

Luciano Ramalho
luciano@ramalho.org



outubro/2012

Objetos Pythonicos

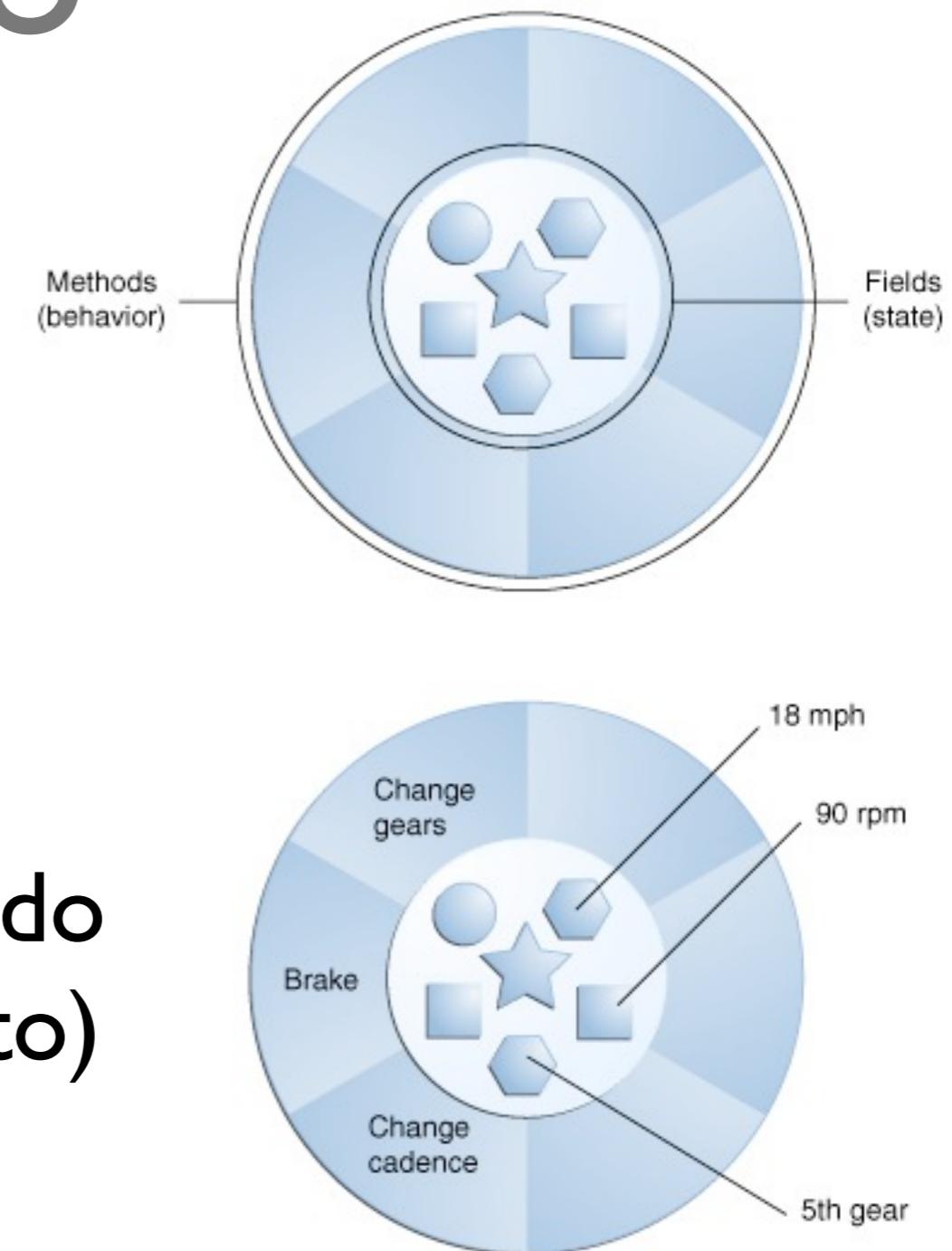
compacto

Orientação a objetos e padrões de projeto em Python

Objetos

Conceito: “objeto”

- Um componente de software que inclui dados (**campos**) e comportamentos (**métodos**)
- Em geral, os **campos** são manipulados pelos **métodos** do próprio objeto (encapsulamento)



Figuras: bycicle (bicicleta), The Java Tutorial

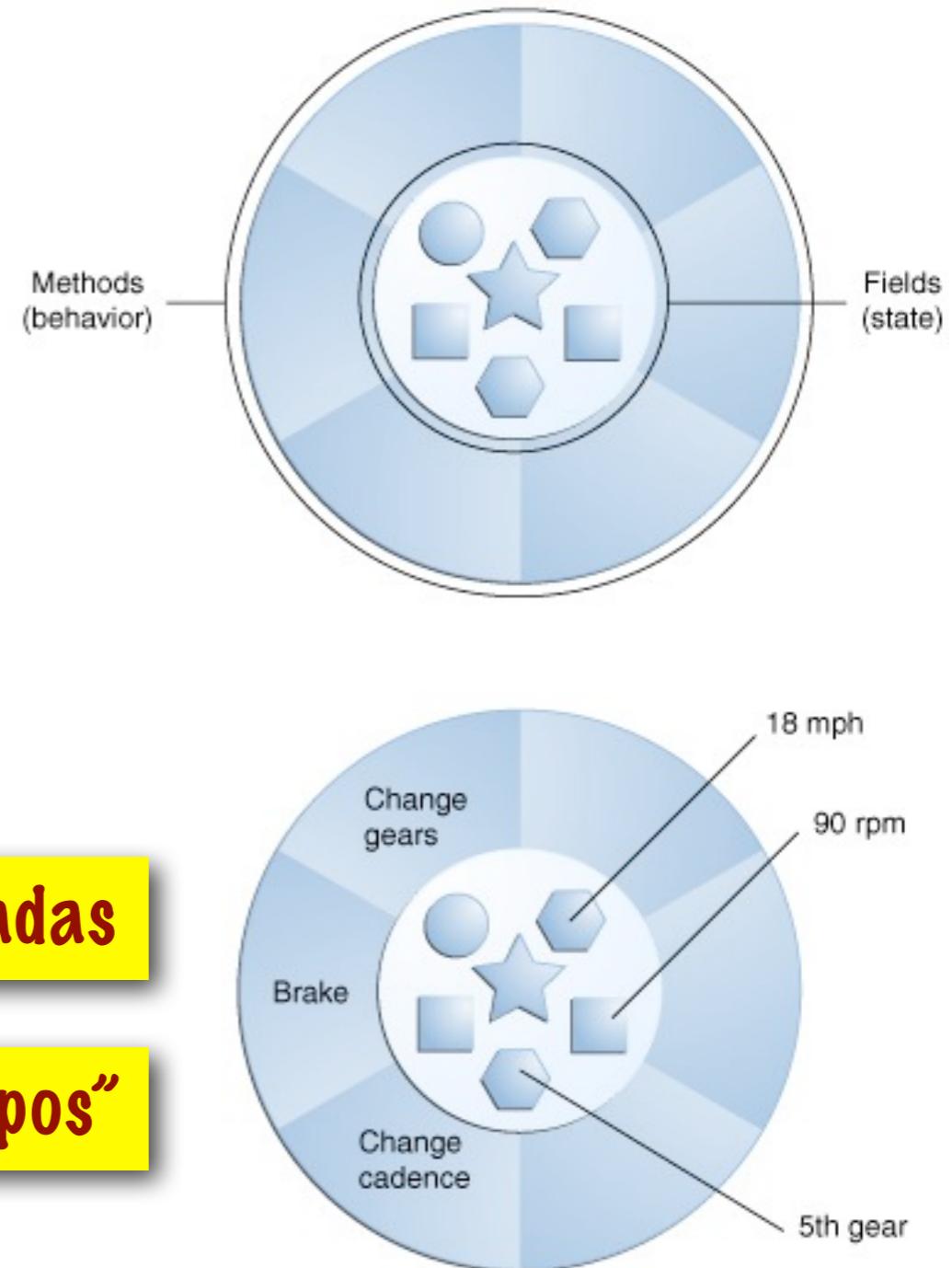
<http://docs.oracle.com/javase/tutorial/java/concepts/object.html>

Terminologia pythonica

- Objetos possuem **atributos**
- Os **atributos** de um objeto podem ser:
 - **métodos**
 - **atributos de dados**

funções vinculadas

"campos"



Figuras: bycicle (bicicleta), The Java Tutorial

<http://docs.oracle.com/javase/tutorial/java/concepts/object.html>

Exemplo: um objeto dict

```
>>> d = { 'AM': 'Manaus', 'PE': 'Recife', 'PR': 'Curitiba' }
>>> d.keys()
[ 'PR', 'AM', 'PE' ]
>>> d.get('PE')
'Recife'
>>> d.pop('PR')
'Curitiba'
>>> d
{ 'AM': 'Manaus', 'PE': 'Recife' }
>>> len(d)
2
>>> d.__len__()
2
```

- Métodos: `keys`, `get`, `pop`, `__len__` etc.

Exemplo: um objeto dict

- `dir` revela atributos de um `dict`
- atributos de dados e métodos

```
>>> dir(d)
['__class__', '__cmp__', '__contains__', '__delattr__', '__delitem__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__gt__', '__hash__', '__init__', '__iter__', '__le__',
'__len__', '__lt__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__setitem__',
'__sizeof__', '__str__', '__subclasshook__', 'clear', 'copy',
'fromkeys', 'get', 'has_key', 'items', 'iteritems', 'iterkeys',
'itervalues', 'keys', 'pop', 'popitem', 'setdefault', 'update',
'velues']
```

Exemplo: um objeto dict

- Sobrecarga de operadores:
- []: __getitem__, __setitem__

```
>>> d
{'AM': 'Manaus', 'PE': 'Recife'}
>>> d['AM']
'Manaus'
>>> d.__getitem__('AM')
'Manaus'
>>> d['MG'] = 'Belo Horizonte'
>>> d.__setitem__('RJ', 'Rio de Janeiro')
>>> d
{'MG': 'Belo Horizonte', 'AM': 'Manaus', 'RJ': 'Rio de Janeiro', 'PE': 'Recife'}
```

Exemplo: um objeto dict

- Atributos de dados: `__class__`, `__doc__`

```
>>> d.__class__
<type 'dict'>
>>> type(d)
<type 'dict'>
>>> print d.__doc__
dict() -> new empty dictionary.
dict(mapping) -> new dictionary initialized from a mapping object's
    (key, value) pairs.
dict(seq) -> new dictionary initialized as if via:
    d = {}
    for k, v in seq:
        d[k] = v
dict(**kwargs) -> new dictionary initialized with the name=value pairs
    in the keyword argument list.  For example:  dict(one=1, two=2)
```

Exemplo: outro dict

```
>>> d = dict()
>>> d.keys()
[]
>>> d.get('bla')
>>> print d.get('bla')
None
>>> d.get('spam')
>>> print d.get('spam')
None
>>> d.pop('ovos')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'ovos'
>>> d
{}
>>> len(d)
0
>>> d.__len__()
0
```

- **d = dict()**
é o mesmo que
d = {}

Exercício: construir e controlar Tkinter.Label

```
import Tkinter
rel = Tkinter.Label()
rel['text'] = '10:42:29'
rel.grid()
rel['font'] = 'Helvetica 120 bold'
```

```
from time import strftime
rel['text'] = strftime('%H:%M:%S')
```

```
def tic():
    rel['text'] = strftime('%H:%M:%S')
```

```
def tac():
    tic()
    rel.after(100, tac)
```

```
tac()
```



Para instalar
Tkinter no
Ubuntu Linux:

\$ sudo apt-get install python-tk

Tudo é um objeto

- Não existem “tipos primitivos” como em Java
 - desde Python 2.2, dezembro de 2001

```
>>> 5 + 3  
8  
>>> 5 .__add__(3)  
8  
>>> type(5)  
<type 'int'>
```

Funções são objetos

```
>>> def factorial(n):
...     '''devolve n!'''
...     return 1 if n < 2 else n * factorial(n-1)
...
>>> factorial(5)
120
>>> fat = factorial
>>> fat
<function factorial at 0x1004b5f50>
>>> fat(42)
1405006117752879898543142606244511569936384000000000L
>>> factorial.__doc__
'devolve n!'
>>> factorial.__name__
'factorial'
>>> factorial.__code__
<code object factorial at 0x1004b84e0, file "<stdin>", line 1>
>>> factorial.__code__.co_varnames
('n',)
```

Funções são objetos

```
>>> factorial.__code__.co_code
'|\x00\x00d\x01\x00j\x00\x00o\x05\x00\x01d\x02\x00S\x01|\x00\x00t\x00\x00|
\x00\x00d\x02\x00\x18\x83\x01\x00\x14S'
>>> from dis import dis
>>> dis(factorial.__code__.co_code)
  0  LOAD_FAST              0  (0)
  3  LOAD_CONST             1  (1)
  6  COMPARE_OP             0  (<)
  9  JUMP_IF_FALSE          5  (to 17)
 12 POP_TOP
 13 LOAD_CONST             2  (2)
 16 RETURN_VALUE
>> 17 POP_TOP
 18 LOAD_FAST              0  (0)
 21 LOAD_GLOBAL             0  (0)
 24 LOAD_FAST              0  (0)
 27 LOAD_CONST             2  (2)
 30 BINARY_SUBTRACT
 31 CALL_FUNCTION           1
 34 BINARY_MULTIPLY
 35 RETURN_VALUE
>>>
```

**Bytecode da
função factorial**

Tipos

Alguns tipos básicos

```
>>> i = 7  
>>> type(i)  
<type 'int'>  
>>> j = 2**100  
>>> type(j)  
<type 'long'>  
>>> f = 7.  
>>> type(f)  
<type 'float'>  
>>> s = 'abc'  
>>> type(s)  
<type 'str'>  
>>> u = u'abc'  
>>> type(u)  
<type 'unicode'>  
>>> b = b'ABC'  
>>> type(b)  
<type 'str'>  
>>> b[0]  
'A'
```

Python 2.7

int e long
unificados no
Python 3

str
X
unicode
X
bytes

Python 3.2

```
>>> i = 7  
>>> type(i)  
<class 'int'>  
>>> j = 2**100  
>>> type(j)  
<class 'int'>  
>>> f = 7.  
>>> type(f)  
<class 'float'>  
>>> s = 'abc'  
>>> type(s)  
<class 'str'>  
>>> u = u'abc'
```

SyntaxError
ou
<class 'str'>
(Python 3.3)

```
>>> b = b'ABC'  
>>> type(b)  
<class 'bytes'>  
>>> b[0]
```

Tipos embutidos (built-in)

- Implementados em C, por eficiência
- Métodos inalteráveis (como “final”, em Java)
- Texto: str, unicode (Python 2 e Python 3)
- Números: int, long (Python 2), float, complex, bool
- Coleções: list, tuple, dict, set, frozenset, bytes (Python 3)
- etc.

Tipagem forte

- O tipo de um objeto nunca muda, e não existem “tipos variantes”
- Raramente Python faz conversão automática

```
>>> a = 10
>>> b = '9'
>>> a + b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> a + int(b)
19
>>> str(a) + b
'109'
>>> 77 * None
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for *: 'int' and 'NoneType'
```

Tipagem dinâmica: variáveis não têm tipo

```
>>> def dobro(x):
...     return x * 2
...
>>> dobro(7)
14
>>> dobro(7.1)
14.2
>>> dobro('bom')
'bombom'
>>> dobro([10, 20, 30])
[10, 20, 30, 10, 20, 30]
>>> dobro(None)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "<stdin>", line 2, in dobro
      TypeError: unsupported operand type(s) for *: 'NoneType' and 'int'
```

Duck typing

- “Se voa como um pato, nada como um pato e grasna como um pato, é um pato.”
- Tipagem dinâmica permite duck typing (tipagem pato): estilo de programação que evita verificar os tipos dos objetos, mas apenas seus métodos
- No exemplo anterior, a função `dobro` funciona com qualquer objeto `x` que consiga fazer `x * 2`
 - `x` implementa o método `__mult__(n)`, para `n` inteiro

Mutabilidade

- Em Python, alguns objetos são mutáveis, outros são imutáveis
- Objetos mutáveis possuem conteúdo ou estado interno (atributos de dados) que podem ser alterados ao longo da sua existência
- Objetos imutáveis não podem ser alterados de nenhuma maneira. Seu estado é congelado no momento da inicialização.

Mutabilidade: exemplos

Mutáveis

- list
- dict
- set
- objetos que permitem a alteração de atributos por acesso direto, setters ou outros métodos

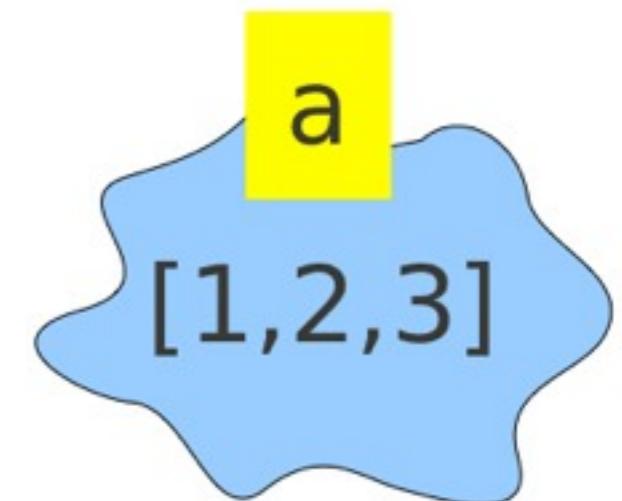
Imutáveis

- tuple
- str, unicode, bytes
- frozenset
- int, float, complex

Variáveis e referências

Variáveis não são “caixas”!

- Abandone essa idéia!
 - Ela é válida em C e Pascal, mas não em Python ou Java
- Variáveis são rótulos (ou post-its) que identificam objetos
- O objeto identificado já existe quando o rótulo é atribuído



Objetos e variáveis

- Objetos existem antes das variáveis
- Em uma atribuição, o lado direito do = é executado primeiro. Prova:

Primeiro o objeto
10 é criado,
depois a variável
a é atribuída a ele

A variável y
nunca chega a
ser criada

```
>>> x = 10
>>> x
10
>>> y = 7 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'y' is not defined
>>>
```

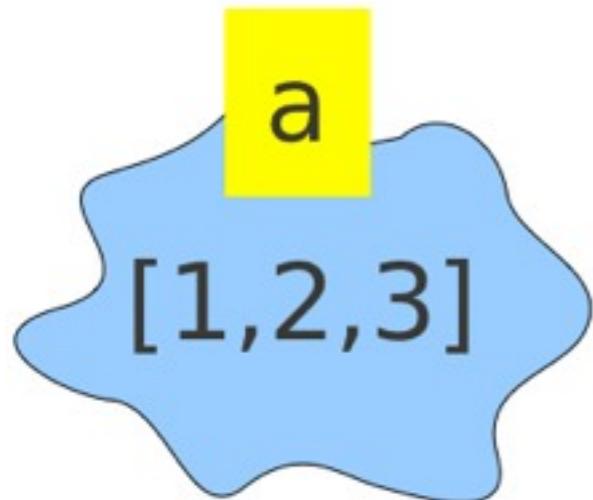
Referências

- Em Python, as variáveis não “contém” objetos, apenas **referências** para objetos
- Isso significa que duas variáveis podem apontar para o mesmo objeto
 - Por isso dá certo pensar em variáveis como “rótulos”
 - O mesmo objeto pode ter dois rótulos

Atribuição

- Atribuição nunca faz cópias de objetos!
 - apenas associa rótulos
 - ou muda os rótulos de lugar

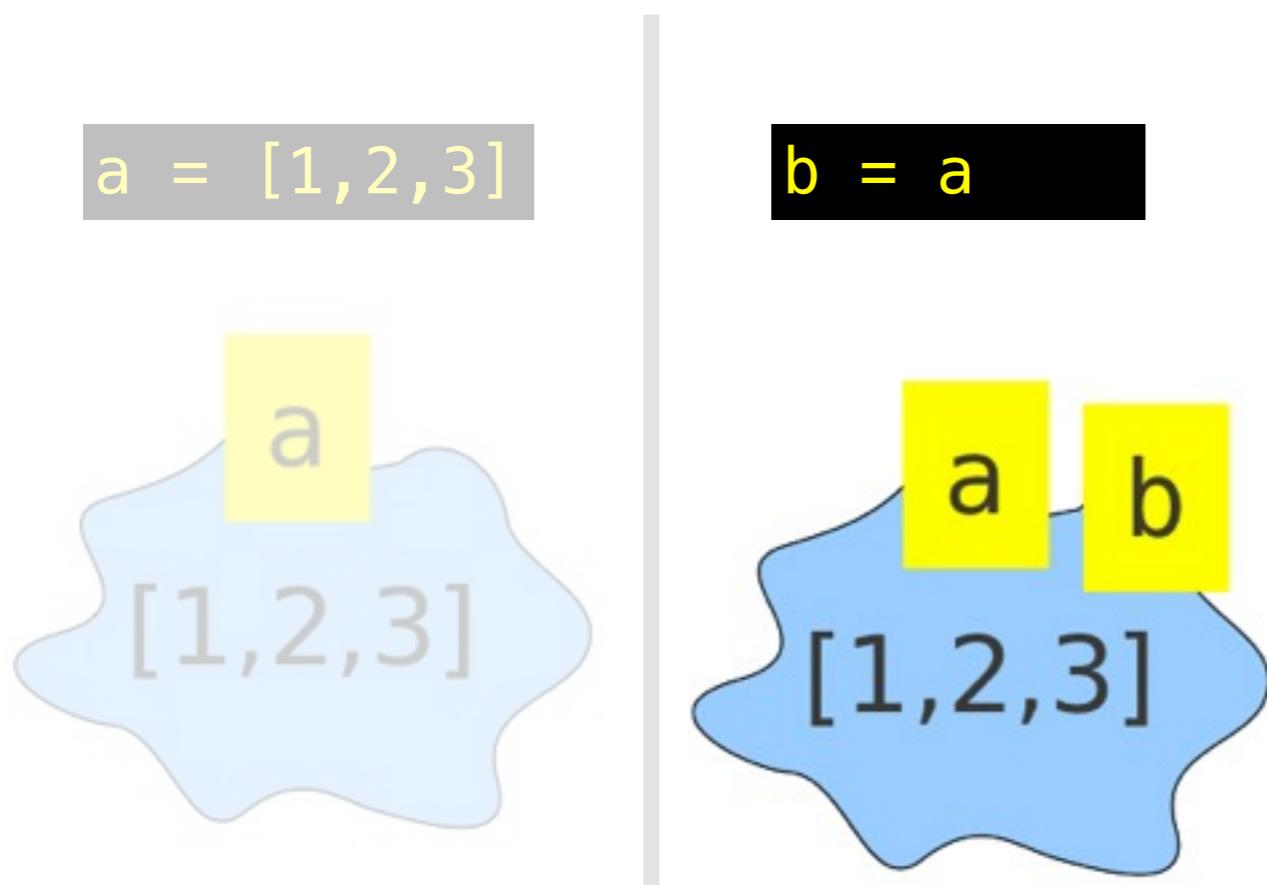
a = [1,2,3]



atribuição da
variável
a
ao objeto
[1, 2, 3]

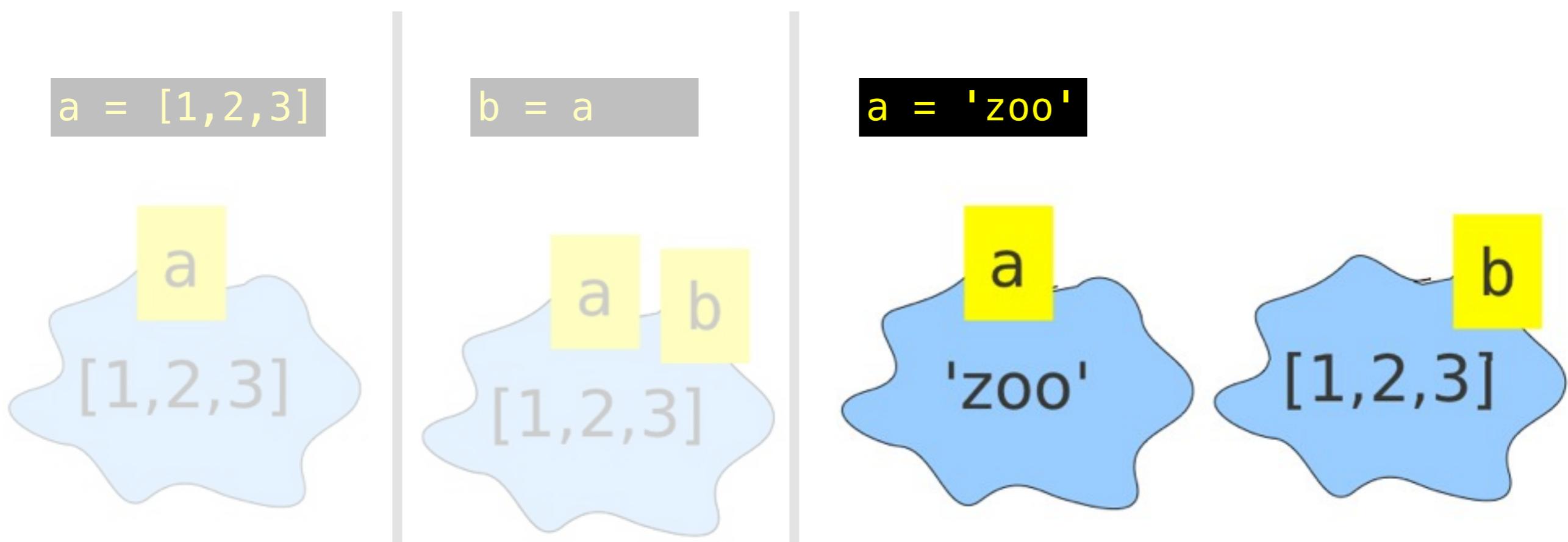
Atribuição

- Atribuição nunca faz cópias de objetos!
 - apenas associa rótulos
 - ou muda os rótulos de lugar

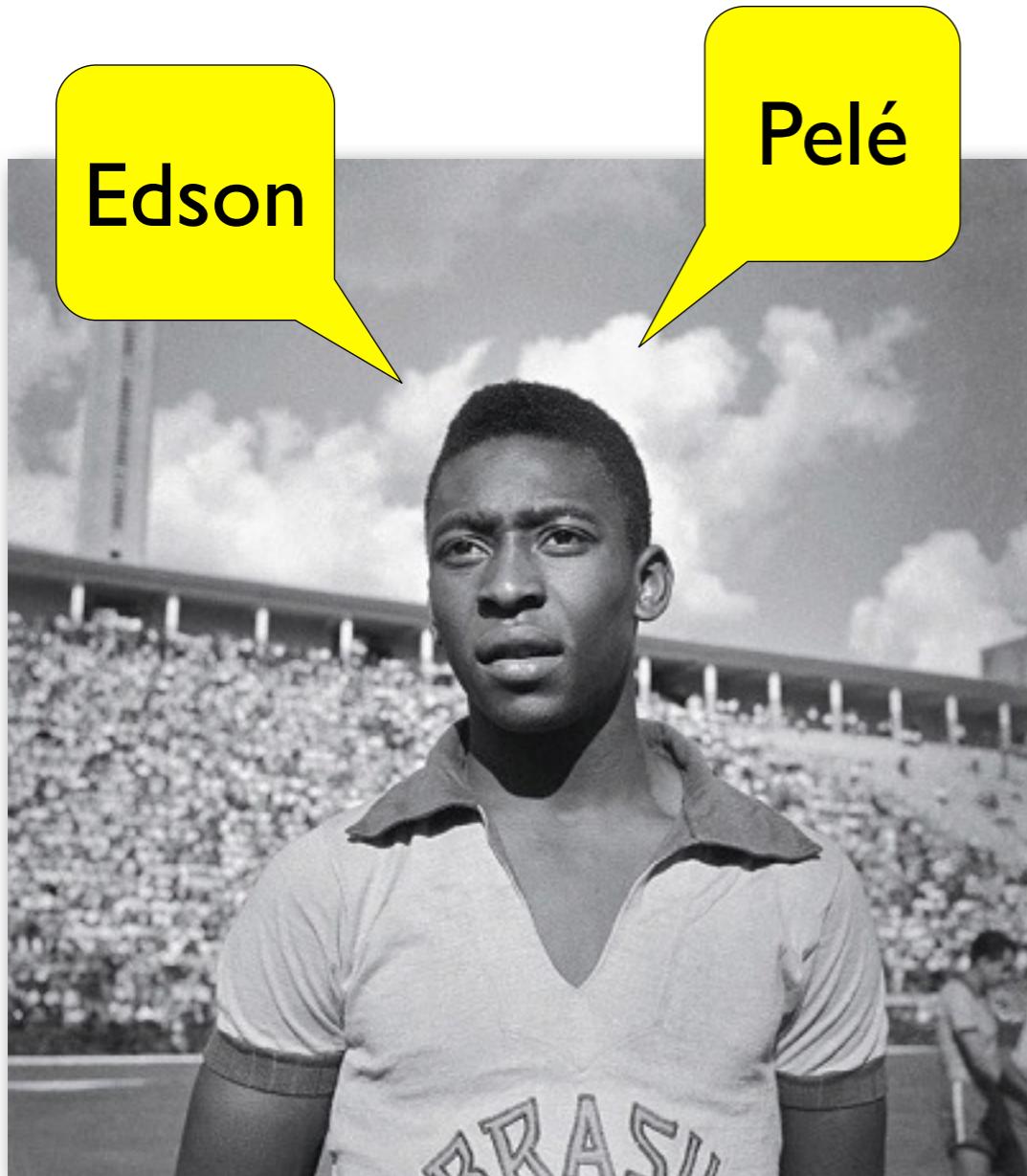


Atribuição

- Atribuição nunca faz cópias de objetos!
 - apenas associa rótulos
 - ou muda os rótulos de lugar



Aliasing (“apelidamento”)



- Uma pessoa pode ser chamada por dois ou mais nomes diferentes
- Um objeto pode ser referenciado através de duas ou mais variáveis diferentes

Aliasing: demonstração

```
>>> a = [21, 52, 73]
>>> b = a
>>> c = a[:]
>>> b is a
True
>>> c is a
False
>>> b == a
True
>>> c == a
True
>>> a, b, c
([21, 52, 73], [21, 52, 73], [21, 52, 73])
>>> b[1] = 999
>>> a, b, c
([21, 999, 73], [21, 999, 73], [21, 52, 73])
```

Identidade e igualdade

- Use **`id(o)`** para ver a identidade de um objeto
- Duas variáveis podem apontar para **o mesmo** objeto
- Neste caso, seus valores são **idênticos**
 - **`a is b`** → True
- Duas variáveis podem apontar para objetos **distintos** com conteúdos **iguais**
 - **`a == b`** → True

Comparando com Java

- Em Java o operador que compara referências é `==`
 - também usado para os tipos primitivos (int etc.)
- Em Python, comparação de referências é com `is`
 - Este operador não pode ser sobreescrito

Comparando com Java (2)

- Em Java, igualdade entre objetos é testada com o método `.equals()`
 - `.equals()` é um método, portanto `x.equals(y)` não funciona se `x` é null
- Em Python, usamos o operador `==`
 - O operador `==` pode ser sobreescarregado em qualquer classe, redefinindo o método `__eq__`
 - Em Python, `None` é um objeto e suporta `==`
 - Mas o teste `x is None` é mais eficiente

Programação orientada a classes

Conceito: “classe”

- Uma categoria, ou tipo, de objeto
 - Uma idéia abstrata, uma forma platônica
- Exemplo: classe “Cão”:
 - Eu digo: “Ontem eu comprei um cão”
 - Você não sabe exatamente qual cão, mas sabe:
 - é um mamífero, quadrúpede, carnívoro
 - pode ser domesticado (normalmente)
 - cabe em um automóvel

Exemplar de cão: instância da classe Cao



```
>>> rex = Cao()
```

instanciação

Classe Cão

```
class Mamifero(object):
    """Lição de casa: implementar"""

class Cão(Mamifero):
    qt_patas = 4
    carnivoro = True
    nervoso = False
    def __init__(self, nome):
        self.nome = nome
    def latir(self, vezes=1):
        # quando nervoso, late o dobro
        vezes = vezes + (self.nervoso * vezes)
        print self.nome + ':' + ' Au!' * vezes
    def __str__(self):
        return self.nome
    def __repr__(self):
        return 'Cão({0!r})'.format(self.nome)
```

instanciação

```
>>> rex = Cão('Rex')
>>> rex
Cão('Rex')
>>> print rex
Rex
>>> rex.qt_patas
4
>>> rex.latir()
Rex: Au!
>>> rex.latir(2)
Rex: Au! Au!
>>> rex.nervoso = True
>>> rex.latir(3)
Rex: Au! Au! Au! Au! Au!
```



<https://github.com/oopy/exemplos/cao.py>

Como atributos são acessados

- Ao buscar **o.a** (atributo **a** do objeto **o** da classe **C**), o interpretador Python faz o seguinte:
 - 1) acessa atributo **a** da instância **o**; caso não exista...
 - 2) acessa atributo **a** da classe **C** de **o** (`type(o)` ou `o.__class__`); caso não exista...
 - 3) busca o atributo **a** nas superclasses de **C**, conforme a MRO (method resolution order)

Classe Cão em Python

```
class Mamifero(object):
    """lição de casa: implementar"""

class Cao(Mamifero):
    qt_patas = 4
    carnivoro = True
    nervoso = False
    def __init__(self, nome):
        self.nome = nome
    def latir(self, vezes=1):
        # quando nervoso, late o dobro
        vezes = vezes + (self.nervoso * vezes)
        print self.nome + ' ' + 'Au!' * vezes
    def __str__(self):
        return self.nome
    def __repr__(self):
        return 'Cao({0!r})'.format(self.nome)
```

`__init__` é o construtor, ou melhor, o inicializador

`self` é o 1º parâmetro formal em todos os métodos de instância



[opy/exemplos/cao.py](https://github.com/opy/exemplos/cao.py)

Classe Cão

```
class Mamifero(object):
    """lição de casa: implementar"""

class Cao(Mamifero):
    qt_patas = 4
    carnivoro = True
    nervoso = False
    def __init__(self, nome):
        self.nome = nome
    def latir(self, vezes=1):
        # quando nervoso, late o dobro
        vezes = vezes + (self.nervoso * vezes)
        print self.nome + ':' + ' Au!' * vezes
    def __str__(self):
        return self.nome
    def __repr__(self):
        return 'Cao({0!r})'.format(self.nome)
```

invocação

```
>>> rex = Cao('Rex')
>>> rex
Cao('Rex')
>>> print rex
Rex
>>> rex.qt_patas
4
>>> rex.latir()
Rex: Au!
>>> rex.latir(2)
Rex•Au! Au!
```

na invocação do método, a instância é passada automaticamente na posição do self



<https://github.com/oopy/exemplos/cao.py>



Classe Cão em Python

```
class Mamifero(object):
    """lição de casa: implementar"""

class Cao(Mamifero):
    qt_patas = 4
    carnivoro = True
    nervoso = False
    def __init__(self, nome):
        self.nome = nome
    def latir(self, vezes=1):
        # quando nervoso, late o dobro
        vezes = vezes + (self.nervoso * vezes)
        print self.nome + ':' + ' Au!' * vezes
    def __str__(self):
        return self.nome
    def __repr__(self):
        return 'Cao({0!r})'.format(self.nome)
```

• atributos de dados na classe funcionam como valores default para os atributos das instâncias

• atributos da instância só podem ser acessados via self



<https://github.com/oficinas-turing/exemplos/cao.py>



Exemplo: tômbola

- Sortear um a um todos os itens de uma coleção finita, sem repetir
- A mesma lógica é usada em sistemas para gerenciar banners online



Interface da tômbola

- Carregar itens
- Misturar itens
- Sortear um item
- Indicar se há mais itens



Projeto da tômbola

Tômbola
itens
carregar
carregada
misturar
sortear

- UML:
diagrama de classe



TDD: Test Driven Design

- Metodologia de desenvolvimento iterativa na qual, para cada funcionalidade nova, um teste é criado antes do código a ser implementado
- Esta inversão ajuda o programador a desenvolver com disciplina apenas uma funcionalidade de cada vez, mantendo o foco no teste que precisa passar
- Cada iteração de teste/implementação deve ser pequena e simples: “baby steps” (passinhos de bebê)

Doctests

- Um dos módulos para fazer testes automatizados na biblioteca padrão de Python
 - o outro módulo é o unittest, da família xUnit
- Doctests foram criados para testar exemplos embutidos na documentação
- Exemplo de uso:

```
$ python -m doctest cao.rst
```



oopy/exemplos/cao.rst



Coding Dojo

- Implementação da classe Tombola, com testes feitos em Doctest

Implementação da tômbola

```
# coding: utf-8

import random

class Tombola(object):
    def __init__(self, itens=None):
        self.itens = list(itens) if itens else []

    def carregar(self, itens):
        self.itens.extend(itens)

    def carregada(self):
        return bool(self.itens)

    def misturar(self):
        random.shuffle(self.itens)

    def sortear(self):
        return self.itens.pop()
```

Tômbola
itens
carregar
carregada
misturar
sortear

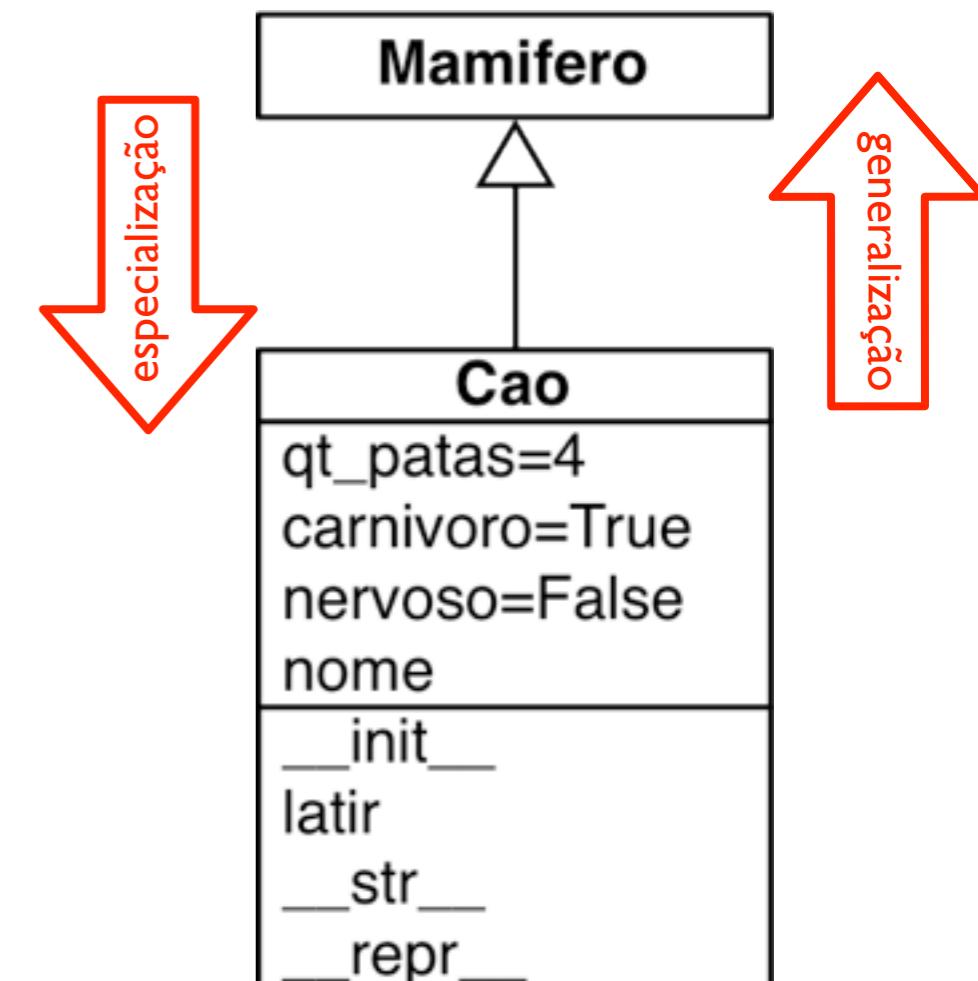
Herança

Mamifero: superclasse de Cao

```
class Mamifero(object):
    """lição de casa: implementar"""

class Cao(Mamifero):
    qt_patas = 4
    carnivoro = True
    nervoso = False
    def __init__(self, nome):
        self.nome = nome
    def latir(self, vezes=1):
        # quando nervoso, late o dobro
        vezes = vezes + (self.nervoso * vezes)
        print self.nome + ':' + ' Au!' * vezes
    def __str__(self):
        return self.nome
    def __repr__(self):
        return 'Cao({0!r})'.format(self.nome)
```

UML
diagrama de classe



<https://github.com/exemplos/cao.py>

Subclasses de Cão

- Continuação de cao.py

```
class Pequines(Cao):  
    nervoso = True  
  
class Mastiff(Cao):  
    def latir(self, vezes=1):  
        # o mastiff não muda seu latido  
        print self.nome + ':' + ' Wuff!'  
  
class SaoBernardo(Cao):  
    def __init__(self, nome):  
        Cao.__init__(self, nome)  
        self.doses = 10  
    def servir(self):  
        if self.doses == 0:  
            raise ValueError('Acabou o conhaque!')  
        self.doses -= 1  
        msg = '{0} serve o conhaque (restam {1} doses)'  
        print msg.format(self.nome, self.doses)
```



Diz a lenda que o cão São Bernardo leva um pequeno barril de conhaque para resgatar viajantes perdidos na neve.

Subclasses de Cão

```
class Pequines(Cao):
    nervoso = True

class Mastiff(Cao):
    def latir(self, vezes=1):
        # o mastiff não muda seu latido quando nervoso
        print self.nome + ':' + ' Wuff!' * vezes

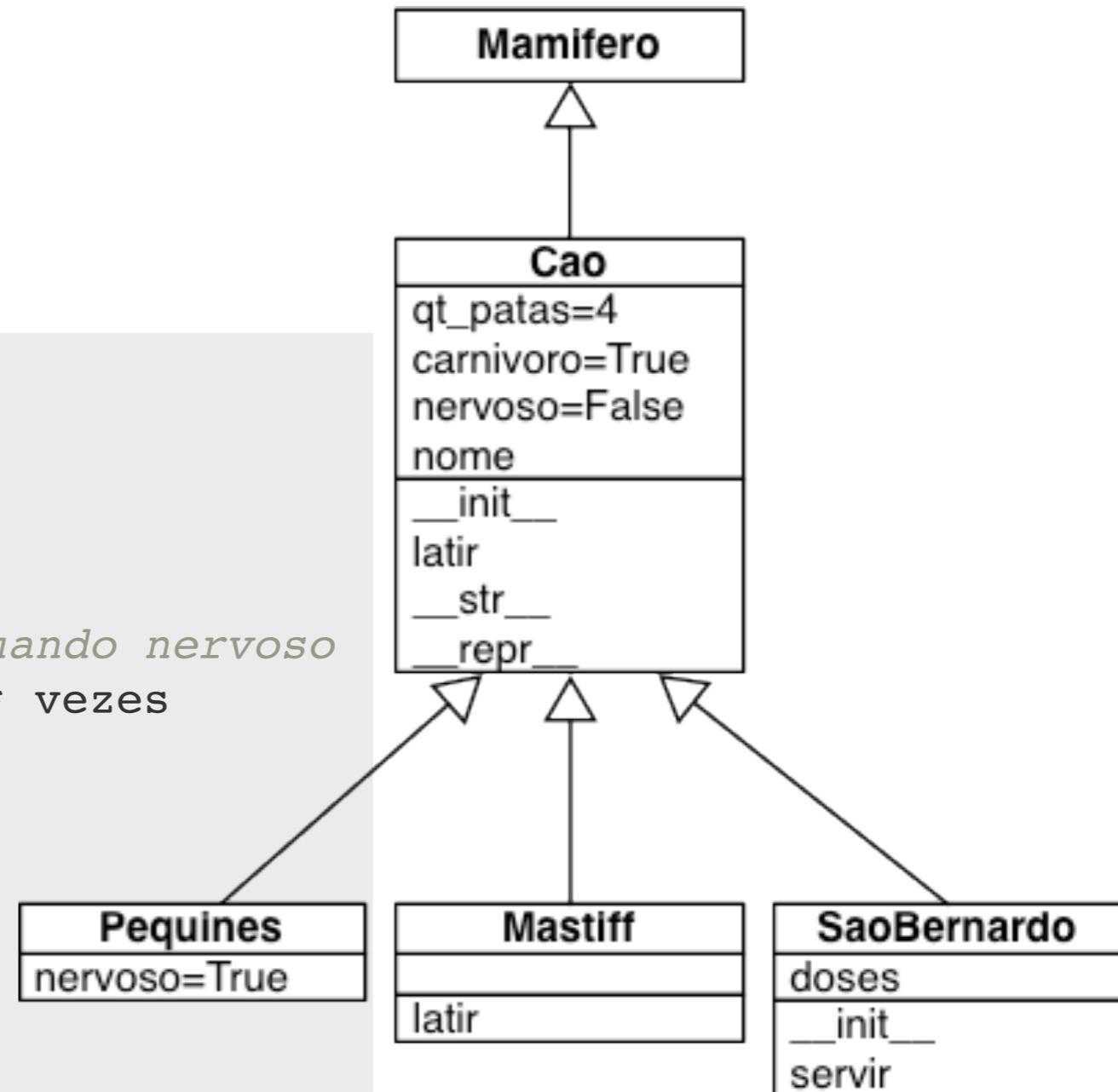
class SaoBernardo(Cao):
    def __init__(self, nome):
        Cao.__init__(self, nome)
        self.doses = 10
    def servir(self):
        if self.doses == 0:
            raise ValueError('Acabou o conhaque!')
        self.doses -= 1
        msg = '{0} serve o conhaque (restam {1} doses)'
        print msg.format(self.nome, self.doses)
```

```
>>> sansao = SaoBernardo('Sansao')
>>> sansao.servir()
Sansao serve o conhaque (restam 9 doses)
>>> sansao.doses = 1
>>> sansao.servir()
Sansao serve o conhaque (restam 0 doses)
>>> sansao.servir()
Traceback (most recent call last):
...
ValueError: Acabou o conhaque!
```

Subclasses de Cao

- Continuação
de cao.py

```
class Pequines(Cao):  
    nervoso = True  
  
class Mastiff(Cao):  
    def latir(self, vezes=1):  
        # o mastiff não muda seu latido quando nervoso  
        print self.nome + ':' + ' Wuff!' * vezes  
  
class SaoBernardo(Cao):  
    def __init__(self, nome):  
        Cao.__init__(self, nome)  
        self.doses = 10  
    def servir(self):  
        if self.doses == 0:  
            raise ValueError('Acabou o conhaque!')  
        self.doses -= 1  
        msg = '{0} serve o conhaque (restam {1} doses)'  
        print msg.format(self.nome, self.doses)
```



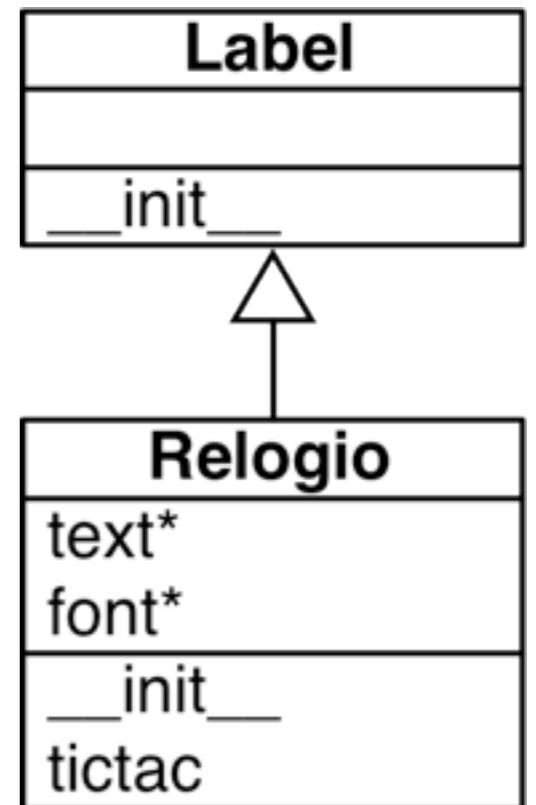
Relógio com classe

```
import Tkinter
from time import strftime

class Relogio(Tkinter.Label):
    def __init__(self):
        Tkinter.Label.__init__(self)
        self.pack()
        self['text'] = strftime('%H:%M:%S')
        self['font'] = 'Helvetica 120 bold'
        self.tictac()

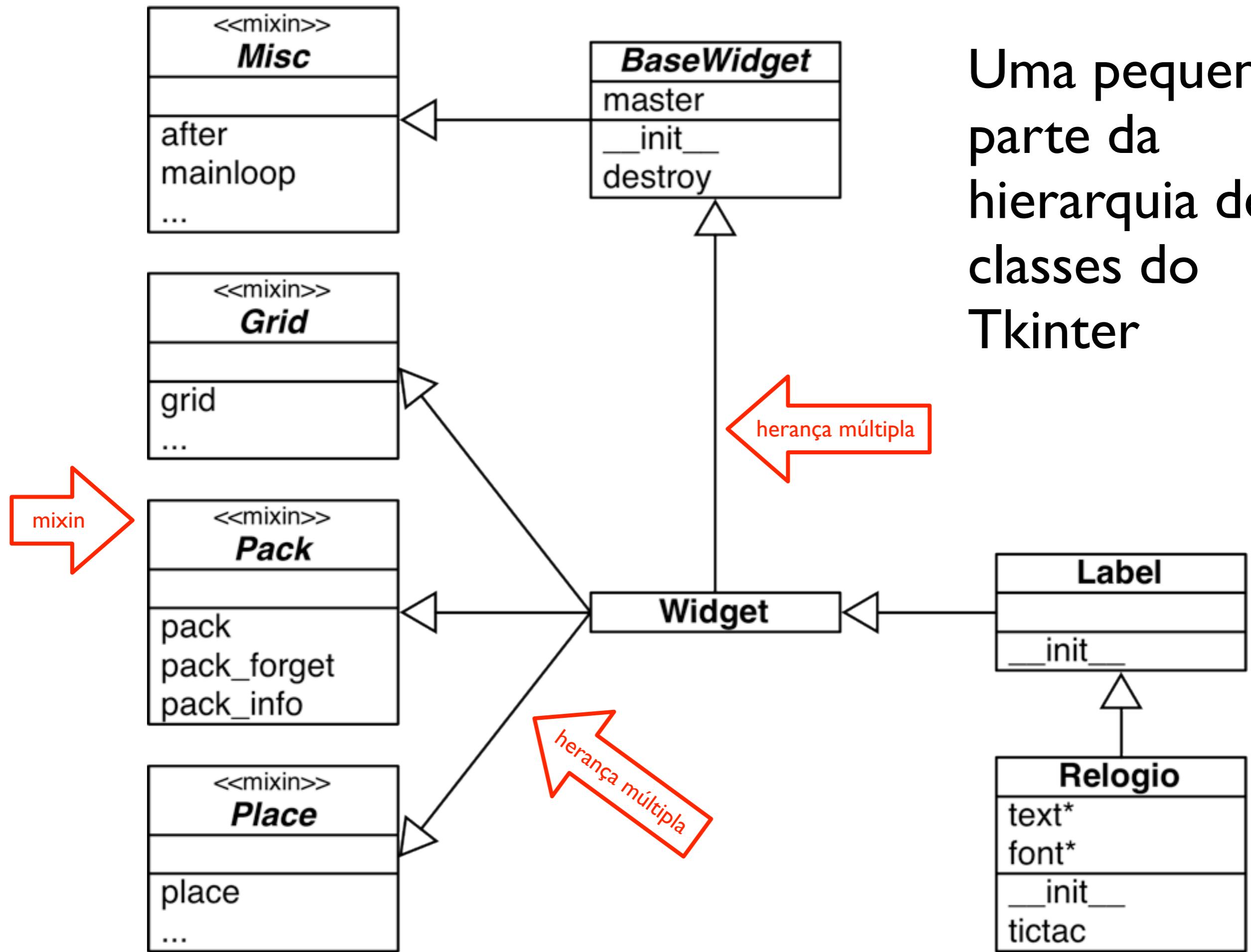
    def tictac(self):
        agora = strftime('%H:%M:%S')
        if agora != self['text']:
            self['text'] = agora
            self.after(100, self.tictac)

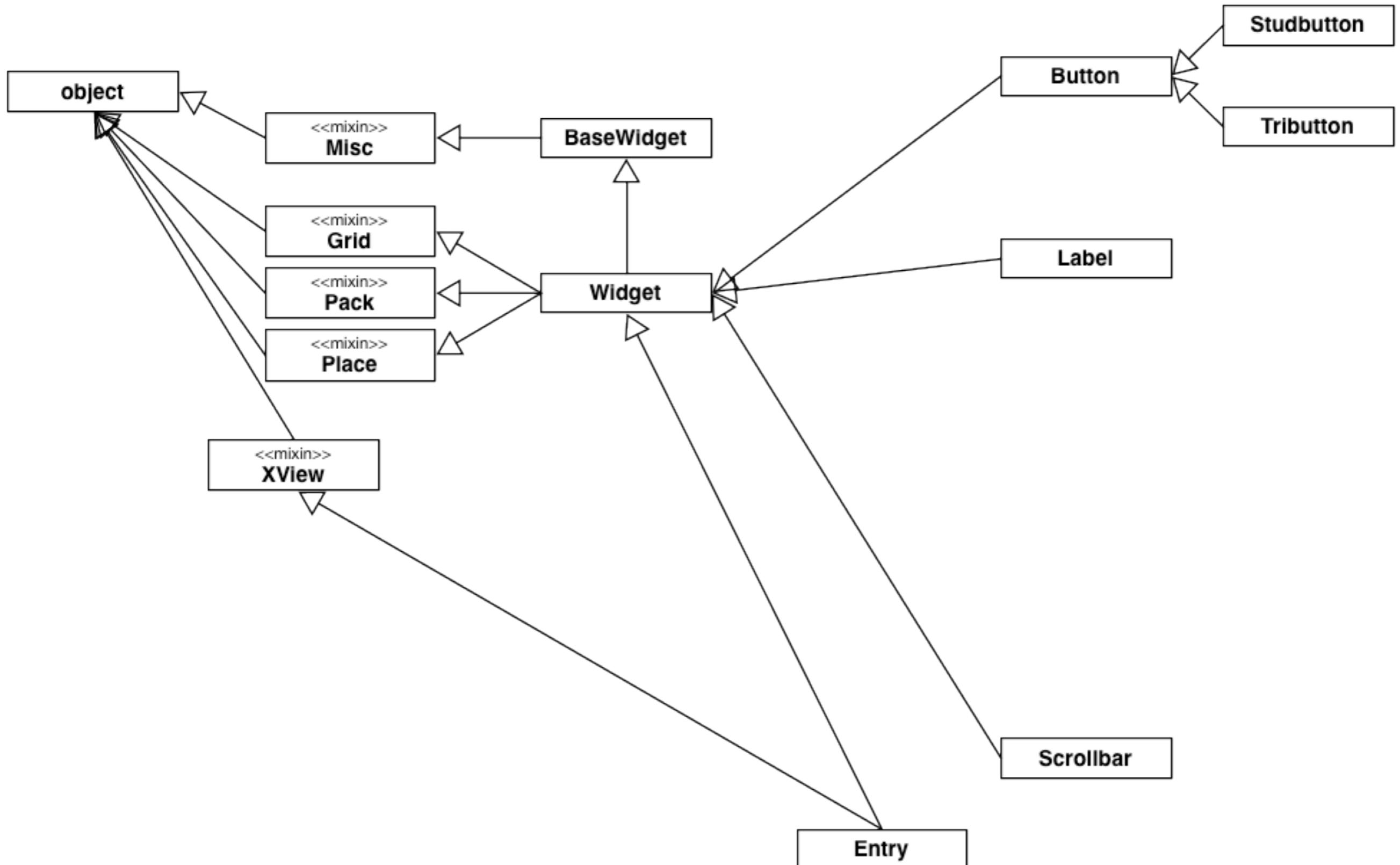
rel = Relogio()
rel.mainloop()
```



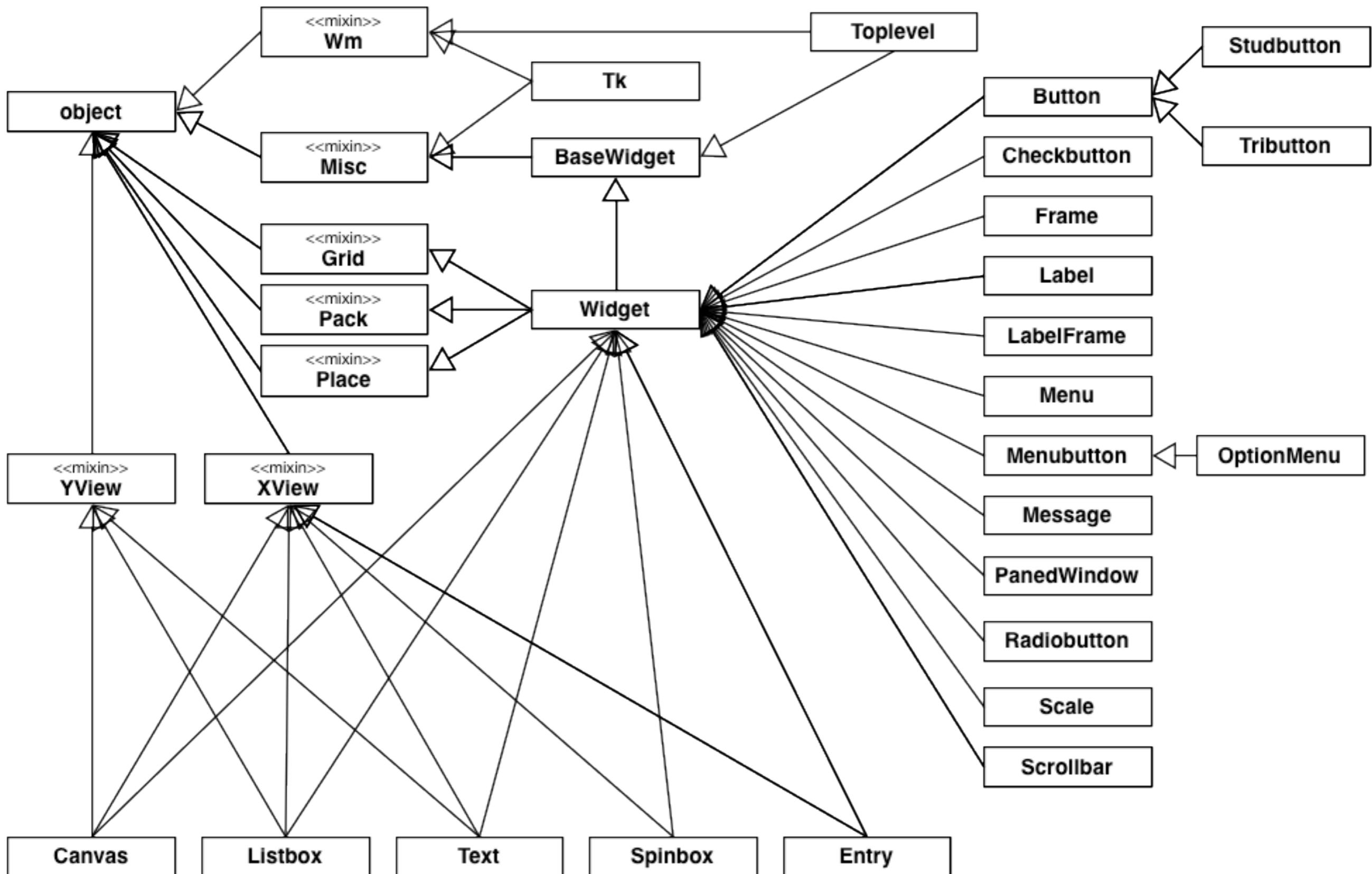
[git](#) [copy/exemplos/relogio_oo.py](#)

Uma pequena parte da hierarquia de classes do Tkinter





Um pouco mais da hierarquia de classes do Tkinter



Hierarquia de classes dos objetos gráficos do Tkinter

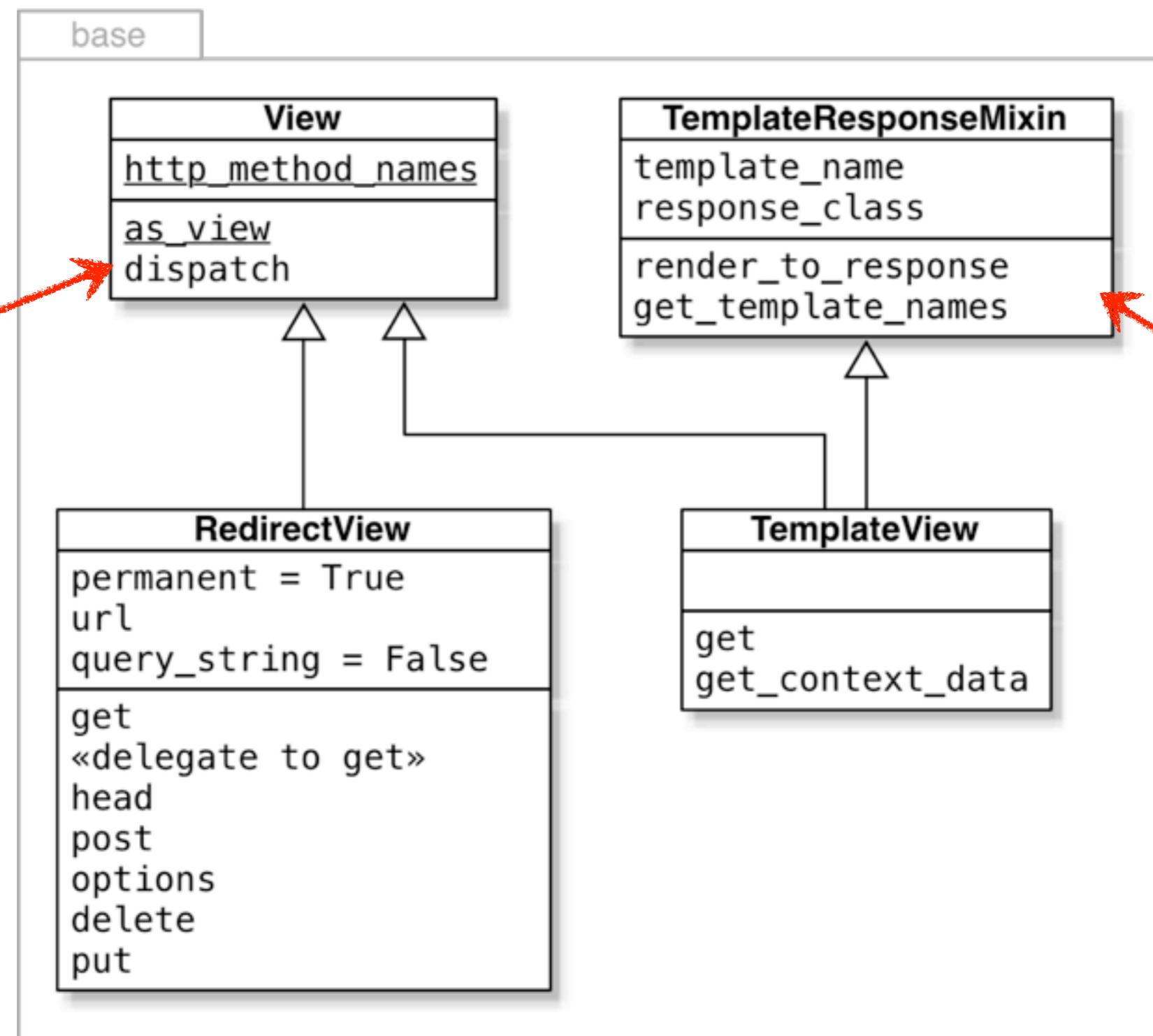
Exemplo de herança múltipla no Django

- Class-based views (CBV): classes para a construção de views, desde o Django 1.3
- Divisão de tarefas para a construção modular de views, diminuindo código repetitivo
- Vamos explorar views básicas e list/detail

API navegável: <http://ccbv.co.uk/>

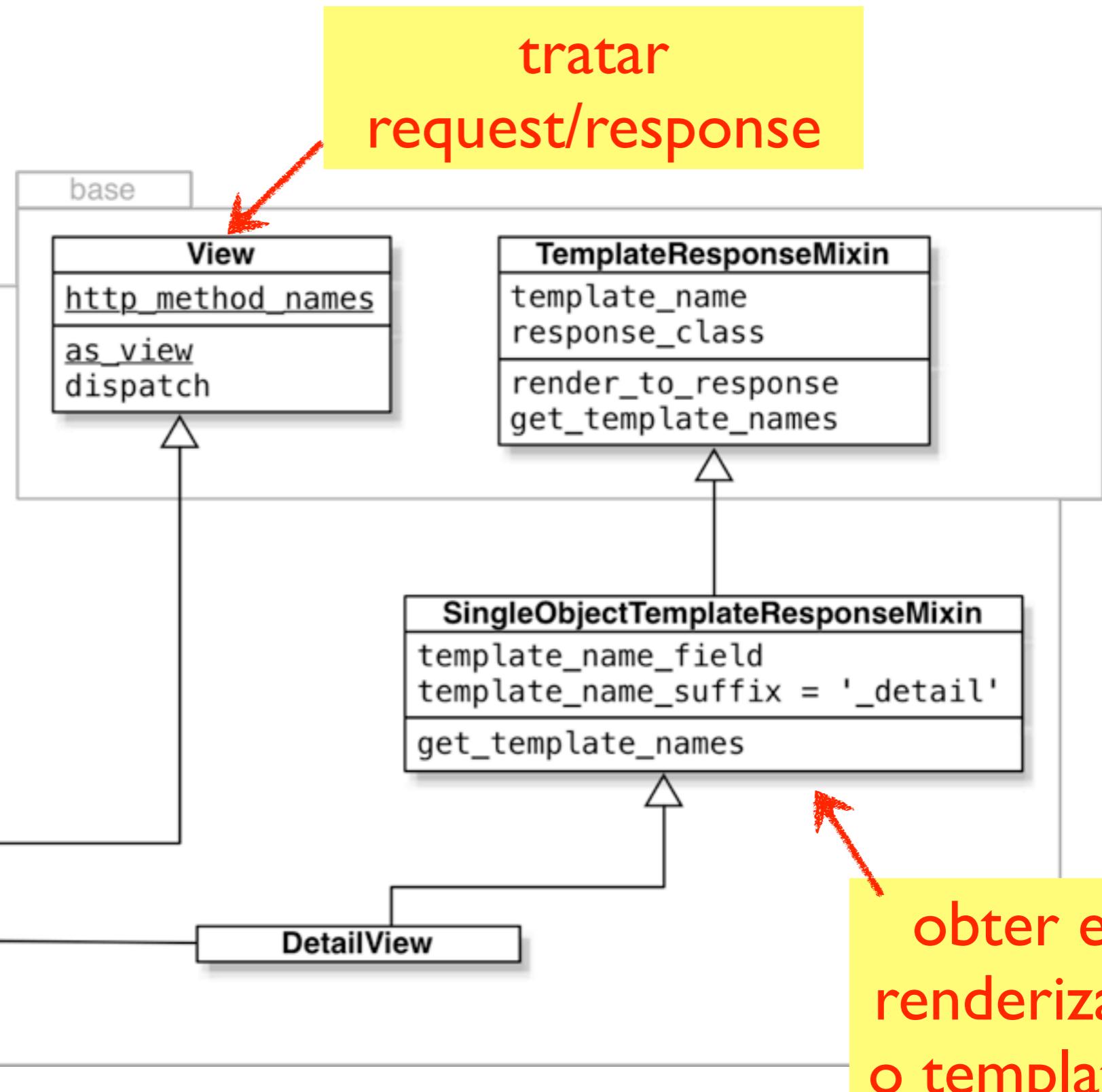
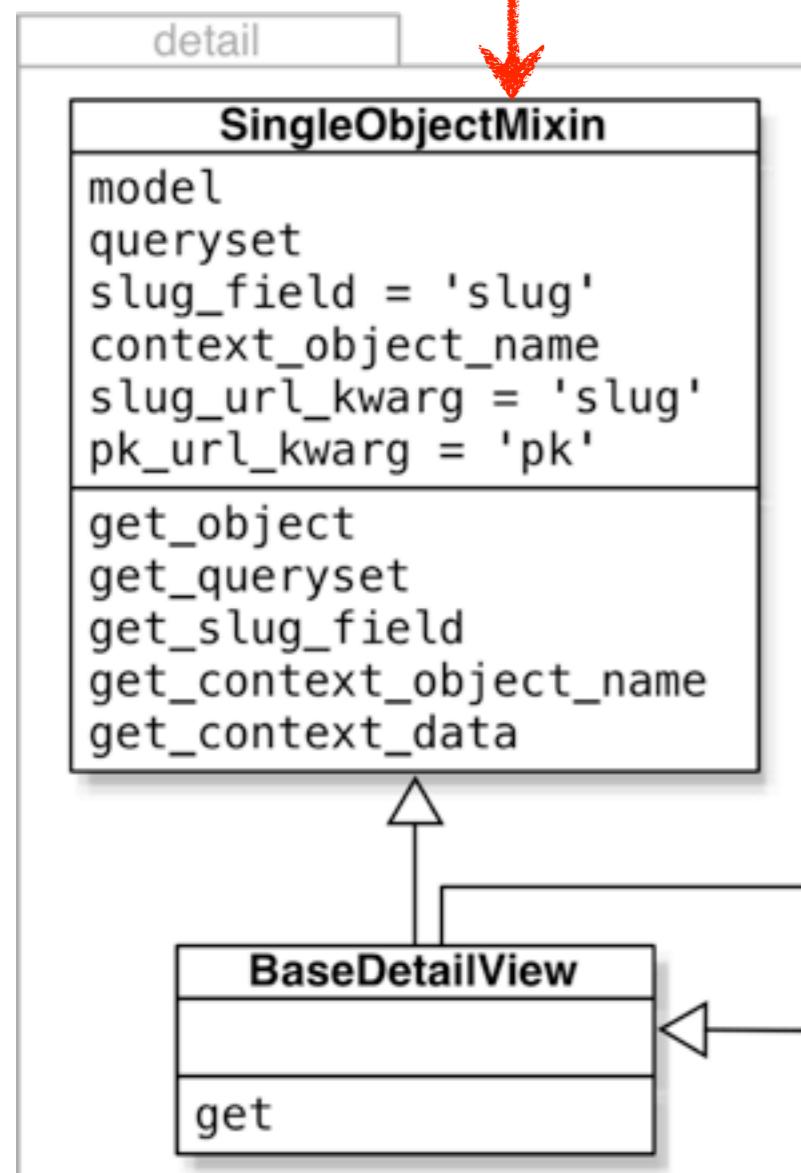
Apostila (em desenvolvimento) com diagramas UML:
<http://turing.com.br/material/acpython/mod3/django/viewsI.html>

CBV: divisão de tarefas



CBV: views de detalhe

identificar e
recuperar o objeto

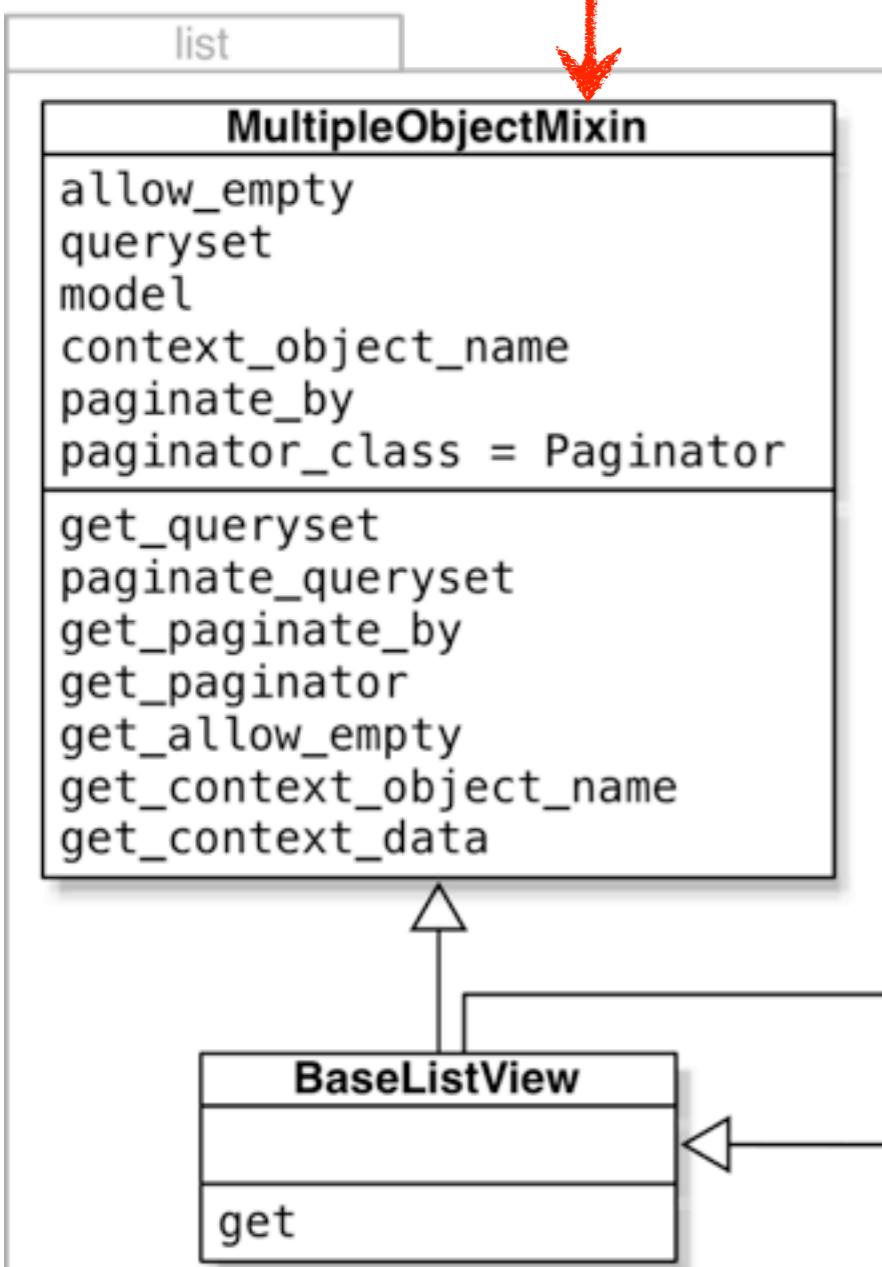


tratar
request/response

obter e
renderizar
o template

CBV: views de listagem

identificar e
recuperar a
lista de objetos



tratar
request/response

obter e
renderizar
o template

Exemplo de uso de CBV

- **django-ibge**: API restful fornecendo JSON para JQuery mostrar regiões, estados e municípios
- No arquivo **municipios/views.py**:
 - uma subclasse bem simples de **ListView**
- No arquivo **municipios/api.py**
 - 4 subclasses de **BaseListView** com **JSONResponseMixin**

<https://github.com/oturing/django-ibge>

Composição

As duas hierarquias de um sistema OO

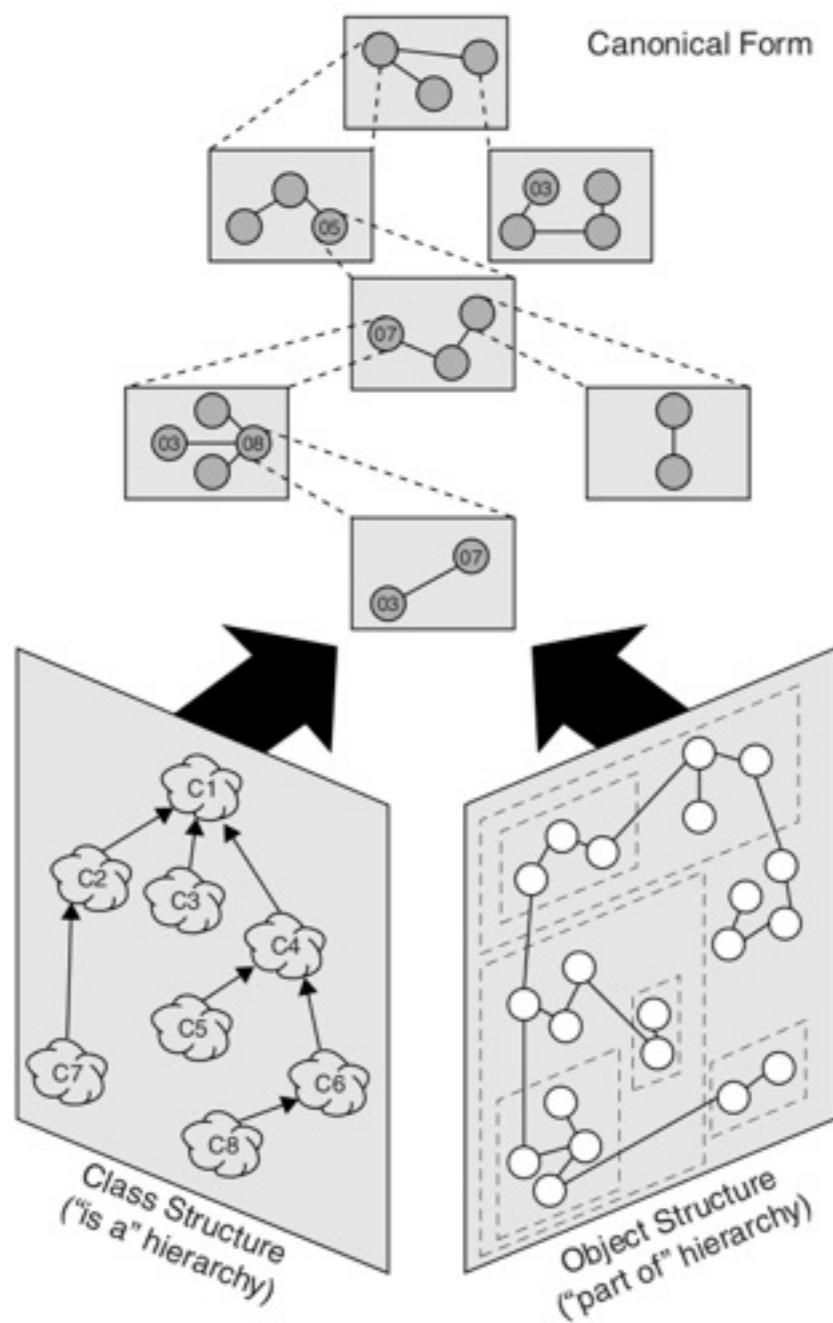
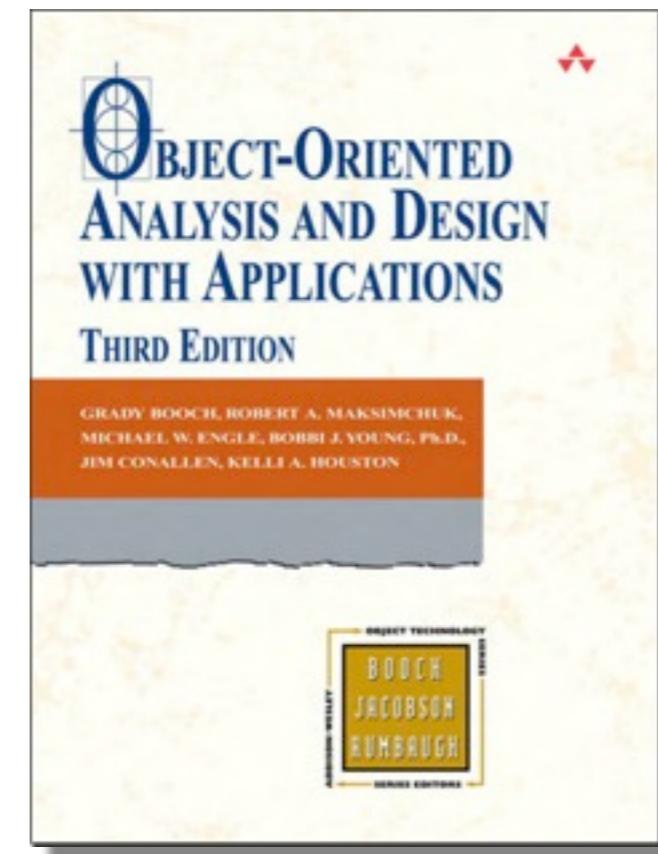


Figure 1–2 The Canonical Form of a Complex System



Object-oriented Analysis
and Design with Applications
3ed. - Booch et. al.

Timer



- Exemplo simples de composição

```
from Tkinter import Frame, Label, Button

class Timer(Frame):
    def __init__(self):
        Frame.__init__(self)
        self.inicio = self.agora = 15
        self.pendente = None # alarme pendente
        self.grid()
        self.mostrador = Label(self, width=2, anchor='e',
                              font='Helvetica 120 bold')
        self.mostrador.grid(column=0, row=0, sticky='nswe')
        self.bt_start = Button(self, text='Start', command=self.start)
        self.bt_start.grid(column=0, row=1, sticky='we')
        self.atualizar_mostrador()

    def atualizar_mostrador(self):
        self.mostrador['text'] = str(self.agora)

    def start(self):
        if self.pendente:
            self.after_cancel(self.pendente)
        self.agora = self.inicio
        self.atualizar_mostrador()
        self.pendente = self.after(1000, self.tictac)

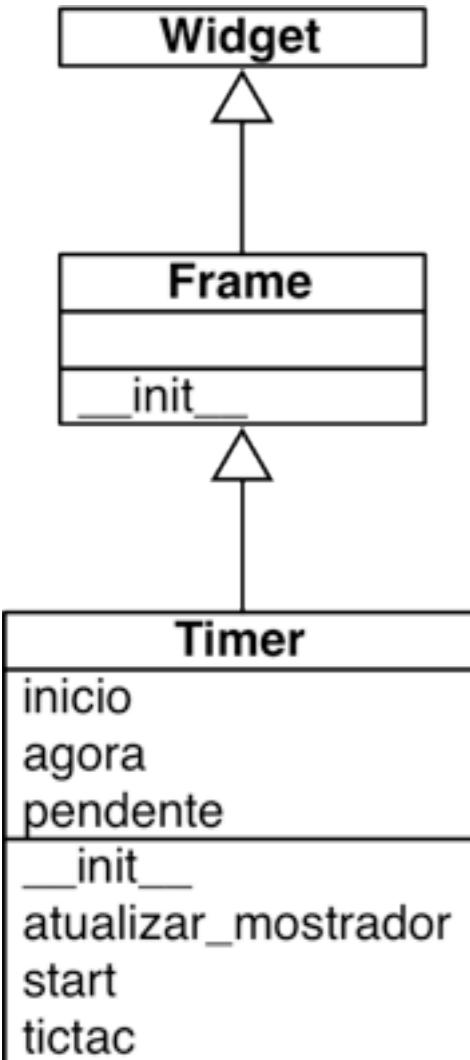
    def tictac(self):
        self.agora -= 1
        self.atualizar_mostrador()
        if self.agora > 0:
            self.pendente = self.after(1000, self.tictac)

timer = Timer()
timer.mainloop()
```



<https://github.com/oopy/exemplos/timer.py>

Timer



```
from Tkinter import Frame, Label, Button

class Timer(Frame):
    def __init__(self):
        Frame.__init__(self)
        self.inicio = self.agora = 15
        self.pendente = None # alarme pendente
        self.grid()
        self.mostrador = Label(self, width=2, anchor='e',
                              font='Helvetica 120 bold')
        self.mostrador.grid(column=0, row=0, sticky='nswe')
        self.bt_start = Button(self, text='Start', command=self.start)
        self.bt_start.grid(column=0, row=1, sticky='we')
        self.atualizar_mostrador()

    def atualizar_mostrador(self):
        self.mostrador['text'] = str(self.agora)

    def start(self):
        if self.pendente:
            self.after_cancel(self.pendente)
        self.agora = self.inicio
        self.atualizar_mostrador()
        self.pendente = self.after(1000, self.tictac)

    def tictac(self):
        self.agora -= 1
        self.atualizar_mostrador()
        if self.agora > 0:
            self.pendente = self.after(1000, self.tictac)

timer = Timer()
timer.mainloop()
```



git <https://github.com/oficinasTuring/oopy/exemplos/timer.py>

Composição

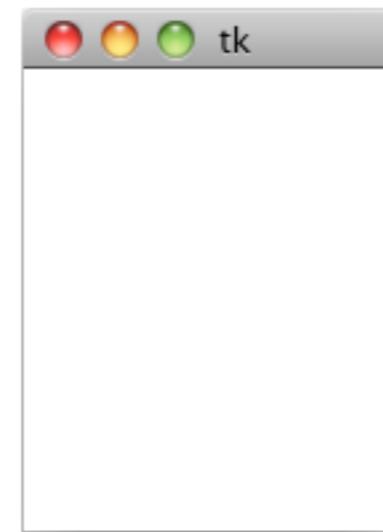
- Arranjo de partes de um sistema
 - componentes, sub-componentes...

15

Start

```
mostrador = Label()
```

```
bt_start = Button()
```



```
timer = Timer()
```

15

Start



```
from Tkinter import Frame, Label, Button

class Timer(Frame):
    def __init__(self):
        Frame.__init__(self)
        self.inicio = self.agora = 15
        self.pendente = None # alarme pendente
        self.grid()
        self.mostrador = Label(self, width=2, anchor='e',
                              font='Helvetica 120 bold')
        self.mostrador.grid(column=0, row=0, sticky='nswe')
        self.bt_start = Button(self, text='Start', command=self.start)
        self.bt_start.grid(column=0, row=1, sticky='we')
        self.atualizar_mostrador()

    def atualizar_mostrador(self):
        self.mostrador['text'] = str(self.agora)

    def start(self):
        if self.pendente:
            self.after_cancel(self.pendente)
        self.agora = self.inicio
        self.atualizar_mostrador()
        self.pendente = self.after(1000, self.tictac)

    def tictac(self):
        self.agora -= 1
        self.atualizar_mostrador()
        if self.agora > 0:
            self.pendente = self.after(1000, self.tictac)

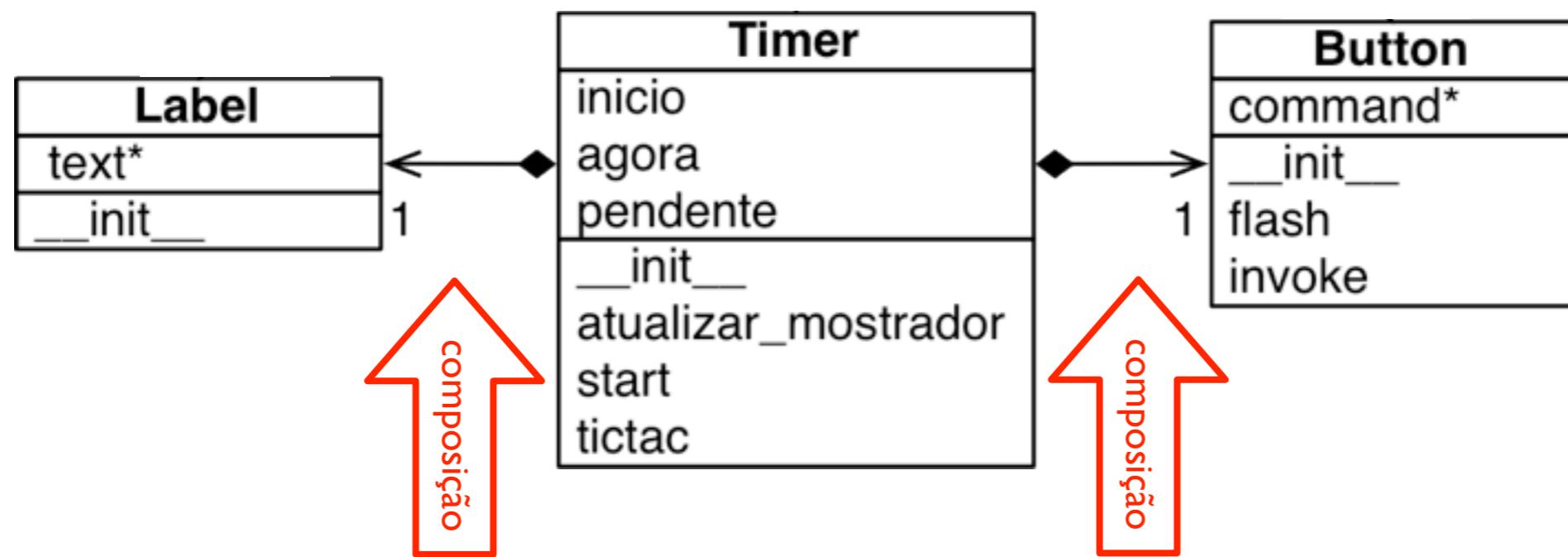
timer = Timer()
timer.mainloop()
```



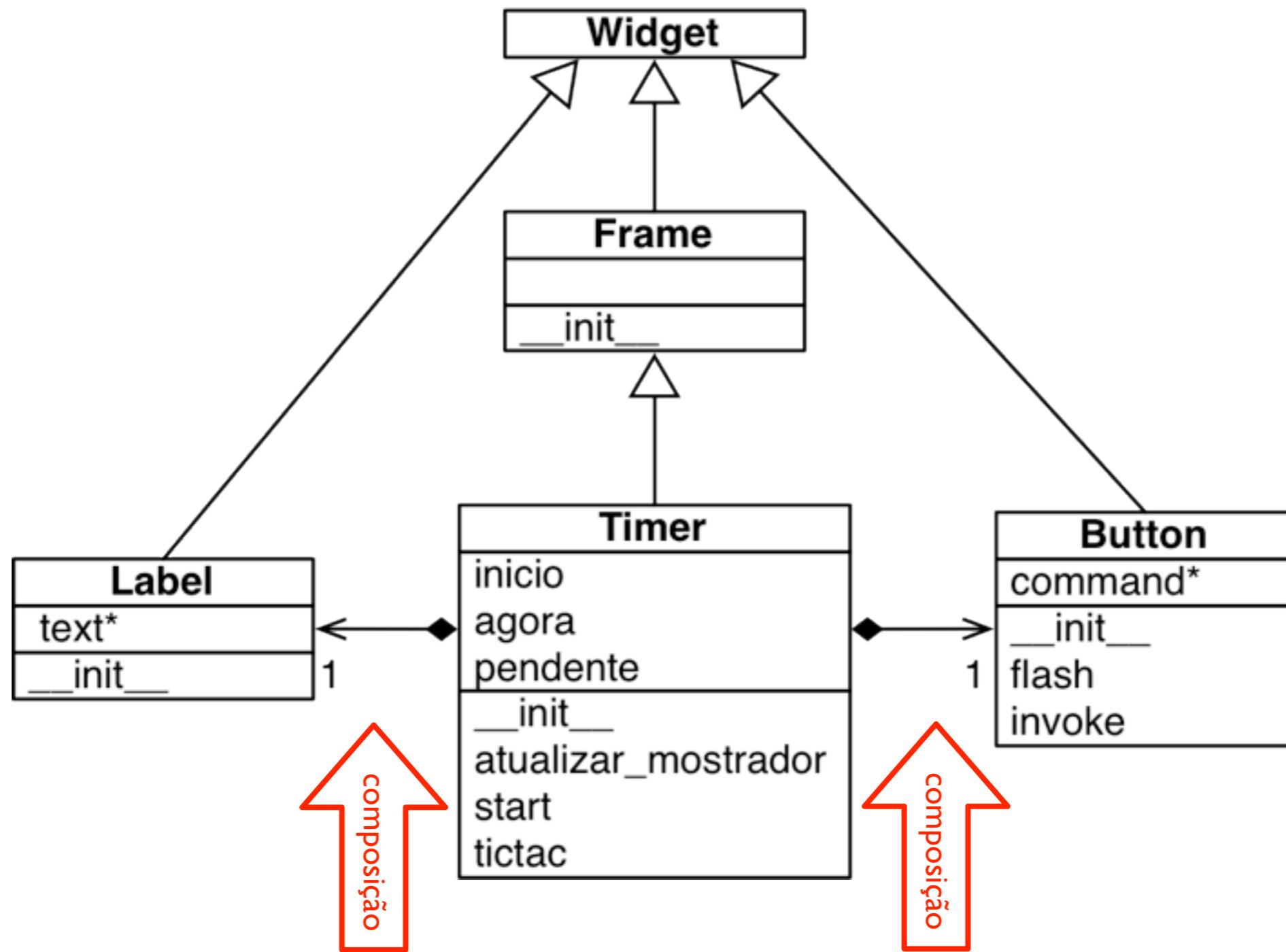
[oopy/exemplos/timer.py](#)



Composição em UML



Composição e herança



Sobrecarga de operadores

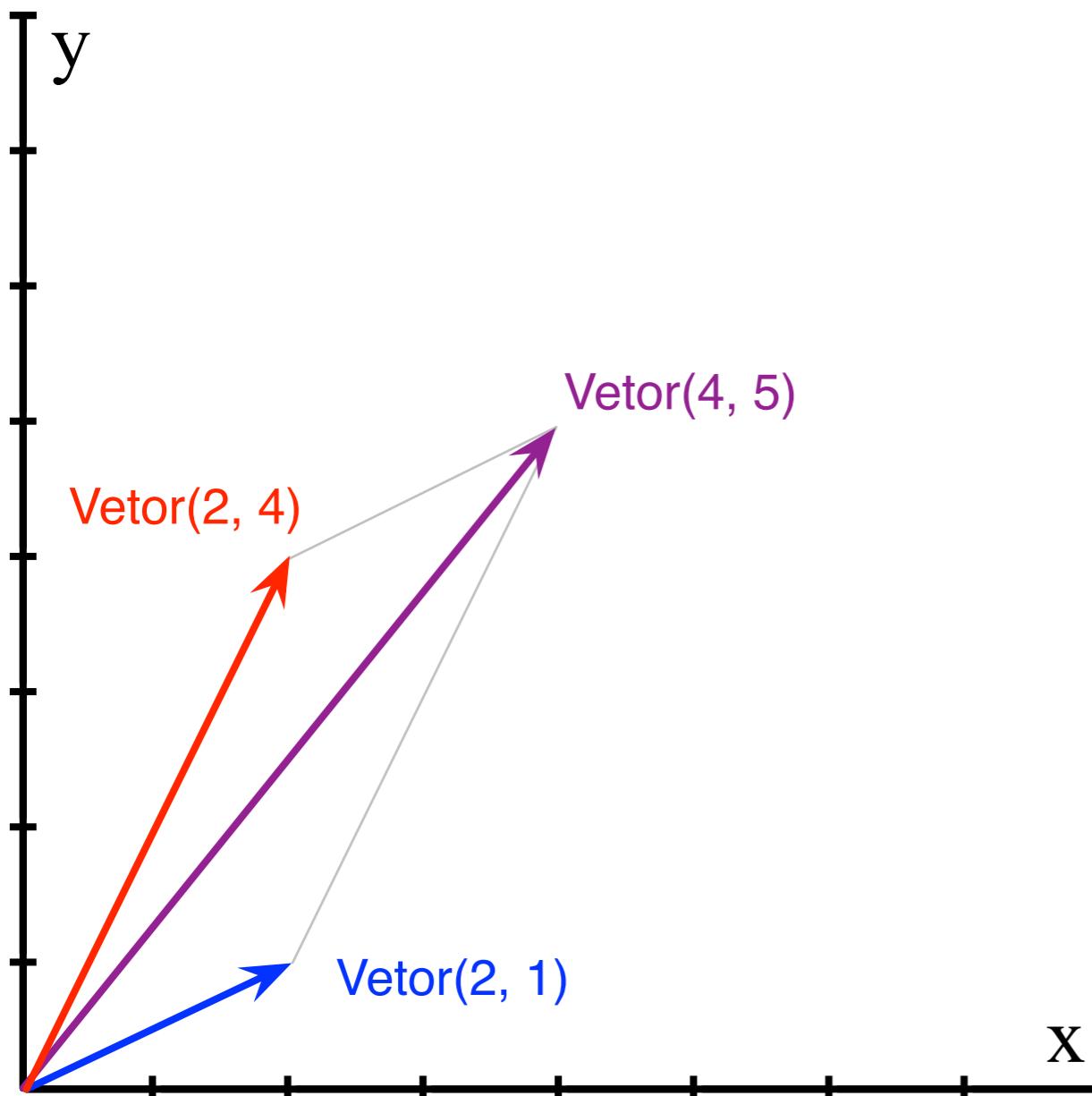
Sobrecarga de operadores

- Python permite que as classes definidas pelo usuário (você!) implementem métodos para os operadores definidos na linguagem
- Não é possível redefinir a função dos operadores nos tipos embutidos
 - isso evita surpresas desagradáveis
- Nem é possível inventar novos operadores
 - não podemos definir ~, <=>, /|\ etc.

Alguns operadores existentes

- Aritméticos: + - * / ** //
- Bitwise: & ^ | << >>
- Acesso a atributos: a.b
- Invocação: f(x)
- Operações em coleções: c[a], len(c), a in c, iter(c)
- Lista completa em Python Reference: Data Model

Exemplo: vetor (2d)



- Campos: x, y
- Métodos:
 - distância
 - abs (distância até 0,0)
 - + (add)
 - * (mul) escalar

```

from math import sqrt

class Vetor(object):

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __repr__(self):
        return 'Vetor(%s, %s)' % (self.x, self.y)

    def distancia(self, v2):
        dx = self.x - v2.x
        dy = self.y - v2.y
        return sqrt(dx*dx + dy*dy)

    def __abs__(self):
        return self.distancia(Vetor(0,0))

    def __add__(self, v2):
        dx = self.x + v2.x
        dy = self.y + v2.y
        return Vetor(dx, dy)

    def __mul__(self, n):
        return Vetor(self.x*n, self.y*n)

```

Vetor

```

>>> from vetor import Vetor
>>> v = Vetor(3, 4)
>>> abs(v)
5.0
>>> v1 = Vetor(2, 4)
>>> v2 = Vetor(2, 1)
>>> v1 + v2
Vetor(4, 5)
>>> v1 * 3
Vetor(6, 12)

```

Objetos invocáveis

- Você pode definir suas próprias funções...
- E também novas classes de objetos que se comportam como funções: objetos invocáveis
 - basta definir um método **`__call__`** para sobrecarregar o operador de invocação: `()`
 - `o(x)`
 - Exemplo: tômbara invocável

Tômbola invocável

- Já que o principal uso de uma instância de tômbola é sortear, podemos criar um atalho:
em vez de **t.sortear()**
apenas **t()**

```
from tombola import Tombola

class TombolaInvocavel(Tombola):
    '''Sorteia itens sem repetir;
    a instância é invocável como uma função'''

    def __call__(self):
        return self.sortear()
```

```
>>> t = TombolaInvocavel()
>>> t.carregar([1, 2, 3])
>>> t()
3
>>> t()
2
```

Encapsulamento

Encapsulamento

- Propriedades:
 - encapsulamento para quem precisa de encapsulamento

```
>>> a = C()  
>>> a.x = 10  
>>> print a.x  
10  
>>> a.x = -10  
>>> print a.x  
0
```

violação de
encapsulamento?

pergunte-me
como!

Propriedade: implementação

- apenas para leitura, via decorator:

```
class C(object):  
    def __init__(self, x):  
        self.__x = x  
    @property  
    def x(self):  
        return self.__x
```

- a notação `__x` protege o atributo contra acessos acidentais (`__x` = dois underscores à esquerda)

Propriedade: implementação 2

- para leitura e escrita (Python >= 2.2):

```
class C(object):
    def __init__(self, x=0):
        self.__x = x
    def getx(self):
        return self.__x
    def setx(self, valor):
        self.__x = valor if valor >= 0 else 0
x = property(getx, setx)
```

Propriedade: implementação 3

- para leitura e escrita (Python >= 2.6):

```
class C(object):
    def __init__(self, x=0):
        self.__x = x
    @property
    def x(self):
        return self.__x
    @x.setter
    def x(self, valor):
        self.__x = valor if valor >= 0 else 0
```

Propriedade: exemplo de uso

```
class ContadorTotalizador(Contador):
    def __init__(self):
        super(ContadorTotalizador, self).__init__()
        self.__total = 0

    def incluir(self, item):
        super(ContadorTotalizador, self).incluir(item)
        self.__total += 1

@property
def total(self):
    return self.__total
```

Atributos protegidos

- Atributos protegidos em Python são salvaguardas
 - servem para evitar atribuição ou sobrescrita acidental
 - não para evitar usos (ou abusos) intencionais



Atributos protegidos

- Atributos protegidos em Python são salvaguardas
 - servem para evitar atribuição ou sobrescrita acidental
 - não para evitar usos (ou abusos) intencionais



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

Decoradores

Decoradores de funções

- Exemplos de decoradores:
 - `@property`, `@x.setter`, `@classmethod`
- Não têm relação com o padrão de projeto “decorator”
- São funções que recebem a função decorada como argumento e produzem uma nova função que substitui a função decorada
- Tema de outro curso...

Decoradores de métodos

- Usados na definição de métodos em classes
 - `@property`, `@x.setter`, `@x.deleter`: definem métodos getter, setter e deleter para propriedades
 - `@classmethod`, `@staticmethod`: definem métodos que não precisam de uma instância para operar
 - `@abstractmethod`, `@abstractproperty`: uso em classes abstratas (veremos logo mais)

classmethod x staticmethod

- Métodos estáticos são como funções simples embutidas em uma classe: não recebem argumentos automáticos
- Métodos de classe recebem a classe como argumento automático

```
class Exemplo(object):  
    @staticmethod  
    def estatico(arg):  
        return arg  
  
    @classmethod  
    def da_classe(cls, arg):  
        return (cls, arg)
```

```
>>> Exemplo.estatico('bar')  
'bar'  
>>> Exemplo.da_classe('fu')  
(<class '__main__.Exemplo'>, 'fu')
```

Carta simples

```
class Carta(object):

    def __init__(self, valor, naipe):
        self.valor = valor
        self.naipe = naipe

    def __repr__(self):
        return 'Carta(%r, %r)' % (self.valor, self.naipe)
```

```
>>> zape = Carta('4', 'paus')
>>> zape.valor
'4'
>>> zape
Carta('4', 'paus')
```

Todas as cartas

```
class Carta(object):

    naipes = 'espadas ouros paus copas'.split()
    valores = '2 3 4 5 6 7 8 9 10 J Q K A'.split()

    def __init__(self, valor, naipe):
        self.valor = valor
        self.naipe = naipe

    def __repr__(self):
        return 'Carta(%r, %r)' % (self.valor, self.naipe)

@classmethod
def todas(cls):
    return [cls(v, n) for n in cls.naipes
            for v in cls.valores]
```

```
>>> monte = Carta.todas()
>>> len(monte)
52
>>> monte[0]
Carta('2', 'espadas')
>>> monte[-3:]
[Carta('Q', 'copas'), Carta('K', 'copas'), Carta('A', 'copas')]
```

Exemplo de classmethod

- É conveniente em **todas** ter acesso à classe para acessar os atributos (**naipes**, **valores**) e para instanciar as cartas

```
class Carta(object):

    naipes = 'espadas ouros paus copas'.split()
    valores = '2 3 4 5 6 7 8 9 10 J Q K A'.split()

    def __init__(self, valor, naipe):
        self.valor = valor
        self.naipe = naipe

    def __repr__(self):
        return 'Carta(%r, %r)' % (self.valor,
                                   self.naipe)

    @classmethod
    def todas(cls):
        return [cls(v, n) for n in cls.naipes
               for v in cls.valores]
```

Decoradores de classes

- Novidade do Python 2.6, ainda pouco utilizada na prática
- Exemplo na biblioteca padrão a partir do Python 2.7:
 - **functools.total_ordering** define automaticamente métodos para os operadores de comparação < > <= >=

Cartas comparáveis

```
from functools import total_ordering

@total_ordering
class Carta(object):

    naipes = 'espadas ouros paus copas'.split()
    valores = '2 3 4 5 6 7 8 9 10 J Q K A'.split()

    def __init__(self, valor, naipe):
        self.valor = valor
        self.naipe = naipe

    def __repr__(self):
        return 'Carta(%r, %r)' % (self.valor, self.naipe)

    @classmethod
    def todas(cls):
        return [cls(v, n) for n in cls.naipes
                for v in cls.valores]

    def __eq__(self, outra):
        return ((self.valor, self.naipe) ==
                (outra.valor, outra.naipe))

    def peso(self):
        return (Carta.naipes.index(self.naipe) +
               len(Carta.naipes) * Carta.valores.index(self.valor))

    def __gt__(self, outra):
        return self.peso() > outra.peso()
```

```
>>> dois >= as_
False
>>> dois <= as_
True
```

Total ordering == lucro!

```
>>> mao = [as_, dois, rei]
>>> sorted(mao)
[Carta('2', 'espadas'), Carta('K', 'copas'), Carta('A', 'copas')]
```

```
from functools import total_ordering

@total_ordering
class Carta(object):

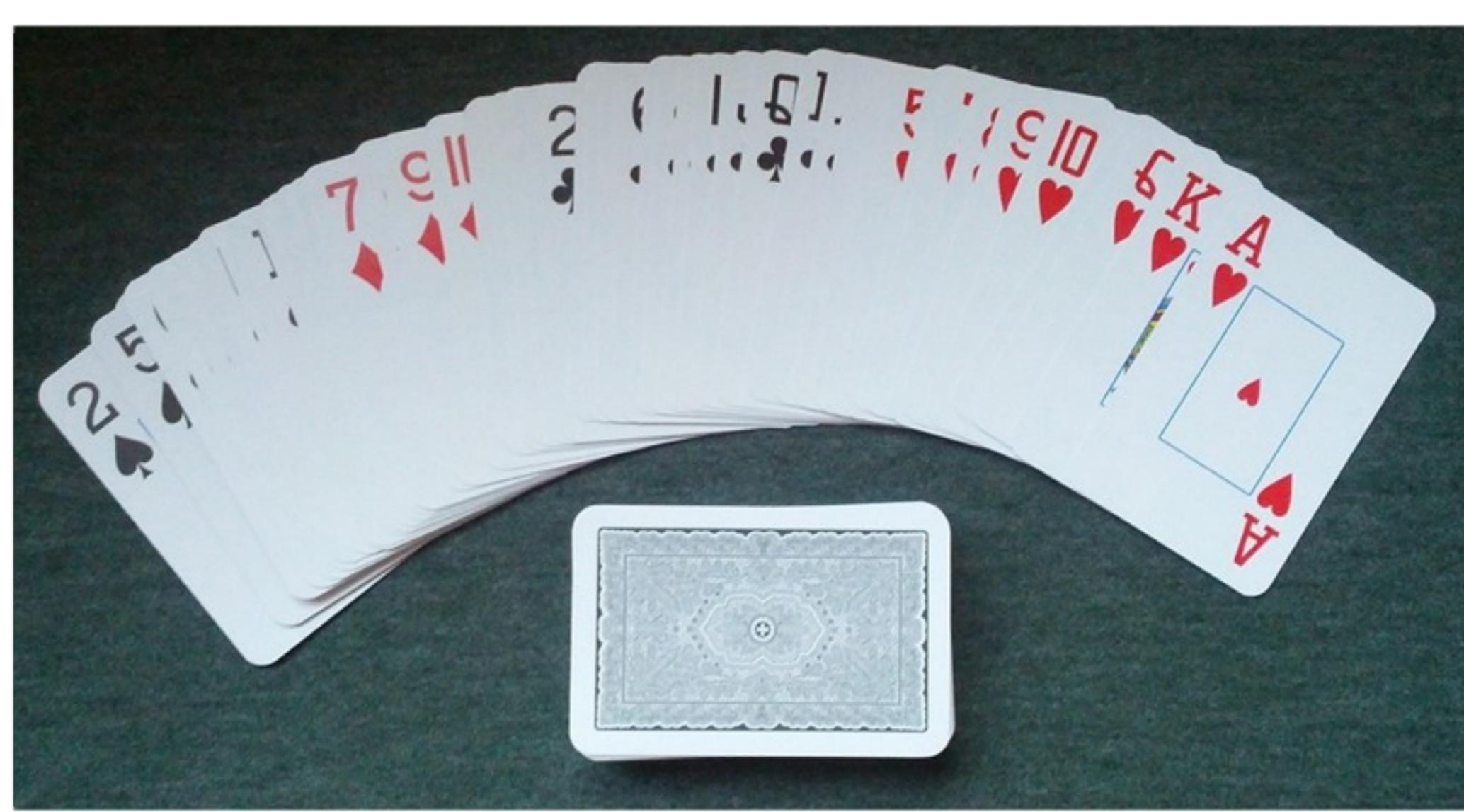
    # ...várias linhas omitidas...

    def __eq__(self, outra):
        return ((self.valor, self.naipes) ==
                (outra.valor, outra.naipes))

    def peso(self):
        return (Carta.naipes.index(self.naipes) +
               len(Carta.naipes) * Carta.valores.index(self.valor))

    def __gt__(self, outra):
        return self.peso() > outra.peso()
```

Baralho polimórfico



Polimorfismo: definição

O conceito de “polimorfismo” significa que podemos tratar instâncias de diferentes classes da mesma maneira.

Assim, podemos enviar uma mensagem a um objeto sem saber de antemão qual é o seu tipo, e o objeto ainda assim fará “a coisa certa”, pelo menos do seu ponto de vista.

Scott Ambler - The Object Primer, 2nd ed. - p. 173

Polimorfismo

- Fatiamento e `len`
 - listas e strings são sequências

```
>>> l = [1, 2, 3]
>>> l[:2]
[1, 2]
>>> 'casa'[:2]
'ca'
>>> len(l)
3
>>> len('casa')
4
```

Polimorfismo

```
>>> s = 'Python: simples e correta'  
>>> for letra in s[:6]: print letra  
P  
y  
t  
h  
o  
n  
>>> for letra in reversed(s): print letra  
...  
o  
r  
e  
t  
a  
n  
h  
o  
s  
e  
l  
p  
s  
i  
m  
p  
l  
e  
s  
e  
c  
o  
r  
r  
e  
t  
a  
r  
a  
o  
n  
:
```

Polimorfismo

```
>>> l = range(10)
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> l[0]
0
>>> l[-1]
9
>>> l[:3]
[0, 1, 2]
>>> for n in reversed(l): print n
...
9
8
7
6
5
4
3
```

Baralho polimórfico 2

- métodos especiais: `__len__`, `__getitem__`
- com esses métodos, Baralho implementa o protocolo das sequências imutáveis

```
class Baralho(object):  
  
    def __init__(self):  
        self.cartas = Carta.todas()  
  
    def __len__(self):  
        return len(self.cartas)  
  
    def __getitem__(self, pos):  
        return self.cartas[pos]
```

Baralho polimórfico 3

```
>>> from baralho import Baralho
>>> b = Baralho()
>>> len(b)
52
>>> b[0], b[1], b[2]
(<A de paus>, <2 de copas>, <3 de copas>)
>>> for carta in reversed(b): print carta
...
<K de ouros>
<Q de ouros>
<J de ouros>
<10 de ouros>
<9 de ouros>
<8 de ouros>
<7 de ouros>
<6 de ouros>
<5 de ouros>
```

Baralho polimórfico 4

```
>>> from baralho import Baralho
>>> b = Baralho()
>>> len(b)
52
>>> b[:3]
[<A de paus>, <2 de paus>, <3 de paus>]
>>> from random import choice
>>> for i in range(5): print choice(b)
...
<0 de copas>
<4 de ouros>
<A de copas>
<5 de ouros>
<9 de paus>
>>> for i in range(5): print choice(b)
...
<3 de paus>
<9 de copas>
```

a mesma carta pode sair duas vezes!

Baralho polimórfico 5

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

Traceback (most recent call last):

```
  File "<stdin>", line 1, in <module>  
  File "/System/Library/Frameworks/Python.framework/  
Versions/2.6/lib/python2.6/random.py", line 275, in  
shuffle
```

```
    x[i], x[j] = x[j], x[i]
```

TypeError: 'Baralho' object does not support item assignment

Python vem com pilhas incluídas!

ooops...



slide
semi
novo

Baralho polimórfico 6

```
>>> def meu_setitem(self, pos, valor):  
...     self.cartas[pos] = valor  
...  
>>> Baralho.__setitem__ = meu_setitem  
>>> shuffle(b)  
>>> b[:5]  
[<J de espadas>, <Q de paus>, <2 de paus>,  
<6 de paus>, <A de espadas>]  
>>>
```

monkey-
patch

agora
funciona!

Baralho polimórfico 7

- fazendo direito (sem monkey-patch)

```
class Baralho(object):

    def __init__(self):
        self.cartas = Carta.todas()

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

    def __getitem__(self, pos):
        return self.cartas[pos]

    def __setitem__(self, pos, item):
        self.cartas[pos] = item
```

Oficinas Turing: computação para programadores

- Próximos lançamentos:
 - I^a turma de **Python para quem usa Django**
 - 3^a turma de **Objetos Pythonicos**
 - 4^a turma de **Python para quem sabe Python**

Para saber mais sobre estes cursos **ONLINE**
escreva para:
ramalho@turing.com.br