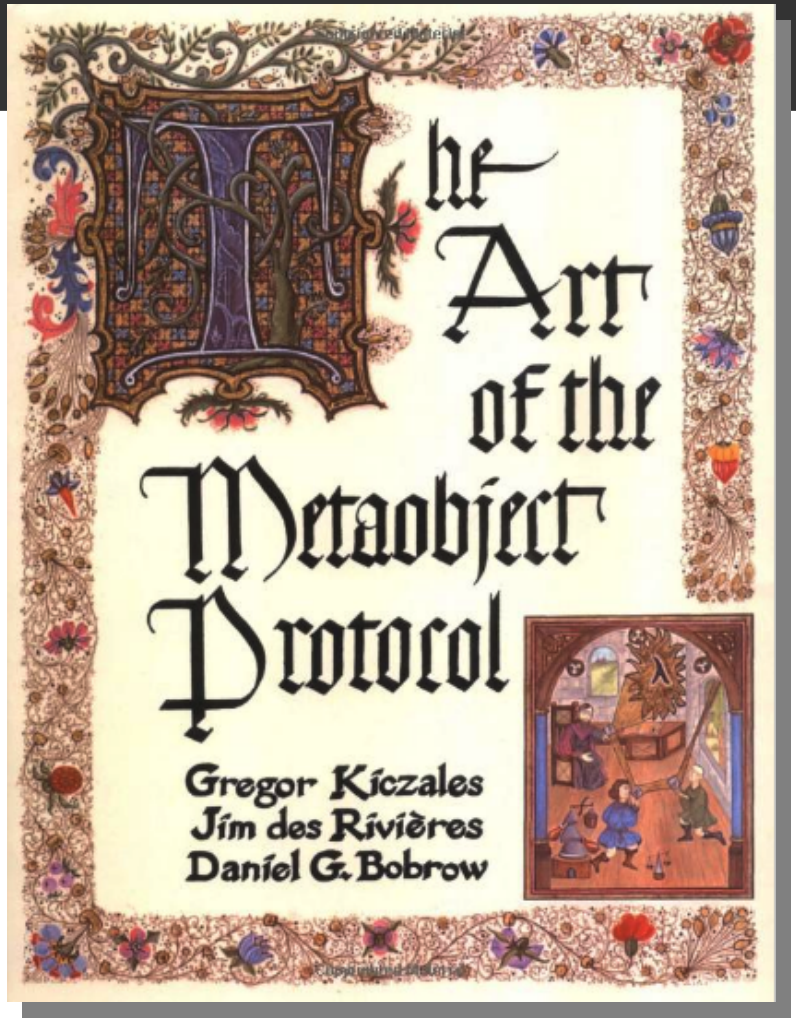
Attribute descriptors

Intercepting attribute
access in a reusable way

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

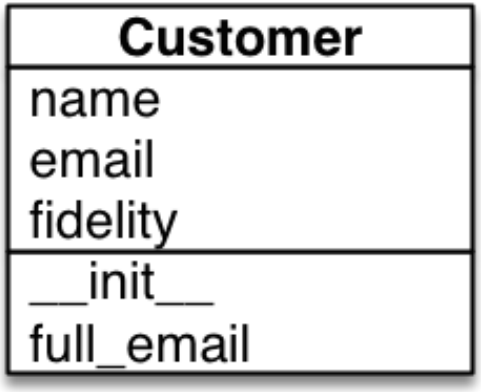
Dynamic attribute access

- 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

PythonPro

Case study: reusing setter logic

1



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

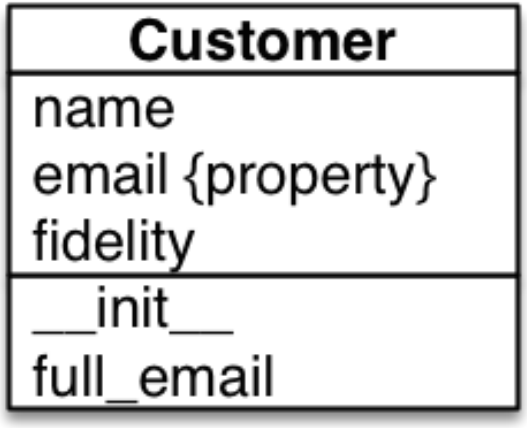
```
>>> joe = Customer('Joseph Blow', '')
>>> joe.full_email()
'Joseph Blow <>'
```

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

```
"""
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

```
"""
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

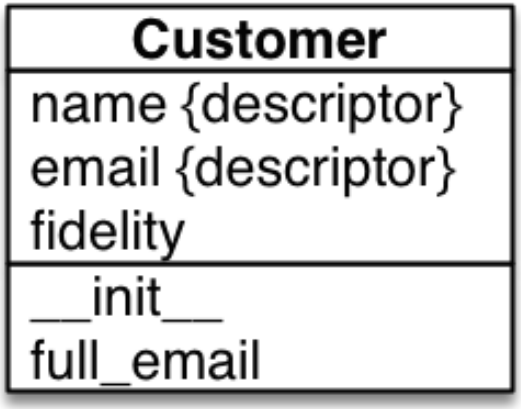
```
      ...
    ValueError: 'email' must not be empty

`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

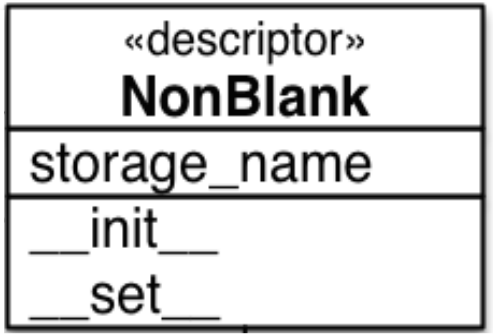
Copy & paste for **name**?!?

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

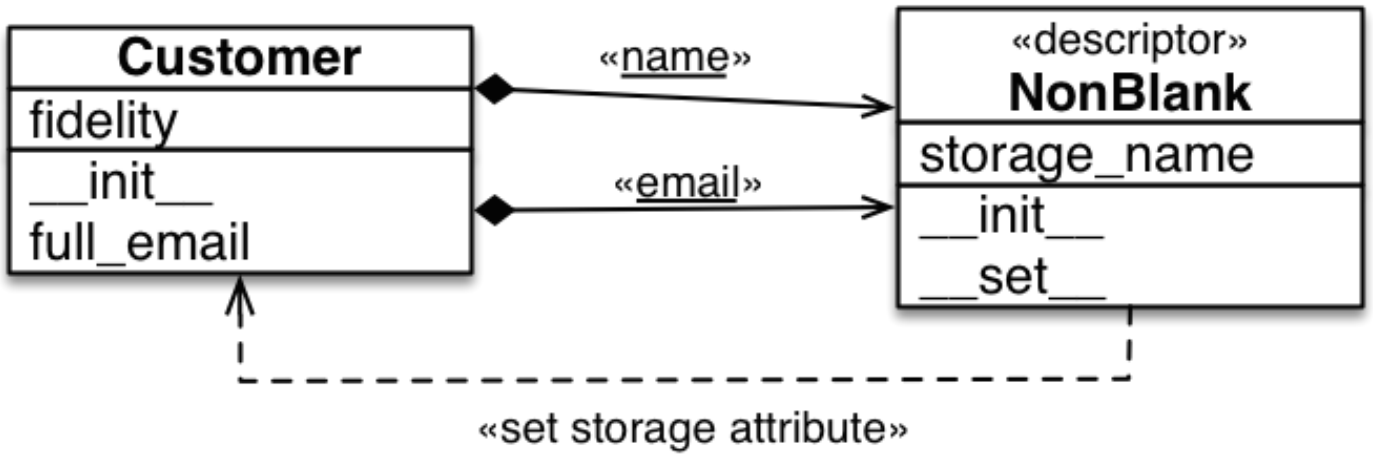


PythonPro

Step 3: create a descriptor

3

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



PythonPro

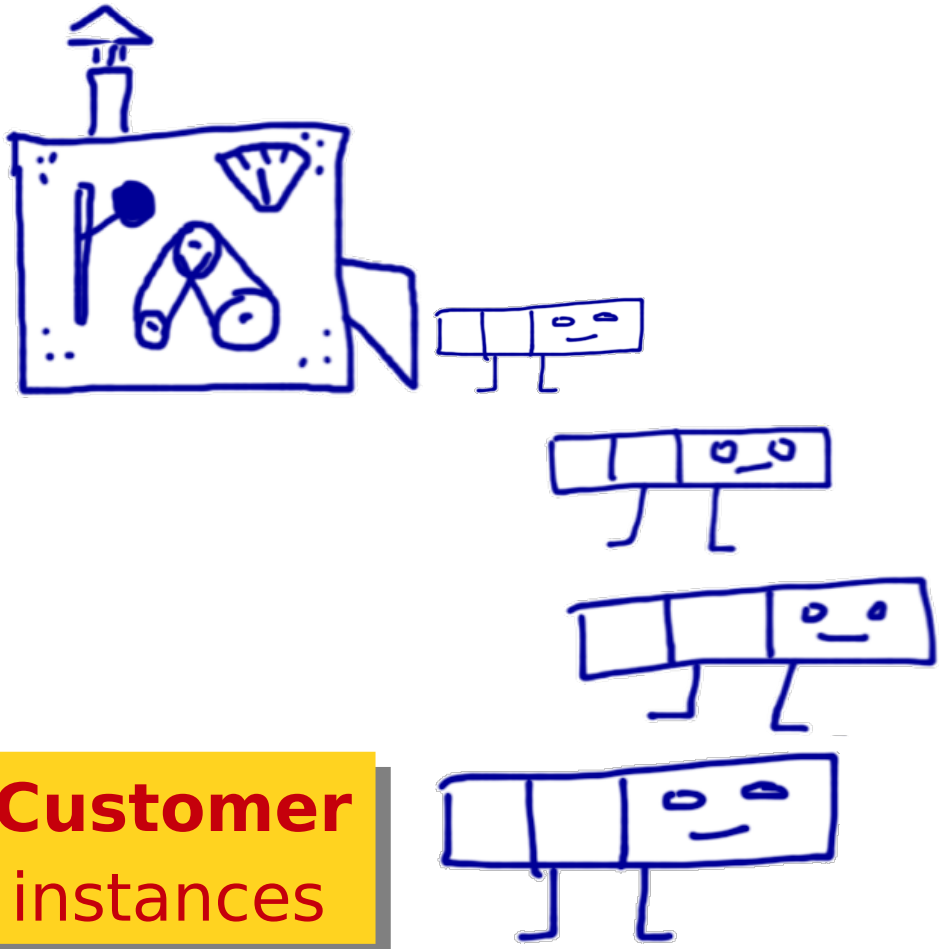
Step 3: cast of characters

Customer class

NonBlank class

PythonPro

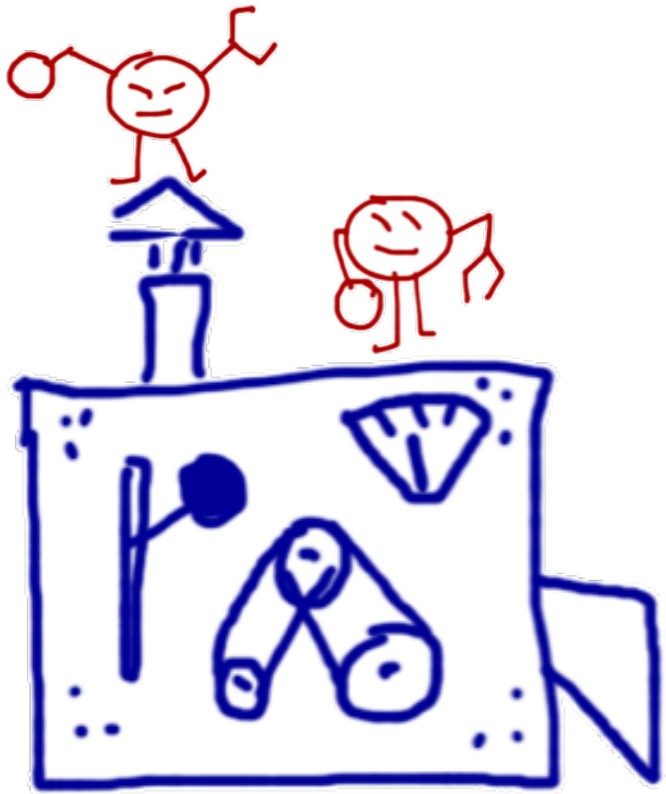
Step 3: cast of characters



Customer instances

NonBlank instances

Step 3: cast of characters



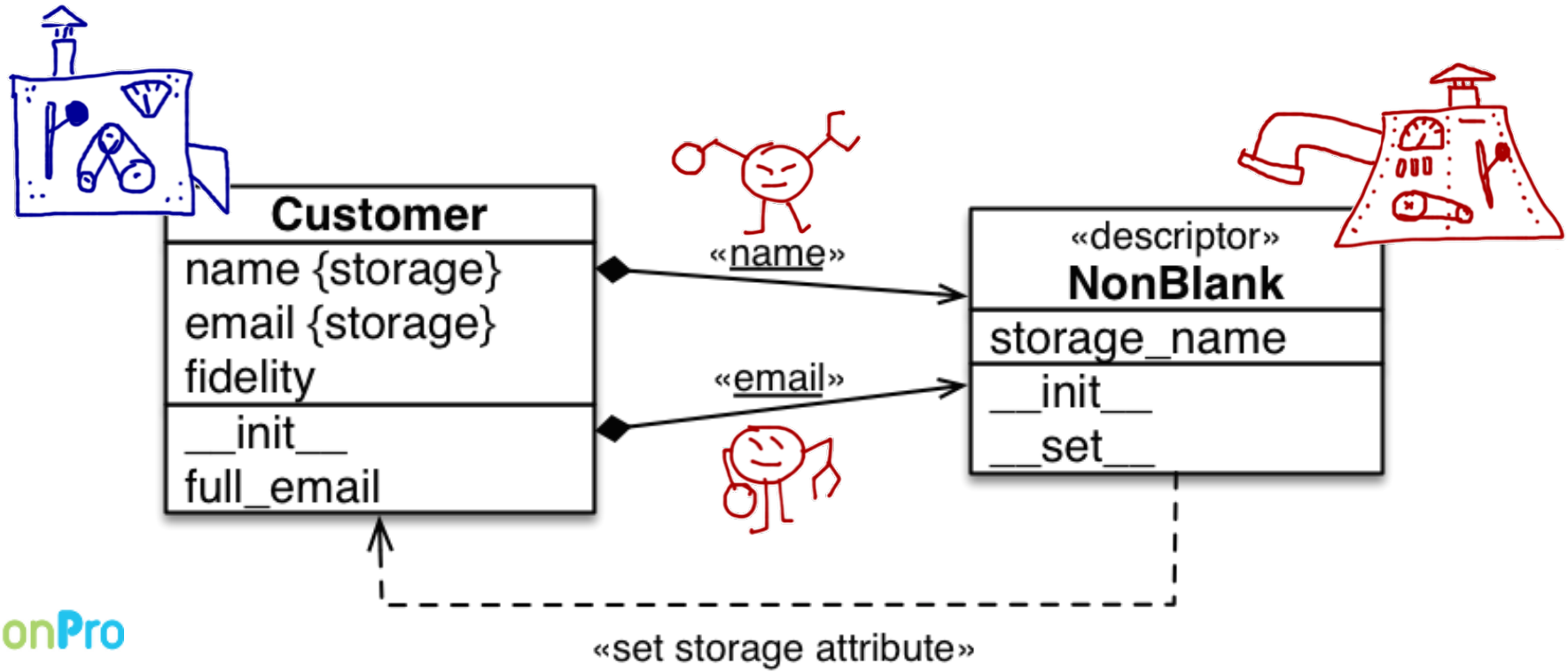
The **Customer** class has 2 instances of **NonBlank** attached to it!

PythonPro

Step 3: how the descriptor works

3

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



PythonPro

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

2

```
    ...
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

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 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)
```

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!

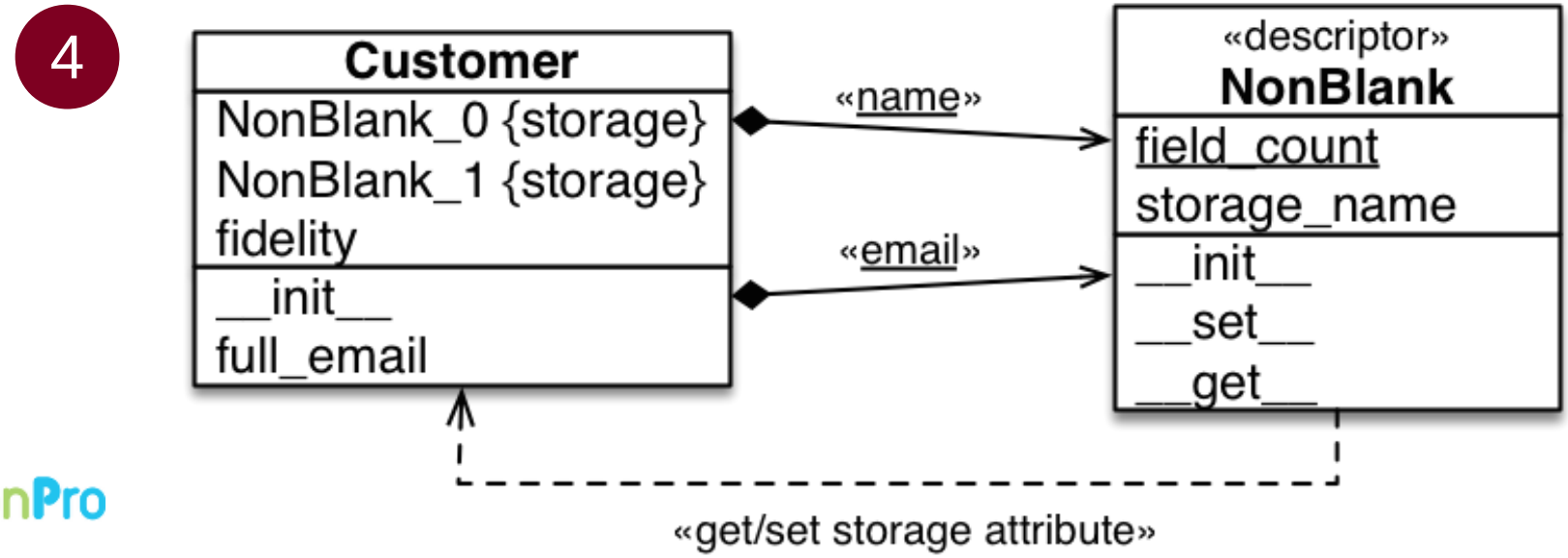
/home/luciano/prj/oscon2014/descriptor/customer3.py

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

"""
```

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

```
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

```
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

"""
```

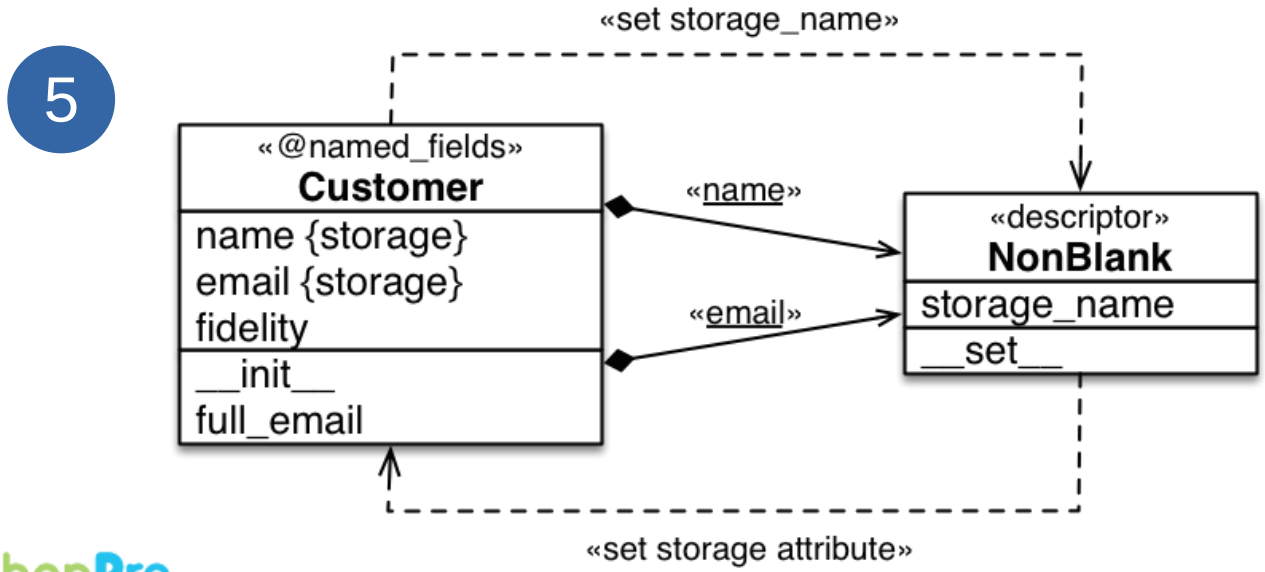
- Error messages no longer refer to the managed field name
- When debugging, we must deal with the fact that **name** is stored in **NonBlank_0** and **email** is in **NonBlank_1**

?

??

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

4

```
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)
```

/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)
```

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)
```

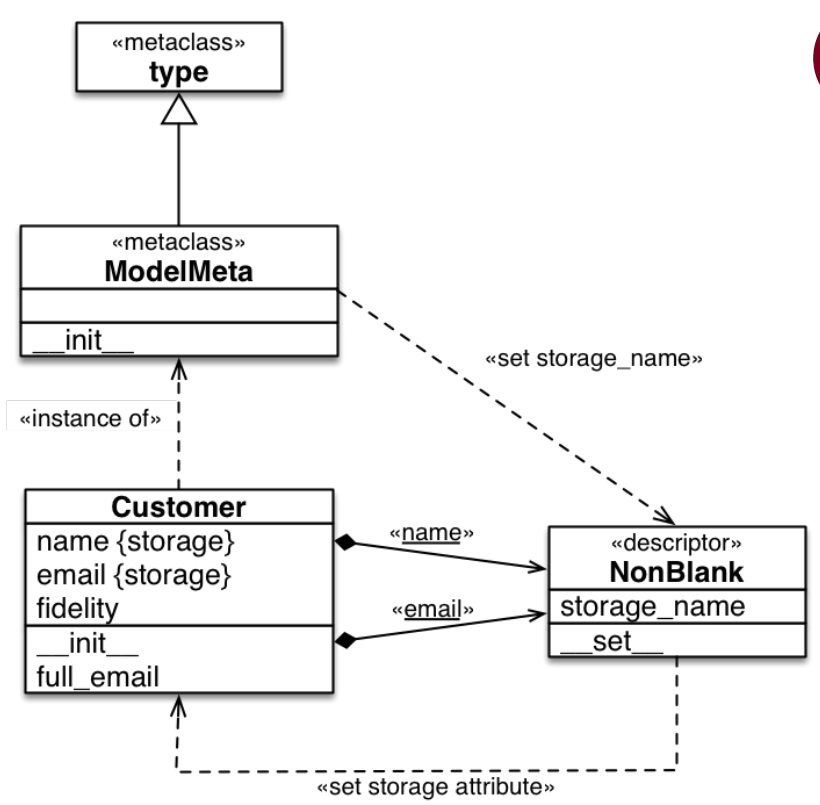
/home/luciano/prj/oscon2014/descriptor/customer5b.py

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

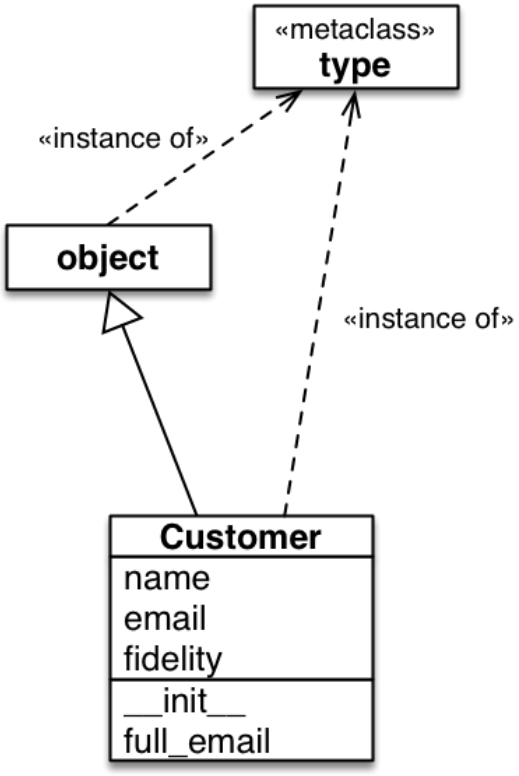
- Not really needed in this example
  - Step 5b is a fine solution
- A metaclass lets you control class construction by implementing:
  - __init__: the class initializer
  - __new__: the class builder



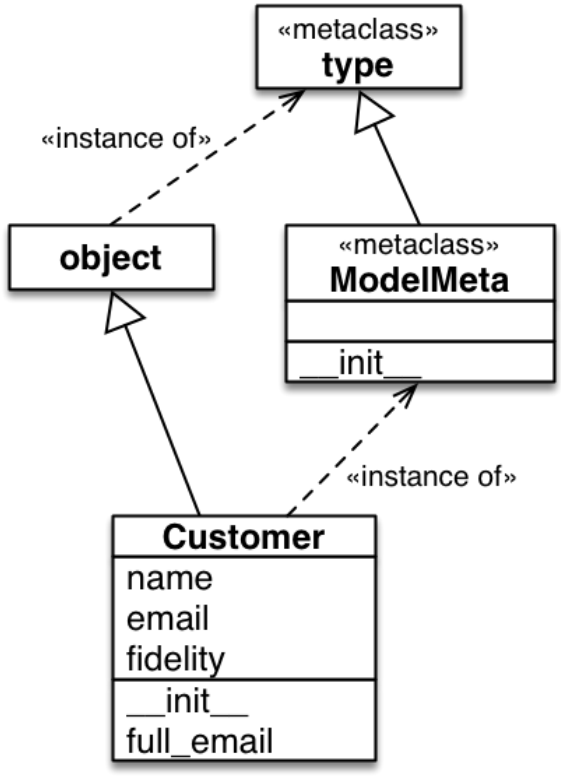
6

PythonPro

Step 6: use a metaclass



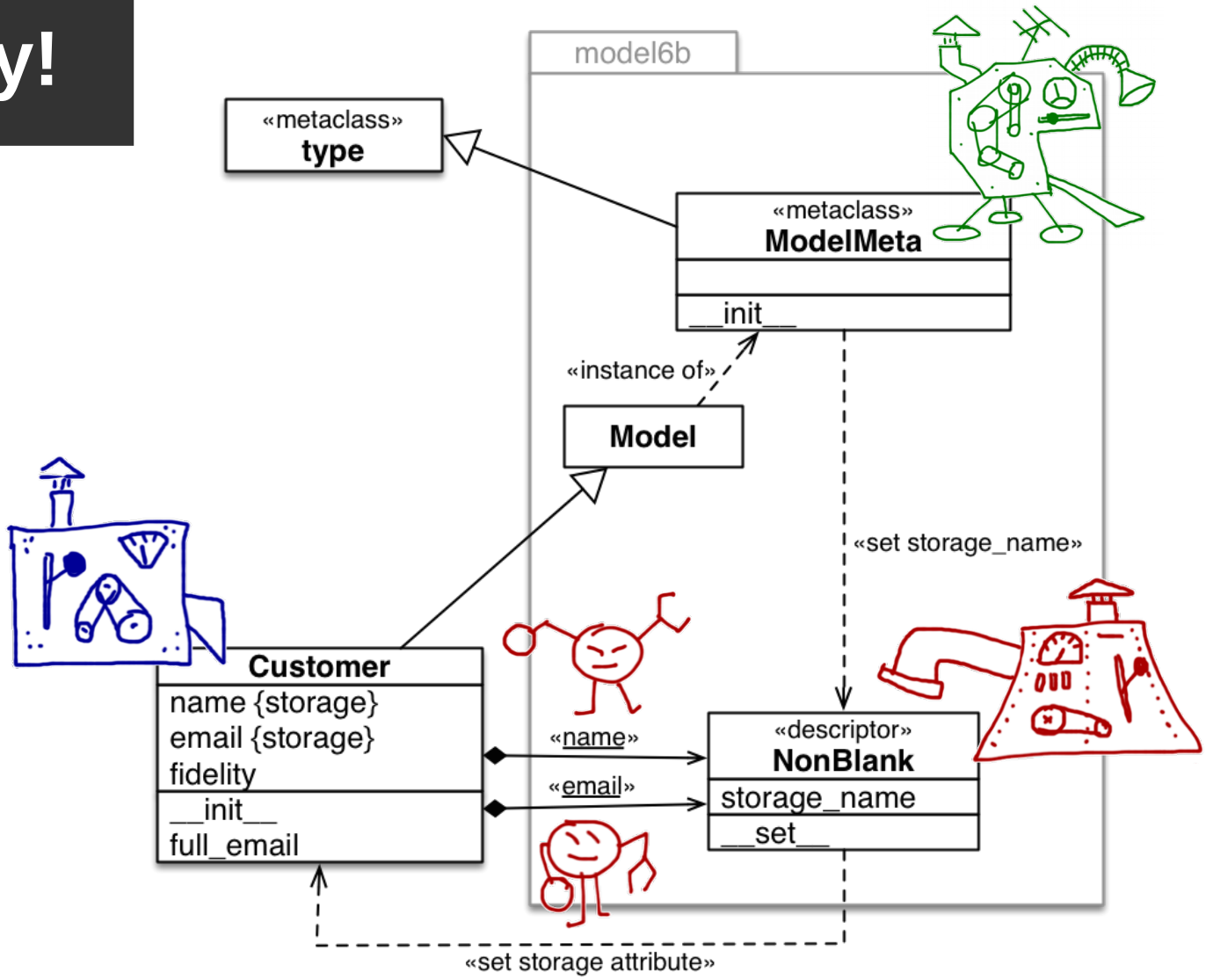
before



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)
```

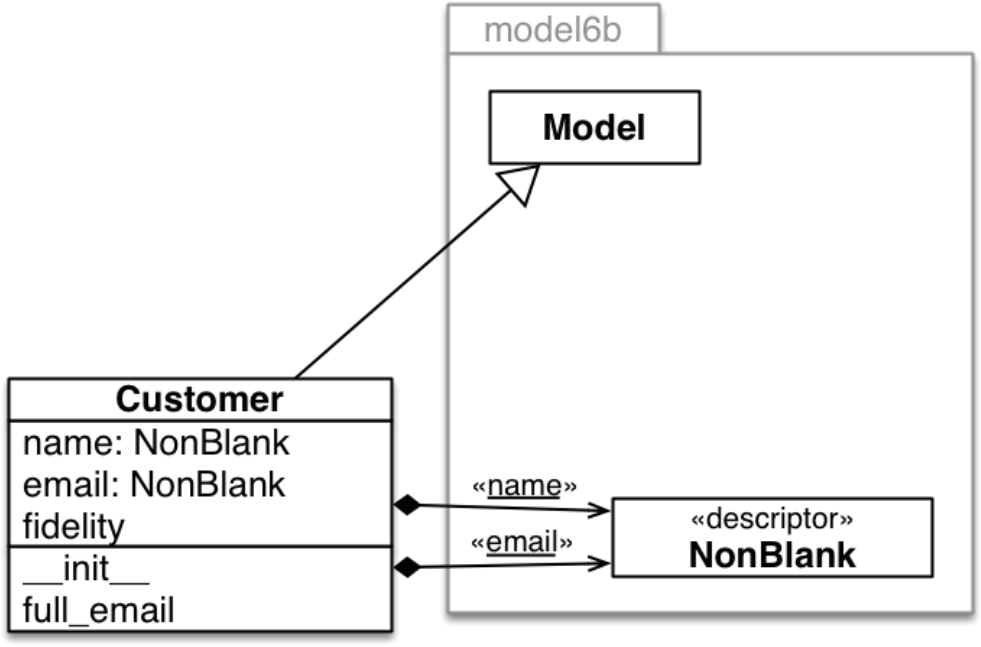
/home/luciano/prj/oscon2014/descriptor/customer6b.py

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/ramalho/full-monty.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

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.