Encapsulation with Descriptors

2012-10-13

Luciano Ramalho
luciano@ramalho.org

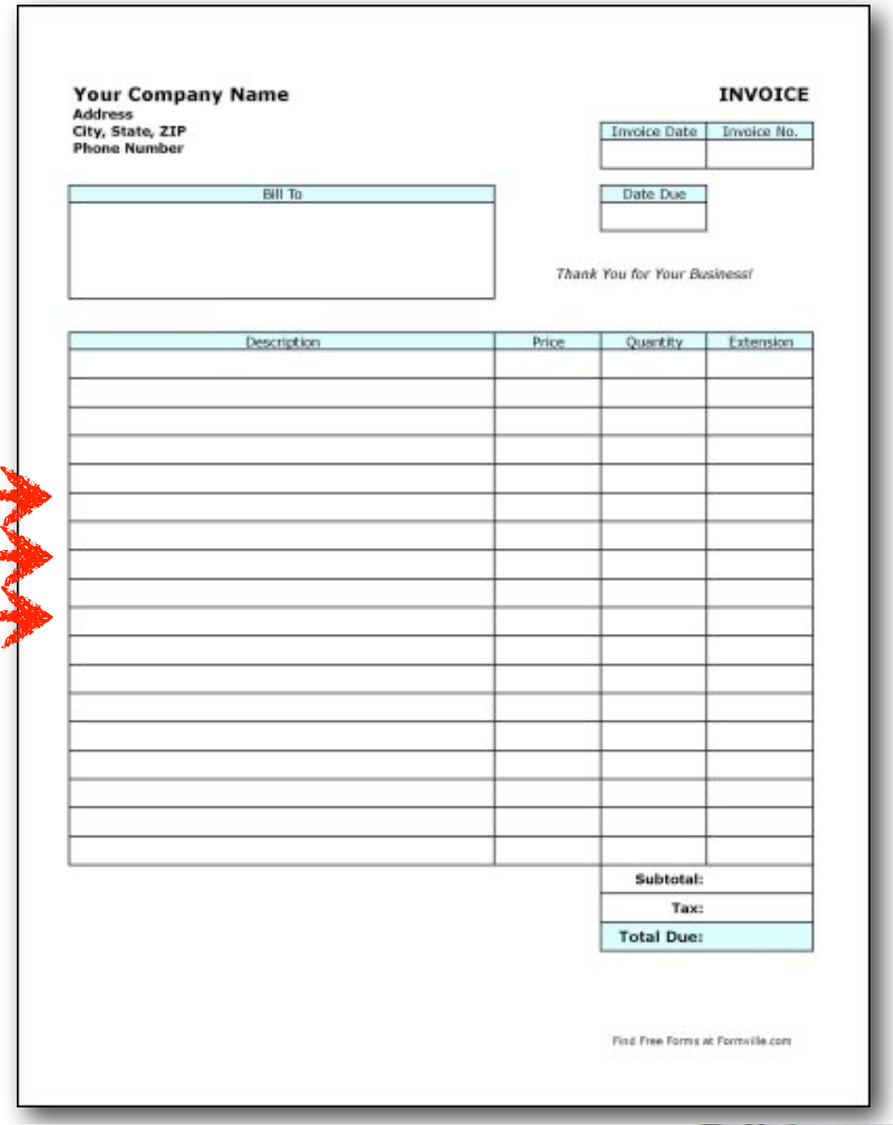
oficinas Turing

Assumptions

- You know the basics of how classes and objects work in Python
  - how to use **super**
  - class versus instance attributes
  - class attribute inheritance (methods and data attributes -- a.k.a. “fields”)
  - attribute protection via name mangling (__x)

The scenario

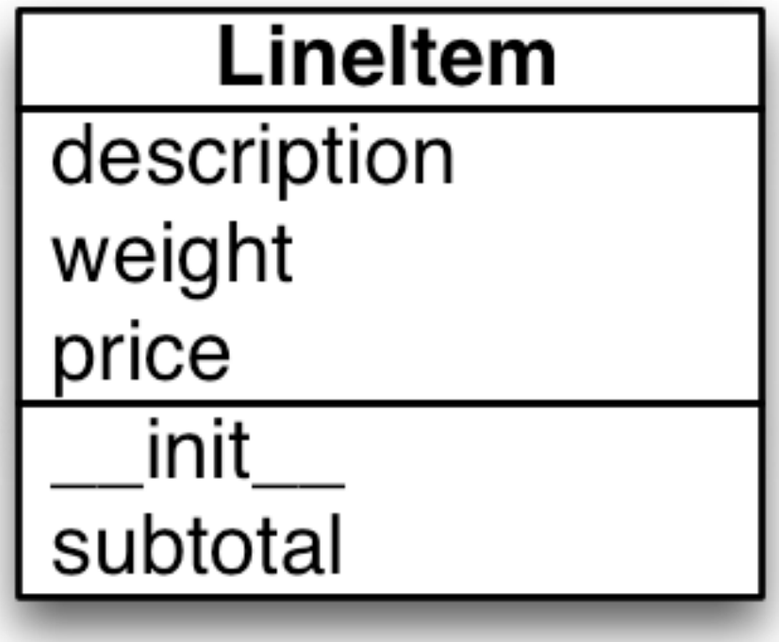
- Selling organic bulk foods
- An order has several **itens**
- Each **item** has a description, weight (kg), unit price (per kg) and a sub-total



❶ the first doctest

```
==============
LineItem tests
==============

A line item for a bulk food order has description, weight and
price fields::

    >>> from bulkfood import LineItem
    >>> raisins = LineItem('Golden raisins', 10, 6.95)
    >>> raisins.weight, raisins.description, raisins.price
    (10, 'Golden raisins', 6.95)

A ``subtotal`` method gives the total price for that line item::

    >>> raisins.subtotal()
    69.5
```

❶ the simplest thing




```
class LineItem(object):

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

❶ perhaps too simple...

```
>>> raisins = LineItem('Golden raisins', 10, 6.95)
>>> raisins.weight = -20
>>> raisins.subtotal()
-139.0
```

a negative amount is due?!

"We found that customers could order a **negative quantity** of books! And we would credit their credit card with the price..." *Jeff Bezos*

amazon.com

Jeff Bezos of Amazon: Birth of a Salesman
WSJ.com - http://j.mp/VZ5not

❶ the classic solution

```
class LineItem(object):

    def __init__(self, description, weight, price):
        self.description = description
        self.set_weight(weight)
        self.price = price

    def subtotal(self):
        return self.get_weight() * self.price

    def get_weight(self):
        return self.__weight

    def set_weight(self, value):
        if value > 0:
            self.__weight = value
        else:
            raise ValueError('value must be > 0')
```

changes in existing code

protected attribute

❶ but it's a breaking change

```
>>> raisins.weight
Traceback (most recent call last):
        ...
AttributeError: 'LineItem' object has no attribute 'weight'
```

- Previously we could get the weight of a line item simply via **`item.weight`**, but not anymore...
- This breaks code
- Python offers another way...

❶ by the way...

- Protected attributes in Python exist only for safety reasons
  - to avoid accidental assignment or overriding
  - not to prevent intentional use (or misuse)

❶ by the way...

- Protected attributes in Python exist only for safety reasons
  - to avoid accidental assignment or overriding
  - not to prevent intentional use (or misuse)

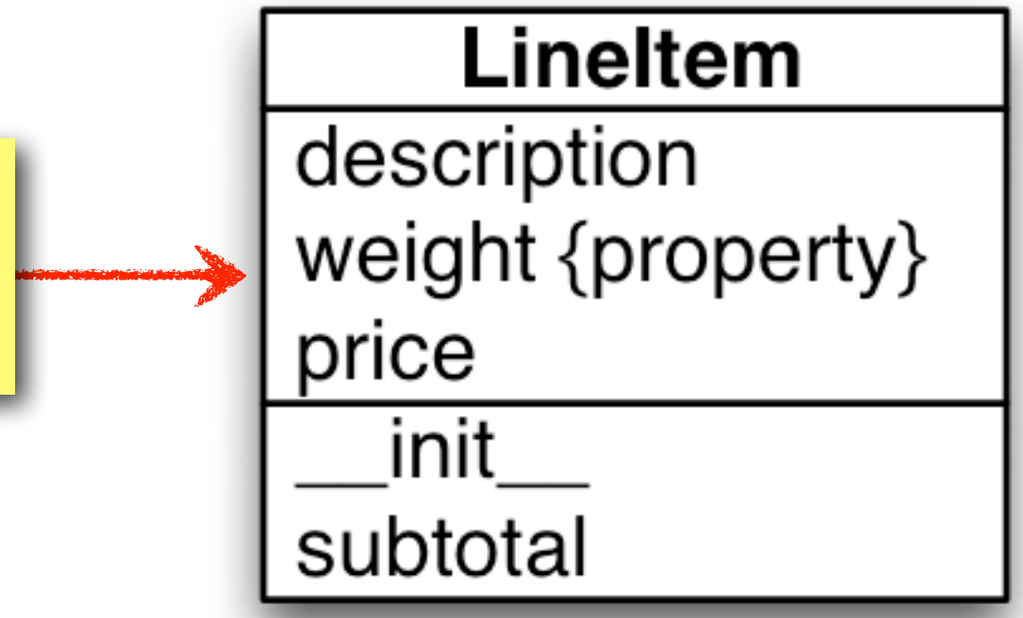
```
>>> raisins._LineItem__weight
10
```

❷ validation with property

```
>>> from bulkfood import LineItem
>>> raisins = LineItem('Golden raisins', 10, 6.95)
>>> raisins.weight = -20
Traceback (most recent call last):
    ...
ValueError: value must be > 0
>>> raisins.weight
10
```

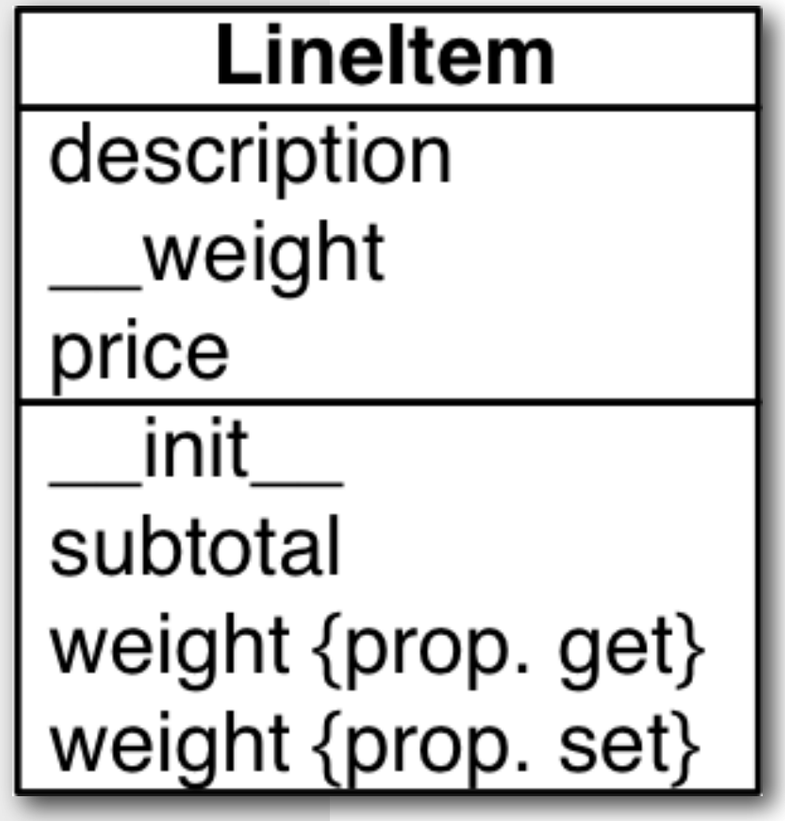
looks like a violation of encapsulation

but business logic is enforced:
weight is now a **property**



❷ property implementation

```
class LineItem(object):

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price

    @property
    def weight(self):
        return self.__weight

    @weight.setter
    def weight(self, value):
        if value > 0:
            self.__weight = value
        else:
            raise ValueError('value must be > 0')
```



❷ property implementation

```
class LineItem(object):

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price

    @property
    def weight(self):
        return self.__weight

    @weight.setter
    def weight(self, value):
        if value > 0:
            self.__weight = value
        else:
            raise ValueError('value must be > 0')
```

within **__init__** the **property** is already in use

❷ property implementation

```
class LineItem(object):

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price

    @property
    def weight(self):
        return self.__weight

    @weight.setter
    def weight(self, value):
        if value > 0:
            self.__weight = value
        else:
            raise ValueError('value must be > 0')
```

the protected **__weight** attribute is touched only within the property methods

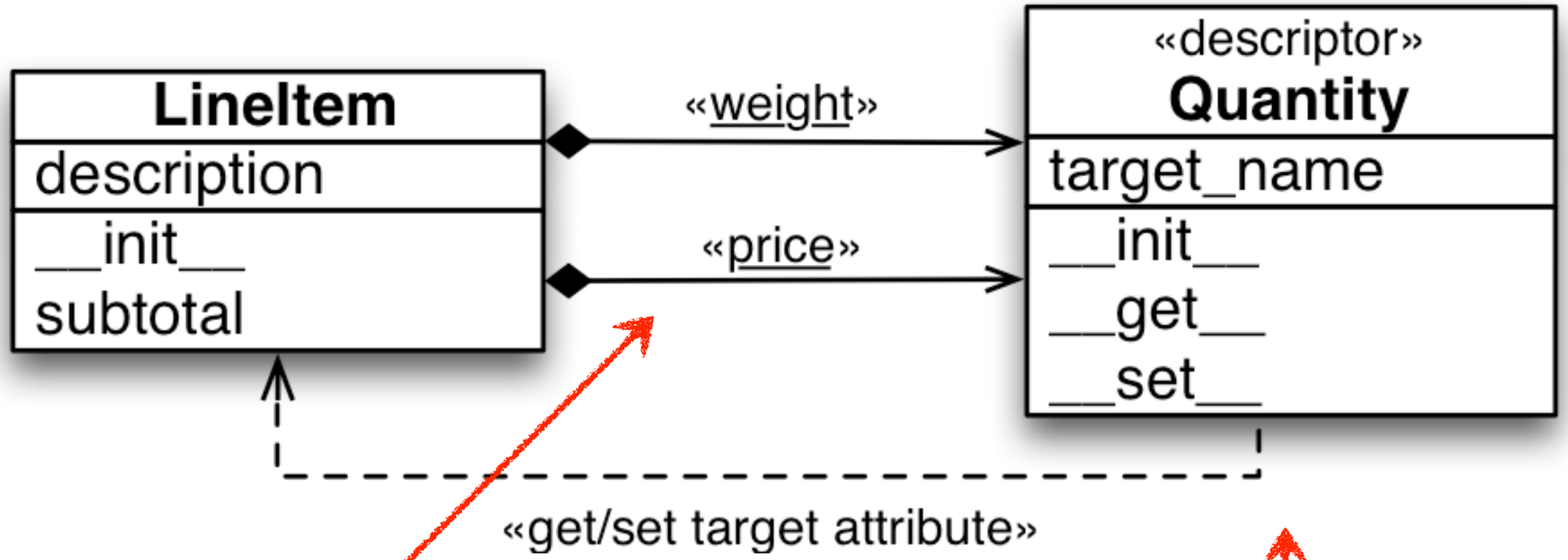
❷ property implementation

```
class LineItem(object):

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price

    @property
    def weight(self):
        return self.__weight

    @weight.setter
    def weight(self, value):
        if value > 0:
            self.__weight = value
        else:
            raise ValueError('value must be > 0')
```

what if we need the same logic applied to the **price** attribute?

duplicate the setter and getter?!?

❸ validation with descriptor



weight and **price** are attributes of the **LineItem** class

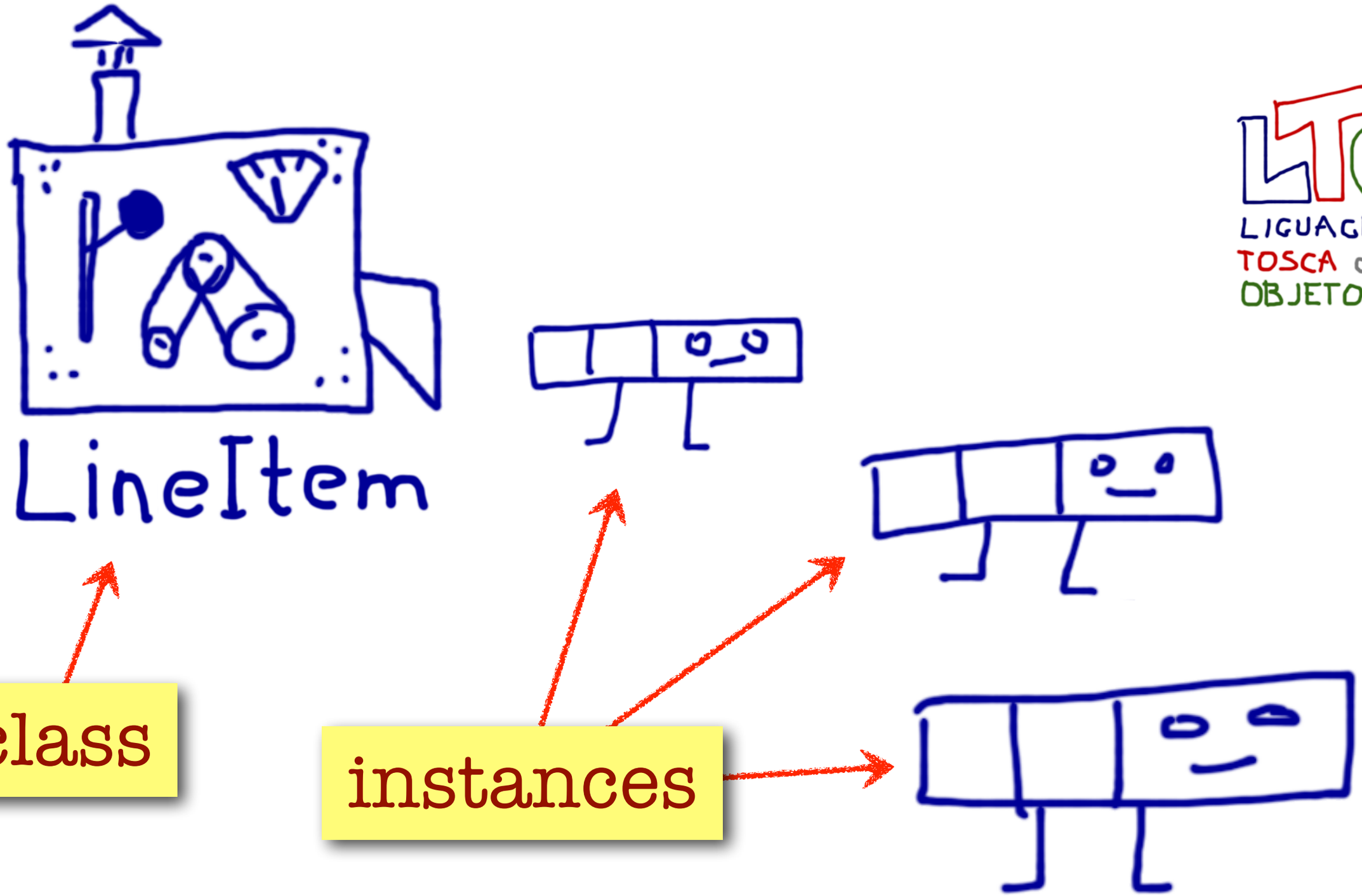
the get/set logic is moved to the **Quantity** descriptor class, where it can be reused via composition

❸

descriptor implementation

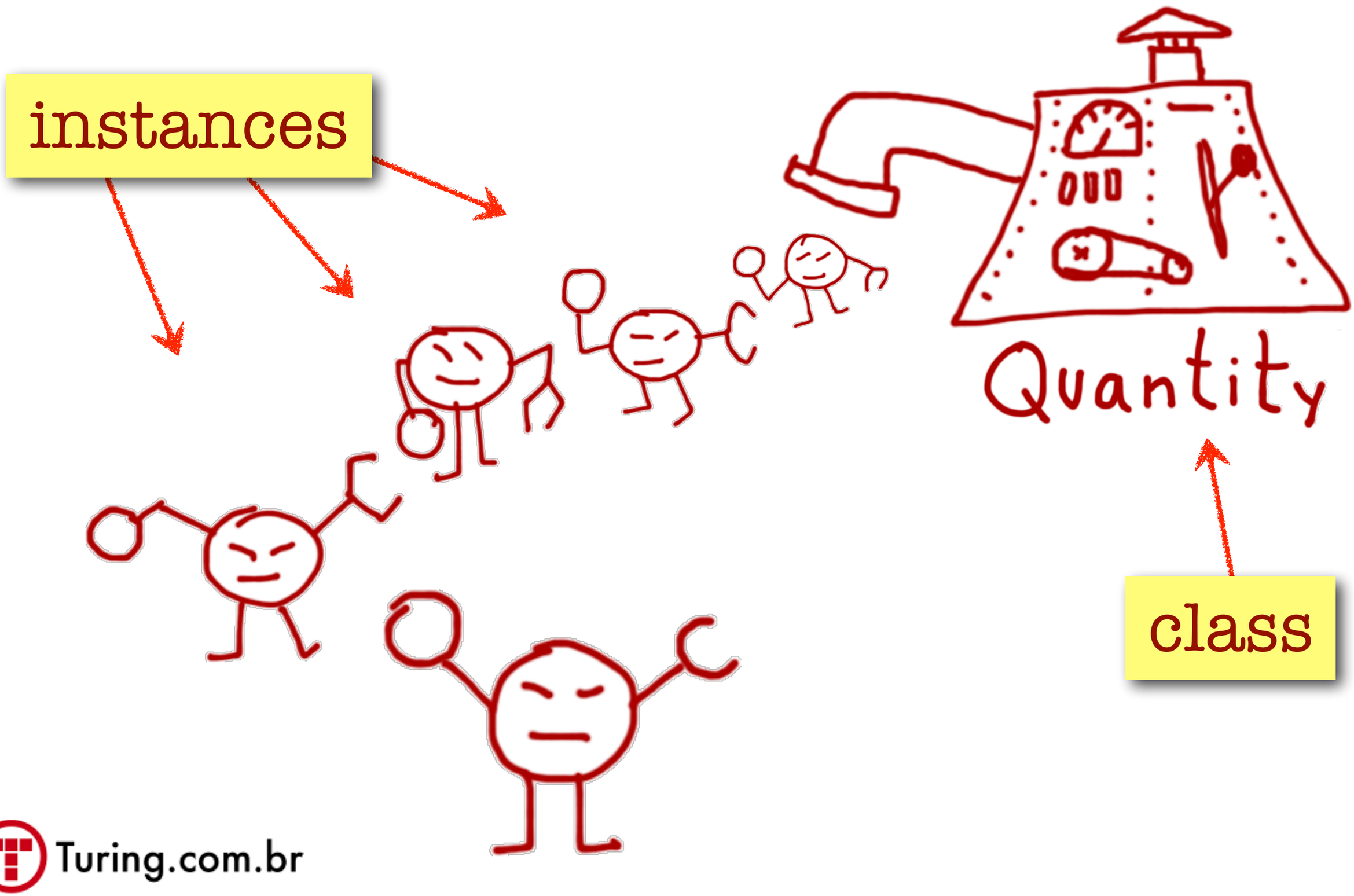
```
class Quantity(object):
    __counter = 0

    def __init__(self):
        prefix = '_' + self.__class__.__name__
        key = self.__class__.__counter
        self.target_name = '%s_%s' % (prefix, key)
        self.__class__.__counter += 1

    def __get__(self, instance, owner):
        return getattr(instance, self.target_name)

    def __set__(self, instance, value):
        if value > 0:
            setattr(instance, self.target_name, value)
        else:
            raise ValueError('value must be > 0')


class LineItem(object):
    weight = Quantity()
    price = Quantity()

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

❸ a class builds instances



LTO
LIGUAGEM
TOSCA de
OBJETOS

❸ a class builds instances



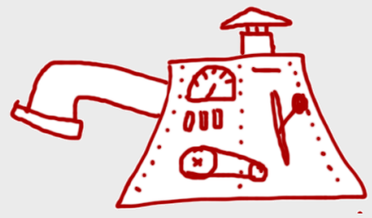
❸

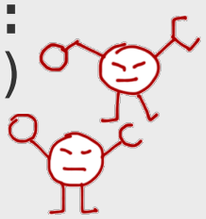
descriptor

managed class

managed attributes

```
class Quantity(object):
    __counter = 0

    def __init__(self):
        prefix = '_' + self.__class__.__name__
        key = self.__class__.__counter
        self.target_name = '%s_%s' % (prefix, key)
        self.__class__.__counter += 1

    def __get__(self, instance, owner):
        return getattr(instance, self.target_name)

    def __set__(self, instance, value):
        if value > 0:
            setattr(instance, self.target_name, value)
        else:
            raise ValueError('value must be > 0')


class LineItem(object):
    weight = Quantity()
    price = Quantity()

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```





❸ descriptor instances attach to managed class

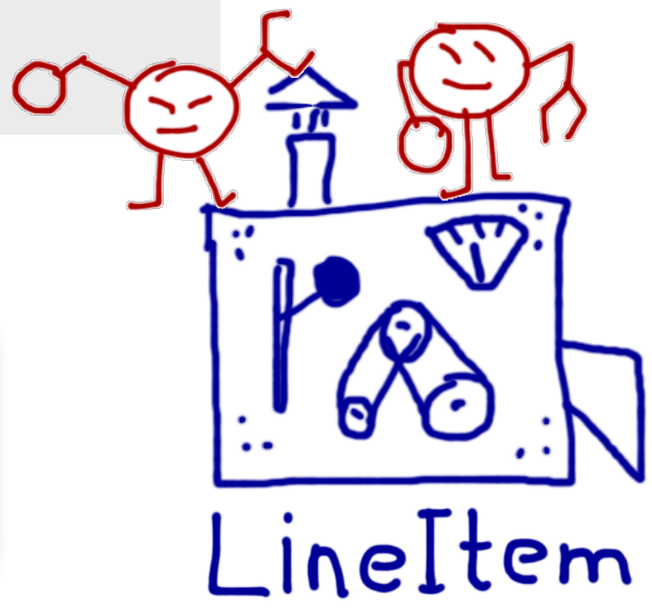
the **LineItem** class has 2 instances of **Quantity** bound to it

❸ use of the descriptor

```
class LineItem(object):
    weight = Quantity()
    price = Quantity()

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

the **Quantity** instances are created at import time

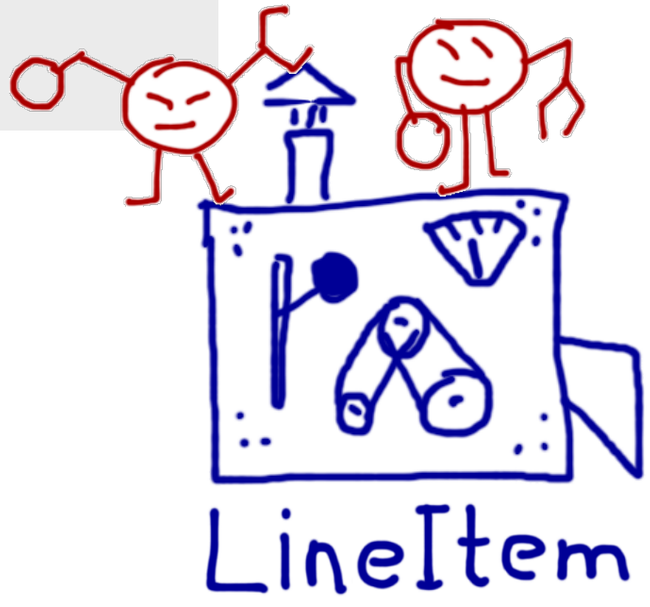
import time: when a module is loaded and its classes and functions are defined



❸ use of the descriptor

```
class LineItem(object):
    weight = Quantity()
    price = Quantity()

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

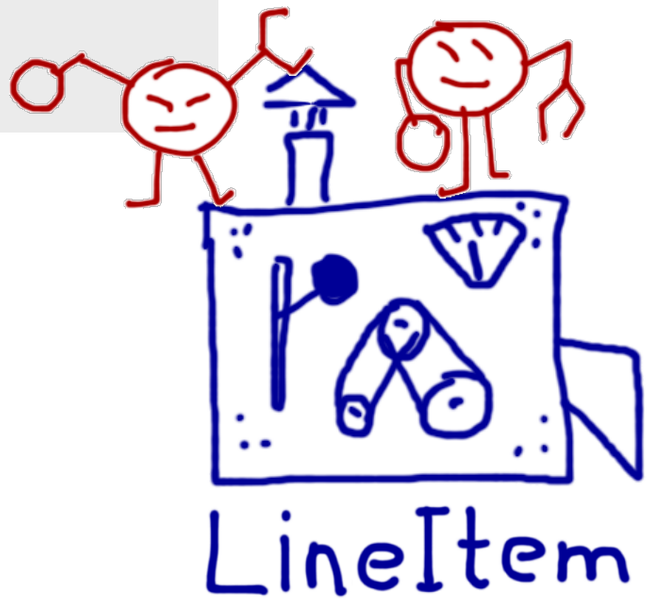
each **Quantity** instance manages one **LineItem** attribute



❸ use of the descriptor

every access to **weight** and **price** goes through their descriptors

```
class LineItem(object):
    weight = Quantity()
    price = Quantity()

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```



❸ descriptor implementation


```
class Quantity(object):
    __counter = 0

    def __init__(self):
        prefix = '_' + self.__class__.__name__
        key = self.__class__.__counter
        self.target_name = '%s_%s' % (prefix, key)
        self.__class__.__counter += 1

    def __get__(self, instance, owner):
        return getattr(instance, self.target_name)

    def __set__(self, instance, value):
        if value > 0:
            setattr(instance, self.target_name, value)
        else:
            raise ValueError('value must be > 0')
```

a descriptor is a class that defines **__get__** , **__set__** or **__delete__**

❸ descriptor implementation


```
class Quantity(object):
    __counter = 0

    def __init__(self):
        prefix = '_' + self.__class__.__name__
        key = self.__class__.__counter
        self.target_name = '%s_%s' % (prefix, key)
        self.__class__.__counter += 1

    def __get__(self, instance, owner):
        return getattr(instance, self.target_name)

    def __set__(self, instance, value):
        if value > 0:
            setattr(instance, self.target_name, value)
        else:
            raise ValueError('value must be > 0')
```

self is the descriptor instance, bound to **weight** or **price**

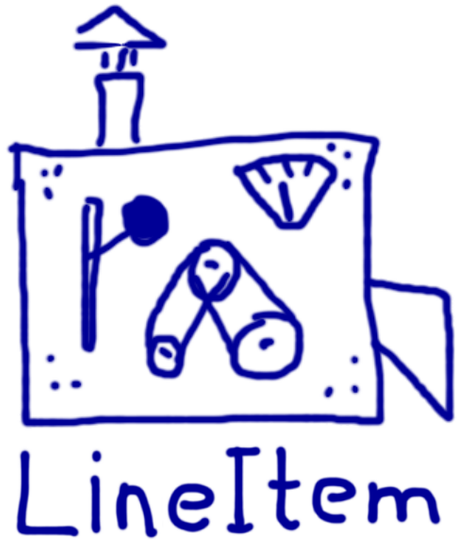
❸ descriptor implementation


```
class Quantity(object):
    __counter = 0

    def __init__(self):
        prefix = '_' + self.__class__.__name__
        key = self.__class__.__counter
        self.target_name = '%s_%s' % (prefix, key)
        self.__class__.__counter += 1

    def __get__(self, instance, owner):
        return getattr(instance, self.target_name)

    def __set__(self, instance, value):
        if value > 0:
            setattr(instance, self.target_name, va
        else:
            raise ValueError('value must be > 0')
```

self is the descriptor instance, bound to **weight** or **price**

instance is the **LineItem** instance being handled

❸ descriptor implementation



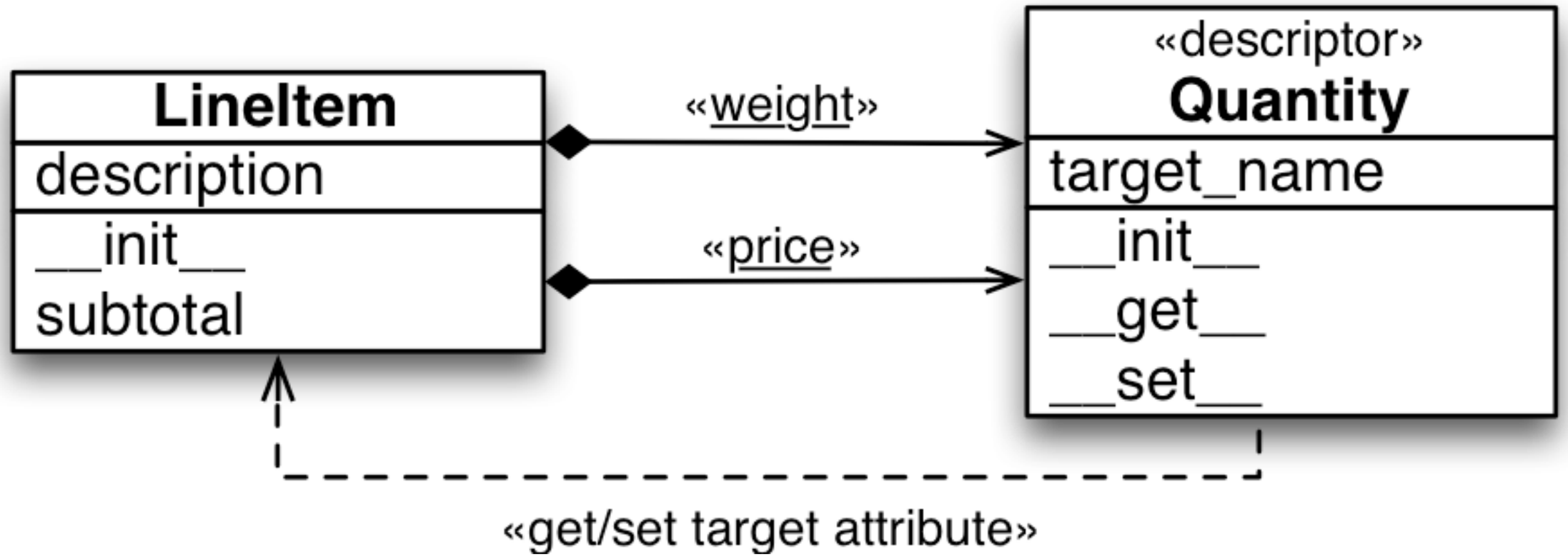

```
class Quantity(object):
    __counter = 0

    def __init__(self):
        prefix = '_' + self.__class__.__name__
        key = self.__class__.__counter
        self.target_name = '%s_%s' % (prefix, key)
        self.__class__.__counter += 1

    def __get__(self, instance, owner):
        return getattr(instance, self.target_name)

    def __set__(self, instance, value):
        if value > 0:
            setattr(instance, self.target_name, va
        else:
            raise ValueError('value must be > 0')
```

owner is **LineItem**, the class to which **instance** belongs

instance is the **LineItem** instance being handled

self is the descriptor instance, bound to **weight** or **price**

❸ descriptor implementation

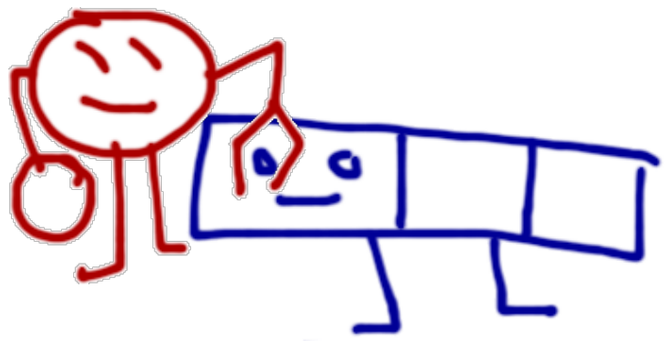


__get__ and **__set__** read and write the target attribute in each **LineItem** instance

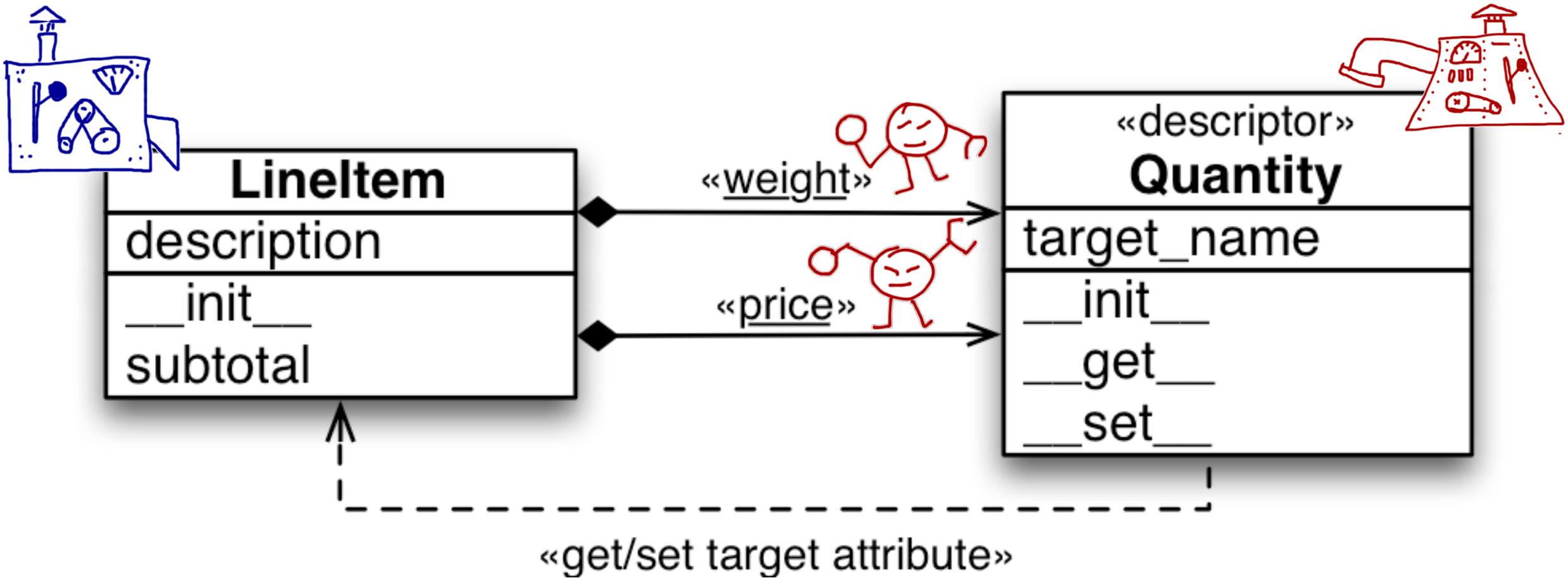
❸ descriptor implementation

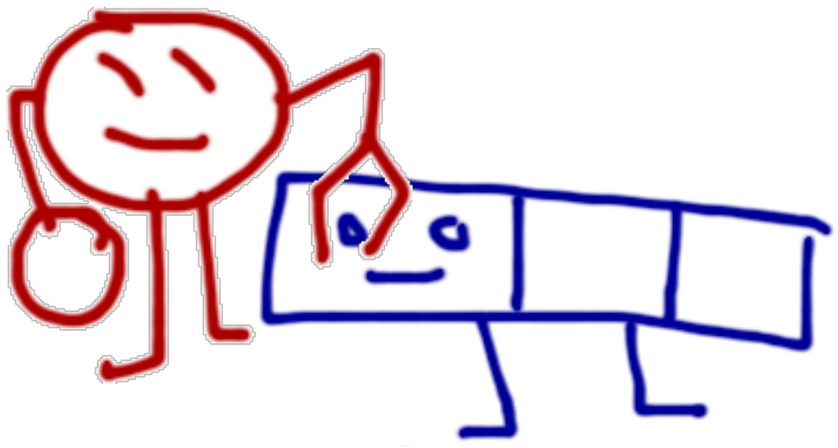

```
class Quantity(object):
    __counter = 0

    def __init__(self):
        prefix = '_' + self.__class__.__name__
        key = self.__class__.__counter
        self.target_name = '%s_%s' % (prefix, key)
        self.__class__.__counter += 1

    def __get__(self, instance, owner):
        return getattr(instance, self.target_name)

    def __set__(self, instance, value):
        if value > 0:
            setattr(instance, self.target_name, value)
        else:
            raise ValueError('value must be > 0')
```

target_name is the name of the **LineItem** instance attribute where where we store the value of the attribute managed by this descriptor instance (**self**)



❸ descriptor implementation

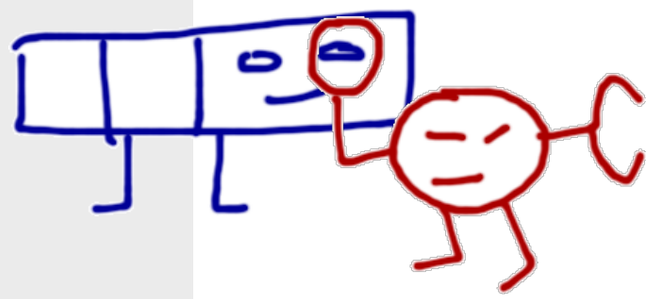




__get__ and **__set__** read and write the target attribute in each **LineItem** instance

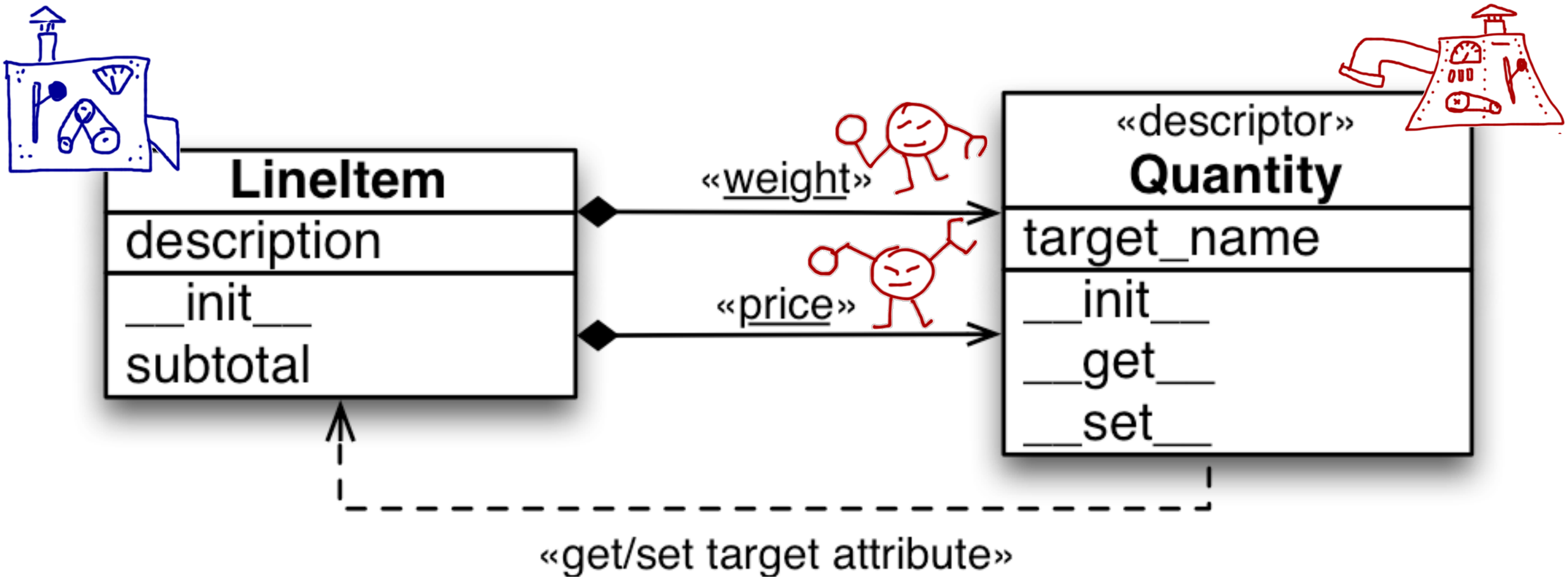
❸ descriptor implementation

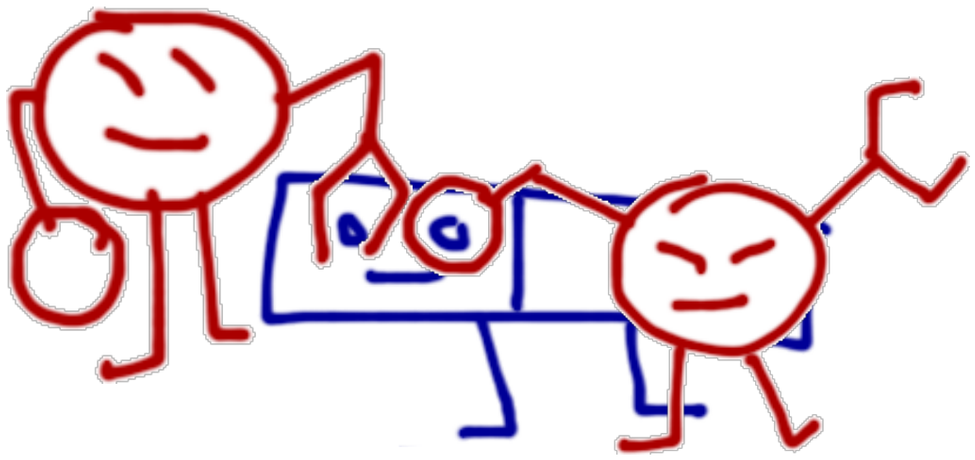

```
class Quantity(object):
    __counter = 0

    def __init__(self):
        prefix = '_' + self.__class__.__name__
        key = self.__class__.__counter
        self.target_name = '%s_%s' % (prefix, key)
        self.__class__.__counter += 1

    def __get__(self, instance, owner):
        return getattr(instance, self.target_name)

    def __set__(self, instance, value):
        if value > 0:
            setattr(instance, self.target_name, value)
        else:
            raise ValueError('value must be > 0')
```



__get__ and **__set__** use the **getattr** and **setattr** built-in functions to read and write the target attribute in the managed instance

❸ descriptor implementation





each descriptor instance manages a specific **LineItem** instance attribute, and needs a specific **target_name**

❸ descriptor implementation


```
class Quantity(object):
    __counter = 0

    def __init__(self):
        prefix = '_' + self.__class__.__name__
        key = self.__class__.__counter
        self.target_name = '%s_%s' % (prefix, key)
        self.__class__.__counter += 1

    def __get__(self, instance, owner):
        return getattr(instance, self.target_name)

    def __set__(self, instance, value):
        if value > 0:
            setattr(instance, self.target_name, value)
        else:
            raise ValueError('value must be > 0')
```

each **Quantity** instance must create and use a unique target attribute name

❸ the target attributes

```
>>> raisins = LineItem('Golden raisins', 10, 6.95)
>>> dir(raisins)
['_Quantity_0', '_Quantity_1', '__class__', '__delattr__' ...
 'description', 'price', 'subtotal', 'weight']
>>> raisins.weight
10
>>> raisins._Quantity_0
10
>>> raisins.price
6.95
>>> raisins._Quantity_1
6.95
```

_Quantity_0 and **_Quantity_1** are the names of the instance attributes where the **weight** and **price** values of each **LineItem** instance are stored

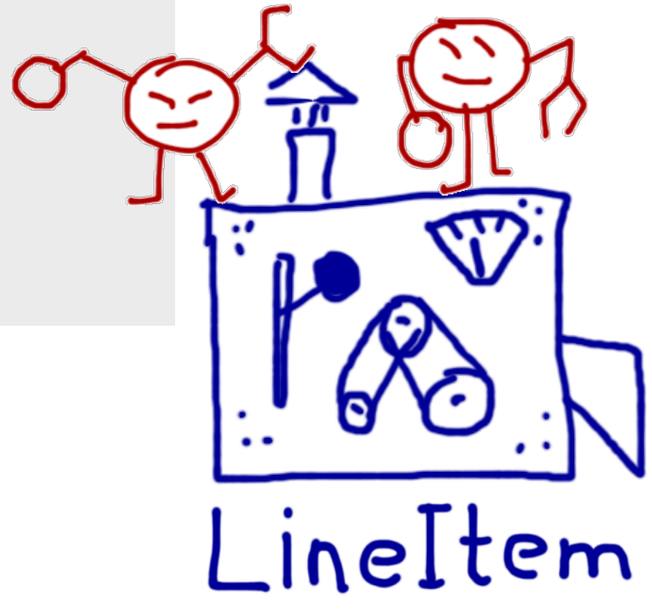
❸ room for improvement

- It would be better if the target attributes were proper protected attributes
  - **`_LineItem__weight`** instead of **`_Quantity_0`**
- To do so, we must find out the name of the managed attribute (eg. **`weight`**)
  - this is not as easy at it sounds
  - the extra complication may not be worthwhile

❸ the challenge

```
class LineItem(object):
    weight = Quantity()
    price = Quantity()

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

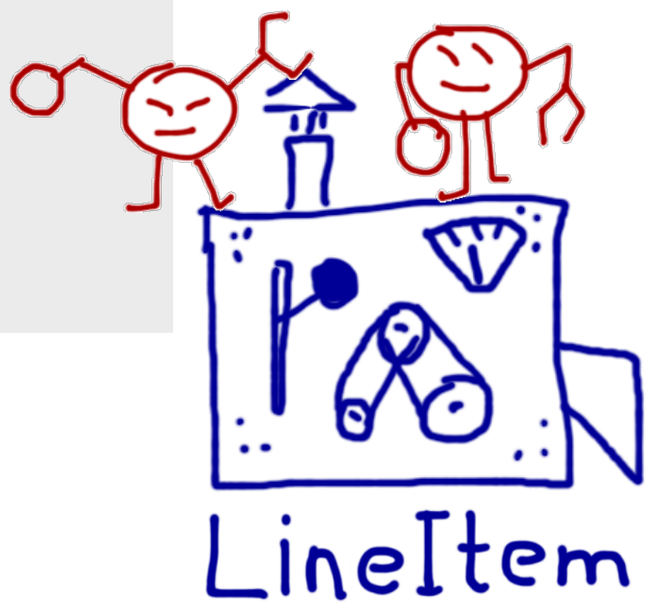
when each descriptor is instantiated, the **LineItem** class does not exist, and the managed attributes don't exist either



❸ the challenge

for example, the **weight** attribute is only created after **Quantity()** is evaluated

```
class LineItem(object):
    weight = Quantity()
    price = Quantity()

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

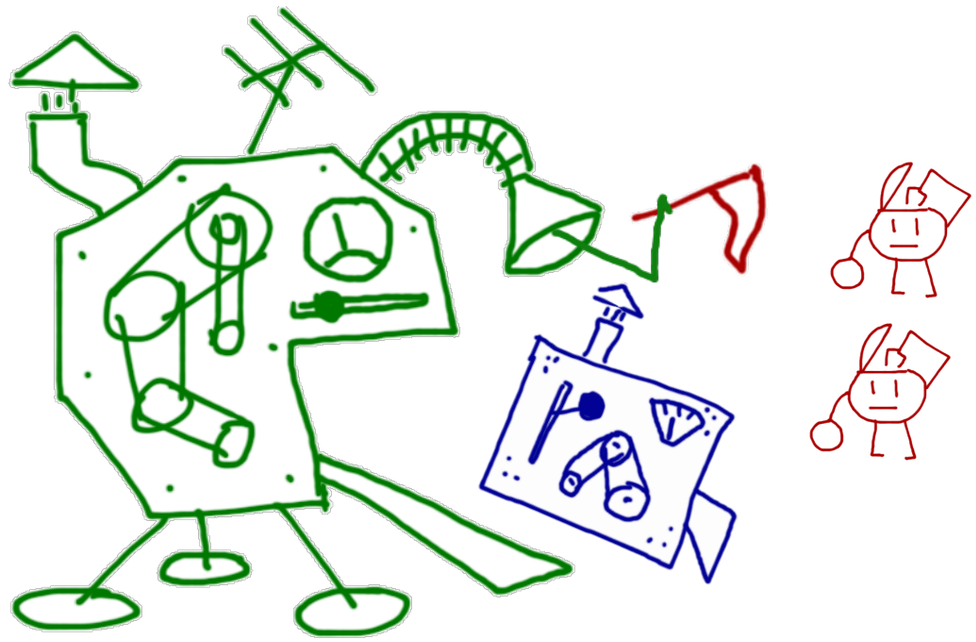


What now?

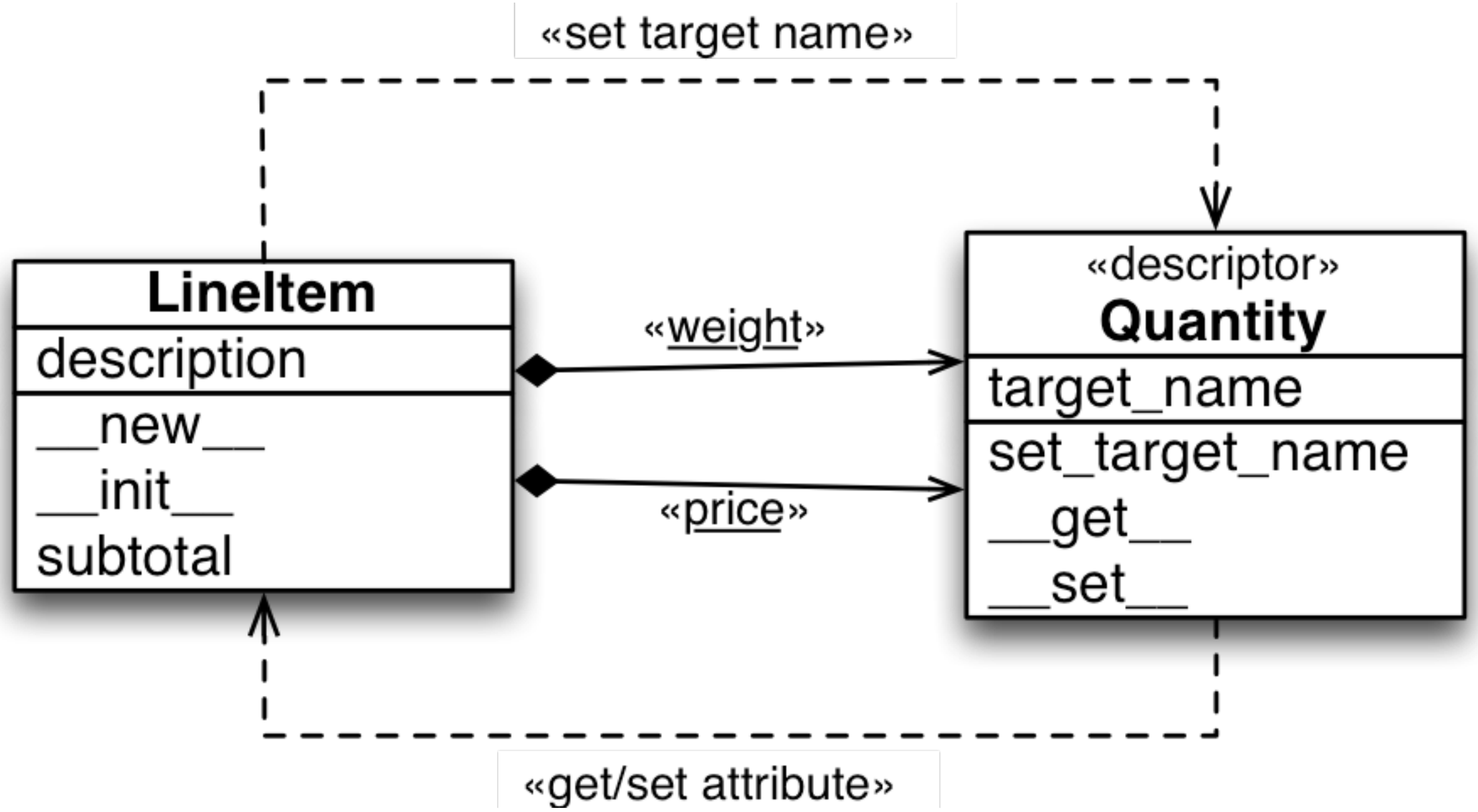
- If the descriptors need to know the names of the class attributes they are bound to...

  (perhaps you need to persist to a database and want to use descriptive column names)

- ...then you will need to control the construction of the managed class with a **metaclass**



❹ setting descriptive names



LineItem.__new__ calls **«quantity».set_target_name**

❹ setting descriptive names

LineItem.__new__ calls
«quantity».set_target_name

```
class LineItem(object):
    weight = Quantity()
    price = Quantity()

    def __new__(cls, *args, **kwargs):
        for key, attr in cls.__dict__.items():
            if isinstance(attr, Quantity):
                attr.set_target_name('__' + cls.__name__, key)
        return super(LineItem, cls).__new__(cls, *args, **kwargs)

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

❹ setting descriptive names

«quantity».set_target_name
sets the target attribute name

```
class Quantity(object):

    def set_target_name(self, prefix, key):
        self.target_name = '%s_%s' % (prefix, key)

    def __get__(self, instance, owner):
        return getattr(instance, self.target_name)

    def __set__(self, instance, value):
        if value > 0:
            setattr(instance, self.target_name, value)
        else:
            raise ValueError('value must be > 0')
```

❹ setting descriptive names

```
class LineItem(object):
    weight = Quantity()
    price = Quantity()

    def __new__(cls, *args, **kwargs):
        for key, attr in cls.__dict__.items():
            if isinstance(attr, Quantity):
                attr.set_target_name('__' + cls.__name__, key)
        return super(LineItem, cls).__new__(cls, *args, **kwargs)

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price


    def subtotal(self):
        return self.weight * self.price
```

when **__init__** runs, each descriptor already has a proper target attribute name

❹ setting descriptive names

LineItem.__new__ calls **«quantity».set_target_name**

❹ usar nomes descritivos

«quantidade».set_nome
redefine o **nome_alvo**

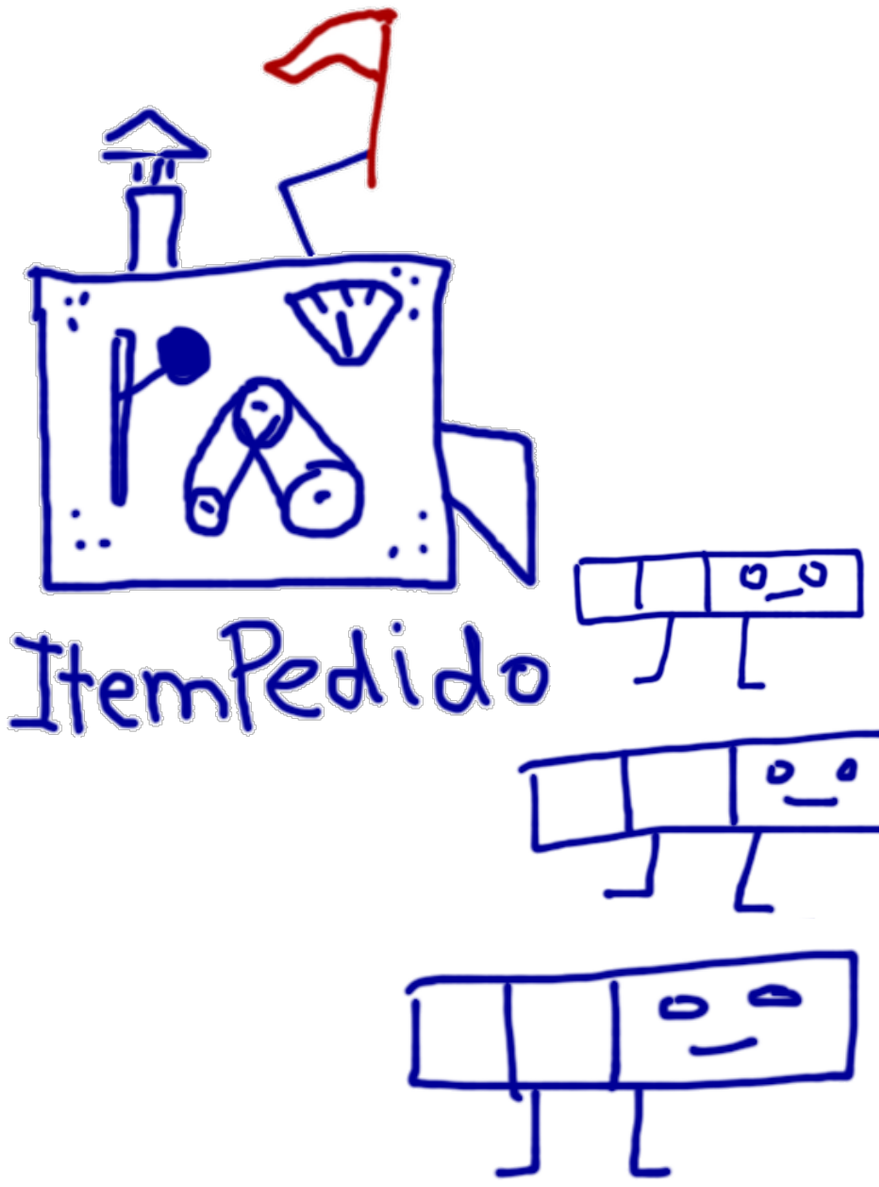
❹ nomes descritivos

```
>>> ervilha = ItemPedido('ervilha partida', .5, 3.95)
>>> ervilha.descricao, ervilha.peso, ervilha.preco
('ervilha partida', 0.5, 3.95)
>>> dir(ervilha)
['__ItemPedido_peso', '__ItemPedido_preco',
'__class__', '__delattr__', '__dict__', '__doc__',
'__format__', '__getattribute__', '__hash__',
'__init__', '__module__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'descricao', 'peso', 'preco',
'subtotal']
```

nesta implementação e nas próximas, os nomes dos atributos-alvo seguem a convenção de atributos protegidos de Python

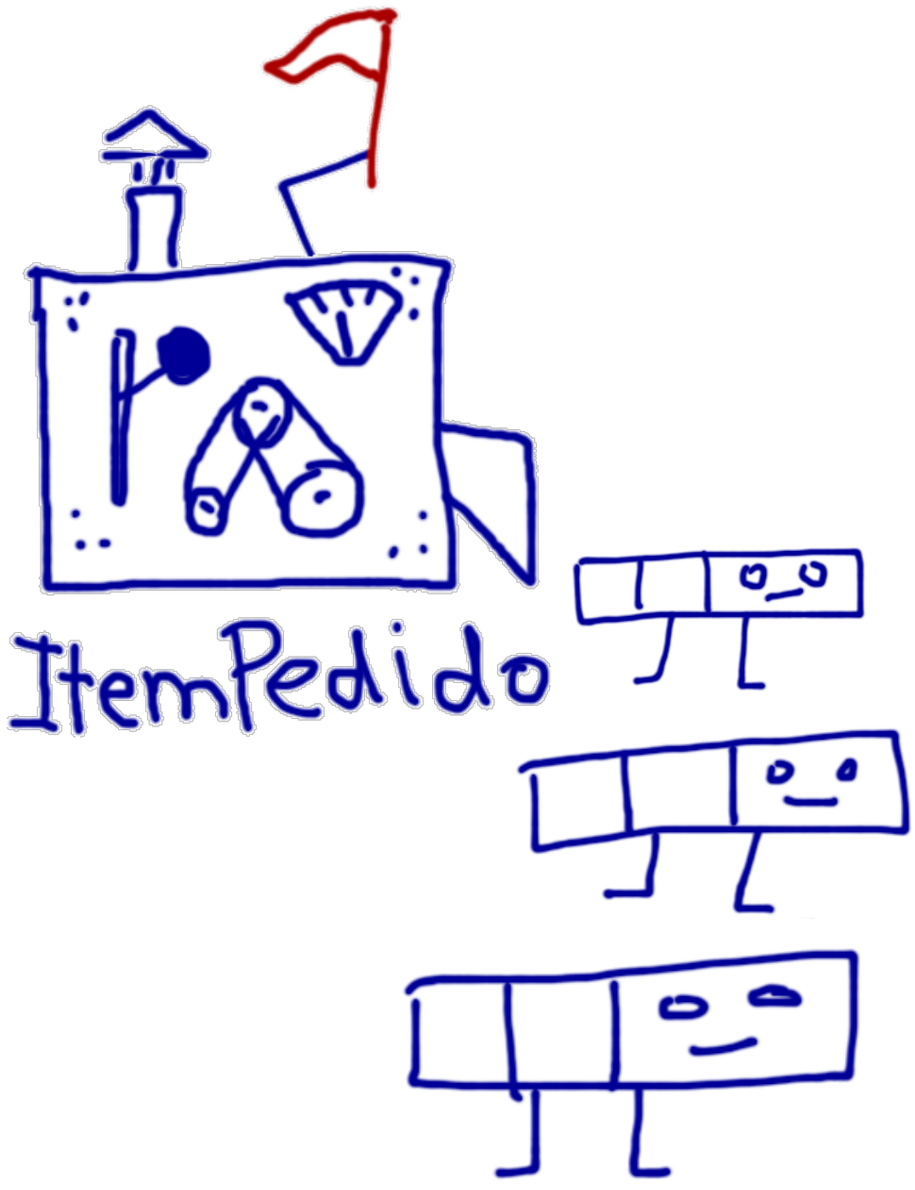
❹ funciona, mas custa caro

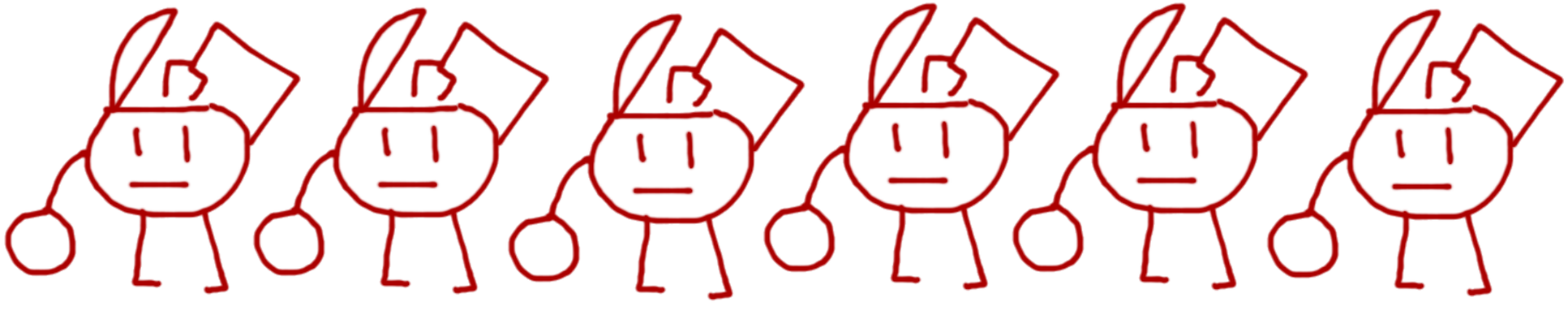
- **ItemPedido** aciona **__new__** para construir cada nova instância
- Porém a associação dos descritores é com a classe **ItemPedido**: o nome do atributo-alvo nunca vai mudar, uma vez definido corretamente



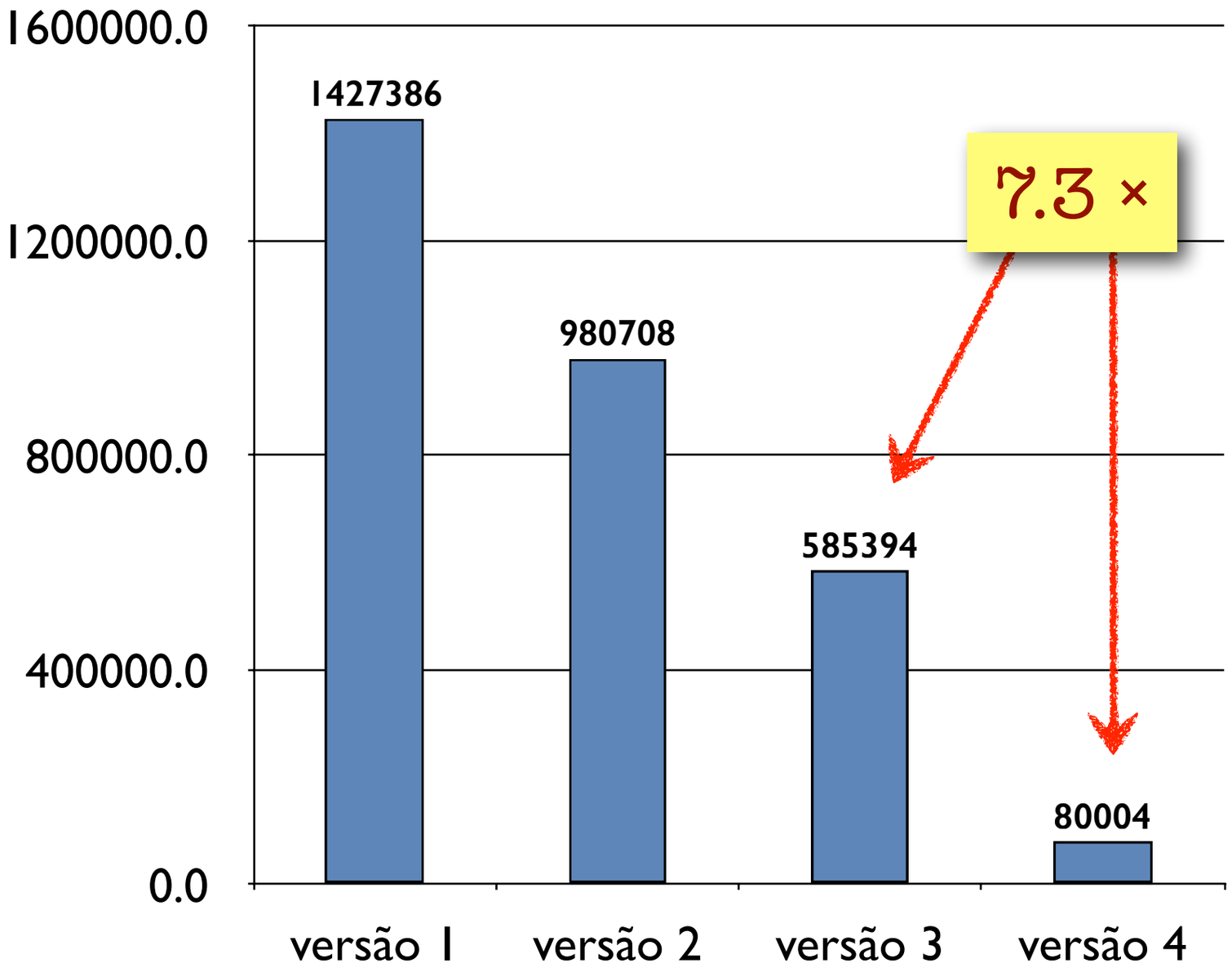
❹ funciona, mas custa caro

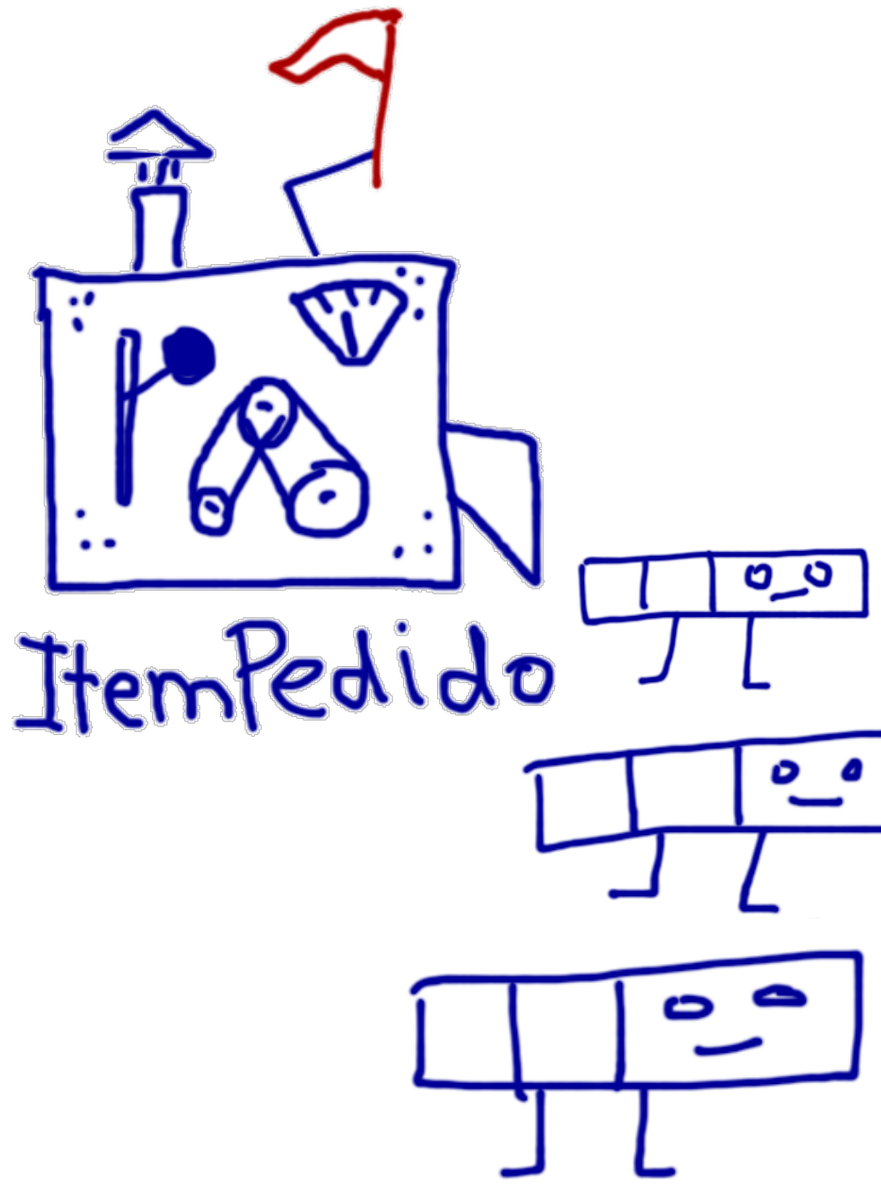
- Isso significa que para cada nova instância de **ItemPedido** que é criada, **«quantidade».set_nome** é invocado duas vezes
- Mas o nome do atributo-alvo não tem porque mudar na vida de uma **«quantidade»**





❹ funciona, mas custa caro





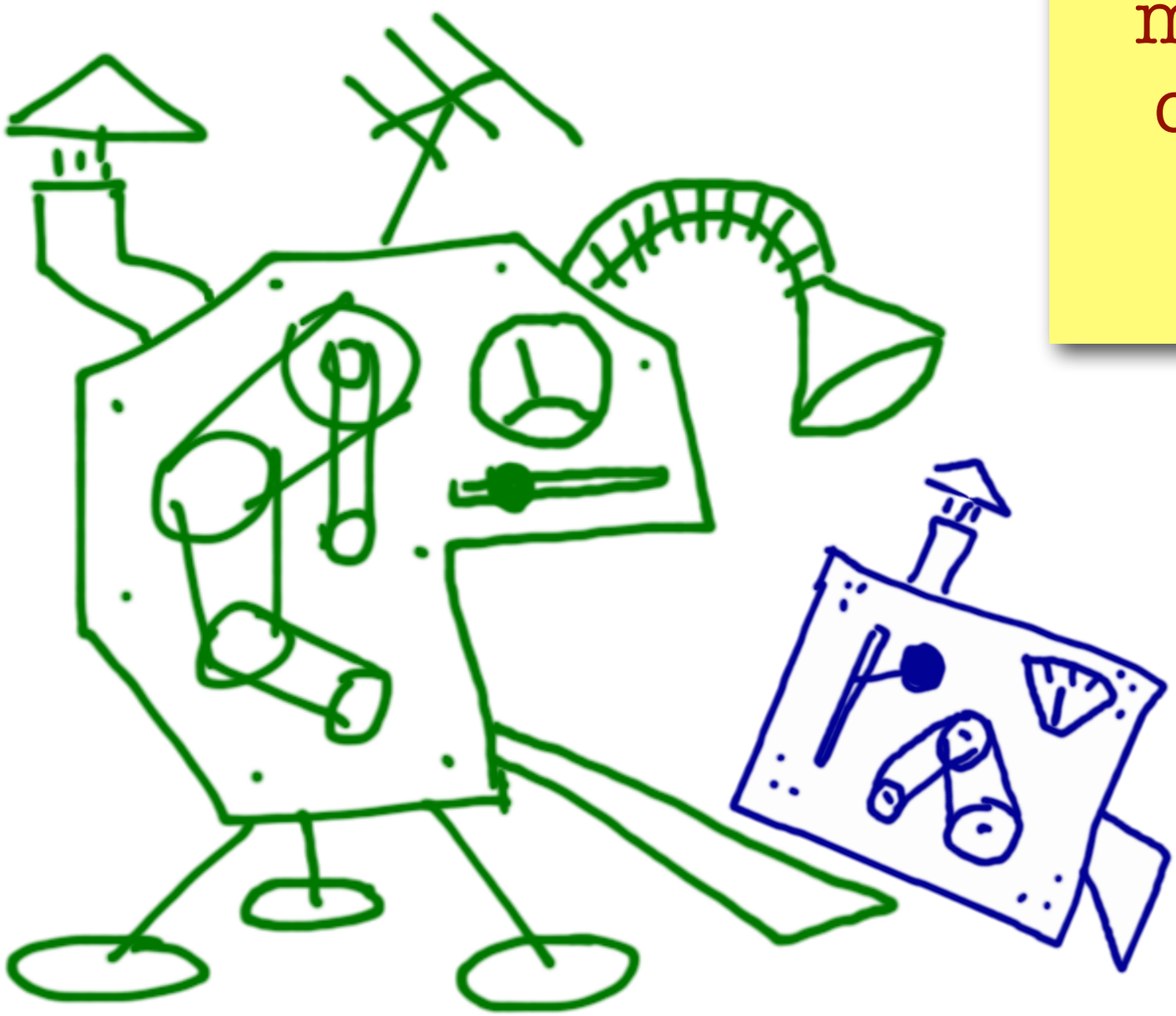
Número de instâncias de **ItemPedido** criadas por segundo (MacBook Pro 2011, Intel Core i7)


5

❺ como evitar trabalho inútil

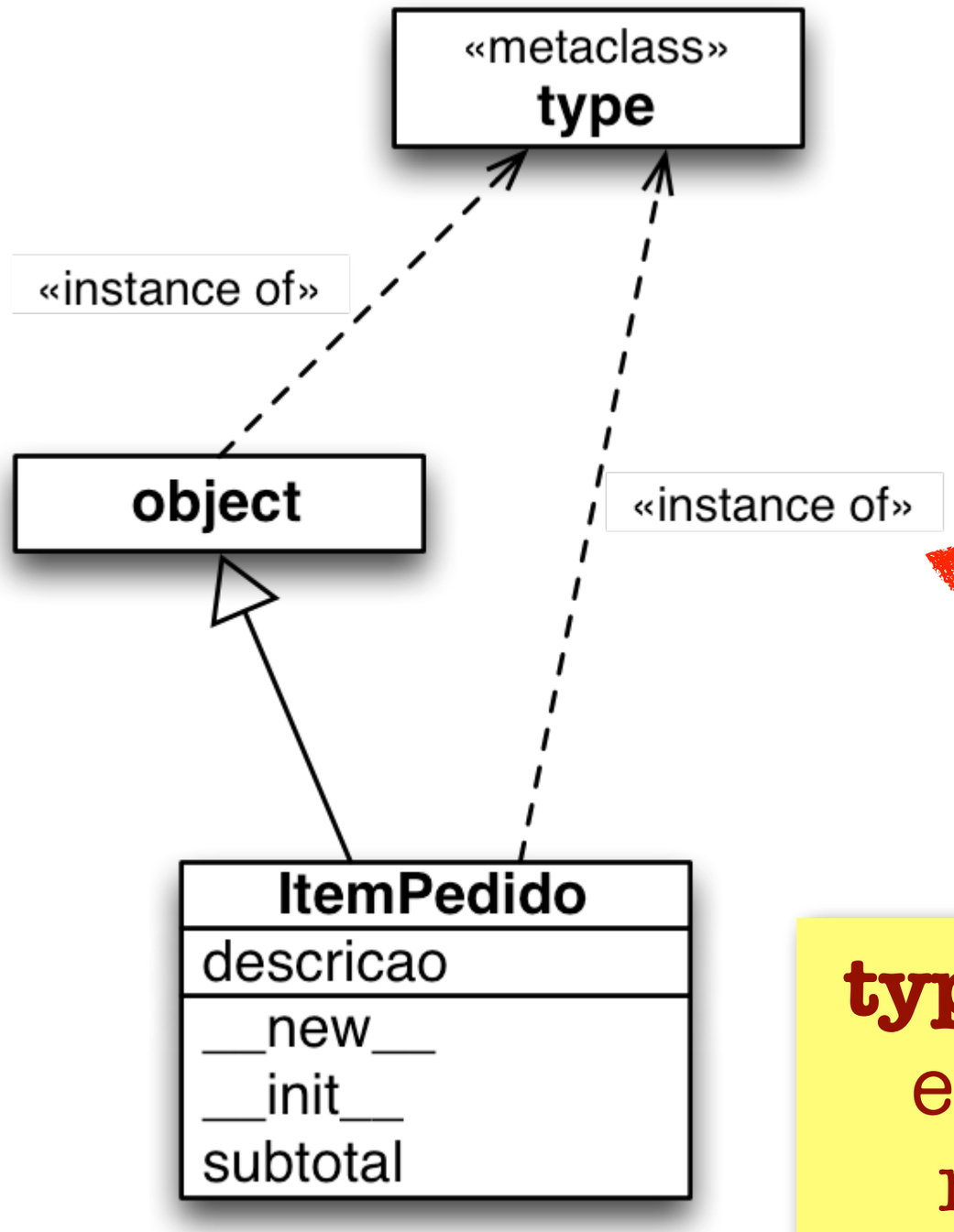
- **ItemPedido.__new__** resolveu mas de modo ineficiente.
- Cada «**quantidade**» deve receber o nome do seu atributo-alvo apenas uma vez, quando a própria classe **ItemPedido** for criada
- Para isso precisamos de uma...

❺ metaclasses create classes!



metaclasses are classes whoses instances are classes

❺ metaclasses criam classes!



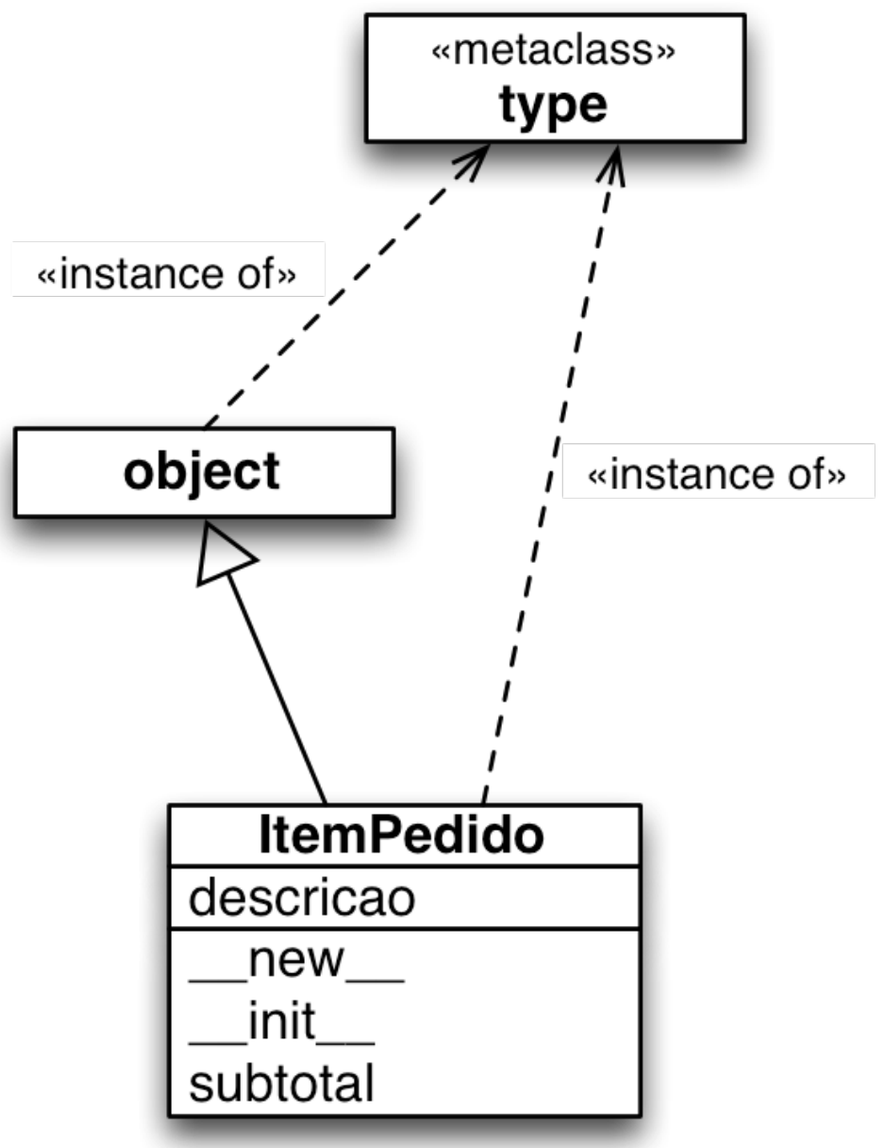
metaclasses são classes cujas instâncias são classes

ItemPedido é uma instância de **type**

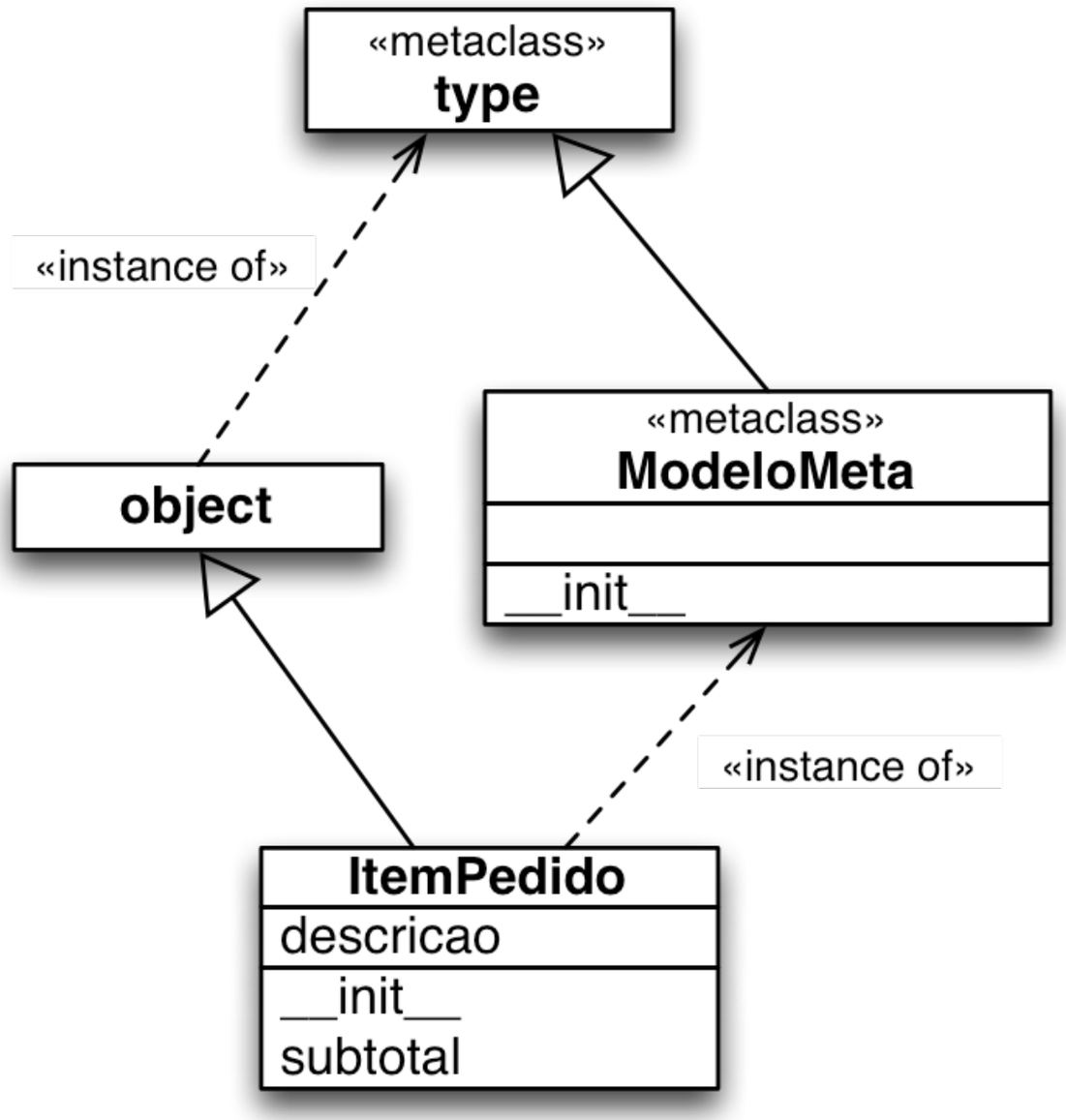
type é a metaclasse default em Python: a classe que normalmente constroi outras classes

❺ nossa metaclasse

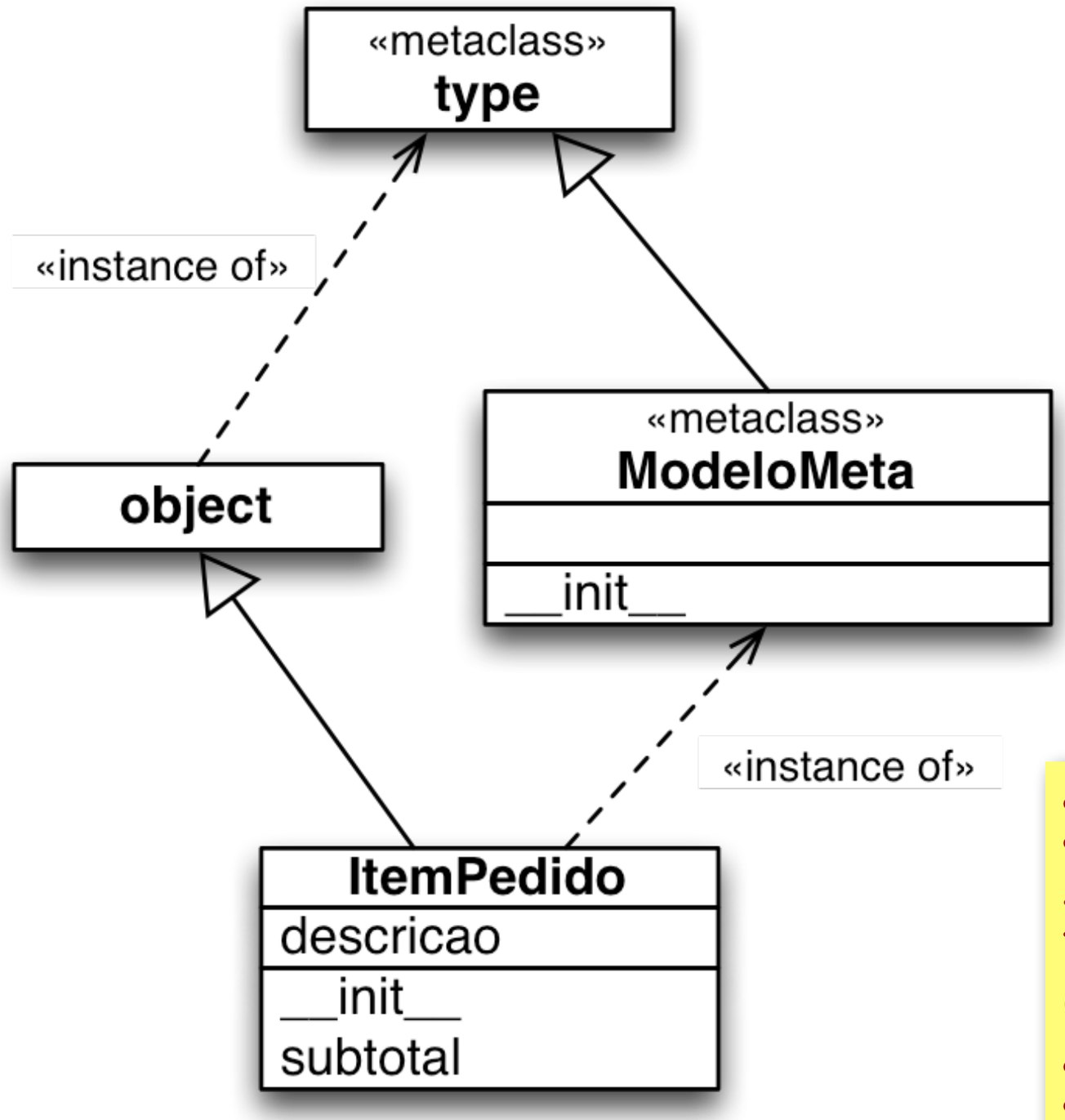
antes



depois



❺ nossa metaclasse

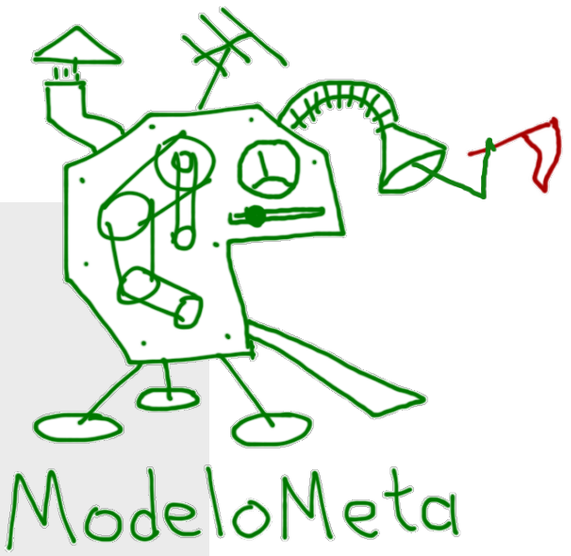


ModeloMeta é a metaclasse que vai construir a classe **ItemPedido**

ModeloMeta.__init__ fará apenas uma vez o que antes era feito em **ItemPedido.__new__** a cada nova instância

❺ nossa metaclasse

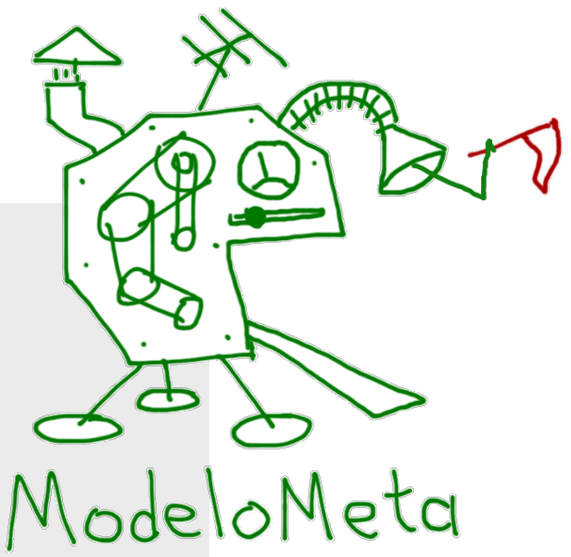
```
class ModeloMeta(type):

    def __init__(cls, nome, bases, dic):
        super(ModeloMeta, cls).__init__(nome, bases, dic)
        for chave, atr in dic.items():
            if hasattr(atr, 'set_nome'):
                atr.set_nome('__' + nome, chave)


class ItemPedido(object):
    __metaclass__ = ModeloMeta

    peso = Quantidade()
    preco = Quantidade()

    def __init__(self, descricao, peso, preco):
        self.descricao = descricao
        self.peso = peso
        self.preco = preco

    def subtotal(self):
        return self.peso * self.preco
```



Assim dizemos que a classe **ItemPedido** herda de **object**, mas é uma instância (construida por) **ModeloMeta**

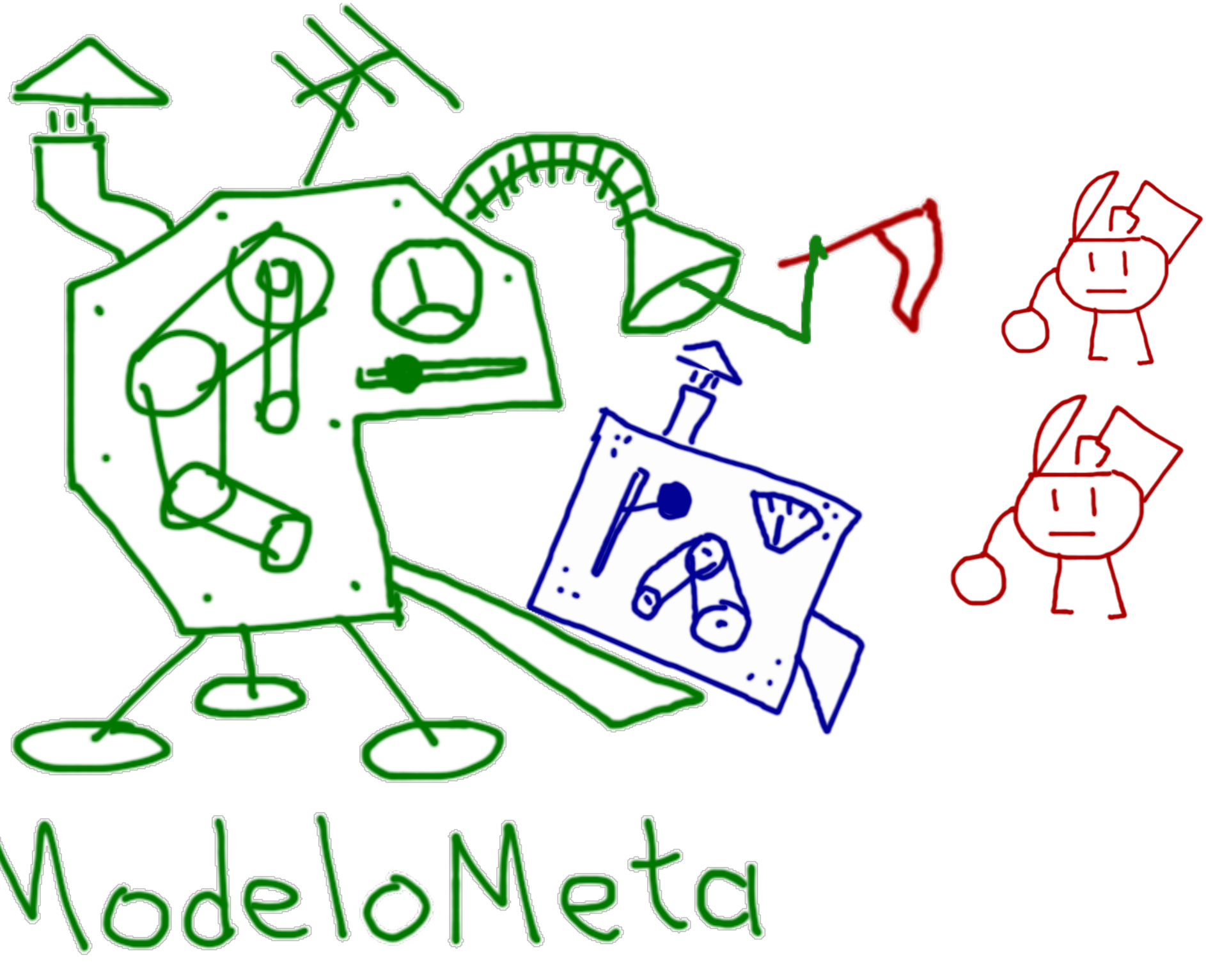
❺ nossa metaclasse

```
class ModeloMeta(type):

    def __init__(cls, nome, bases, dic):
        super(ModeloMeta, cls).__init__(nome, bases, dic)
        for chave, atr in dic.items():
            if hasattr(atr, 'set_nome'):
                atr.set_nome('__' + nome, chave)


class ItemPedido(object):
    __metaclass__ = ModeloMeta

    peso = Quantidade()
    preco = Quantidade()

    def __init__(self, descricao, peso, preco):
        self.descricao = descricao
        self.peso = peso
        self.preco = preco

    def subtotal(self):
        return self.peso * self.preco
```

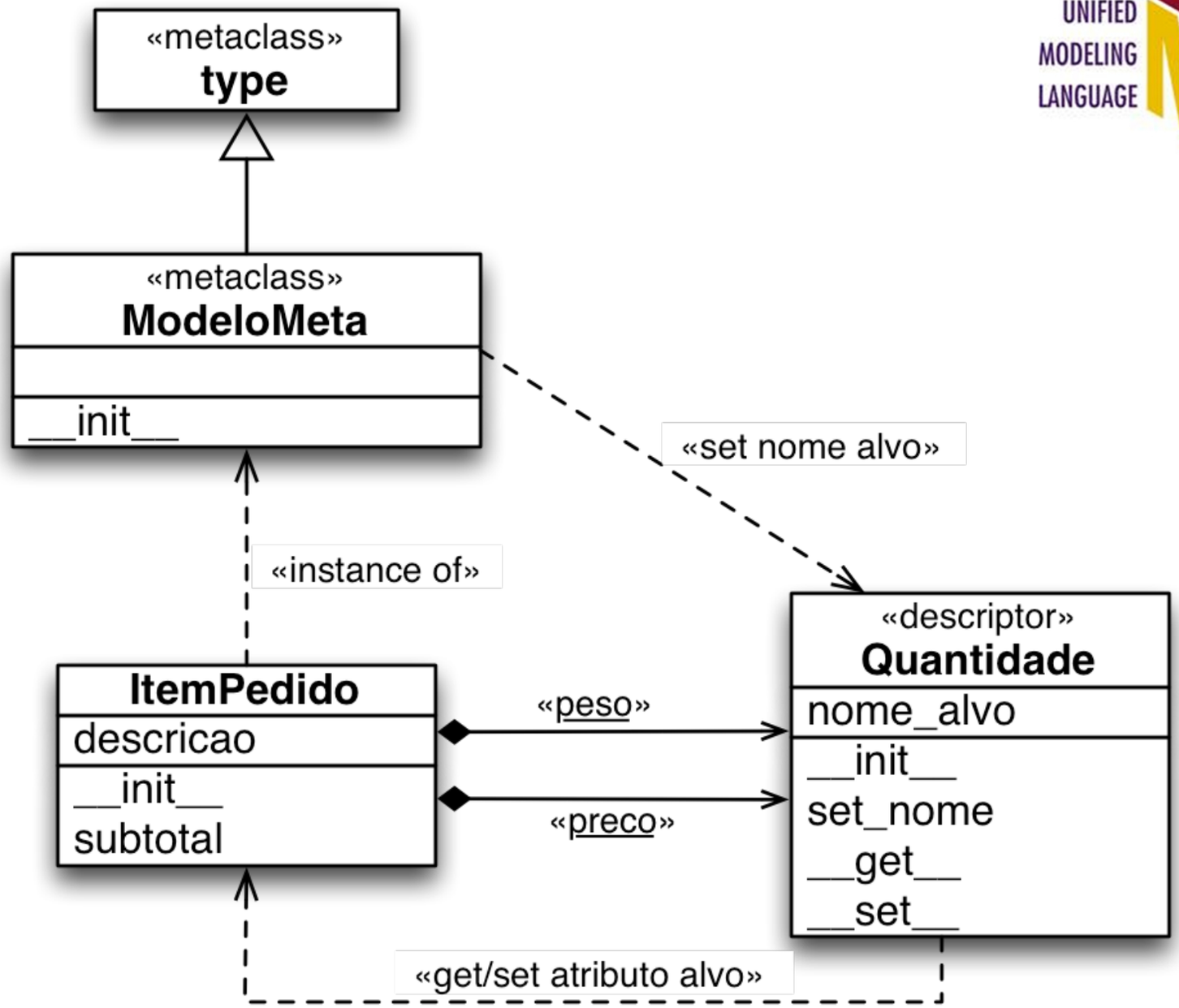


Este **__init__** invoca **«quantidade».set_nome**, para cada descritor, uma vez só, na inicialização da classe **ItemPedido**

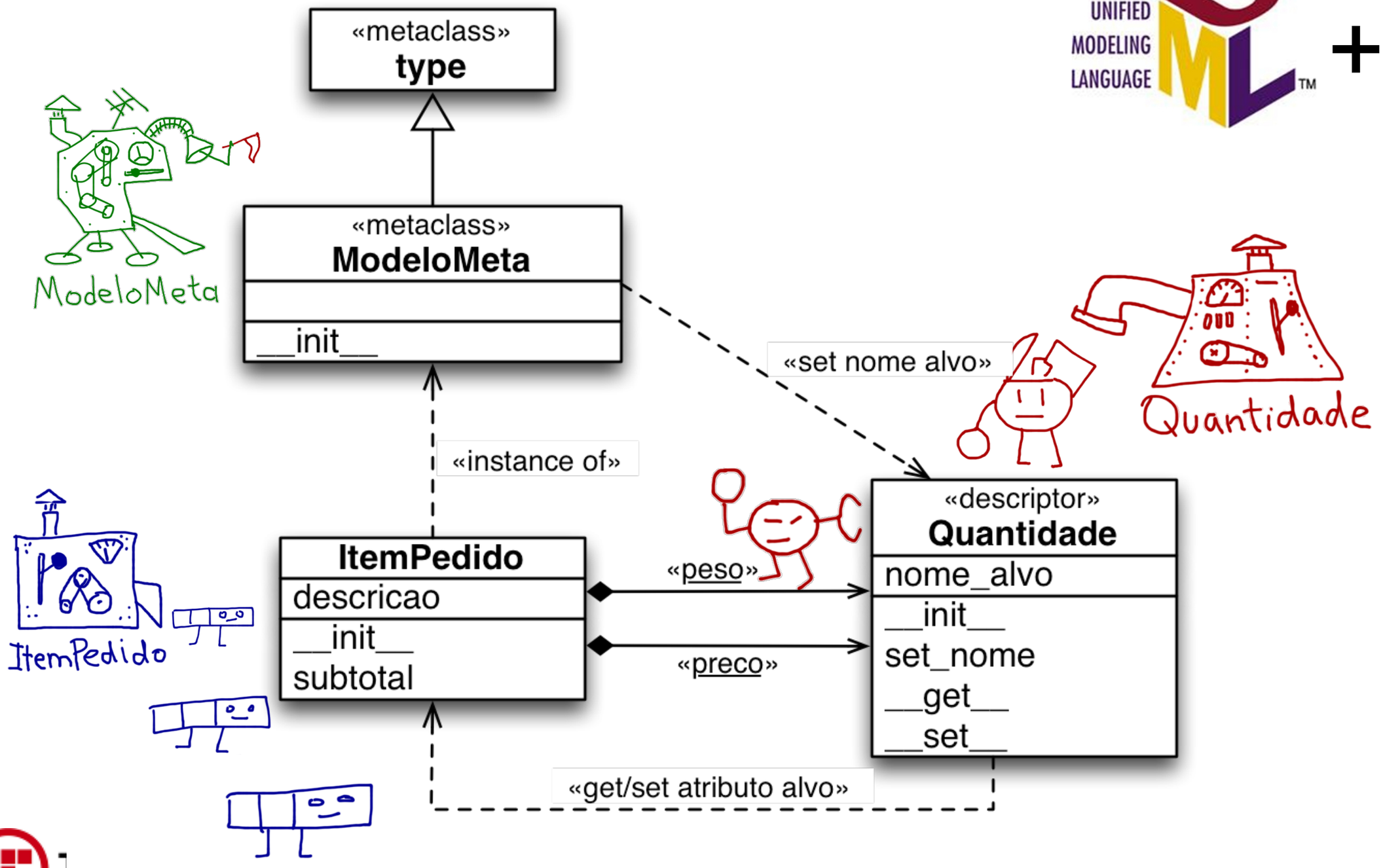
❺ nossa metaclasse em ação



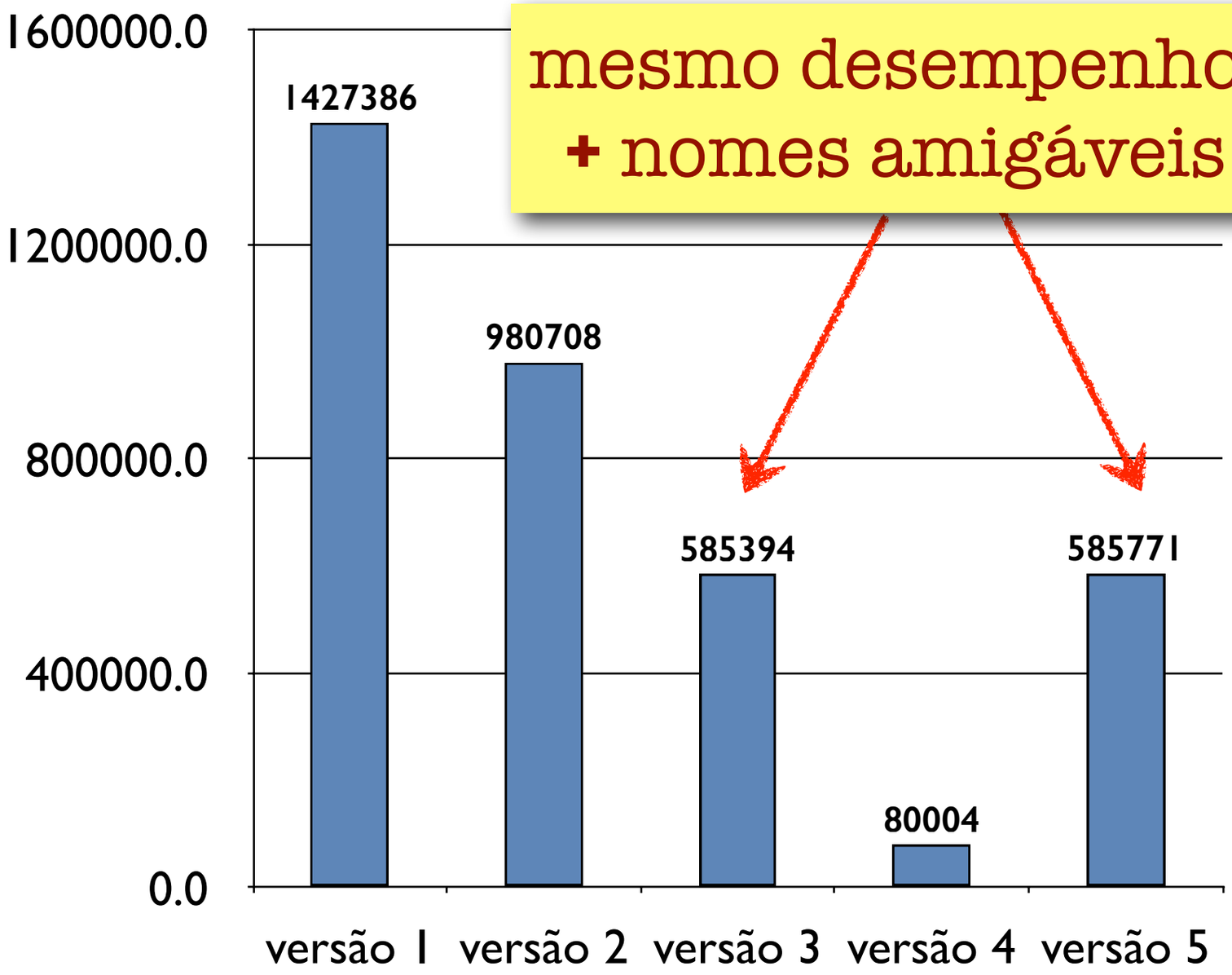
❺ nossa metaclasse em ação



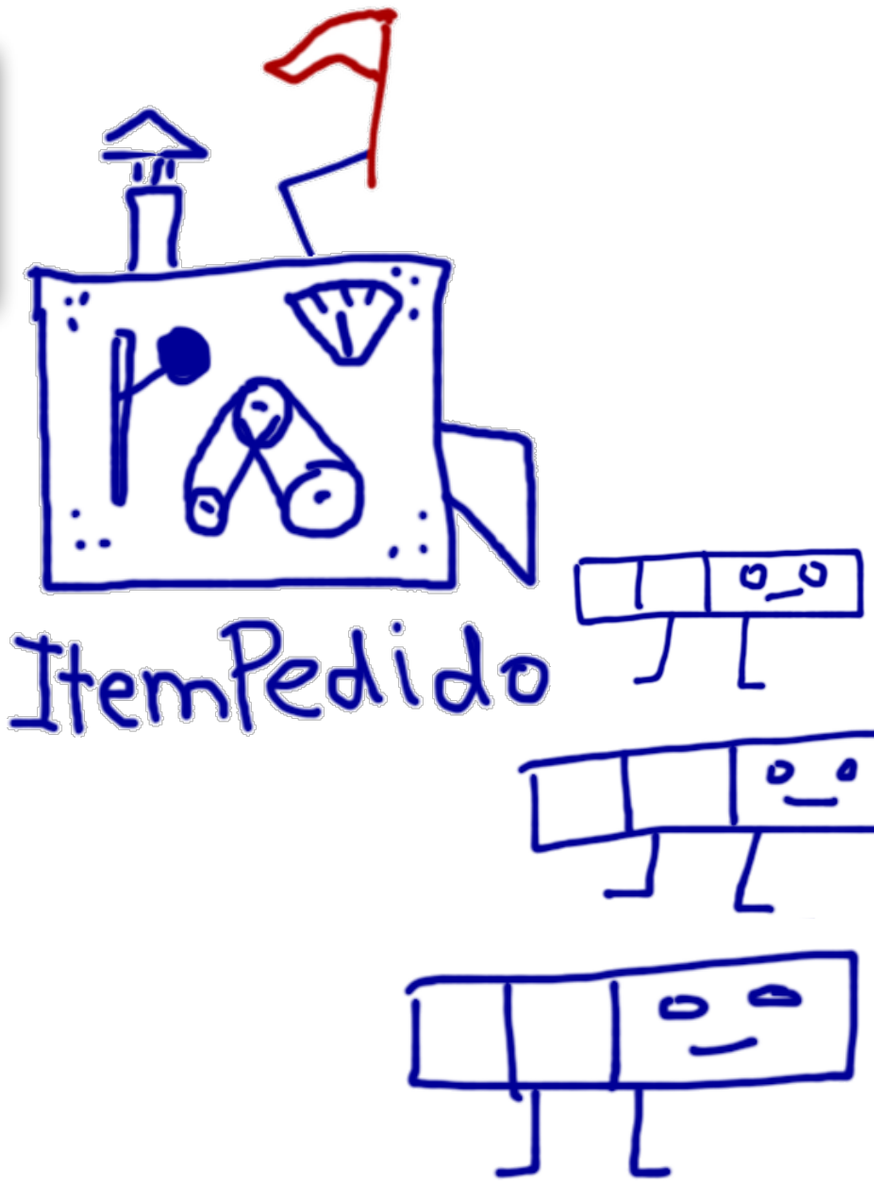
❺ nossa metaclasse em ação



❺ desempenho melhor



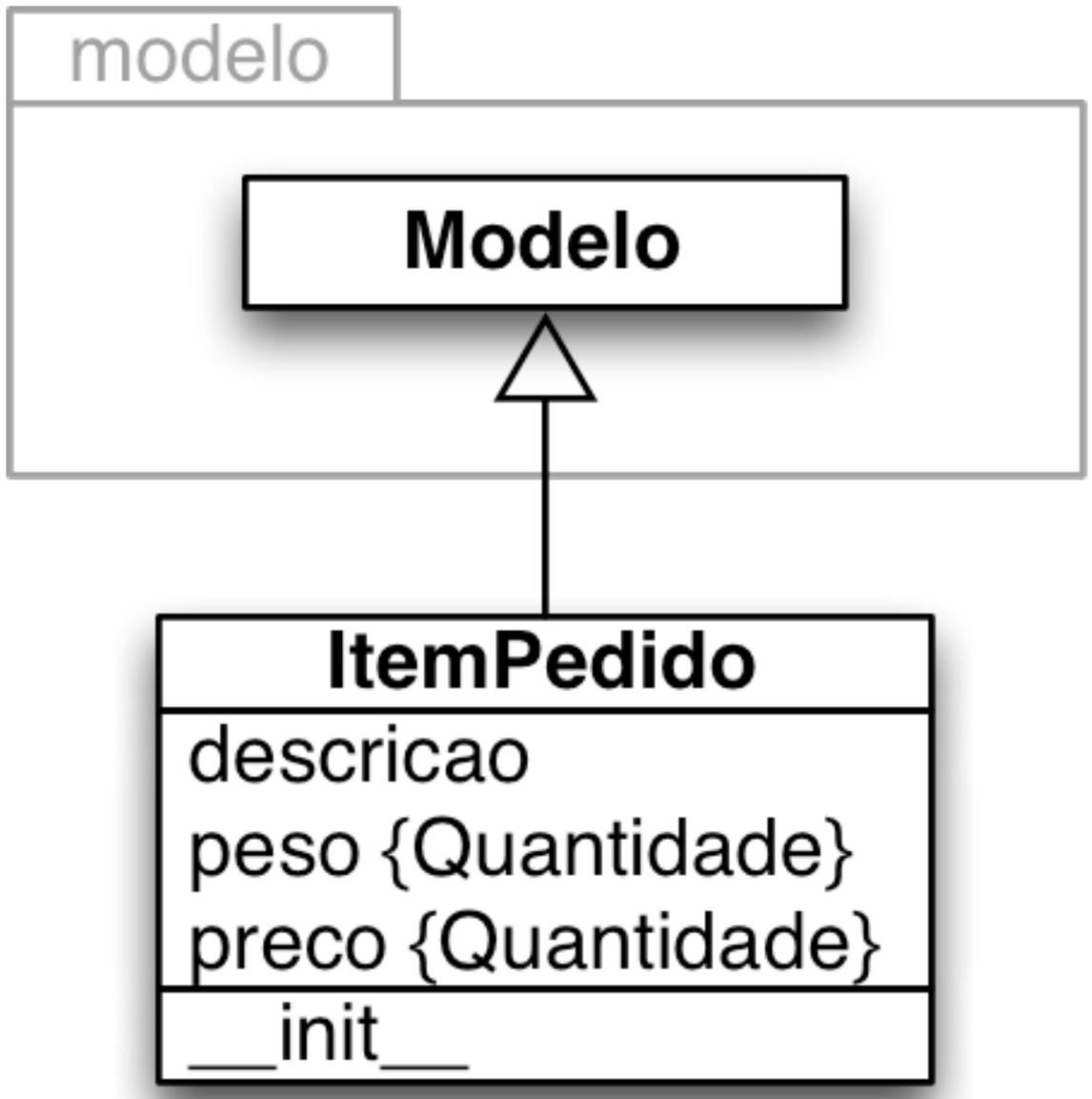
mesmo desempenho,
+ nomes amigáveis



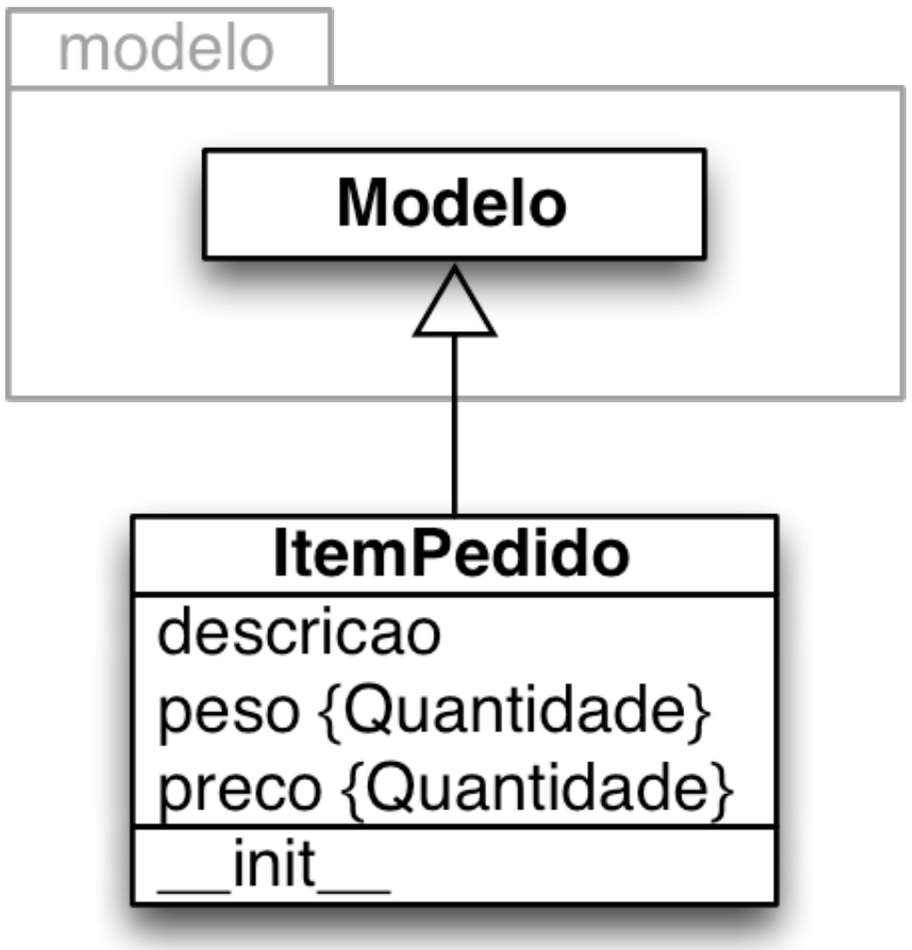
Número de instâncias de **ItemPedido** criadas por segundo (MacBook Pro 2011, Intel Core i7)

6

❻ simplicidade aparente



❻ o poder da abstração



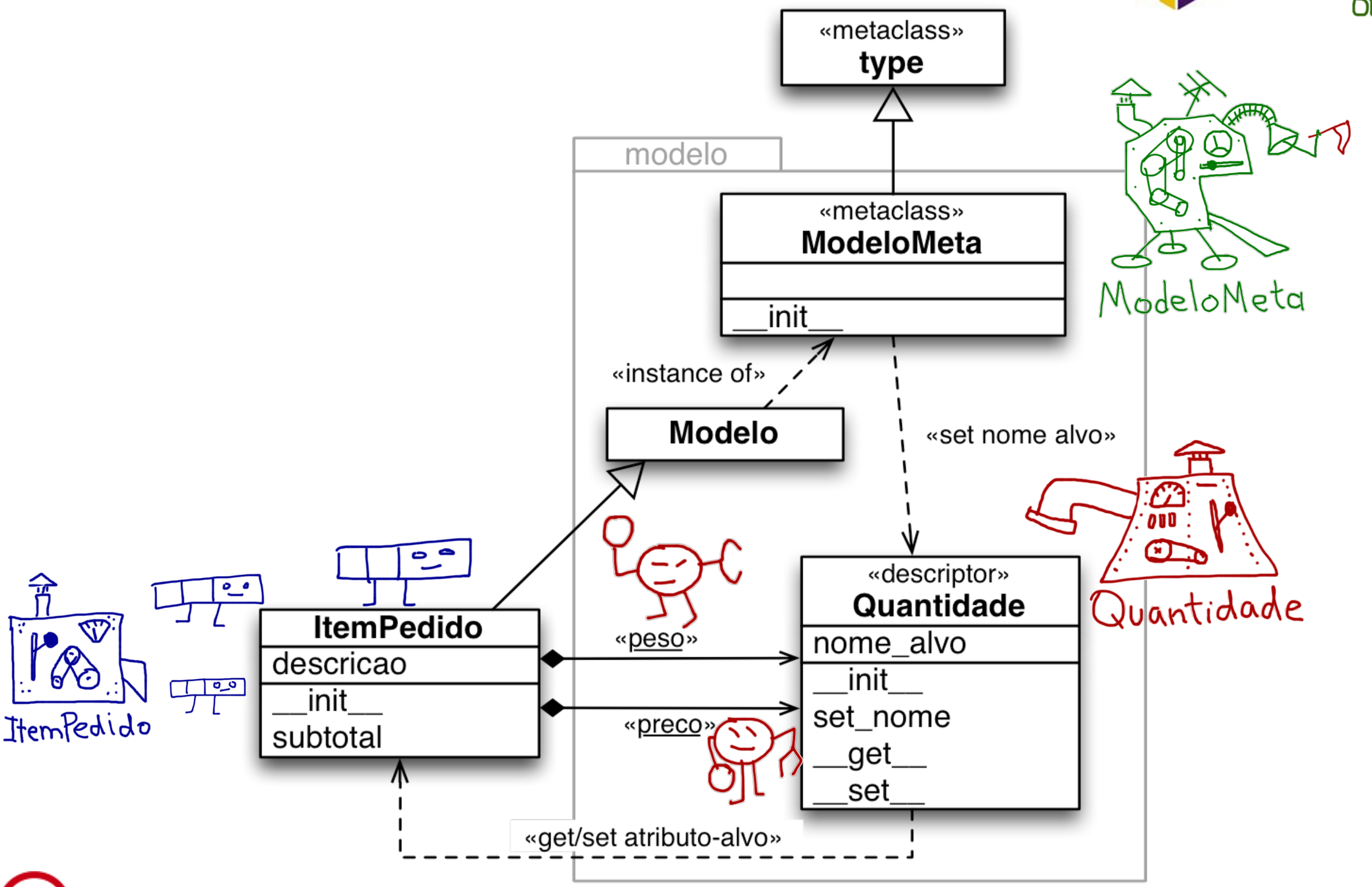

```
from modelo import Modelo, Quantidade

class ItemPedido(Modelo):

    peso = Quantidade()
    preco = Quantidade()

    def __init__(self, descricao, peso, preco):
        self.descricao = descricao
        self.peso = peso
        self.preco = preco
```

❻ módulo modelo.py

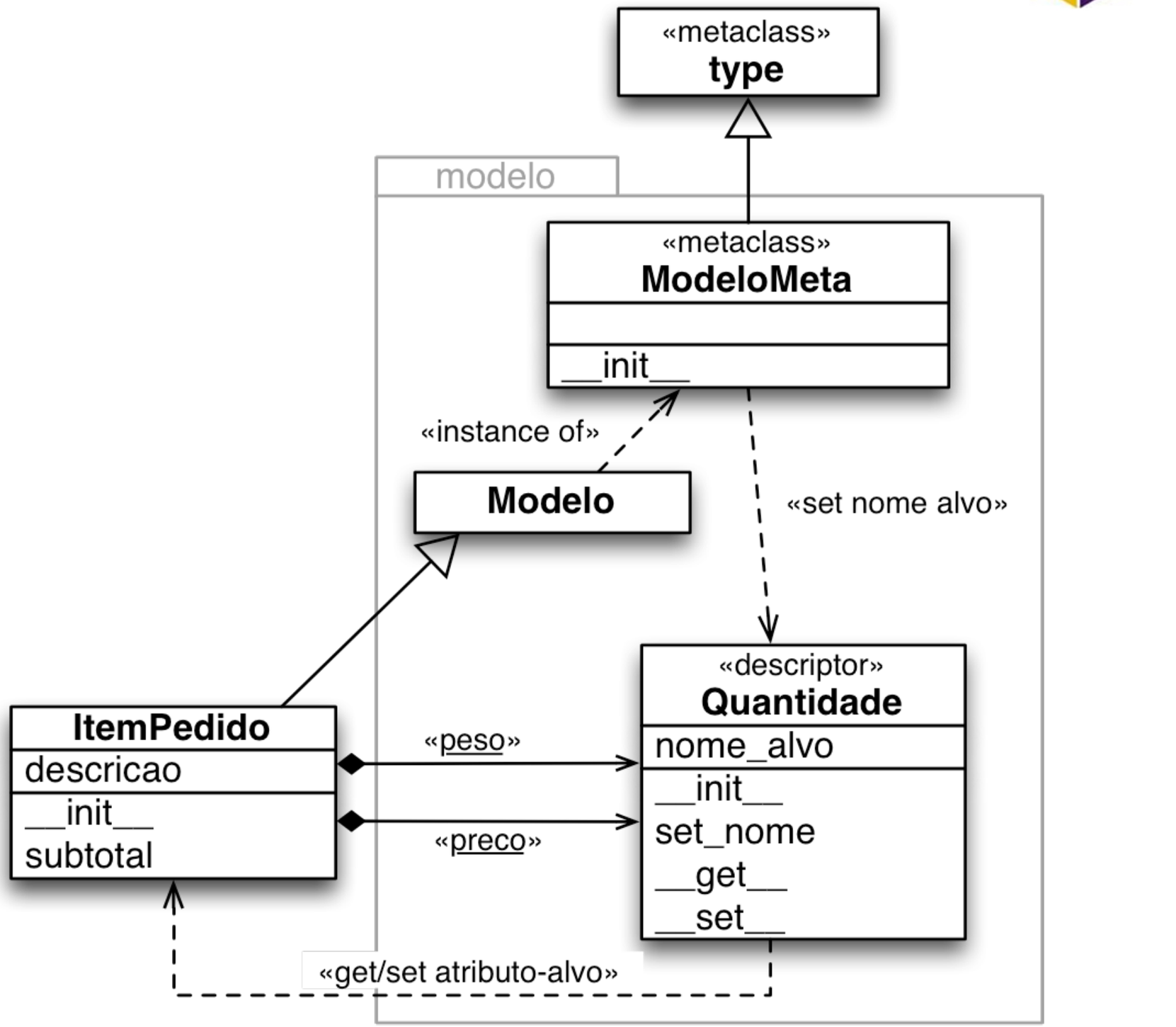
```
class Quantidade(object):

    def __init__(self):
        self.set_nome(self.__class__.__name__, id(self))

    def set_nome(self, prefix, key):
        self.nome_alvo = '%s_%s' % (prefix, key)

    def __get__(self, instance, owner):
        return getattr(instance, self.nome_alvo)

    def __set__(self, instance, value):
        if value > 0:
            setattr(instance, self.nome_alvo, value)
        else:
            raise ValueError('valor deve ser > 0')

class ModeloMeta(type):

    def __init__(cls, nome, bases, dic):
        super(ModeloMeta, cls).__init__(nome, bases, dic)
        for chave, atr in dic.items():
            if hasattr(atr, 'set_nome'):
                atr.set_nome('__' + nome, chave)

class Modelo(object):
    __metaclass__ = ModeloMeta
```

❻ esquema final

UNIFIED MODELING LANGUAGE™ + LTO LIGUAGEM TOSCA de OBJETOS

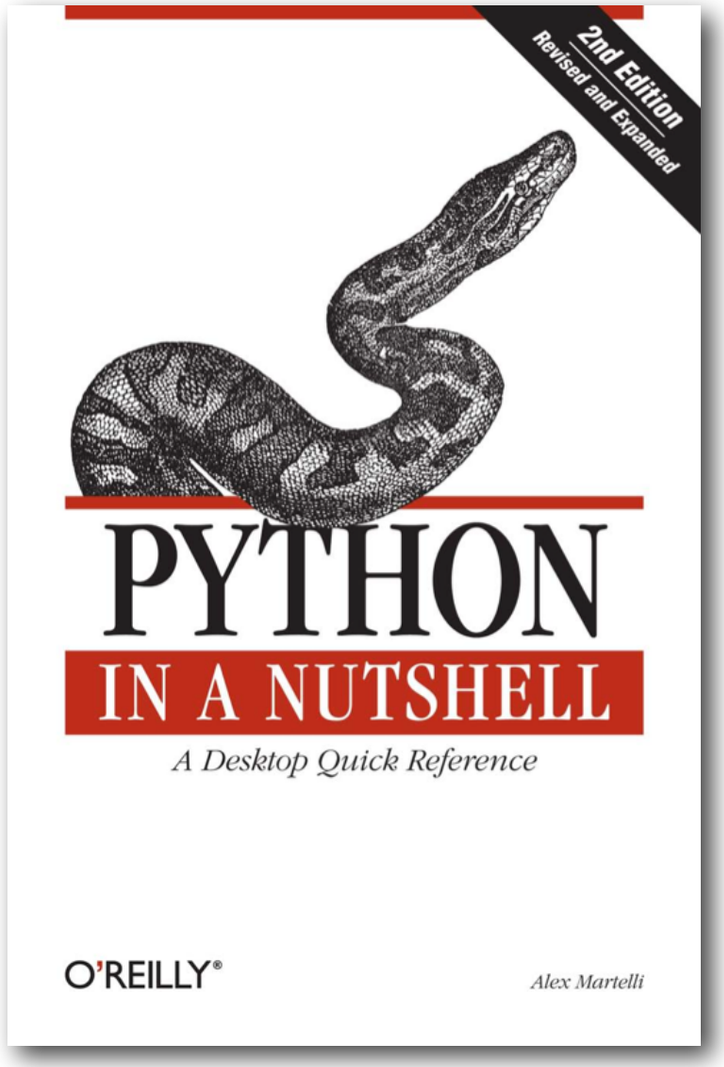


❻ esquema final



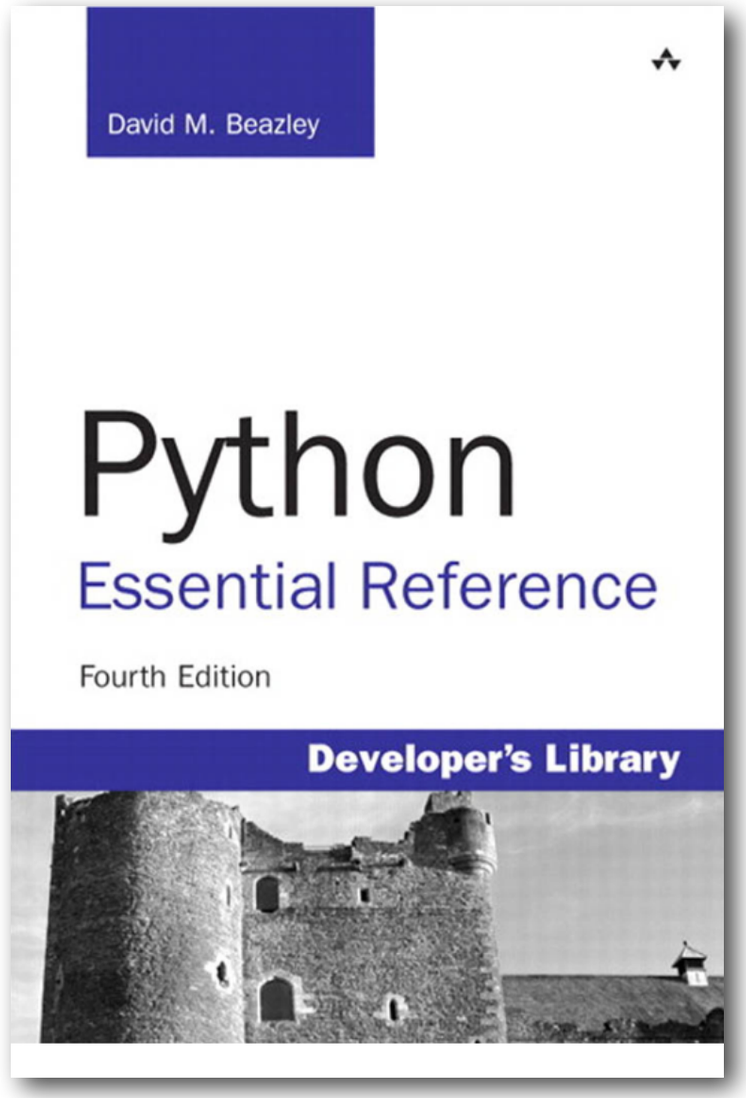
References

- Raymond Hettinger's
  Descriptor HowTo Guide
  (part of the official Python docs.)
- Alex Martelli's
  Python in a Nutshell, 2e.
  (Python 2.5 but still excellent, includes the complete attribute access algorithm)



References

- David Beazley's
  Python Essential Reference,
  4th edition
  (covers Python 2.6)



- Source code and doctests for this talk:
  http://github.com/ramalho/talks/tree/master/encap
  (will move to: https://github.com/oturing/encapy)