .5),  # <5>
...                LineItem('apple', 10, 1.5)]
>>> Order(joe, banana_cart, BulkItemPromo())  # <6>
<Order total: 30.00 due: 28.50>
>>> long_order = [LineItem(str(item_code), 1, 1.0)  # <7>
...                for item_code in range(10)]
>>> Order(joe, long_order, LargeOrderPromo())  # <8>
<Order total: 10.00 due: 9.30>
>>> Order(joe, cart, LargeOrderPromo())
<Order total: 42.00 due: 42.00>
```
class Order:  # the Context

def __init__(self, customer, cart, promotion=None):
    self.customer = customer
    self.cart = list(cart)
    self.promotion = promotion

def total(self):
    if not hasattr(self, '__total'):
        self.__total = sum(item.total() for item in self.cart)
    return self.__total

def due(self):
    if self.promotion is None:
        discount = 0
    else:
        discount = self.promotion.discount(self)
    return self.total() - discount

def __repr__(self):
    fmt = '<Order total: {:.2f} due: {:.2f}>'
    return fmt.format(self.total(), self.due())
class Promotion(ABC):  # the Strategy: an Abstract Base Class
    @abstractmethod
    def discount(self, order):
        """Return discount as a positive dollar amount""

class FidelityPromo(Promotion):  # first Concrete Strategy
"""5% discount for customers with 1000 or more fidelity points""

def discount(self, order):
    return order.total() * .05 if order.customer.fidelity >= 1000 else 0

class BulkItemPromo(Promotion):  # second Concrete Strategy
"""10% discount for each LineItem with 20 or more units""

def discount(self, order):
    discount = 0
    for item in order.cart:
        if item.quantity >= 20:
            discount += item.total() * .1
    return discount
EXERCISE 2

Function-based Strategy implementation
EXERCISE: STRATEGY WITHOUT CLASSES

In this exercise you’ll refactor the Strategy design pattern, replacing the concrete strategy classes with functions. You should already have a local copy of the repository:

https://github.com/fluentpython/pythonic-api

Short link to exercise instructions online:

http://x.co/pythonic2
The `best_promo` strategy tries all available promotion strategies and returns the maximum discount.

```python
promos = [fidelity_promo, bulk_item_promo, large_order_promo]  # <1>
def best_promo(order):  # <2>
    """Select best discount available
    """
    return max(promo(order) for promo in promos)  # <3>
```

However, it depends on that `promos` list, which must be manually edited, duplicating information in the codebase — breaking the DRY principle.
Using the `inspect` module you can programmatically discover objects such as modules, classes, methods and functions.

```python
promos = [func for name, func in inspect.getmembers(promotions, inspect.isfunction)]

def best_promo(order):
    '''Select best discount available'''

    return max(promo(order) for promo in promos)
```

Here, `promotions` is a module imported at the top of this file, and we use `inspect.getmembers` and `inspect.isfunction` to build a list of all functions in the `promotions` module.
WRAP-UP

Considering how to make Pythonic APIs
Ideas to consider:

• study well designed packages from the Standard Library and from PyPI
• implement special methods from the Data Model to emulate the standard behaviors your users expect
• leverage first-class language objects: functions, modules, classes
  • thinking about classes as objects makes Factory patterns redundant!
• with great care: do some metaprogramming to reduce boilerplate code that your users would need to write
• review slide 20: Typical Characteristics of Idiomatic Libraries
Q & A

Come to the *Fluent Python* book signing Wednesday, June 22, 5:40 PM - 6:10

ThoughtWorks®

LUCIANO RAMALHO

Technical Principal

@ramalhoorg
luciano.ramalho@thoughtworks.com