O'REILLY

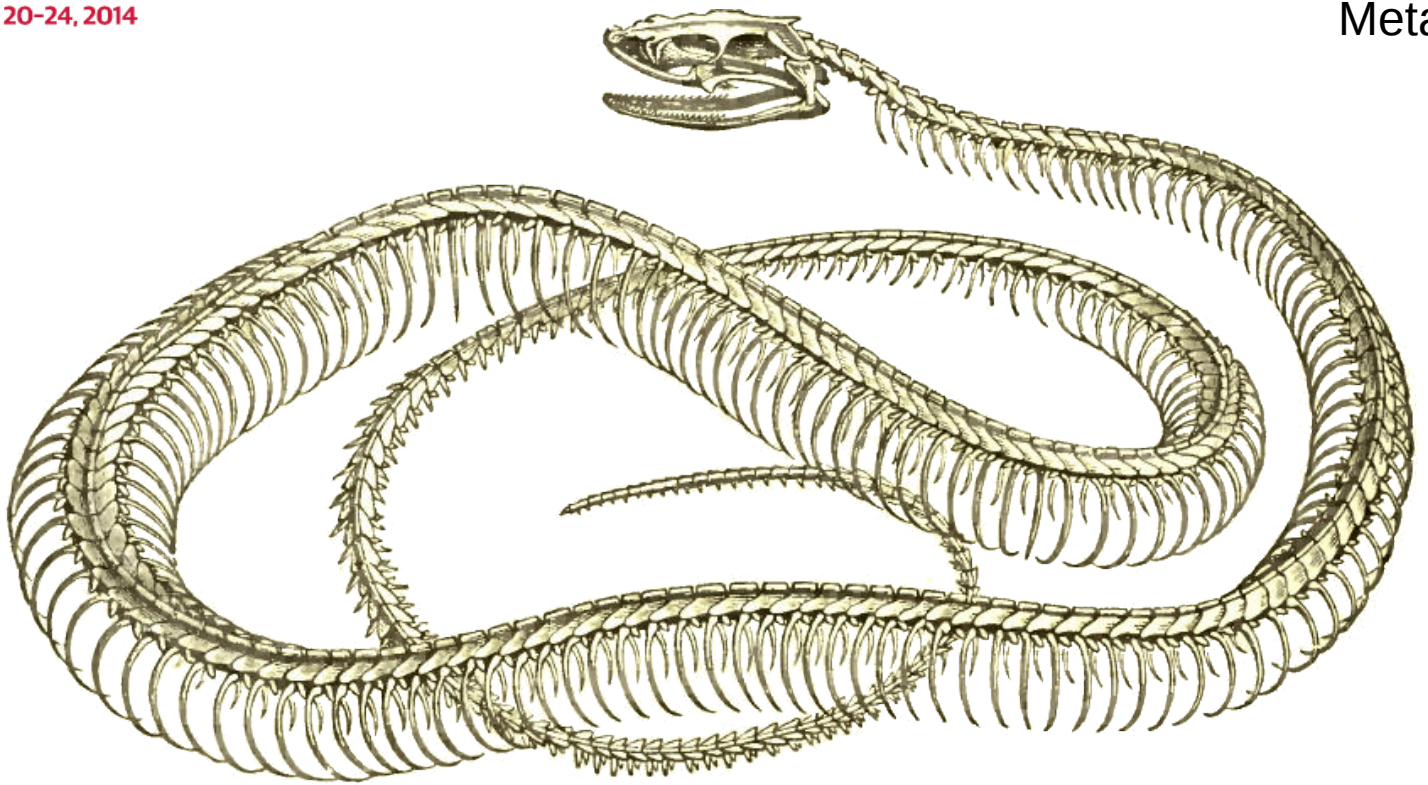
oscon

OPEN SOURCE CONVENTION

PORTLAND, OR JULY 20–24, 2014

Full Monty

Introduction to Python Metaprogramming



SKELETON OF SNAKE.

PythonPro

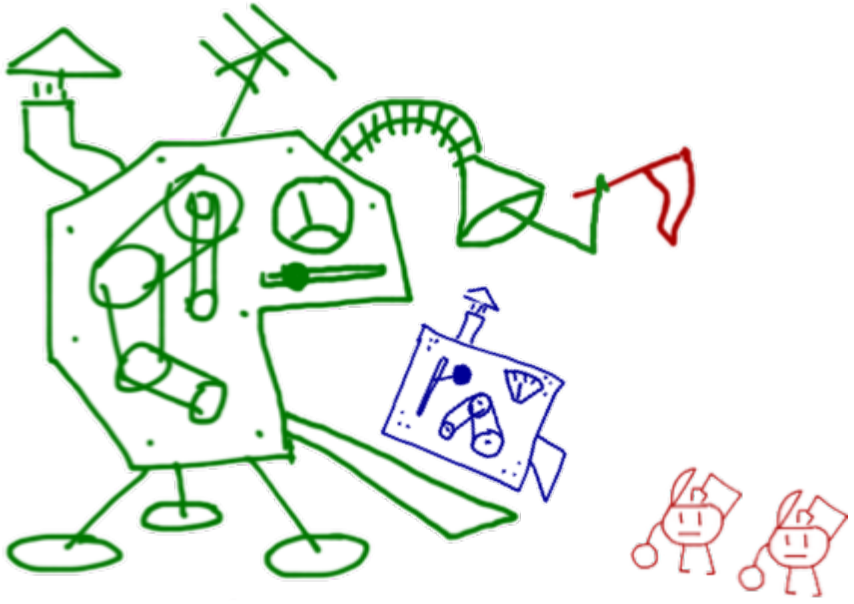
Luciano Ramalho
luciano@ramalho.org
Twitter: @ramalhoorg

Code for this tutorial

https://github.com/pythonprobr/oscon2014

Metaprogramming

- Modifying program structure at run time
- Easier to do in dynamic languages (e.g. Python) where the run-time environment includes the compiler/interpreter



PythonPro

Metaprogramming techniques

- Modifying/creating objects that represent language constructs in memory using the Python Data Model API

- Processing the Python AST (Abstract Syntax Tree)
- Generating byte code
- Generating source code

Agenda

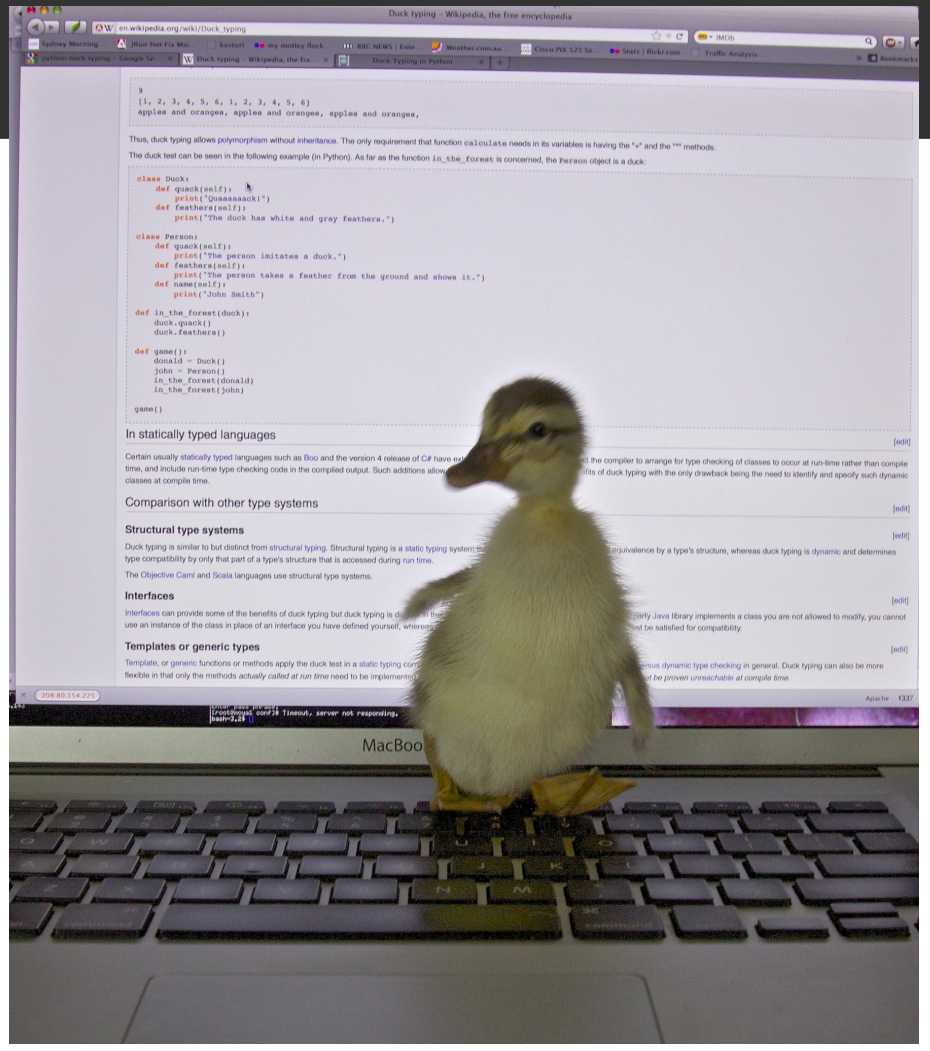
- Data model
  - Standard Type Hierarchy and Special methods
- Functions as objects
  - rethinking “Strategy”
  - introspection
  - function decorators
  - modules as objects
- Attributes
  - dynamic attributes
  - rethinking “Proxy”
  - descriptors: attributes as objects
- Classes as objects
  - class decorators
  - metaclasses

Formal Interfaces x Protocols

- Formal interface:
  - defined via ABCs - Abstract Base Classes
    - since Python 2.6/3.0
  - inherit from ABC, implement **all** methods marked @abstractmethod
  - checked upon object instantiation
- Protocol:
  - informal interface, defined by documentation
  - implementation: just methods
    - no need to inherit from ABC
  - may be partially implemented
    - if you don't need all the functionality of the full protocol

Duck Typing

- It's all about protocols
- The DNA of the bird is irrelevant
- All that matters is what it does
  - the methods it implements



duck typing, an intro (photo by flickr user cskk)

PythonPro

The Python Data Model

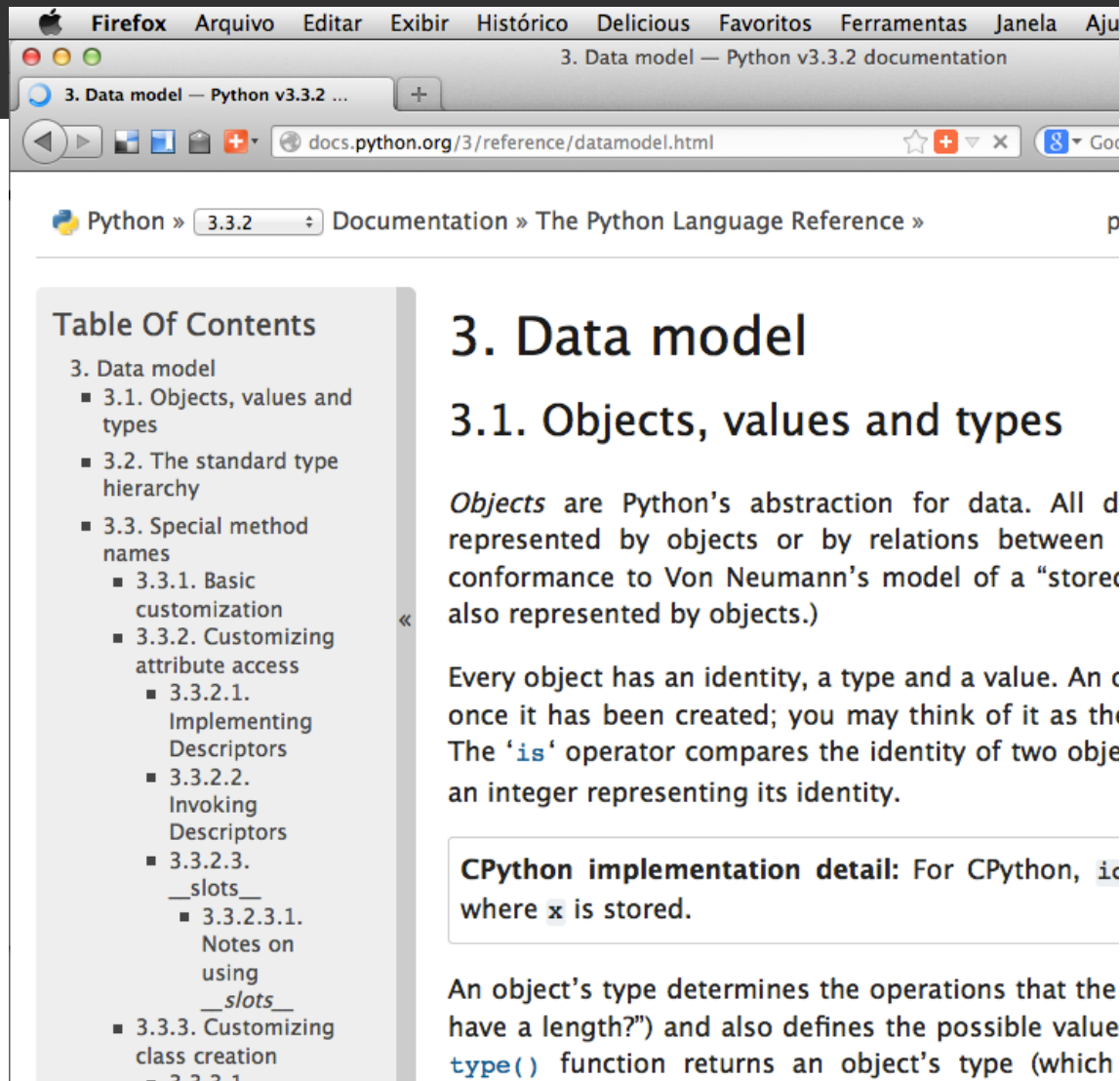
- Pragmatic
- Fixed extension points: special methods (e.g. __new__)
- Syntactic support:
  - keywords: for, yield, with, class, def, etc.
  - operators: almost all, including arithmetic and o.a, f(), d[i]

Not magic!

PythonPro

Special methods

- “Dunder” syntax: __iter__
  - “dunder” is shortcut for “double-underscore-prefix-and-suffix”
- Documented in the Python Language Reference:
  - 3.2 - The standard type hierarchy
  - 3.3 - Special method names



PythonPro

Example 1: Pythonic card deck

PythonPro

Example 1a: basic tests

```
>>> from frenchdeck import FrenchDeck
>>> deck = FrenchDeck()
>>> len(deck)
52
>>> deck[:3]
[Card(rank='2', suit='spades'), Card(rank='3',
suit='spades'), Card(rank='4', suit='spades')]
>>> deck[12::13]
[Card(rank='A', suit='spades'), Card(rank='A',
suit='diamonds'), Card(rank='A', suit='clubs'),
Card(rank='A', suit='hearts')]
```

Example 1b: the `in` operator

```
>>> from frenchdeck import Card
>>> Card('Q', 'hearts') in deck
True
>>> Card('Z', 'clubs') in deck
False
```

Example 1c: iteration

```
>>> for card in deck:
...     print(card)
...
Card(rank='2', suit='spades')
Card(rank='3', suit='spades')
Card(rank='4', suit='spades')
Card(rank='5', suit='spades')
Card(rank='6', suit='spades')
Card(rank='7', suit='spades')
Card(rank='8', suit='spades')
Card(rank='9', suit='spades')
Card(rank='10', suit='spades')
Card(rank='J', suit='spades')
Card(rank='Q', suit='spades')
Card(rank='K', suit='spades')
```

Example 1: implementation

An immutable sequence

- Sequence protocol:
  - __len__
  - __getitem__

```
import collections

Card = collections.namedtuple('Card', ['rank', 'suit'])

class FrenchDeck:
    ranks = [str(n) for n in range(2, 11)] + list('JQKA')
    suits = 'spades diamonds clubs hearts'.split()

    def __init__(self):
        self._cards = [Card(rank, suit)
                       for suit in self.suits
                       for rank in self.ranks]

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

    def __getitem__(self, position):
        return self._cards[position]
```

Exercise 1: Shuffle the deck

- Python has a random.shuffle function ready to use
- Should we implement a shuffle method in FrenchDeck?
- Pythonic alternative: make FrenchDeck a mutable sequence

```
>>> from random import shuffle
>>> l = list(range(10))
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> shuffle(l)
>>> l
[3, 9, 1, 0, 5, 7, 6, 8, 4, 2]
```

Exercise 1: instructions

- Visit https://github.com/pythonprobr/oscon2014.git
- Clone that repo or click “Download Zip” (right column, bottom)
- Go to frenchdeck/ directory
- Run doctests:

```
$ python3 -m doctest frenchdeck.doctest
$ python3 -m doctest shuffle.doctest
```

passes

fails

- Edit frenchdeck.py to make the shuffle.doctest pass

Hint: look up "Emulating container types" in the Python Data Model docs

PythonPro

Mutable Sequence protocol

A mutable sequence

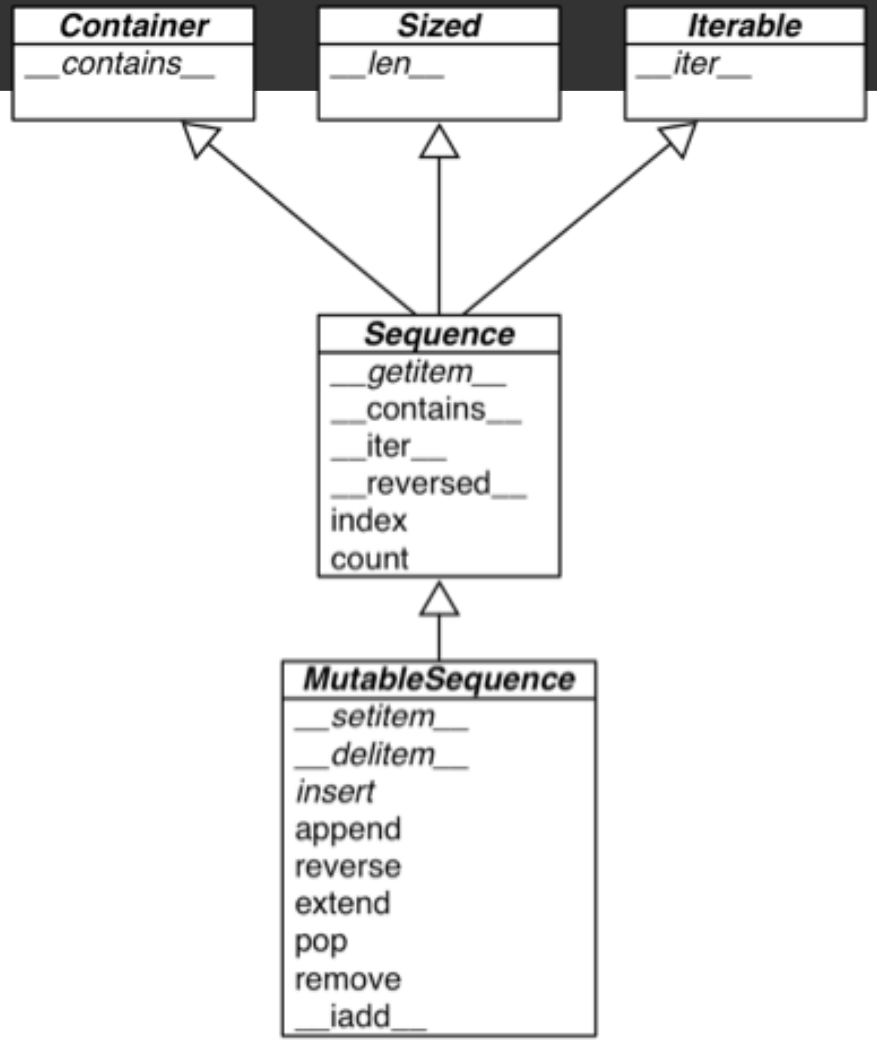
- The complete Mutable sequence protocol:
  - __setitem__
  - __delitem__
  - insert

```
import collections

Card = collections.namedtuple('Card', ['rank', 'suit'])

class FrenchDeck:
    ranks = [str(n) for n in range(2, 11)] + list('JQKA')
    suits = 'spades diamonds clubs hearts'.split()

    def __init__(self):
        self._cards = [Card(rank, suit)
                       for suit in self.suits
                       for rank in self.ranks]

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

    def __getitem__(self, position):
        return self._cards[position]

    def __setitem__(self, position, card):
        self._cards[position] = card
```

Partial implementation is OK for now

collections.abc

- Formal interfaces defined as abstract classes:
  - Container, Sized, Iterable
  - Sequence
  - MutableSequence
  - Mapping, MutableMapping
  - Set, MutableSet
  - etc.



PythonPro

Sequence ABC

- Sequence ABC:
  - __len__
  - __getitem__
- MutableSequence ABC:
  - __setitem__
  - __delitem__
  - insert

A mutable sequence

```
import collections
from collections import abc

Card = collections.namedtuple('Card', ['rank', 'suit'])

class FrenchDeck(abc.MutableSequence):
    ranks = [str(n) for n in range(2, 11)] + list('JQKA')
    suits = 'spades diamonds clubs hearts'.split()

    def __init__(self):
        self._cards = [Card(rank, suit)
                       for suit in self.suits
                       for rank in self.ranks]

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

    def __getitem__(self, position):
        return self._cards[position]

    def __setitem__(self, position, card):
        self._cards[position] = card

    def __delitem__(self, position):
        del self._cards[position]

    def insert(self, position, card):
        self._cards.insert(position, card)
```

Must also implement

PythonPro

Recap: special methods

- Emulate built-in collections
- Overload arithmetic, comparison and boolean operators
  - __add__, __lt__, __xor__...
  - **+ * / - == != <= >= & ^ | << >>**
- First steps into metaprogramming
- Attribute and item access operators
  - __getattr__, __setitem__...
  - a**.**x b**[**i**]** k **in** c
- Call operator: __call__
  - f**(**x**)** # same as f.__call__(x)
- Think about functions as objects

PythonPro

Insight: why is len a function?

- Sequences are a family of standard types unrelated by inheritance but united by common protocols
  - len(s), e in s
  - s[i], s[a:b:c]
  - for e in s, reversed(s)
  - s1 + s2, s * n
- Just as numeric types are united by common arithmetic operators and functions

Nobody complains that ***abs(x)*** *should be* ***x.abs()*** *or claims that* ***x.add(y)*** *is better than* ***x + y***

Insight: why len is a function

- Pragmatics:
  - len(s) is more readable than s.len() – Guido's opinion
  - len(s) is shorter than s.len()
  - len(s) is faster than s.len()
- Why len(s) is faster than s.len()
  - for built-in types, the CPython implementation of len(s) simply gets the value of the ob_size field in a PyVarObject C struct
  - no attribute lookup and no method call

Insight: __len__ increases consistency

- The __**len**__ special method is a fallback called by the implementation of **len** for user defined types
- With this, **len(s)** is fast for built-ins but *also* works for any sequence s
  - no need to remember whether to write s.length or s.length() depending on the type of s

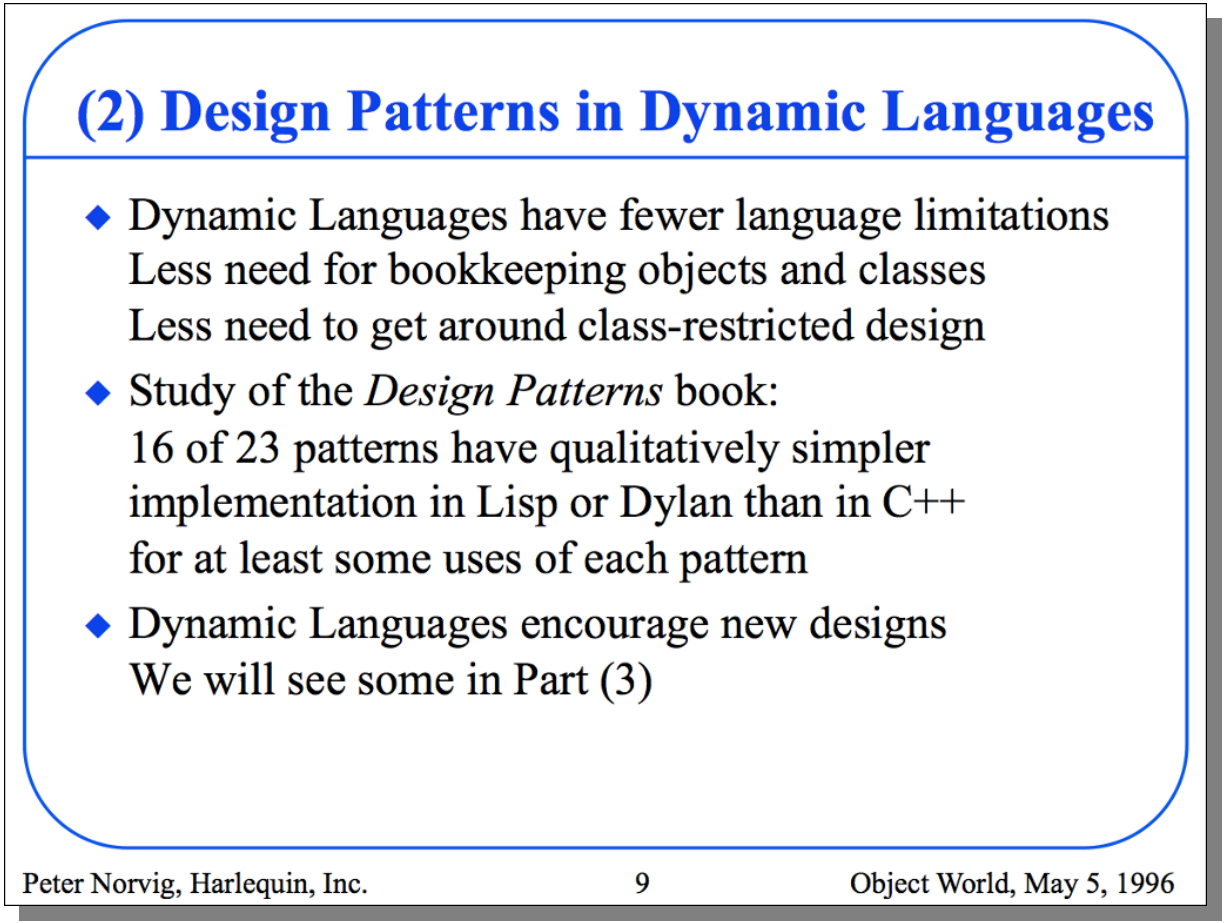
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
(from The Zen of Python)

Case study

Refactoring the Strategy Pattern
with function objects

Rethinking Design Patterns

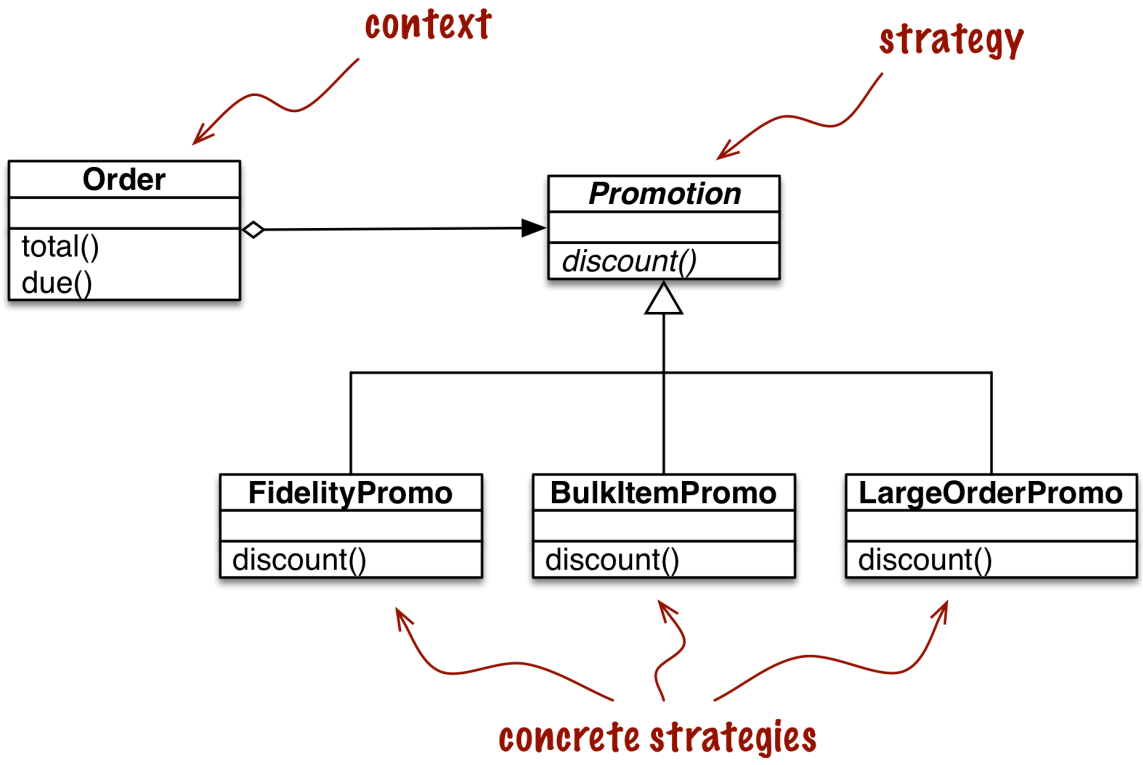
- Peter Norvig's 1996 presentation “Design Patterns in Dynamic Programming”
- Most refactorings Norvig suggests also apply to Python



The Strategy Pattern

- From the GoF* book:

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



PythonPro

* Design Patterns – Gamma, Helm, Johnson & Vlissides (a.k. a. the “Gang of Four”)

Strategy at work

- Order constructor takes 3 arguments:
  - customer
  - shopping cart
  - promotional discount strategy

```
>>> 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>
```

PythonPro

Classic Strategy

- The **context** is an Order
- An Order has:
  - a Customer
  - a collection of LineItems
  - a Promotion (the strategy)

The Context

```
class Order:

    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())
```

Calls promotion.discount()

Classic Strategy (3)

- Promotion ABC defines the Strategy interface
- FidelityPromo is the first concrete strategy

The Strategy

```
class Promotion(metaclass=ABCMeta):

    @abstractmethod
    def discount(self, order):
        """Return discount as positive dollar amount"""
```

A concrete strategy

```
class FidelityPromo(Promotion):
    """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)
```

Classic Strategy (4)

- BulkItemPromo is the second concrete strategy

```
class BulkItemPromo(Promotion):
    """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
```

Another concrete strategy

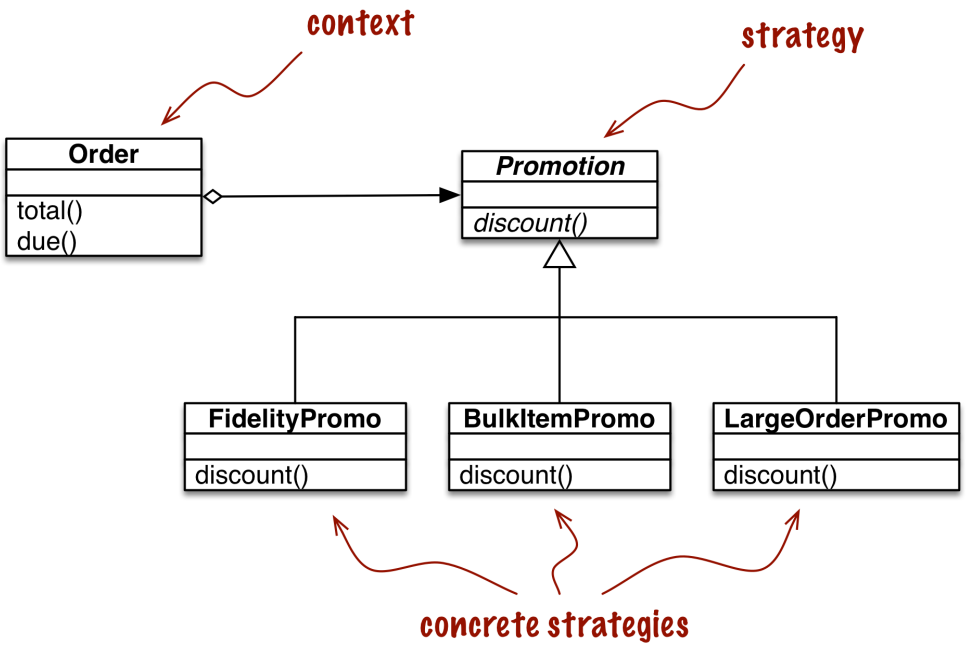
Classic Strategy (5)

- LargeOrderPromo is the third concrete strategy

```
class LargeOrderPromo(Promotion):
    """7% discount for orders with 10 or more distinct items"""

    def discount(self, order):
        distinct_items = {item.product for item in order.cart}
        if len(distinct_items) >= 10:
            return order.total() * .07
        return 0
```

Third concrete strategy

Strategy at work



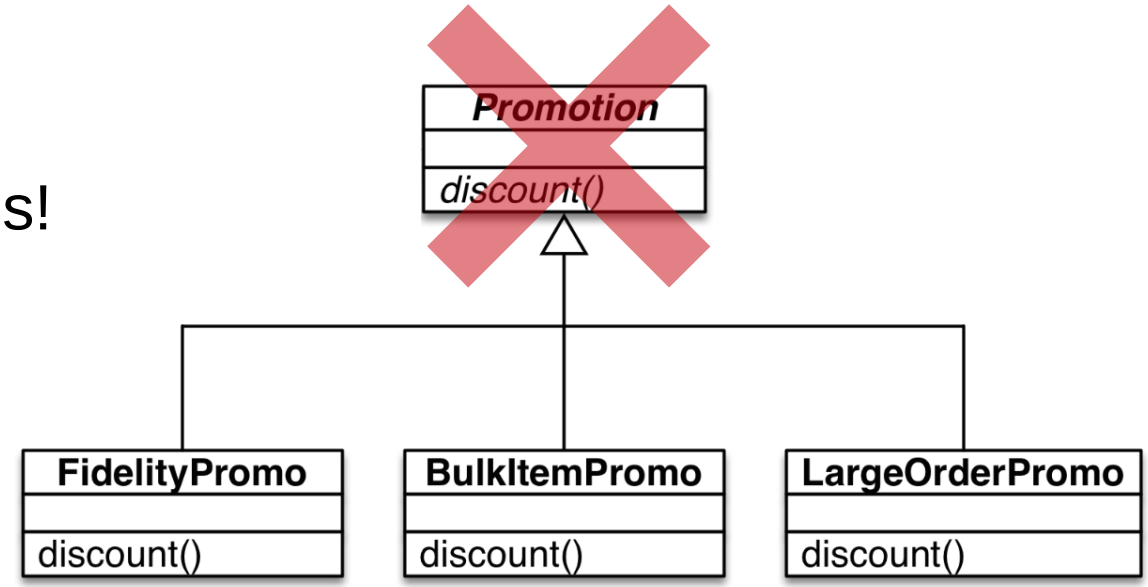
- A concrete strategy is instantiated for each order

```
>>> joe = Customer('John Doe', 0)
>>> ann = Customer('Ann Smith', 1100)
>>> cart = [LineItem('banana', 4, .5),
...         LineItem('apple', 10, 1.5),
...         LineItem('watermellon', 5, 5.0)]
>>> Order(joe, cart, FidelityPromo())
<Order total: 42.00 due: 42.00>
>>> Order(ann, cart, FidelityPromo())
<Order total: 42.00 due: 39.90>
```

PythonPro

Rethinking Strategy

- Concrete strategies (in this example):
  - have only one method, discount()
  - have no internal state
- Could be simple functions!



PythonPro

Functions as objects

Building a list of functions:
static and dynamic solutions

Choosing the best strategy

- Given an order, the **best_promo** function tests every other promotion on the order and returns the maximum discount

List of function objects

```
promos = [fidelity_promo, bulk_item_promo, large_order_promo]

def best_promo(order):
    """Select best discount available
    """
    return max(promo(order) for promo in promos)
```

Using introspection to build list of functions

- Get all names from the **current module** global dictionary
- Select those that end with "**_promo**"
- Ignore "**best_promo**" (to avoid infinite recursion)

```
promos = [globals()[name] for name in globals()
          if name.endswith('_promo')
          and name != 'best_promo']
```

Complete source code: `http://bit.ly/strategy_best2`

Using introspection to build list of functions (2)

- Get strategy functions from a **promotions** module
- Use **inspect** module to make it easier
- **best_promo** should not be in the **promotions** module

```
import promotions
# ...

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

Complete source code: `http://bit.ly/strategy_best3`

Using introspection to build list of functions (3)

- Use a decorator to register strategy functions

```
promos = []

def promotion(promo_func):
    promos.append(promo_func)
    return promo_func

@promotion
def fidelity_promo(order):
    """5% discount for customers with 1000 or more fidelity points"""
    return order.total() * .05 if order.customer.fidelity >= 1000 else 0
```

PythonPro

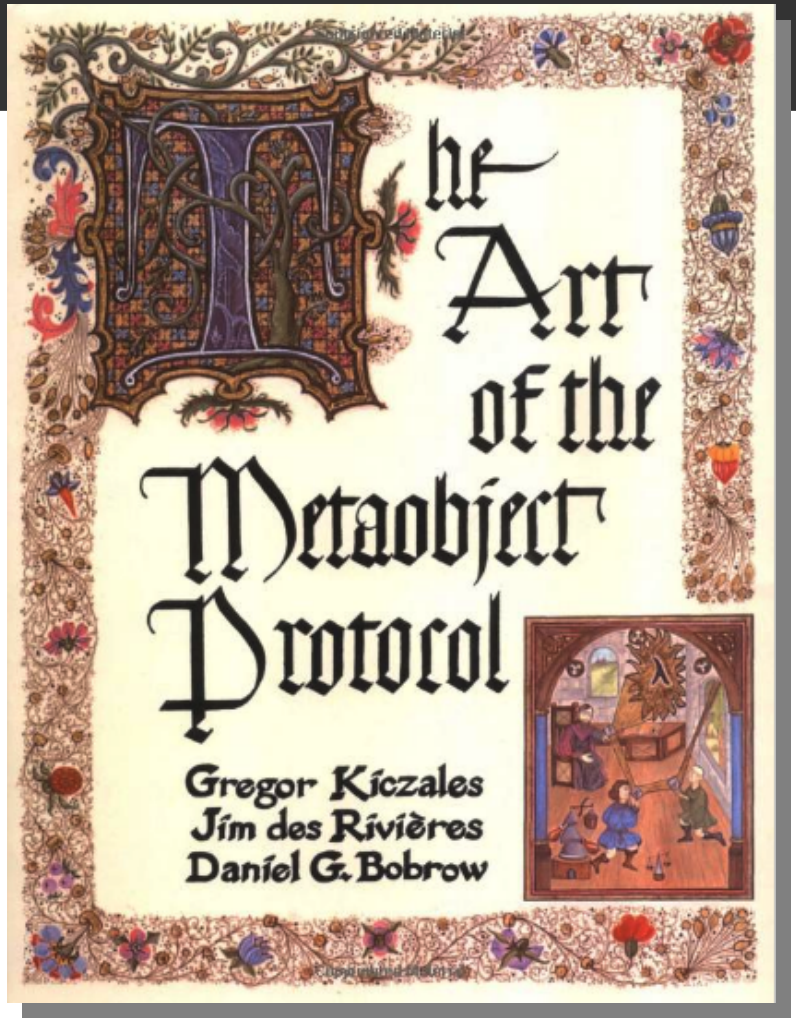
Complete source code: `http://bit.ly/strategy_best4`

Recap: functions and modules as objects

- Organize functions into data structures (list, dict...)
- Module is a built-in data structure that holds functions, classes and other names
- **globals()**, **locals()** and **vars()** return mappings of names and objects in different scopes
- **inspect** module has many functions that help with introspection of modules, classes, functions etc.
  - getmembers, isfunction etc.
- Decorators can change the behavior of functions or merely register

Metaobject Protocol

- Background: implementation of OO features in Common Lisp
- Key idea: think of language constructs as objects
  - functions, classes, modules, object accessors/mutators...
- Provide API for run-time handling of language constructs



PythonPro

Attribute descriptors

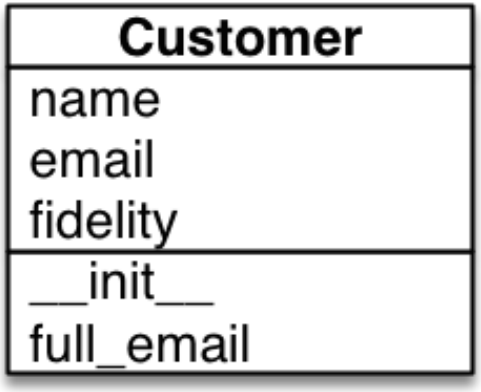
Intercepting attribute
access in a reusable way

Dynamic attribute access

- Built-in functions
  - getattr(obj, name)
  - setattr(obj, name, value)
- Special methods
  - __getattr__(self, name)
    - called when instance and class lookup fails
  - __setattr__(self, name, value)
    - **always** called for instance attribute assignment
  - __getattribute__(self, name, value)
    - almost always called for instance attribute access
    - low-level hook

Case study: reusing setter logic

1



- A simple **Customer** class
- The **name** and **email** fields should never be blank

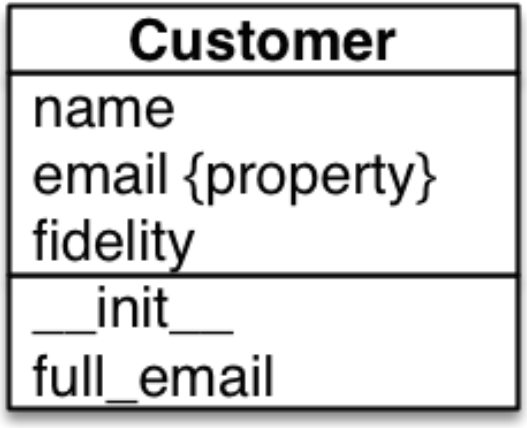
```
>>> joe = Customer('Joseph Blow', '')
>>> joe.full_email()
'Joseph Blow <>'
```

Problem

PythonPro

Step 2: rewrite email as a property

2

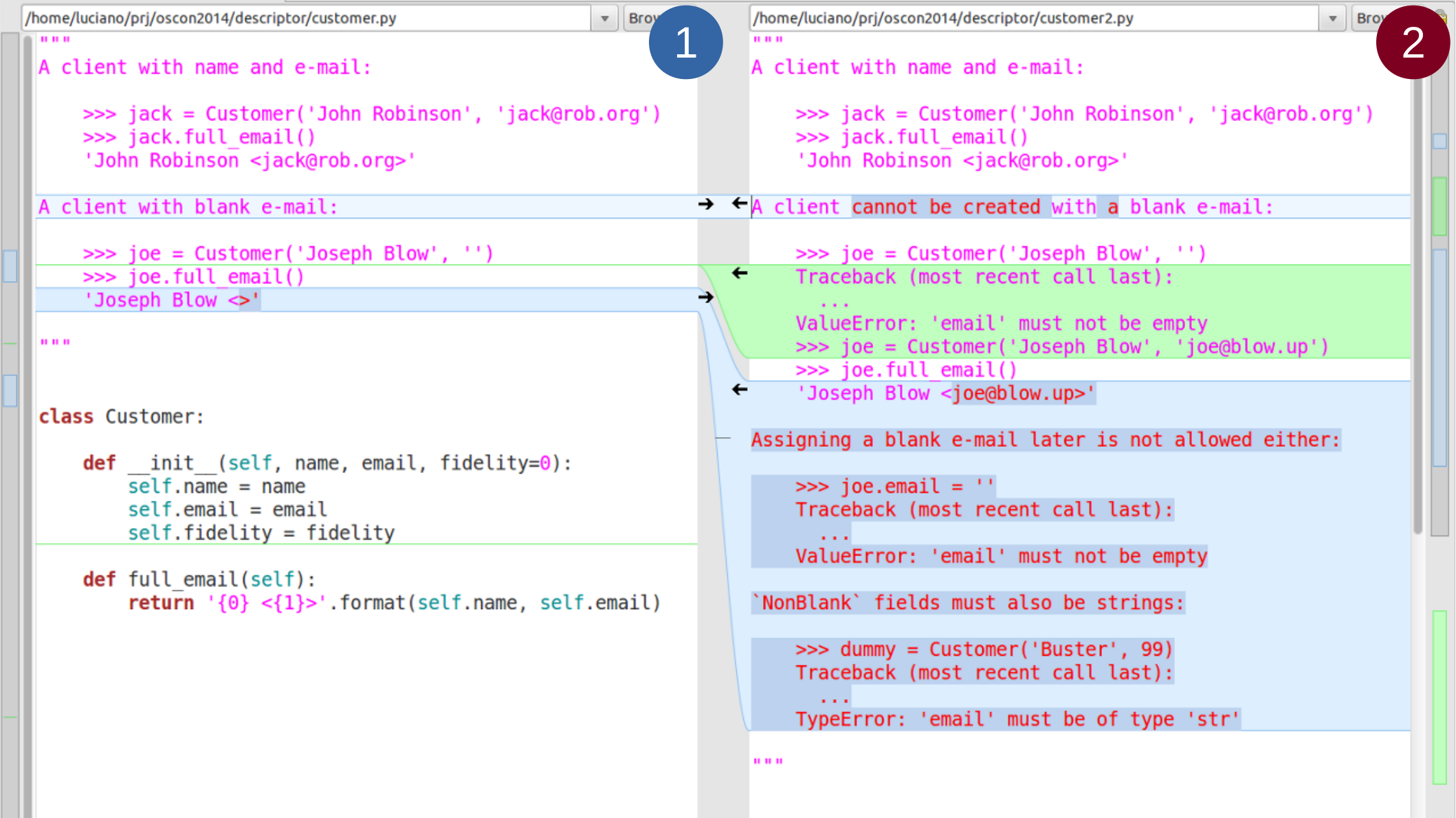


- **Customer** class with **email** property
- The **email** field can never be blank

```
>>> joe = Customer('Joseph Blow', '')
Traceback (most recent call last):
  ...
ValueError: 'email' must not be empty
```

*Problem solved (for **email**)*

PythonPro


/home/luciano/prj/oscon2014/descriptor/customer.py
1
"""
A client with name and e-mail:
>>> jack = Customer('John Robinson', 'jack@rob.org')
>>> jack.full_email()
'John Robinson <jack@rob.org>'
A client with blank e-mail:
>>> joe = Customer('Joseph Blow', '')
>>> joe.full_email()
'Joseph Blow <>'
"""
class Customer:
def __init__(self, name, email, fidelity=0):
self.name = name
self.email = email
self.fidelity = fidelity
def full_email(self):
return '{0} <{1}>'.format(self.name, self.email)
/home/luciano/prj/oscon2014/descriptor/customer2.py
2
"""
A client with name and e-mail:
>>> jack = Customer('John Robinson', 'jack@rob.org')
>>> jack.full_email()
'John Robinson <jack@rob.org>'
A client cannot be created with a blank e-mail:
>>> joe = Customer('Joseph Blow', '')
Traceback (most recent call last):
...
ValueError: 'email' must not be empty
>>> joe = Customer('Joseph Blow', 'joe@blow.up')
>>> joe.full_email()
'Joseph Blow <joe@blow.up>'
Assigning a blank e-mail later is not allowed either:
>>> joe.email = ''
Traceback (most recent call last):
...
ValueError: 'email' must not be empty
`NonBlank` fields must also be strings:
>>> dummy = Customer('Buster', 99)
Traceback (most recent call last):
...
TypeError: 'email' must be of type 'str'
"""

/home/luciano/prj/oscon2014/descriptor/customer.py

```
"""
A client with name and e-mail:

    >>> jack = Customer('John Robinson', 'jack@rob.org')
    >>> jack.full_email()
    'John Robinson <jack@rob.org>'

A client with blank e-mail:

    >>> joe = Customer('Joseph Blow', '')
    >>> joe.full_email()
    'Joseph Blow <>'

"""


class Customer:

    def __init__(self, name, email, fidelity=0):
        self.name = name
        self.email = email
        self.fidelity = fidelity

    def full_email(self):
        return '{0} <{1}>'.format(self.name, self.email)
```

1

/home/luciano/prj/oscon2014/descriptor/customer2.py

```
      ...
    ValueError: 'email' must not be empty

`NonBlank` fields must also be strings:

    >>> dummy = Customer('Buster', 99)
    Traceback (most recent call last):
      ...
    TypeError: 'email' must be of type 'str'

"""


class Customer:

    def __init__(self, name, email, fidelity=0):
        self.name = name
        self.email = email
        self.fidelity = fidelity

    @property
    def email(self):
        return self.__email

    @email.setter
    def email(self, value):
        if not isinstance(value, str):
            raise TypeError("'email' must be of type 'str'")
        elif len(value) == 0:
            raise ValueError("'email' must not be empty")
        self.__email = value

    def full_email(self):
        return '{0} <{1}>'.format(self.name, self.email)
```

2

/home/luciano/prj/oscon2014/descriptor/customer.py

```
"""
A client with name and e-mail:

    >>> jack = Customer('John Robinson', 'jack@rob.org')
    >>> jack.full_email()
    'John Robinson <jack@rob.org>'

A client with blank e-mail:

    >>> joe = Customer('Joseph Blow', '')
    >>> joe.full_email()
    'Joseph Blow <>'

"""


class Customer:

    def __init__(self, name, email, fidelity=0):
        self.name = name
        self.email = email
        self.fidelity = fidelity

    def full_email(self):
        return '{0} <{1}>'.format(self.name, self.email)
```

1

/home/luciano/prj/oscon2014/descriptor/customer2.py

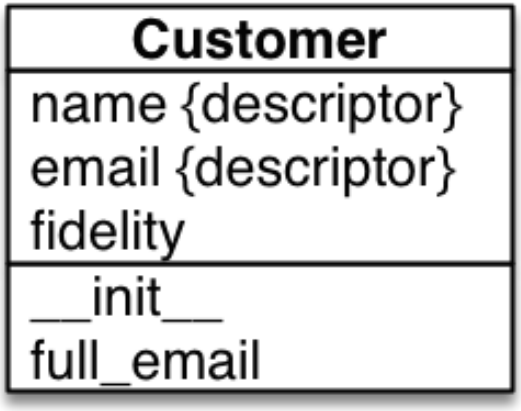
```
      ...
    ValueError: 'email' must not be empty

`NonBlank` fields must also be strings:

    >>> dummy = Customer('Buster', 99)
    Traceback (most recent call last):
      ...
    TypeError: 'email' must be of type 'str'

"""


class Customer:

    def __init__(self, name, email, fidelity=0):
        self.name = name
        self.email = email
        self.fidelity = fidelity

    @property
    def email(self):
        return self.__email

    @email.setter
    def email(self, value):
        if not isinstance(value, str):
            raise TypeError("'email' must be of type 'str'")
        elif len(value) == 0:
            raise ValueError("'email' must not be empty")
        self.__email = value

    def full_email(self):
        return '{0} <{1}>'.format(self.name, self.email)
```

2

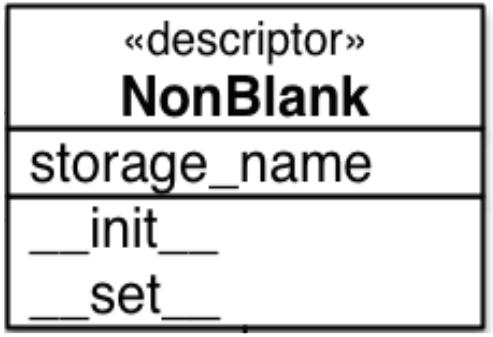
*Copy & paste for **name**???*

Step 3: create a descriptor

3



- A **descriptor** encapsulates attribute getting/setting logic in a reusable way
- Any class that implements **__get__** or **__set__** can be used as a descriptor

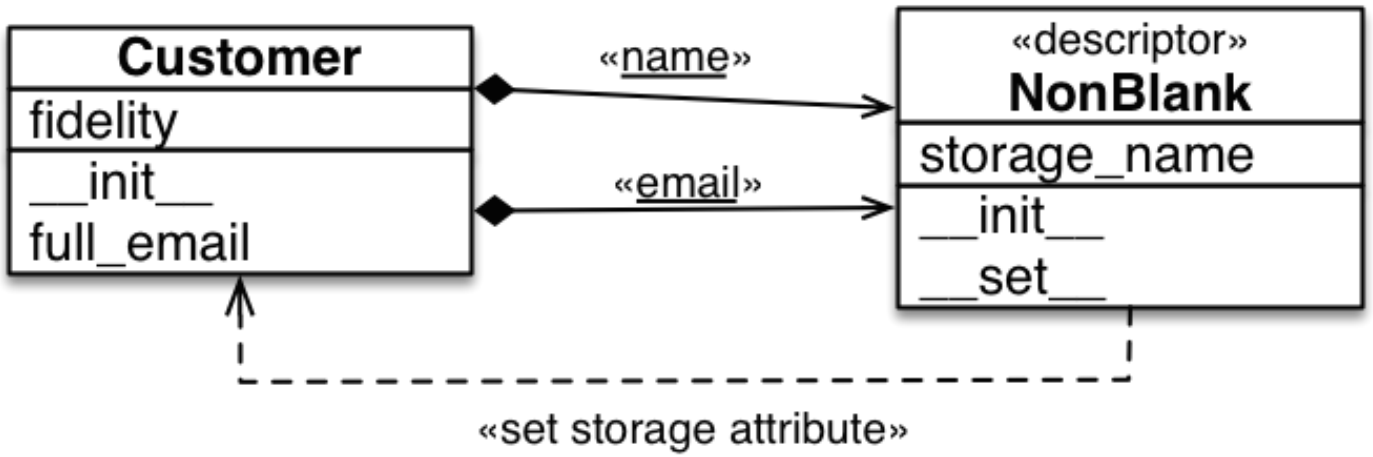


PythonPro

Step 3: create a descriptor

3

- **NonBlank** is an *overriding* descriptor
  - it implements **__set__**



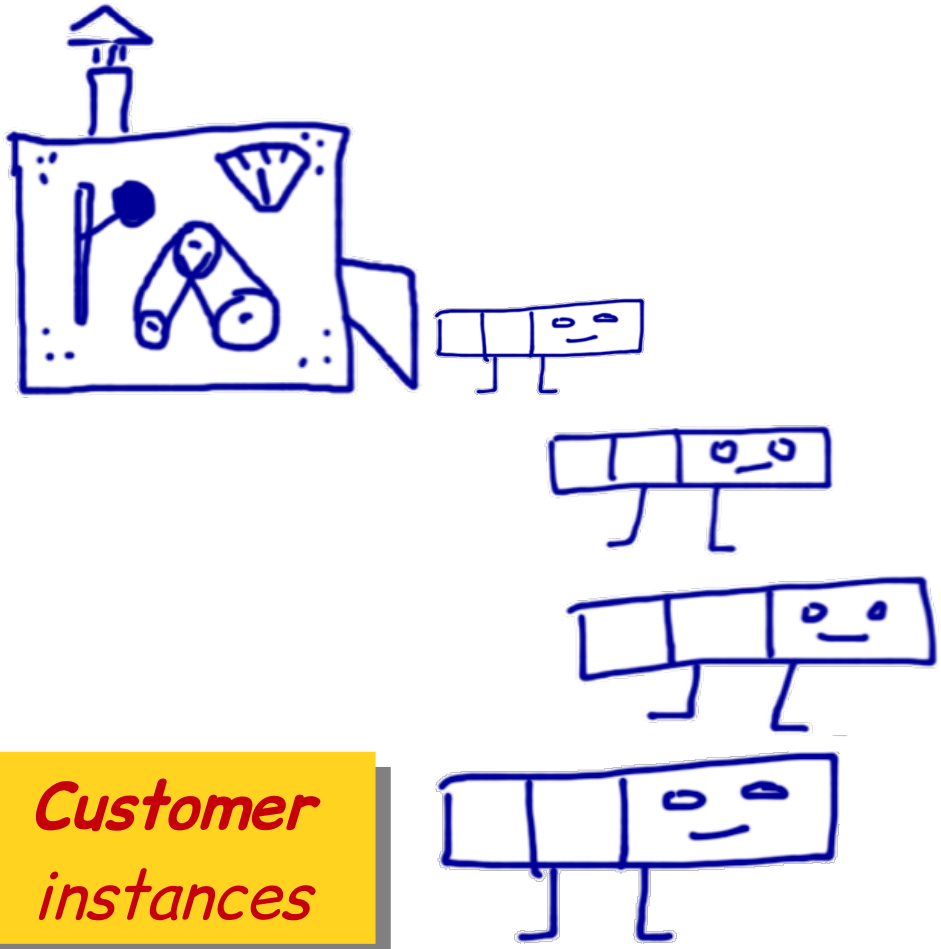
PythonPro

Step 3: cast of characters

Customer class

NonBlank class

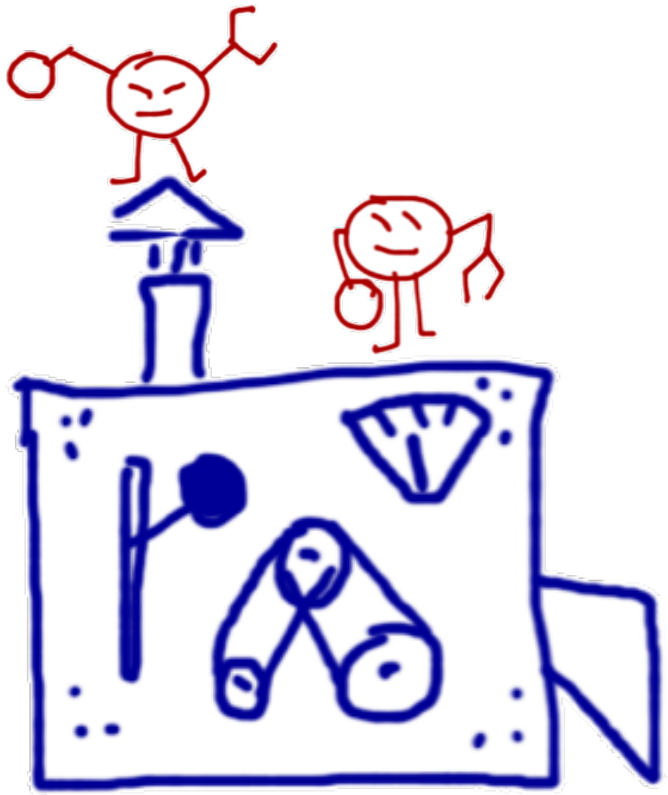
Step 3: cast of characters



Customer *instances*

NonBlank *instances*

Step 3: cast of characters



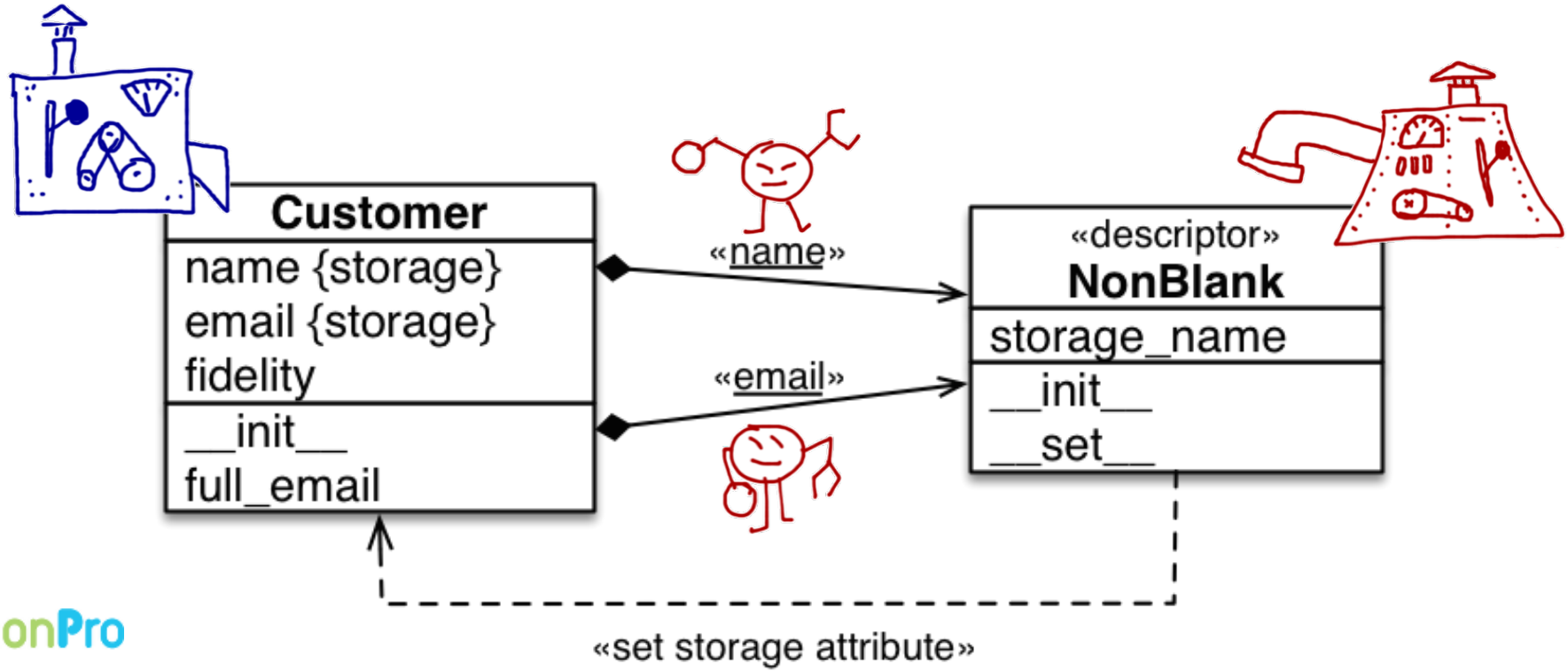
*The **Customer** class has 2 instances of **NonBlank** attached to it!*

PythonPro

Step 3: how the descriptor works

3

- Each instance of **NonBlank** manages one attribute



/home/luciano/prj/oscon2014/descriptor/customer2.py

```
    ...
    ValueError: 'email' must not be empty

`NonBlank` fields must also be strings:

    >>> dummy = Customer('Buster', 99)
    Traceback (most recent call last):
      ...
    TypeError: 'email' must be of type 'str'

"""


class Customer:

    def __init__(self, name, email, fidelity=0):
        self.name = name
        self.email = email
        self.fidelity = fidelity

    @property
    def email(self):
        return self.__email

    @email.setter
    def email(self, value):
        if not isinstance(value, str):
            raise TypeError("'email' must be of type 'str'")
        elif len(value) == 0:
            raise ValueError("'email' must not be empty")
        self.__email = value

    def full_email(self):
        return '{0} <{1}>'.format(self.name, self.email)
```

/home/luciano/prj/oscon2014/descriptor/customer3.py

```
    >>> dummy = Customer('Buster', 99)
    Traceback (most recent call last):
      ...
    TypeError: 'email' must be of type 'str'

"""


class NonBlank:

    def __init__(self, storage_name):
        self.storage_name = storage_name

    def __set__(self, instance, value):
        if not isinstance(value, str):
            raise TypeError("%r must be of type 'str'" % se
        elif len(value) == 0:
            raise ValueError('%r must not be empty' % self.
        instance.__dict__[self.storage_name] = value


class Customer:

    name = NonBlank('name')
    email = NonBlank('email')

    def __init__(self, name, email, fidelity=0):
        self.name = name
        self.email = email
        self.fidelity = fidelity

    def full_email(self):
        return '{0} <{1}>'.format(self.name, self.email)
```

Step 3: an issue

- Each **NonBlank** instance must be created with an explicit **storage_name**
- If the name given does not match the actual attribute, reading and writing will access different data!

Error prone

/home/luciano/prj/oscon2014/descriptor/customer3.py

```
    >>> dummy = Customer('Buster', 99)
    Traceback (most recent call last):
      ...
    TypeError: 'email' must be of type 'str'

"""


class NonBlank:

    def __init__(self, storage_name):
        self.storage_name = storage_name

    def __set__(self, instance, value):
        if not isinstance(value, str):
            raise TypeError("%r must be of type 'str'" % se
        elif len(value) == 0:
            raise ValueError('%r must not be empty' % self.
        instance.__dict__[self.storage_name] = value


class Customer:

    name = NonBlank('name')
    email = NonBlank('email')

    def __init__(self, name, email, fidelity=0):
        self.name = name
        self.email = email
        self.fidelity = fidelity

    def full_email(self):
        return '{0} <{1}>'.format(self.name, self.email)
```

PythonPro

Step 3: the challenge

- The right side of an assignment runs before the variable is touched
- There is no way a **NonBlank** instance can know to which variable it will be assigned when the instance is created!

*The **name** class attribute does not exist when **NonBlank()** is instantiated!*

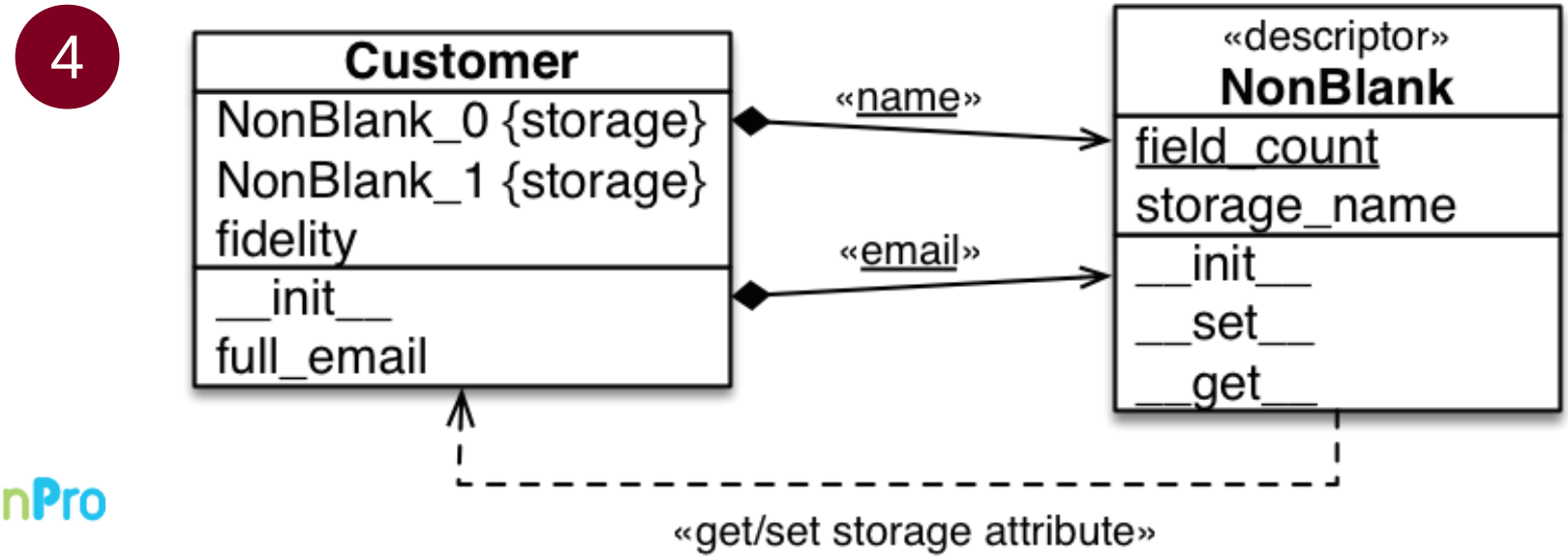
/home/luciano/prj/oscon2014/descriptor/customer3.py

3

```
    >>> dummy = Customer('Buster', 99)
    Traceback (most recent call last):
      ...
    TypeError: 'email' must be of type 'str'

"""


class NonBlank:

    def __init__(self, storage_name):
        self.storage_name = storage_name

    def __set__(self, instance, value):
        if not isinstance(value, str):
            raise TypeError("%r must be of type 'str'" % se
        elif len(value) == 0:
            raise ValueError('%r must not be empty' % self.
        instance.__dict__[self.storage_name] = value


class Customer:

    name = NonBlank('name')
    email = NonBlank('email')

    def __init__(self, name, email, fidelity=0):
        self.name = name
        self.email = email
        self.fidelity = fidelity

    def full_email(self):
        return '{0} <{1}>'.format(self.name, self.email)
```

Step 4: automate storage_name generation

- **NonBlank** has a **field_count** class attribute to count its instances
- **field_count** is used to generate a unique **storage_name** for each **NonBlank** instance (i.e. NonBlank_0, NonBlank_1)



PythonPro

/home/luciano/prj/oscon2014/descriptor/customer3.py

```
    >>> dummy = Customer('Buster', 99)
    Traceback (most recent call last):
      ...
    TypeError: 'email' must be of type 'str'

"""


class NonBlank:

    def __init__(self, storage_name):
        self.storage_name = storage_name

    def __set__(self, instance, value):
        if not isinstance(value, str):
            raise TypeError("%r must be of type 'str'" % se
        elif len(value) == 0:
            raise ValueError('%r must not be empty' % self.
        instance.__dict__[self.storage_name] = value


class Customer:

    name = NonBlank('name')
    email = NonBlank('email')

    def __init__(self, name, email, fidelity=0):
        self.name = name
        self.email = email
        self.fidelity = fidelity

    def full_email(self):
        return '{0} <{1}>'.format(self.name, self.email)
```

3

/home/luciano/prj/oscon2014/descriptor/customer4.py

```
class NonBlank:
    field_count = 0

    def __init__(self):
        cls = self.__class__
        self.storage_name = '%s_%s' % (cls.__name__, cls.fi
        cls.field_count += 1

    def __set__(self, instance, value):
        if not isinstance(value, str):
            raise TypeError("'NonBlank' must be of type 'st
        elif len(value) == 0:
            raise ValueError("'NonBlank' must not be empty"
        instance.__dict__[self.storage_name] = value

    def __get__(self, instance, owner):
        if instance is None:
            return self
        else:
            return getattr(instance, self.storage_name)


class Customer:

    name = NonBlank()
    email = NonBlank()

    def __init__(self, name, email, fidelity=0):
        self.name = name
        self.email = email
        self.fidelity = fidelity

    def full_email(self):
        return '{0} <{1}>'.format(self.name, self.email)
```

4

No duplication

/home/luciano/prj/oscon2014/descriptor/customer3.py

3

```
A client cannot be created with a blank e-mail:

    >>> joe = Customer('Joseph Blow', '')
    Traceback (most recent call last):
      ...
    ValueError: 'email' must not be empty
    >>> joe = Customer('Joseph Blow', 'joe@blow.up')
    >>> joe.full_email()
    'Joseph Blow <joe@blow.up>'

Assigning a blank e-mail later is not allowed either:

    >>> joe.email = ''
    Traceback (most recent call last):
      ...
    ValueError: 'email' must not be empty

`NonBlank` fields must also be strings:

    >>> dummy = Customer('Buster', 99)
    Traceback (most recent call last):
      ...
    TypeError: 'email' must be of type 'str'

"""


class NonBlank:

    def __init__(self, storage_name):
        self.storage_name = storage_name

    def __set__(self, instance, value):
        if not isinstance(value, str):
```

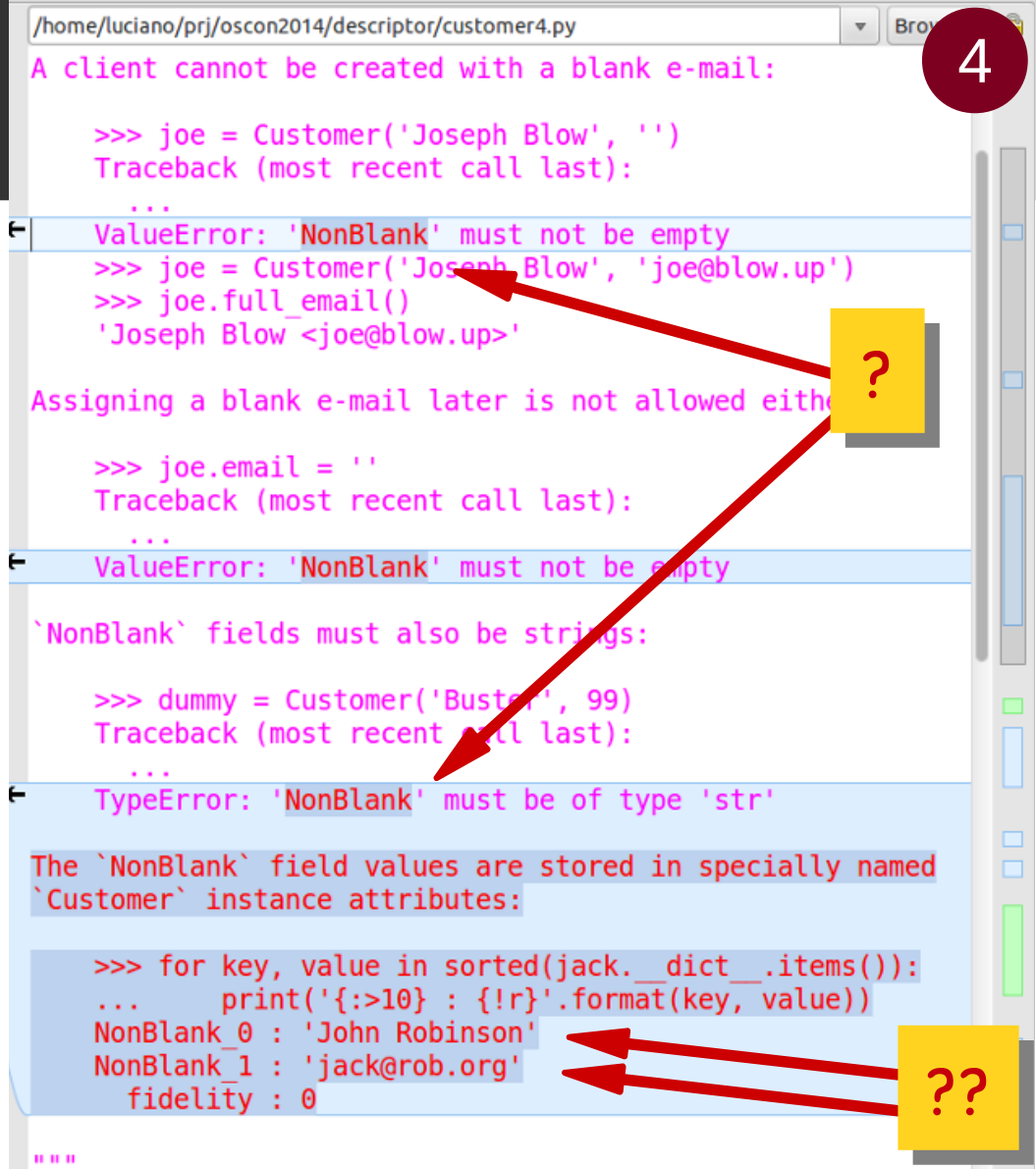
/home/luciano/prj/oscon2014/descriptor/customer4.py

4

```
A client cannot be created with a blank e-mail:

    >>> joe = Customer('Joseph Blow', '')
    Traceback (most recent call last):
      ...
    ValueError: 'NonBlank' must not be empty
    >>> joe = Customer('Joseph Blow', 'joe@blow.up')
    >>> joe.full_email()
    'Joseph Blow <joe@blow.up>'

Assigning a blank e-mail later is not allowed either:

    >>> joe.email = ''
    Traceback (most recent call last):
      ...
    ValueError: 'NonBlank' must not be empty

`NonBlank` fields must also be strings:

    >>> dummy = Customer('Buster', 99)
    Traceback (most recent call last):
      ...
    TypeError: 'NonBlank' must be of type 'str'

The `NonBlank` field values are stored in specially named
`Customer` instance attributes:

    >>> for key, value in sorted(jack.__dict__.items()):
    ...     print('{:>10} : {!r}'.format(key, value))
    NonBlank_0 : 'John Robinson'
    NonBlank_1 : 'jack@rob.org'
      fidelity : 0

"""
```

Step 4: issues

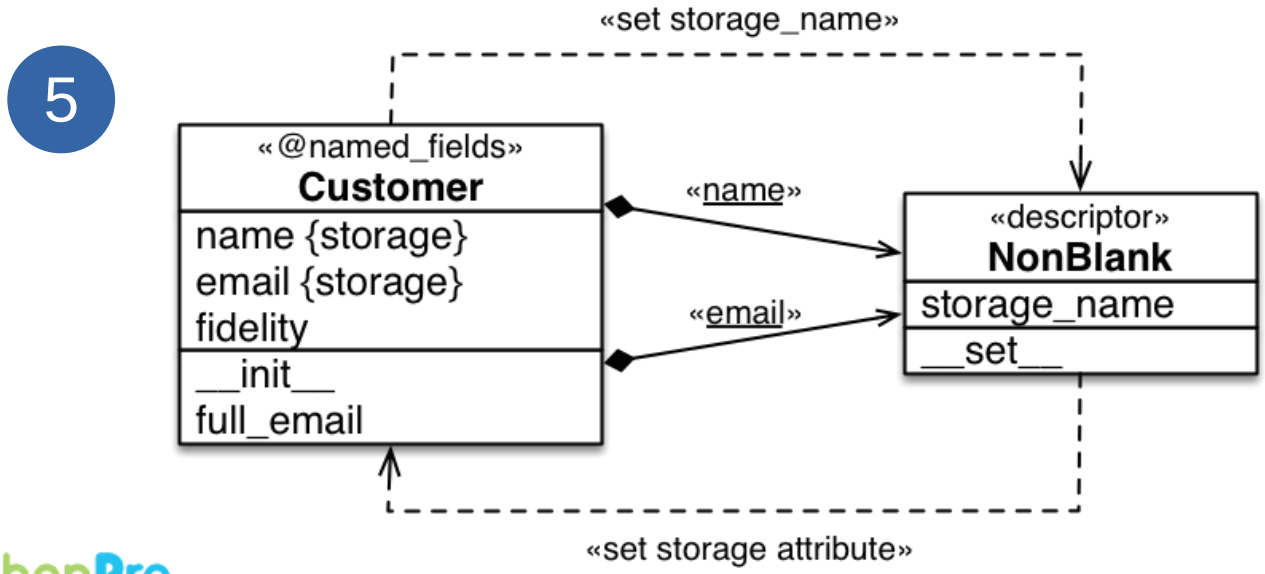
- Error messages no longer refer to the managed field name
- When debugging, we must deal with the fact that **name** is stored in **NonBlank_0** and **email** is in **NonBlank_1**



PythonPro

Step 5: set storage_name with class decorator

- Create a class decorator **named_fields** to set the **storage_name** of each **NonBlank** instance immediately after the **Customer** class is created



Full-on metaprogramming: treating a class like an object!

PythonPro

/home/luciano/prj/oscon2014/descriptor/customer4.py

4

```
"""
A client with name and e-mail:

    >>> jack = Customer('John Robinson', 'jack@rob.org')
    >>> jack.full_email()
    'John Robinson <jack@rob.org>'

A client cannot be created with a blank e-mail:

    >>> joe = Customer('Joseph Blow', '')
    Traceback (most recent call last):
      ...
    ValueError: 'NonBlank' must not be empty
    >>> joe = Customer('Joseph Blow', 'joe@blow.up')
    >>> joe.full_email()
    'Joseph Blow <joe@blow.up>'

Assigning a blank e-mail later is not allowed either:

    >>> joe.email = ''
    Traceback (most recent call last):
      ...
    ValueError: 'NonBlank' must not be empty

`NonBlank` fields must also be strings:

    >>> dummy = Customer('Buster', 99)
    Traceback (most recent call last):
      ...
    TypeError: 'NonBlank' must be of type 'str'

The `NonBlank` field values are stored in specially named
`Customer` instance attributes:
```

/home/luciano/prj/oscon2014/descriptor/customer5.py

5

```
"""
A client with name and e-mail:

    >>> jack = Customer('John Robinson', 'jack@rob.org')
    >>> jack.full_email()
    'John Robinson <jack@rob.org>'

A client cannot be created with a blank e-mail:

    >>> joe = Customer('Joseph Blow', '')
    Traceback (most recent call last):
      ...
    ValueError: 'email' must not be empty
    >>> joe = Customer('Joseph Blow', 'joe@blow.up')
    >>> joe.full_email()
    'Joseph Blow <joe@blow.up>'

Assigning a blank e-mail later is not allowed either:

    >>> joe.email = ''
    Traceback (most recent call last):
      ...
    ValueError: 'email' must not be empty

`NonBlank` fields must also be strings:

    >>> dummy = Customer('Buster', 99)
    Traceback (most recent call last):
      ...
    TypeError: 'email' must be of type 'str'

The `NonBlank` field values are stored in properly named
`Customer` instance attributes:
```

/home/luciano/prj/oscon2014/descriptor/customer4.py

```
class NonBlank:
    field_count = 0

    def __init__(self):
        cls = self.__class__
        self.storage_name = '%s_%s' % (cls.__name__, cls.fi
        cls.field_count += 1

    def __set__(self, instance, value):
        if not isinstance(value, str):
            raise TypeError("'NonBlank' must be of type 'st
        elif len(value) == 0:
            raise ValueError("'NonBlank' must not be empty"
        instance.__dict__[self.storage_name] = value

    def __get__(self, instance, owner):
        if instance is None:
            return self
        else:
            return getattr(instance, self.storage_name)


class Customer:

    name = NonBlank()
    email = NonBlank()

    def __init__(self, name, email, fidelity=0):
        self.name = name
        self.email = email
        self.fidelity = fidelity

    def full_email(self):
        return '{0} <{1}>'.format(self.name, self.email)
```

4

/home/luciano/prj/oscon2014/descriptor/customer5.py

```
"""


class NonBlank:

    def __set__(self, instance, value):
        if not isinstance(value, str):
            raise TypeError("%r must be of type 'str'" % se
        elif len(value) == 0:
            raise ValueError("%r must not be empty" % self.
        instance.__dict__[self.storage_name] = value


def named_fields(cls):
    for name, attr in cls.__dict__.items():
        if isinstance(attr, NonBlank):
            attr.storage_name = name
    return cls


@named_fields
class Customer:

    name = NonBlank()
    email = NonBlank()

    def __init__(self, name, email, fidelity=0):
        self.name = name
        self.email = email
        self.fidelity = fidelity

    def full_email(self):
        return '{0} <{1}>'.format(self.name, self.email)
```

5

Step 5b: create a reusable module

- Move the decorator **named_fields** and the descriptor **NonBlank** to a separate module so they can be reused
  - that is the whole point: being able to abstract getter/setter logic for reuse
  - in our code: model5b.py is the “reusable” module

/home/luciano/prj/oscon2014/descriptor/customer5.py

5

```
"""

class NonBlank:

    def __set__(self, instance, value):
        if not isinstance(value, str):
            raise TypeError("%r must be of type 'str'" % se
        elif len(value) == 0:
            raise ValueError("%r must not be empty" % self.
        instance.__dict__[self.storage_name] = value


def named_fields(cls):
    for name, attr in cls.__dict__.items():
        if isinstance(attr, NonBlank):
            attr.storage_name = name
    return cls


@named_fields
class Customer:

    name = NonBlank()
    email = NonBlank()

    def __init__(self, name, email, fidelity=0):
        self.name = name
        self.email = email
        self.fidelity = fidelity

    def full_email(self):
        return '{0} <{1}>'.format(self.name, self.email)
```

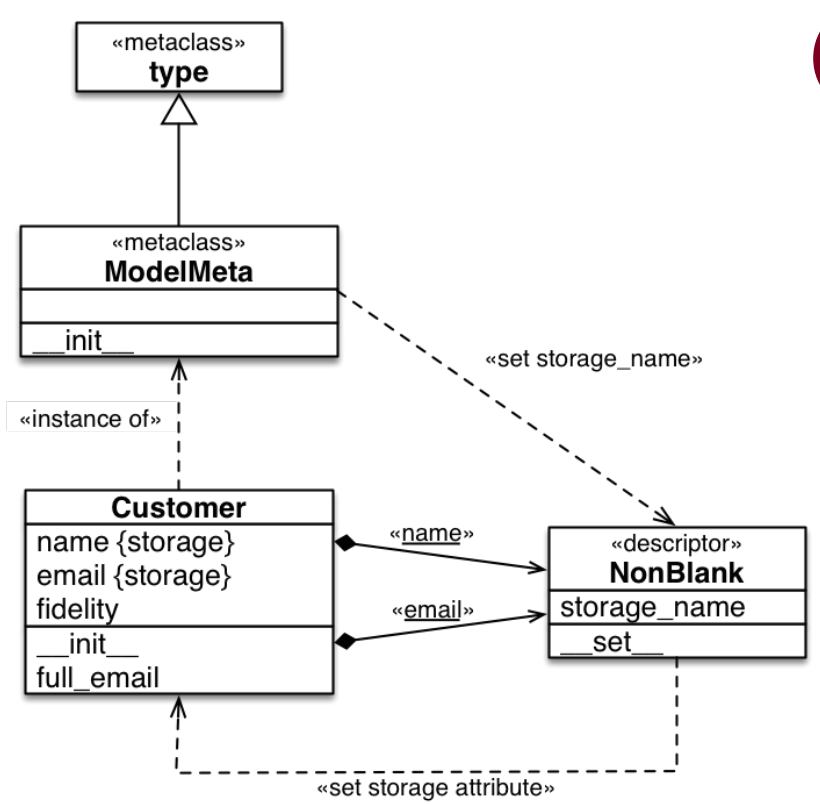
/home/luciano/prj/oscon2014/descriptor/customer5b.py

5b

```
`NonBlank` fields must also be strings:

    >>> dummy = Customer('Buster', 99)
    Traceback (most recent call last):
      ...
    TypeError: 'email' must be of type 'str'

The `NonBlank` field values are stored in properly named
`Customer` instance attributes:

    >>> for key, value in sorted(jack.__dict__.items()):
    ...     print('{:>10} : {!r}'.format(key, value))
         email : 'jack@rob.org'
       fidelity : 0
           name : 'John Robinson'

"""

from model5b import named_fields, NonBlank


@named_fields
class Customer:

    name = NonBlank()
    email = NonBlank()

    def __init__(self, name, email, fidelity=0):
        self.name = name
        self.email = email
        self.fidelity = fidelity

    def full_email(self):
        return '{0} <{1}>'.format(self.name, self.email)
```

Step 6: replace decorator with metaclass

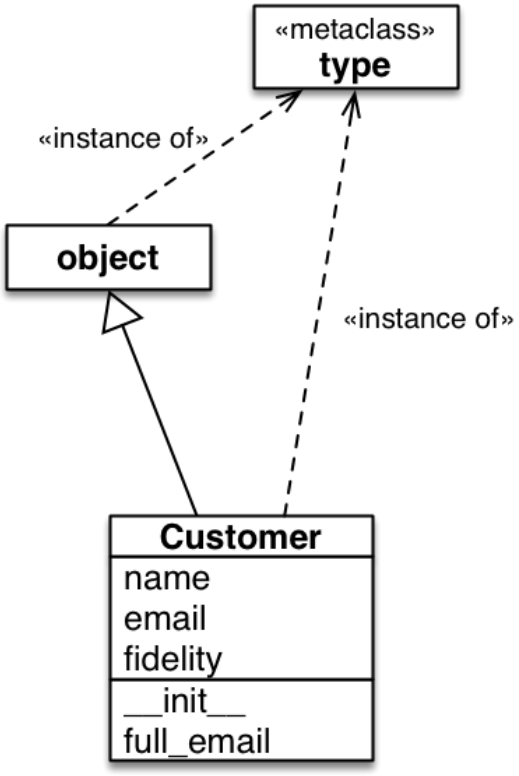
- Not really needed in this example
  - Step 5b is a fine solution
- A metaclass lets you control class construction by implementing:
  - __init__: the class initializer
  - __new__: the class builder



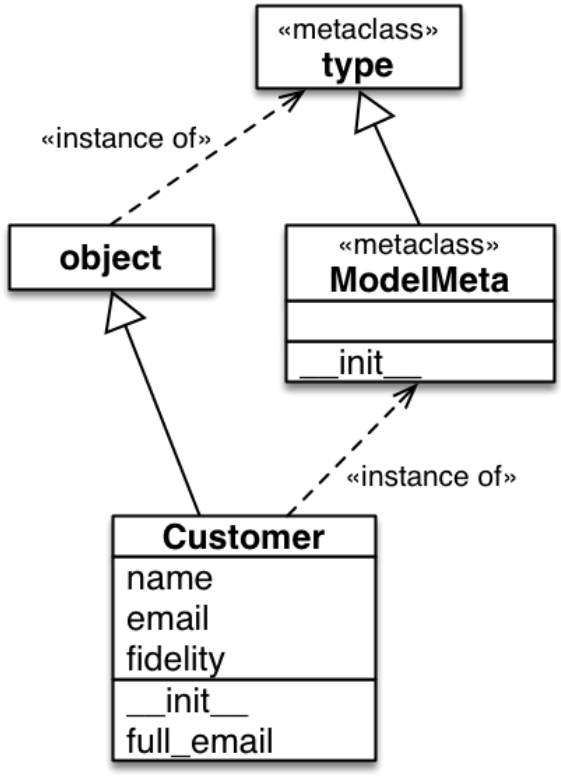
6

PythonPro

Step 6: use a metaclass



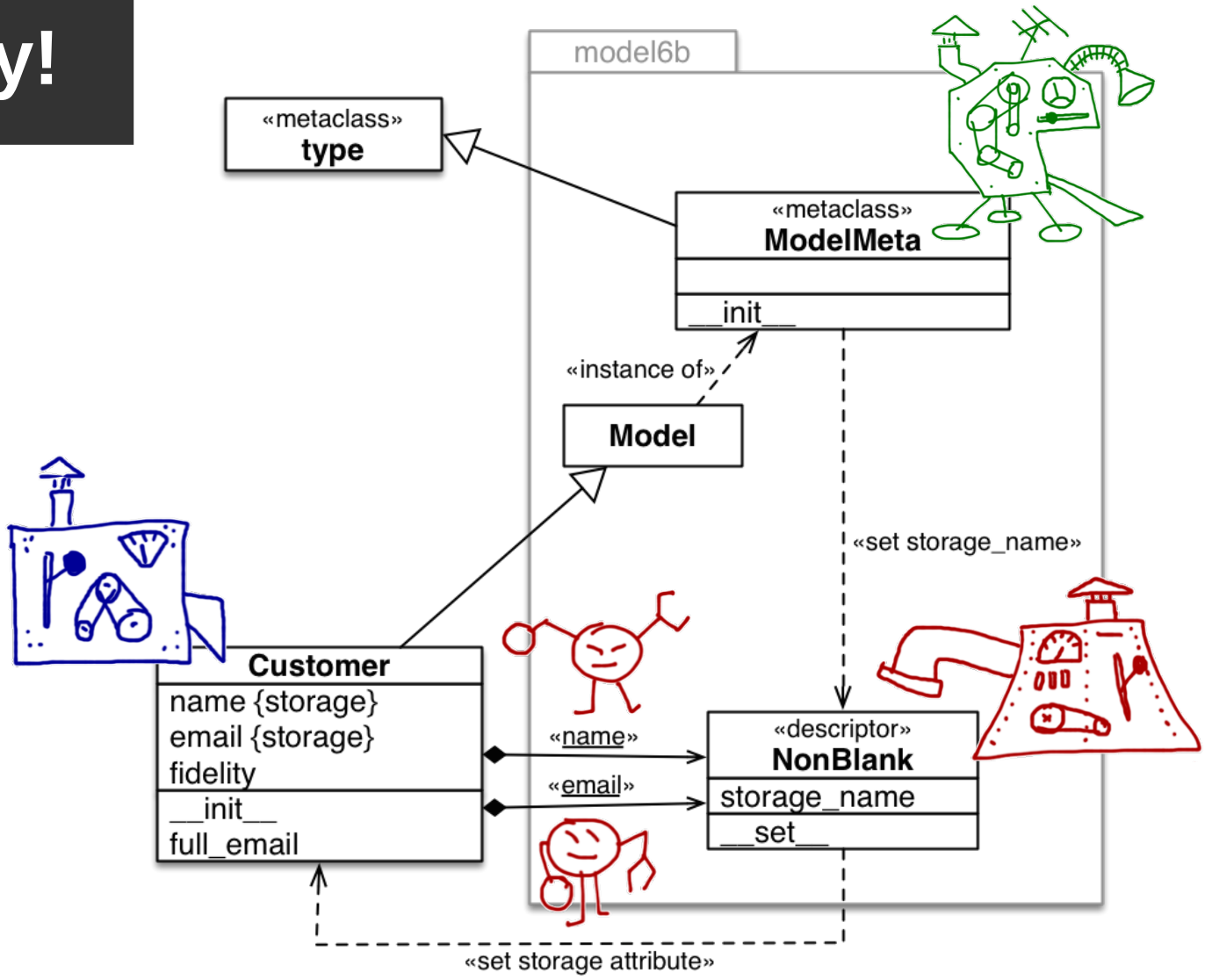
before



after

PythonPro

The full monty!



PythonPro

/home/luciano/prj/oscon2014/descriptor/customer5.py

5

```
"""


class NonBlank:

    def __set__(self, instance, value):
        if not isinstance(value, str):
            raise TypeError("%r must be of type 'str'" % se
        elif len(value) == 0:
            raise ValueError("%r must not be empty" % self.
        instance.__dict__[self.storage_name] = value


def named_fields(cls):
    for name, attr in cls.__dict__.items():
        if isinstance(attr, NonBlank):
            attr.storage_name = name
    return cls


@named_fields
class Customer:

    name = NonBlank()
    email = NonBlank()

    def __init__(self, name, email, fidelity=0):
        self.name = name
        self.email = email
        self.fidelity = fidelity

    def full_email(self):
        return '{0} <{1}>'.format(self.name, self.email)
```

/home/luciano/prj/oscon2014/descriptor/customer6.py

6

```
class NonBlank:

    def __set__(self, instance, value):
        if not isinstance(value, str):
            raise TypeError("%r must be of type 'str'" % se
        elif len(value) == 0:
            raise ValueError("%r must not be empty" % self.
        instance.__dict__[self.storage_name] = value


class ModelMeta(type):

    def __init__(cls, name, bases, dic):
        super().__init__(name, bases, dic)

        for name, attr in dic.items():
            if isinstance(attr, NonBlank):
                attr.storage_name = name


class Customer(metaclass=ModelMeta):

    name = NonBlank()
    email = NonBlank()

    def __init__(self, name, email, fidelity=0):
        self.name = name
        self.email = email
        self.fidelity = fidelity

    def full_email(self):
        return '{0} <{1}>'.format(self.name, self.email)
```

/home/luciano/prj/oscon2014/descriptor/customer6.py

6

```
class NonBlank:

    def __set__(self, instance, value):
        if not isinstance(value, str):
            raise TypeError("%r must be of type 'str'" % se
        elif len(value) == 0:
            raise ValueError("%r must not be empty" % self.
        instance.__dict__[self.storage_name] = value


class ModelMeta(type):

    def __init__(cls, name, bases, dic):
        super().__init__(name, bases, dic)

        for name, attr in dic.items():
            if isinstance(attr, NonBlank):
                attr.storage_name = name


class Customer(metaclass=ModelMeta):

    name = NonBlank()
    email = NonBlank()

    def __init__(self, name, email, fidelity=0):
        self.name = name
        self.email = email
        self.fidelity = fidelity

    def full_email(self):
        return '{0} <{1}>'.format(self.name, self.email)
```

/home/luciano/prj/oscon2014/descriptor/customer6b.py

6b

```
`NonBlank` fields must also be strings:

    >>> dummy = Customer('Buster', 99)
    Traceback (most recent call last):
      ...
    TypeError: 'email' must be of type 'str'

The `NonBlank` field values are stored in properly named
`Customer` instance attributes:

    >>> for key, value in sorted(jack.__dict__.items()):
    ...     print('{:>10} : {!r}'.format(key, value))
         email : 'jack@rob.org'
       fidelity : 0
           name : 'John Robinson'

"""


from model6b import Model, NonBlank

class Customer(Model):

    name = NonBlank()
    email = NonBlank()

    def __init__(self, name, email, fidelity=0):
        self.name = name
        self.email = email
        self.fidelity = fidelity

    def full_email(self):
        return '{0} <{1}>'.format(self.name, self.email)
```

/home/luciano/prj/oscon2014/descriptor/customer5b.py

5b

```
`NonBlank` fields must also be strings:

    >>> dummy = Customer('Buster', 99)
    Traceback (most recent call last):
      ...
    TypeError: 'email' must be of type 'str'

The `NonBlank` field values are stored in properly named
`Customer` instance attributes:

    >>> for key, value in sorted(jack.__dict__.items()):
    ...     print('{:>10} : {!r}'.format(key, value))
         email : 'jack@rob.org'
       fidelity : 0
           name : 'John Robinson'

"""

from model5b import named_fields, NonBlank


@named_fields
class Customer:

    name = NonBlank()
    email = NonBlank()

    def __init__(self, name, email, fidelity=0):
        self.name = name
        self.email = email
        self.fidelity = fidelity

    def full_email(self):
        return '{0} <{1}>'.format(self.name, self.email)
```

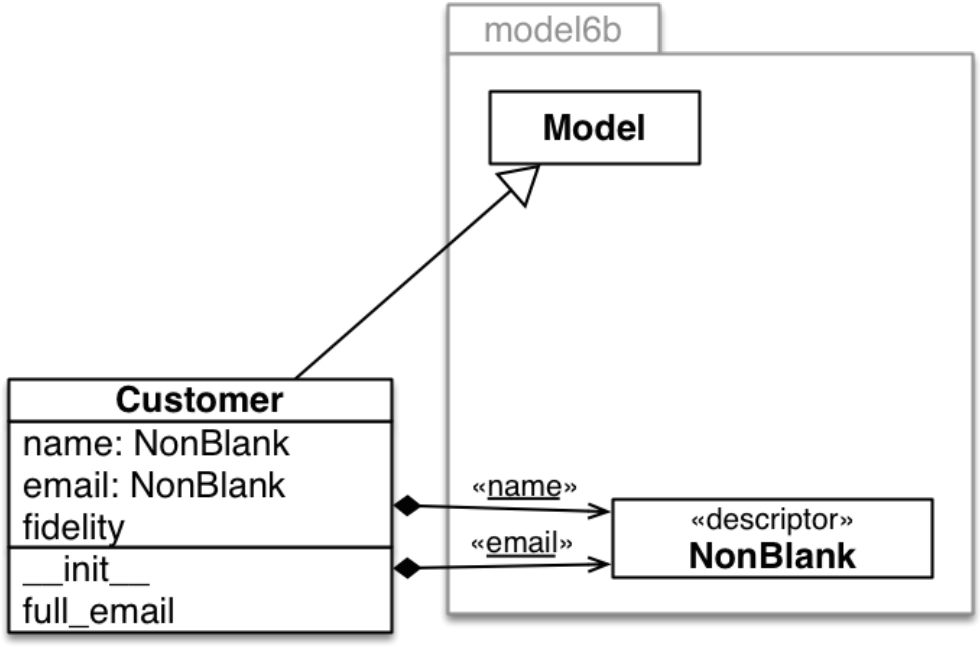
/home/luciano/prj/oscon2014/descriptor/customer6b.py

6b

```
`NonBlank` fields must also be strings:

    >>> dummy = Customer('Buster', 99)
    Traceback (most recent call last):
      ...
    TypeError: 'email' must be of type 'str'

The `NonBlank` field values are stored in properly named
`Customer` instance attributes:

    >>> for key, value in sorted(jack.__dict__.items()):
    ...     print('{:>10} : {!r}'.format(key, value))
         email : 'jack@rob.org'
       fidelity : 0
           name : 'John Robinson'

"""


from model6b import Model, NonBlank


class Customer(Model):

    name = NonBlank()
    email = NonBlank()

    def __init__(self, name, email, fidelity=0):
        self.name = name
        self.email = email
        self.fidelity = fidelity

    def full_email(self):
        return '{0} <{1}>'.format(self.name, self.email)
```

Class decorator x metaclass recap

- Class decorator is easier to create and understand
- Metaclass is more powerful and easier to hide away into a module



PythonPro

For the framework user: the illusion of simplicity

Back to LineItem

- The simplest thing that could possibly work™

```
class LineItem:

    def __init__(self, product, quantity, price):
        self.product = product
        self.quantity = quantity
        self.price = price

    def total(self):
        return self.price * self.quantity
```

Exercise 3: Validade LineItem numeric fields

- Implement a **Quantity** descriptor that only accepts numbers > 0
- Test it
  - -f option makes it easier to test incrementally (e.g. when doing TDD)

```
$ python3 -m doctest lineitem.py -f
```

-f is the same as -o FAIL_FAST;
FAIL_FAST flag is new in Python 3.4

Exercise 3: instructions

- Visit https://github.com/pythonprobr/oscon2014.git
- Clone that repo or click “Download Zip” (right column, bottom)
- Go to descriptor/ directory
- Edit lineitem.py:
  - add doctests to check validation of values < 0
  - implement descriptor make the test pass

Same as Ex. 1

PythonPro

Methods as descriptors

How unbound methods
are bound to instances

Unbound method

- Just a function that happens to be a class attribute
  - Rememeber __setitem__ in the FrenchDeck interactive demo

```
>>> LineItem.total
<function LineItem.total at 0x101812c80>
>>> LineItem.total()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: total() missing 1 required positional argument: 'self'
>>> LineItem.total(bananas)
69.6
```

Bound method

- Partial application of a function to an instance, binding the first argument (self) to the instance.

```
>>> bananas = LineItem('banana', 12, 5.80)
>>> bananas.total()
69.6
>>> bananas.total
<bound method LineItem.total of <LineItem object at 0x...>>
>>> f = bananas.total
>>> f()
69.6
```

Partial application

```
>>> from operator import mul
>>> mul(2, 5)
10
>>> def bind(func, first):
...     def inner(second):
...         return func(first, second)
...     return inner
...
>>> triple = bind(mul, 3)
>>> triple(7)
21
```

- Create a new function that calls an existing function with some arguments fixed
  - The new function takes fewer arguments than the old function
- **functools.partial()**
  - More general than bind()
  - Ready to use

How a method becomes bound

- Every Python function is a descriptor (implements **__get__**)
- When called through an instance, the **__get__** method returns a function object with the first argument bound to the instance

```
>>> dir(LineItem.total)
['__annotations__', '__call__', '__class__', '__closure__',
'__code__', '__defaults__', '__delattr__', '__dict__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__get__'
'__getattribute__', '__globals__', '__gt__', '__hash__',
'__init__', '__kwdefaults__', '__le__', '__lt__', '__module__',
'__name__', '__ne__', '__new__', '__qualname__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__']
```

Anatomy of a bound method

- The bound method has an attribute **__self__** that points to the instance to which it is bound, and **__func__** is a reference to the original, unbound function

```
>>> bananas.total
<bound method LineItem.total of <LineItem object at 0x...>>
>>> dir(bananas.total)
['__call__', '__class__', '__delattr__', '__dir__', '__doc__', '__eq__',
'__format__', '__func__', '__ge__', '__get__', '__getattribute__',
'__gt__', '__hash__', '__init__', '__le__', '__lt__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__self__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__']
>>> bananas.total.__self__ is bananas
True
>>> bananas.total.__func__ is LineItem.total
True
```

Descriptors recap

- Any object implementing **__get__** or **__set__** works as a descriptor when it is an attribute of a class
- A descriptor **y** intercepts access in the form **x.y** when **x** is an instance of the class or the class itself.
- It is not necessary to implement **__get__** if the storage attribute in the instance has the same name as the descriptor in the class
- Every Python function implements **__get__**. The descriptor mechanism turns functions into bound methods when they are acessed via an instance.