Luciano Ramalho
luciano@ramalho.org

@pythonprobr

novembro/2013

Objetos Pythonicos

Orientação a objetos e padrões de projeto em Python

Aula 3

Variáveis e referências
+
Iteráveis, iteradores e geradores

@pythonprobr

Objetivos desta aula

- Apresentar os conceitos de referências e *aliasing*
- Iniciar a discussão sobre padrões de projeto, mostrando iteradores em Python

Implementação da tômbola

- Python 2.2 a 2.7

```
# coding: utf-8

import random

class Tombola(object):
    itens = None

    def carregar(self, seq):
        self.itens = list(seq)

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

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

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

Uma tômbola com defeito

- Esta implementação tem um bug sutil, porém grave

```
# coding: utf-8
# COM DEFEITO!!!
import random

class Tombola(object):
    itens = []

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

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

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

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

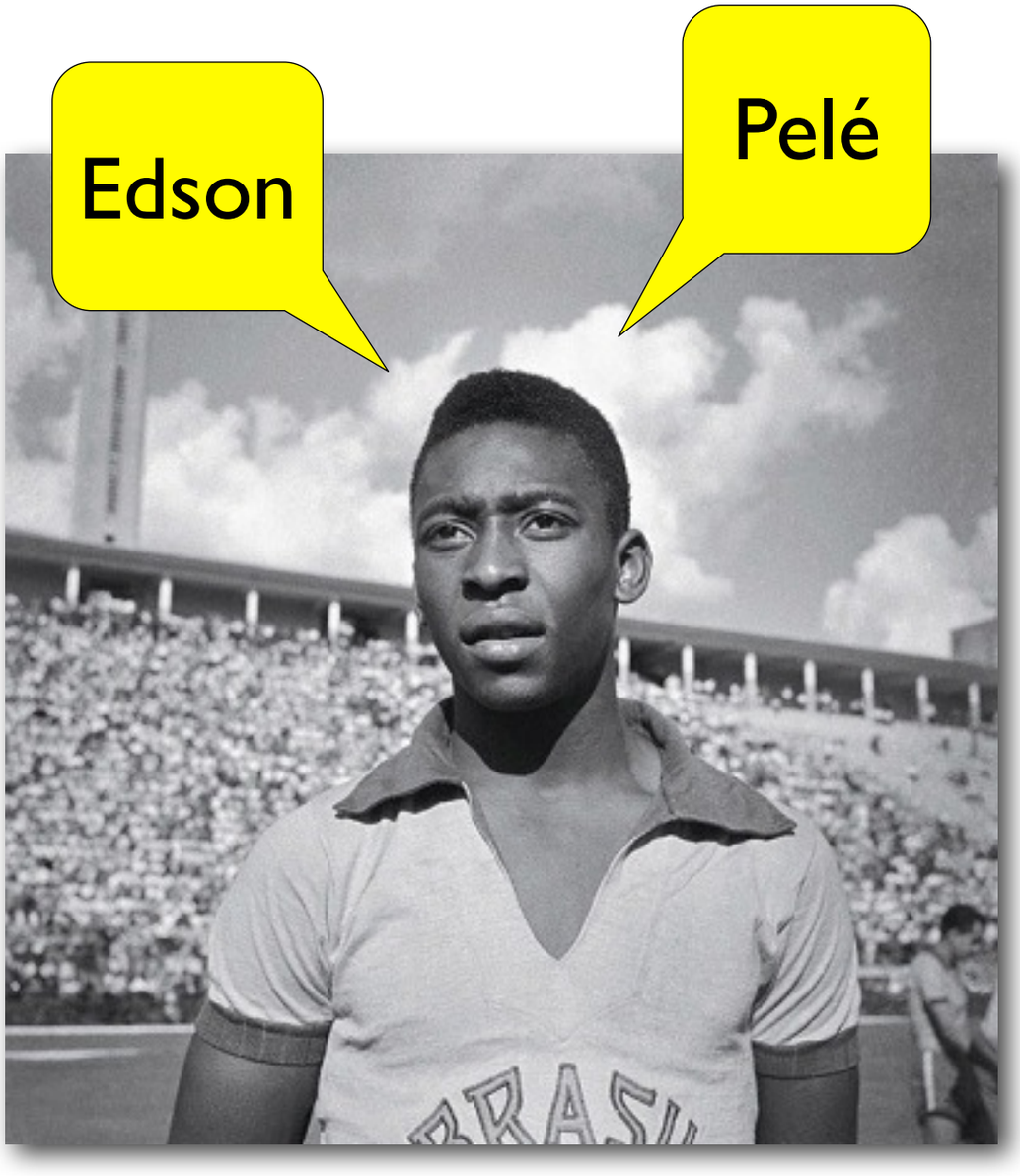
Mutabilidade

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

Mutabilidade: exemplos

- Alguns tipos mutáveis:
  - list
  - dict
  - set
  - objetos que permitem a alteração de atributos por acesso direto, setters ou outros métodos
- Alguns tipos imutáveis:
  - str

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

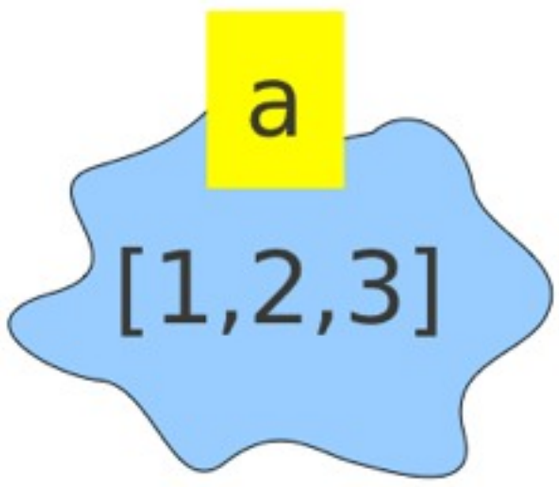
Aliasing (“apelidamento”)

- Em Python, as variáveis não “contém” objetos, apenas referências para objetos
  - Esqueça a metáfora da variável como “caixa”
- Isso significa que duas variáveis podem apontar para o mesmo objeto
  - Adote a metáfora da variável como “rótulo”
  - O mesmo objeto pode ter dois rótulos

Variáveis e referências

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

```
a = [1,2,3]
```



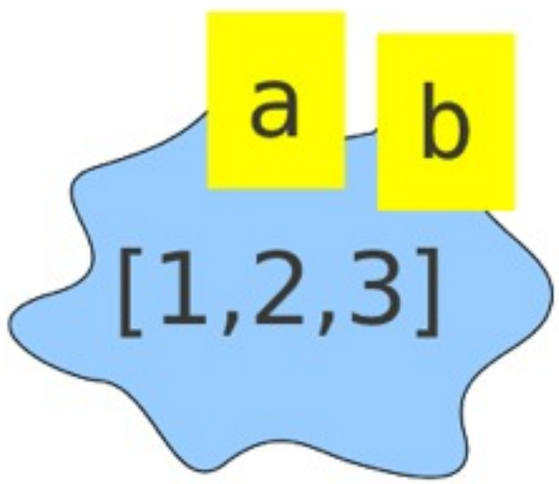
Variáveis e referências

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

```
a = [1,2,3]
```



```
b = a
```



Variáveis e referências

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

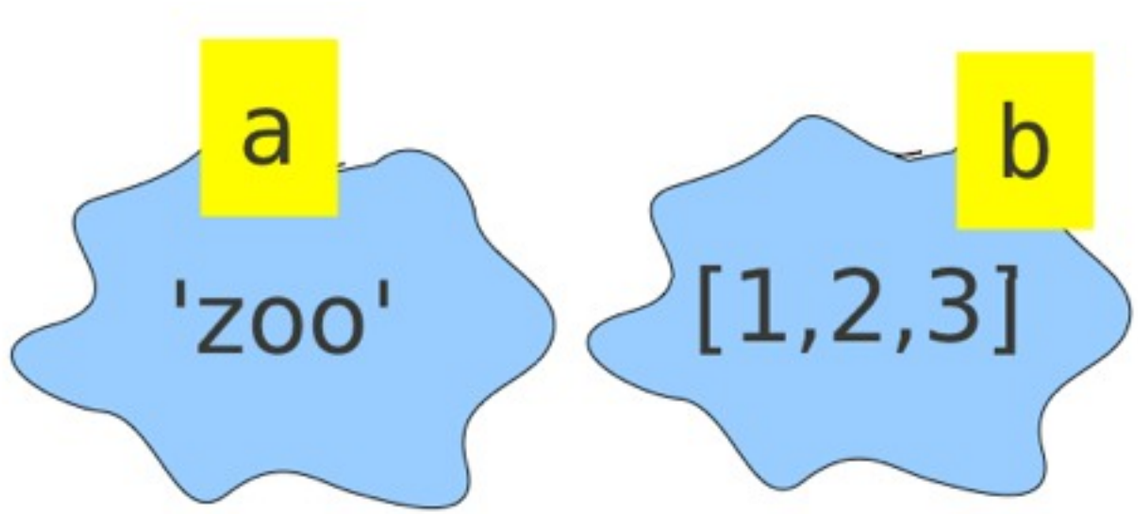
```
a = [1,2,3]
```



```
b = a
```



```
a = 'zoo'
```



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 sobrecarregado

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 sobrecarregado em qualquer classe, redefinindo o método __eq__
  - Em Python, None é um objeto e suporta ==
  - Mas o teste **x is None** é mais eficiente

Iteráveis,
iteradores
e geradores

Comparando: C e Python

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int i;
    for(i = 0; i < argc; i++)
        printf("%s\n", argv[i]);
    return 0;
}
```

```python
import sys

for arg in sys.argv:
    print arg
```

Iteração em Java

```java
class Argumentos {
    public static void main(String[] args) {
        for (int i=0; i < args.length; i++)
            System.out.println(args[i]);
    }
}
```

```
$ java Argumentos alfa bravo charlie
alfa
bravo
charlie
```

Iteração em Java ≥ 1.5

ano: 2004

- *Enhanced* ***for*** (**for** melhorado)

```
class Argumentos2 {
    public static void main(String[] args) {
        for (String arg : args)
            System.out.println(arg);
    }
}
```

```
$ java Argumentos2 alfa bravo charlie
alfa
bravo
charlie
```

ABC language
the mother of Python

"It all started with ABC, a wonderful teaching language that I had helped create in the early eighties. It was an incredibly elegant and powerful language, aimed at non-professional programmers."

Guido van Rossum, ABC team member and creator of Python[1]

ABC 1.05

```
>>> HOW TO SIEVE TO n:
HOW TO SIEVE TO n:
   PUT {2..n} IN numbers
   WHILE numbers <> {}:
      PUT min numbers IN p
      WRITE p
      FOR m IN {1..floor(n/p)}:
         IF m*p in numbers:
            REMOVE m*p FROM numbers

>>> SIEVE TO 50
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
```

SIEVE TO procedure definition

condition must be a **test**

21 years later, Python still has a lot of ABC in it

Python 3.2

```
>>> def sieve(n):
...     numbers = set(range(2, n+1))
...     while numbers:
...         p = min(numbers)
...         print(p, end=' ')
...         for m in range(1, int(n/p)+1):
...             if m*p in numbers:
...                 numbers.remove(m*p)
...
>>> sieve(50)
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
```

1. Foreword for "Programming Python" (1st ed.) http://www.python.org/doc/essays/foreword/

ABC 1.05

```
>>> HOW TO SIEVE TO n:
HOW TO SIEVE TO n:
    PUT {2..n} IN numbers
    WHILE numbers <> {}:
        PUT min numbers IN p
        WRITE p
        FOR m IN {1..floor(n/p)}:
            IF m*p in numbers:
                REMOVE m*p FROM numbers

>>> SIEVE TO 50
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
```

ano:
1984!

Exemplos de iteração

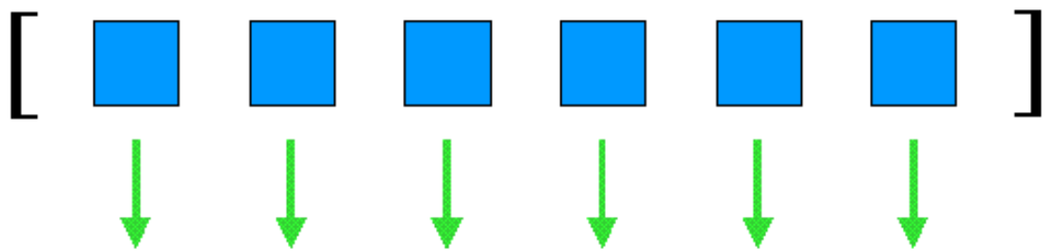
- Iteração em Python não se limita a tipos primitivos
- Exemplos
  - string
  - arquivo
  - Django QuerySet
  - Baralho (em:“OO em Python sem Sotaque”)

https://slideshare.net/ramalho/

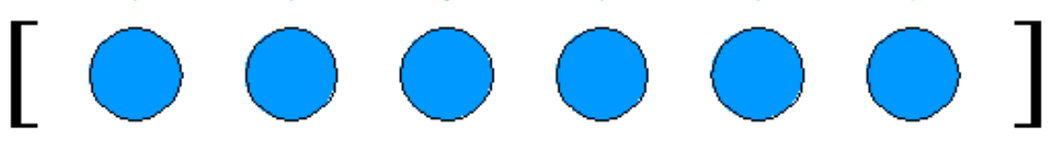
List comprehensions

- Expressões que consomem iteráveis e produzem listas

qualquer iterável



resultado: uma lista



```
>>> s = 'abracadabra'
>>> l = [ord(c) for c in s]
>>> [ord(c) for c in s]
[97, 98, 114, 97, 99, 97, 100, 97, 98, 114, 97]
```

Set & dict comprehensions

- Expressões que consomem iteráveis e produzem sets ou dicts

```
>>> s = 'abracadabra'
>>> {c for c in s}
set(['a', 'r', 'b', 'c', 'd'])
>>> {c:ord(c) for c in s}
{'a': 97, 'r': 114, 'b': 98, 'c': 99, 'd': 100}
```

Em Python o comando **for** itera sobre... "iteráveis"

- Definicão preliminar informal:
  - "iterável" = que pode ser iterado
    - assim como:
      "desmontável" = que pode ser desmontado
- Iteráveis podem ser usados em outros contextos além do laço for

Tipos iteráveis embutidos

- basestring
  - str
  - unicode
- dict
- file
- frozenset
- list
- set
- tuple
- xrange

Funções embutidas que consomem iteráveis

- all
- any
- filter
- iter
- len
- map
- max
- min
- reduce
- sorted
- sum
- zip

Operações com iteráveis

- Desempacotamento de tupla
  - em atribuições
  - em chamadas de funções

```
>>> a, b, c = 'XYZ'
>>> a
'X'
>>> b
'Y'
>>> c
'Z'
>>> g = (n for n in [1, 2, 3])
>>> a, b, c = g
>>> a
1
>>> b
2
>>> c
3
```

```
>>> def soma(a, b):
...     return a + b
...
>>> soma(1, 2)
3
>>> t = (3, 4)
>>> soma(t)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: soma() takes exactly 2 arguments (1 given)
>>> soma(*t)
7
```

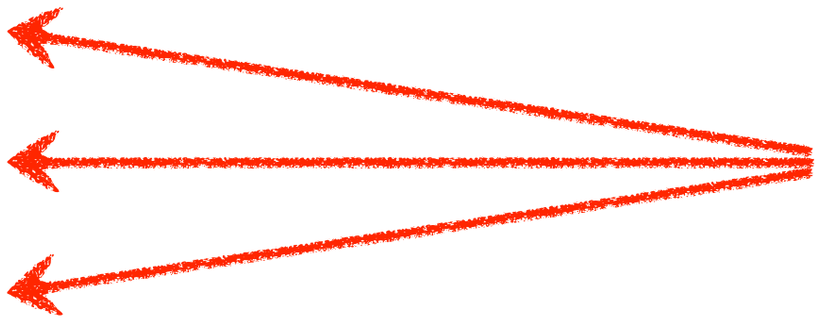
Em Python, um iterável é...

- Um objeto a partir do qual a função **iter** consegue obter um iterador.
- A chamada iter(x):
  - invoca x.__iter__() para obter um iterador
  - **ou**, se x.__iter__ não existe:
    - fabrica um iterador que acessa os itens de x sequenciamente fazendo x[0], x[1], x[2] etc.

Protocolo de sequência

```
>>> t = Trem(4)
>>> len(t)
4
>>> t[0]
'vagao #1'
>>> t[3]
'vagao #4'
>>> t[-1]
'vagao #4'
>>> for vagao in t:
...     print(vagao)
vagao #1
vagao #2
vagao #3
vagao #4
```

__len__



__getitem__

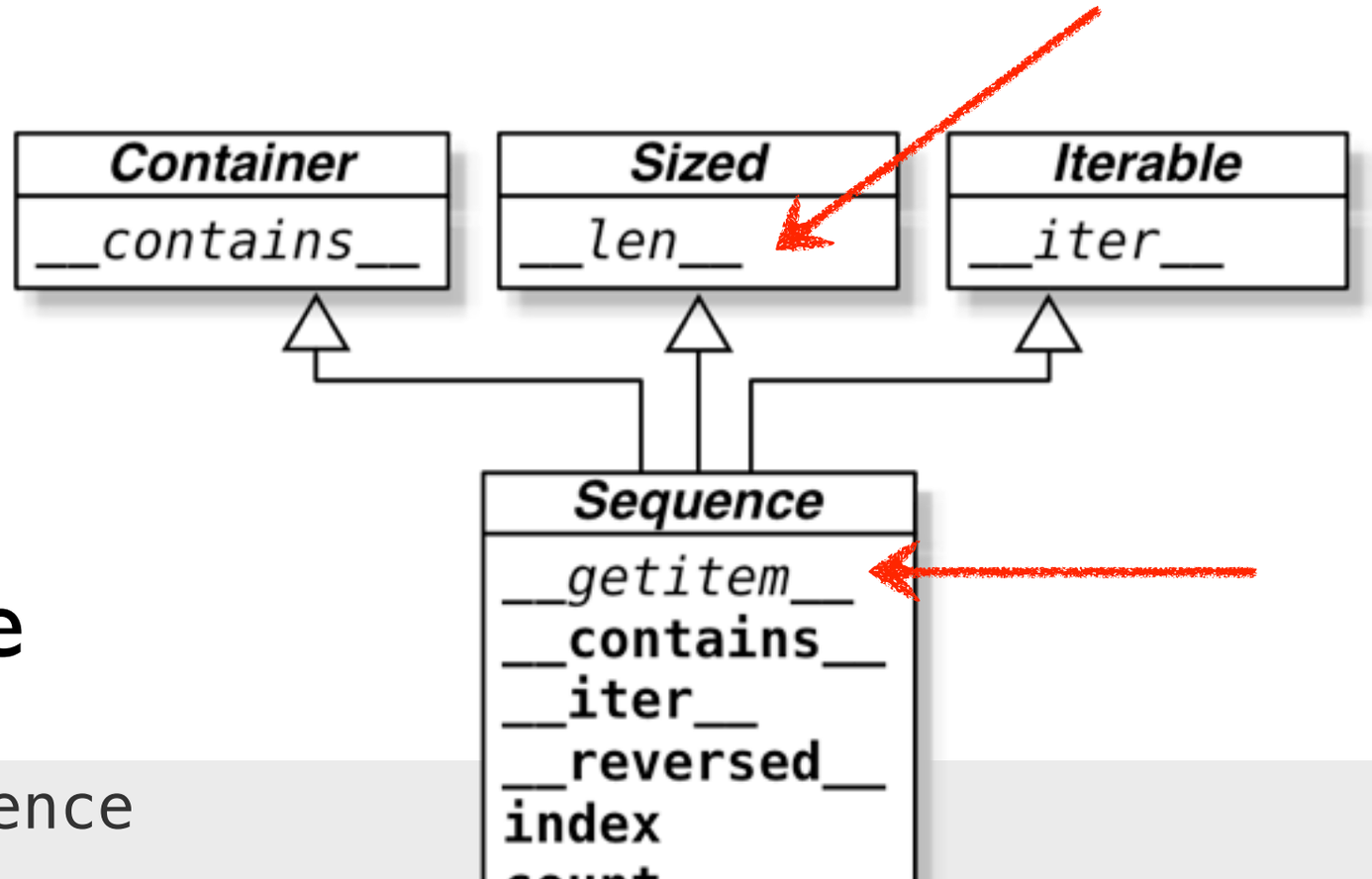
__getitem__

Protocolo de sequência

- implementação “informal” da interface

```
class Trem(object):
    def __init__(self, num_vagoes):
        self.num_vagoes = num_vagoes
    def __len__(self):
        return self.num_vagoes
    def __getitem__(self, pos):
        indice = pos if pos >= 0 else self.num_vagoes + pos
        if 0 <= indice < self.num_vagoes: # indice 2 -> vagao #3
            return 'vagao #%s' % (indice+1)
        else:
            raise IndexError('vagao inexistente %s' % pos)
```

Interface Sequence

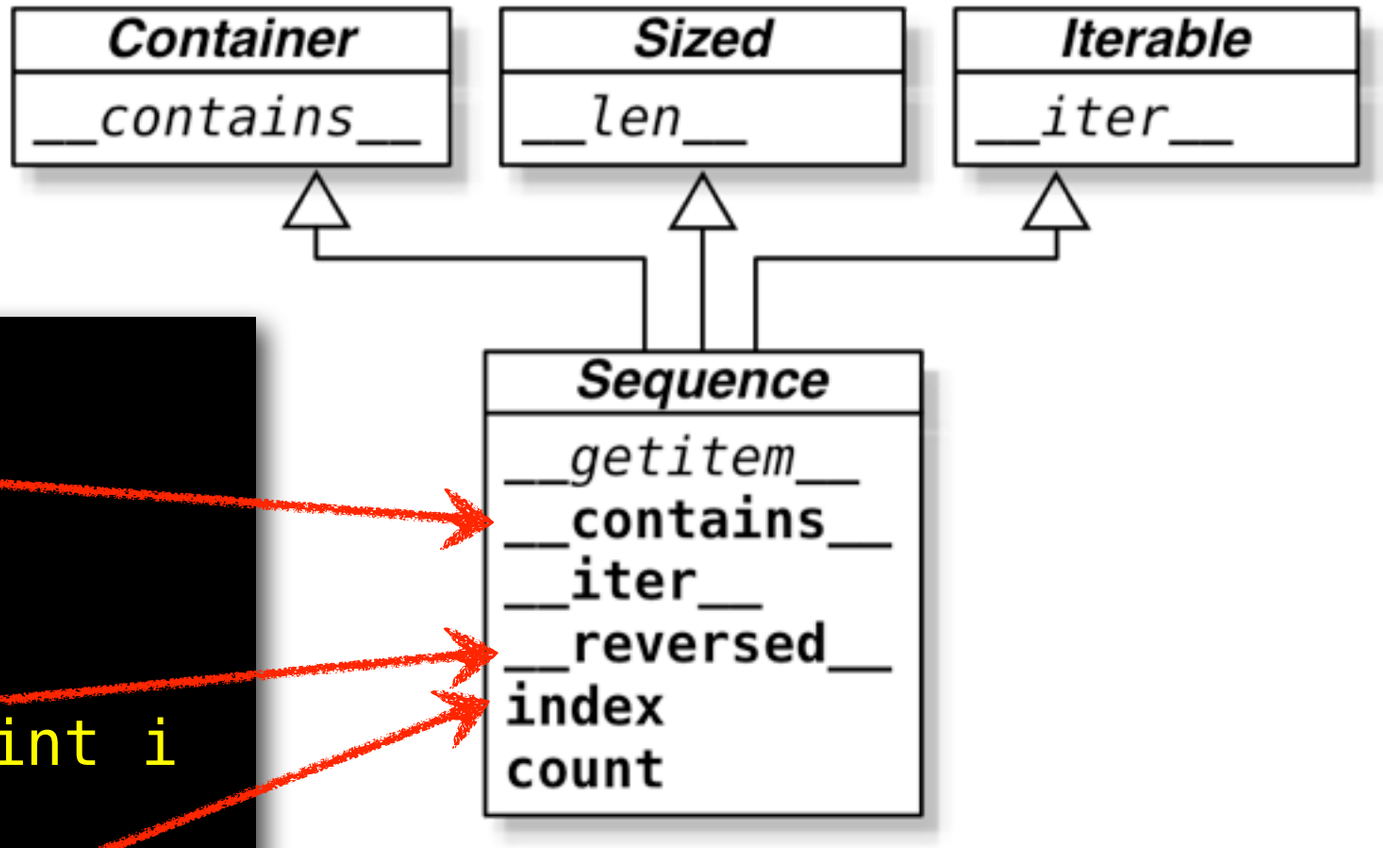


- collections.Sequence

```
from collections import Sequence

class Trem(Sequence):
    def __init__(self, num_vagoes):
        self.num_vagoes = num_vagoes
    def __len__(self):
        return self.num_vagoes
    def __getitem__(self, pos):
        indice = pos if pos >= 0 else self.num_vagoes + pos
        if 0 <= indice < self.num_vagoes: # indice 2 -> vagao #3
            return 'vagao #%s' % (indice+1)
        else:
            raise IndexError('vagao inexistente %s' % pos)
```

Herança de Sequence

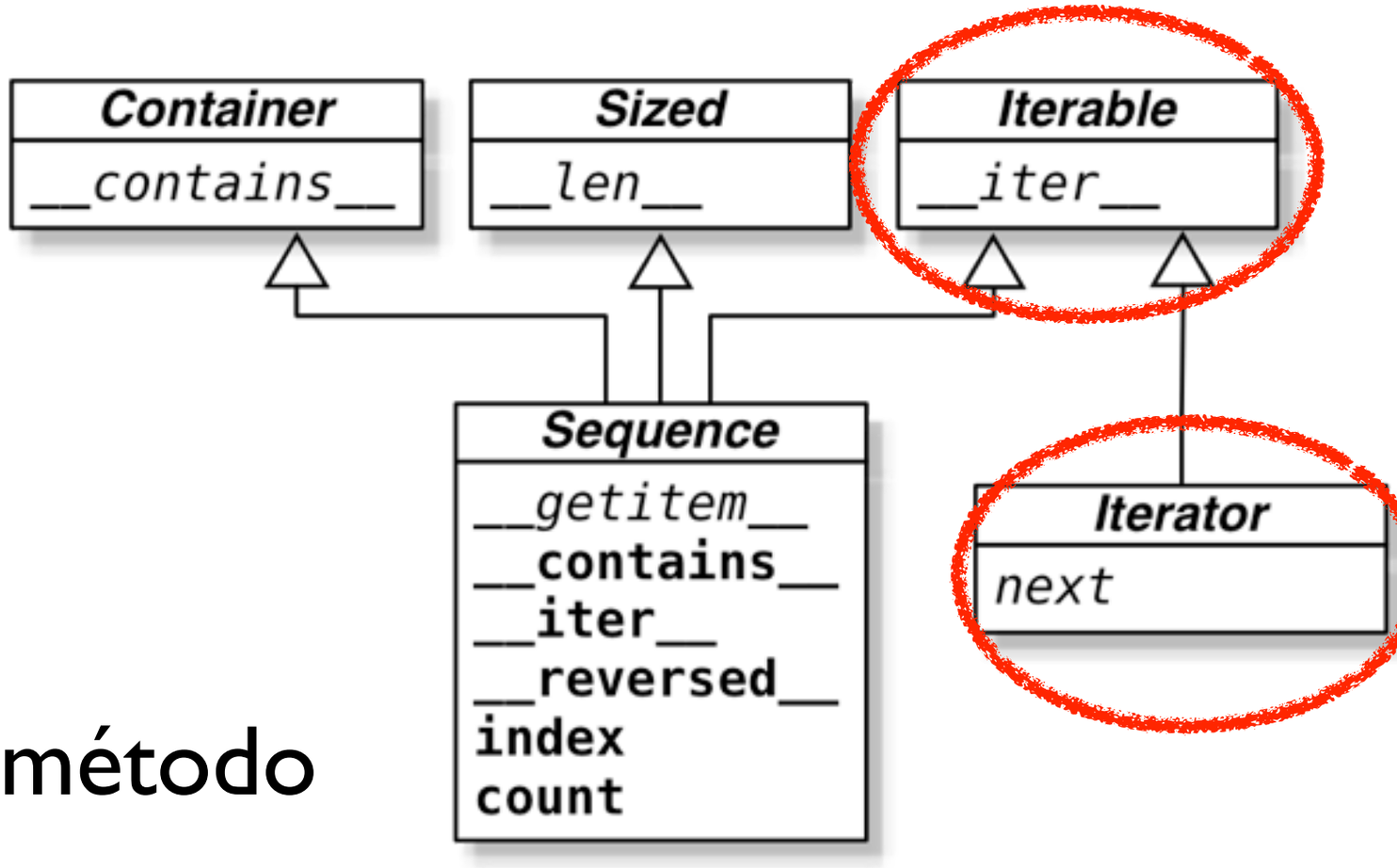



```
>>> t = Trem(4)
>>> 'vagao #2' in t
True
>>> 'vagao #5' in t
False
>>> for i in reversed(t): print i
...
vagao #4
vagao #3
vagao #2
vagao #1
>>> t.index('vagao #2')
1
>>> t.index('vagao #7')
Traceback (most recent call last):
  ...
ValueError
```

```
from collections import Seque

class Trem(Sequence):
    def __init__(self, num_va
        self.num_vagoes = num
    def __len__(self):
        return self.num_vagoe
    def __getitem__(self, pos
        indice = pos if pos >
```

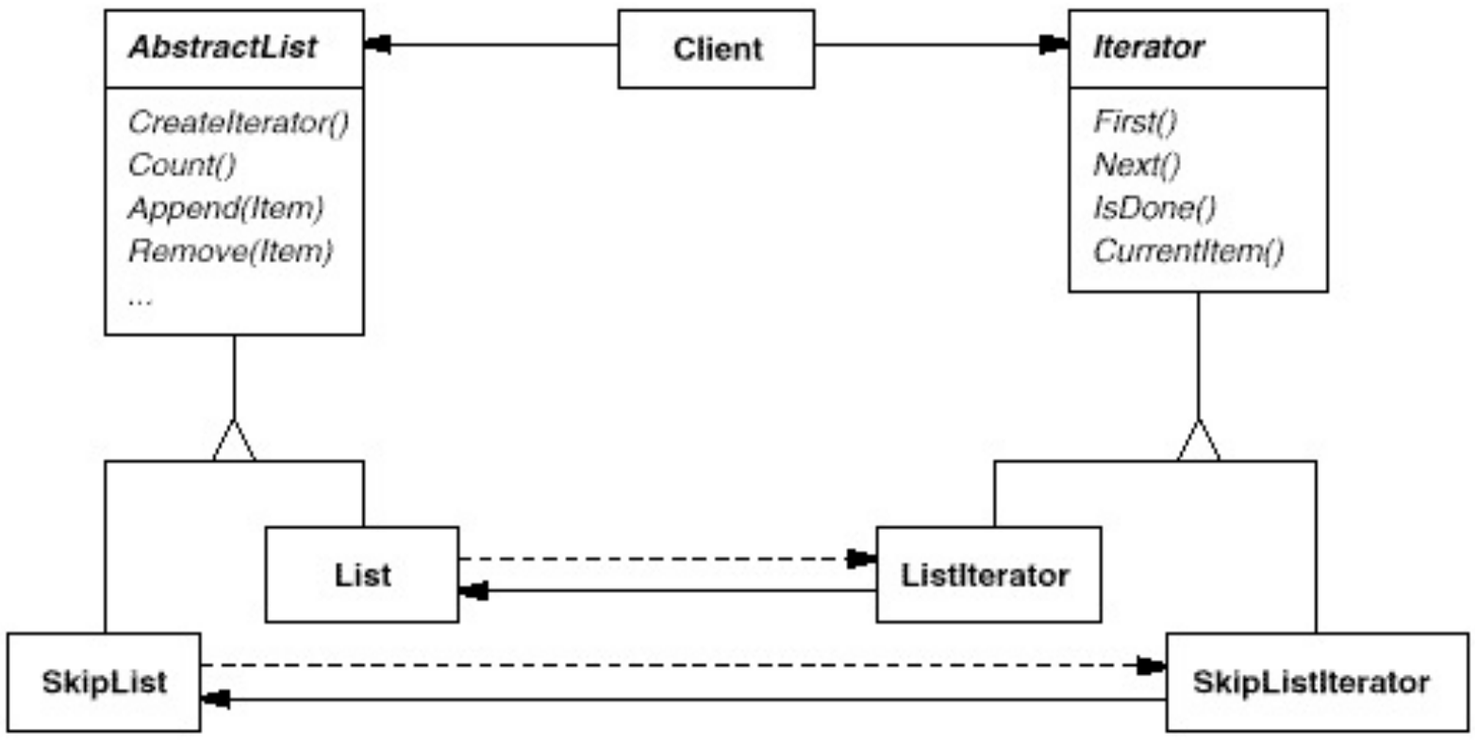
Interface Iterable

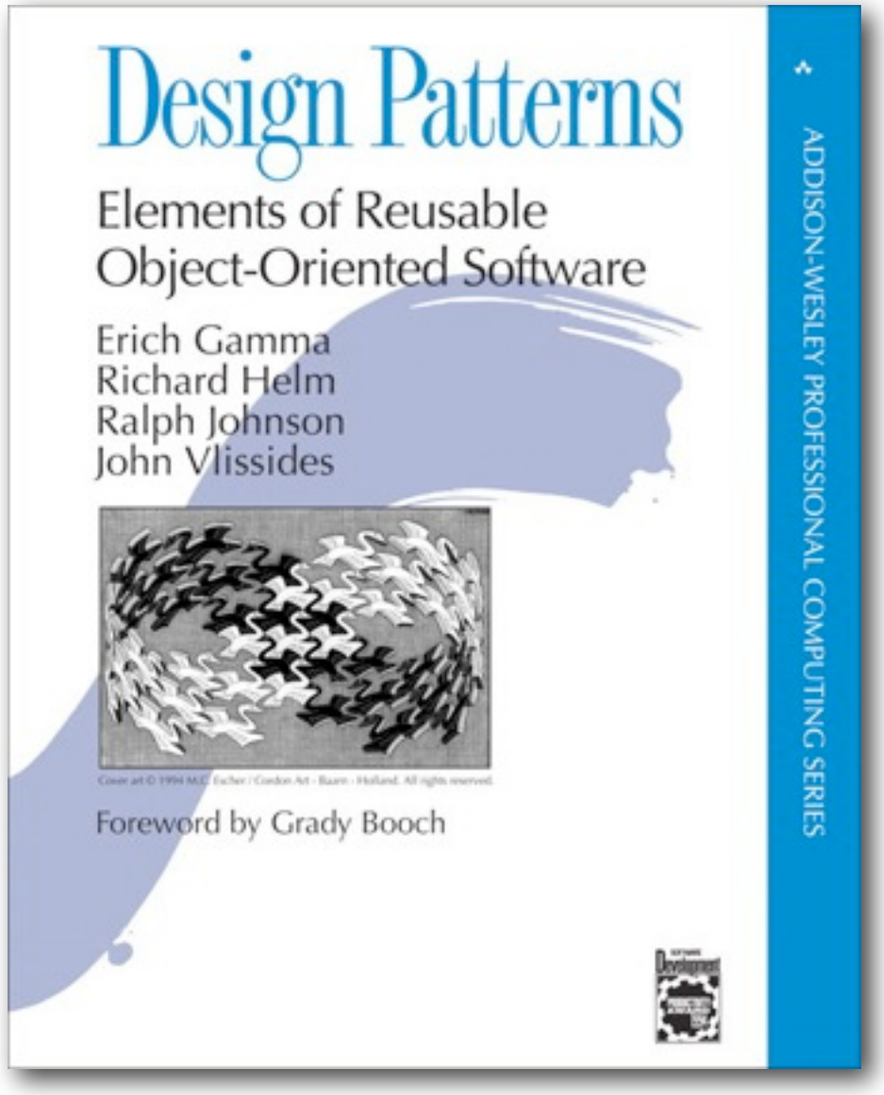


- Iterable provê um método __iter__
- O método __iter__ devolve uma instância de Iterator

Iterator é...

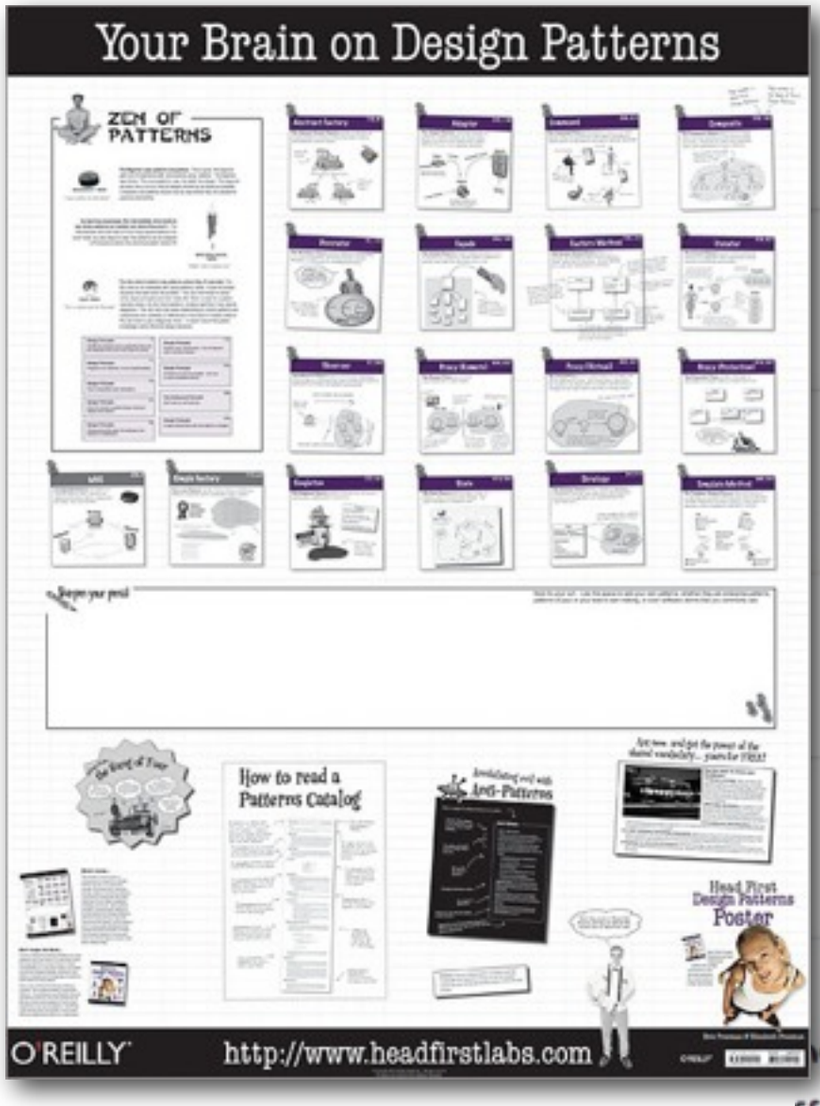
- um padrão de projeto





Design Patterns

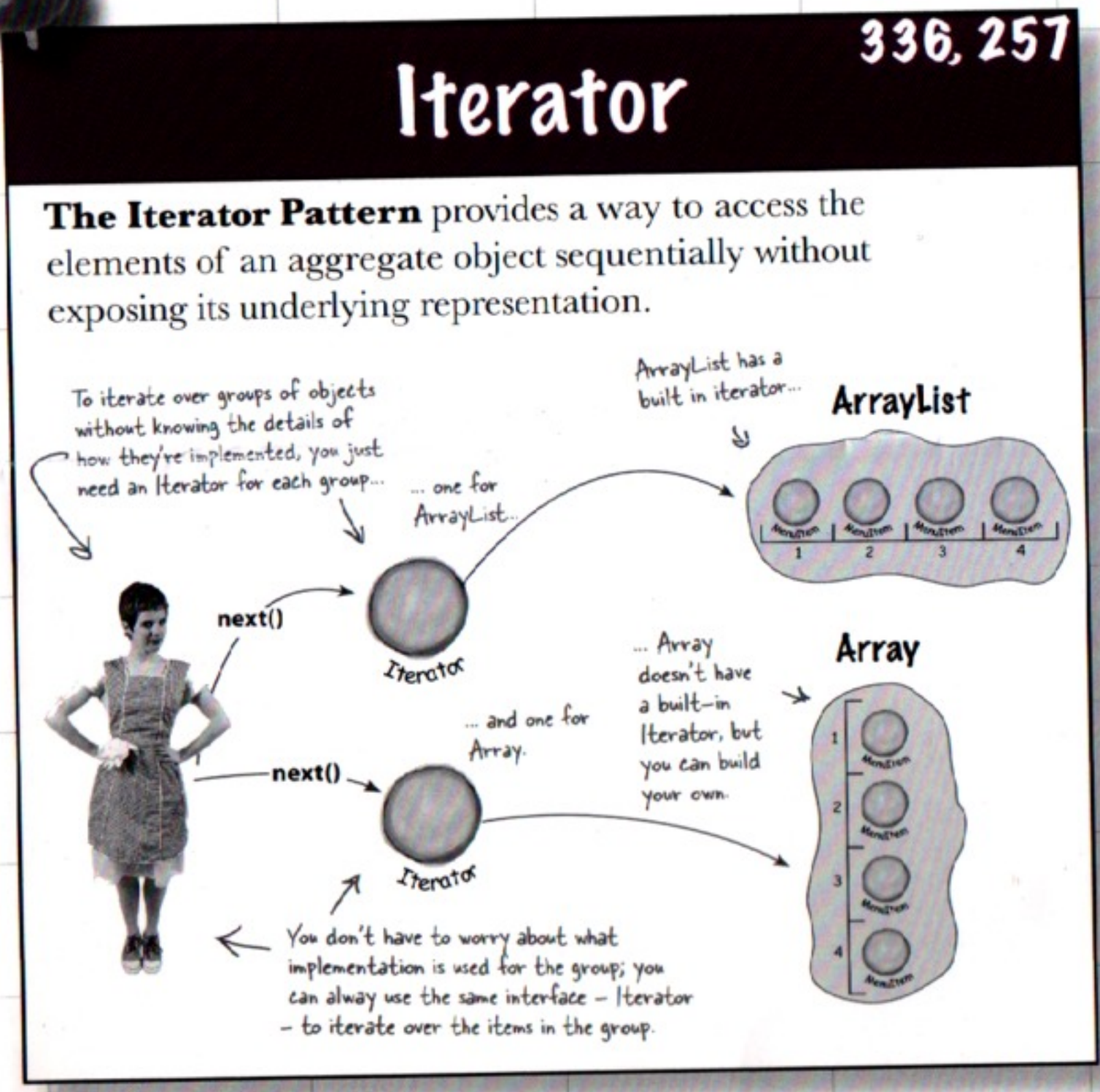
Gamma, Helm, Johnson & Vlissides
Addison-Wesley,
ISBN 0-201-63361-2



Head First Design Patterns Poster

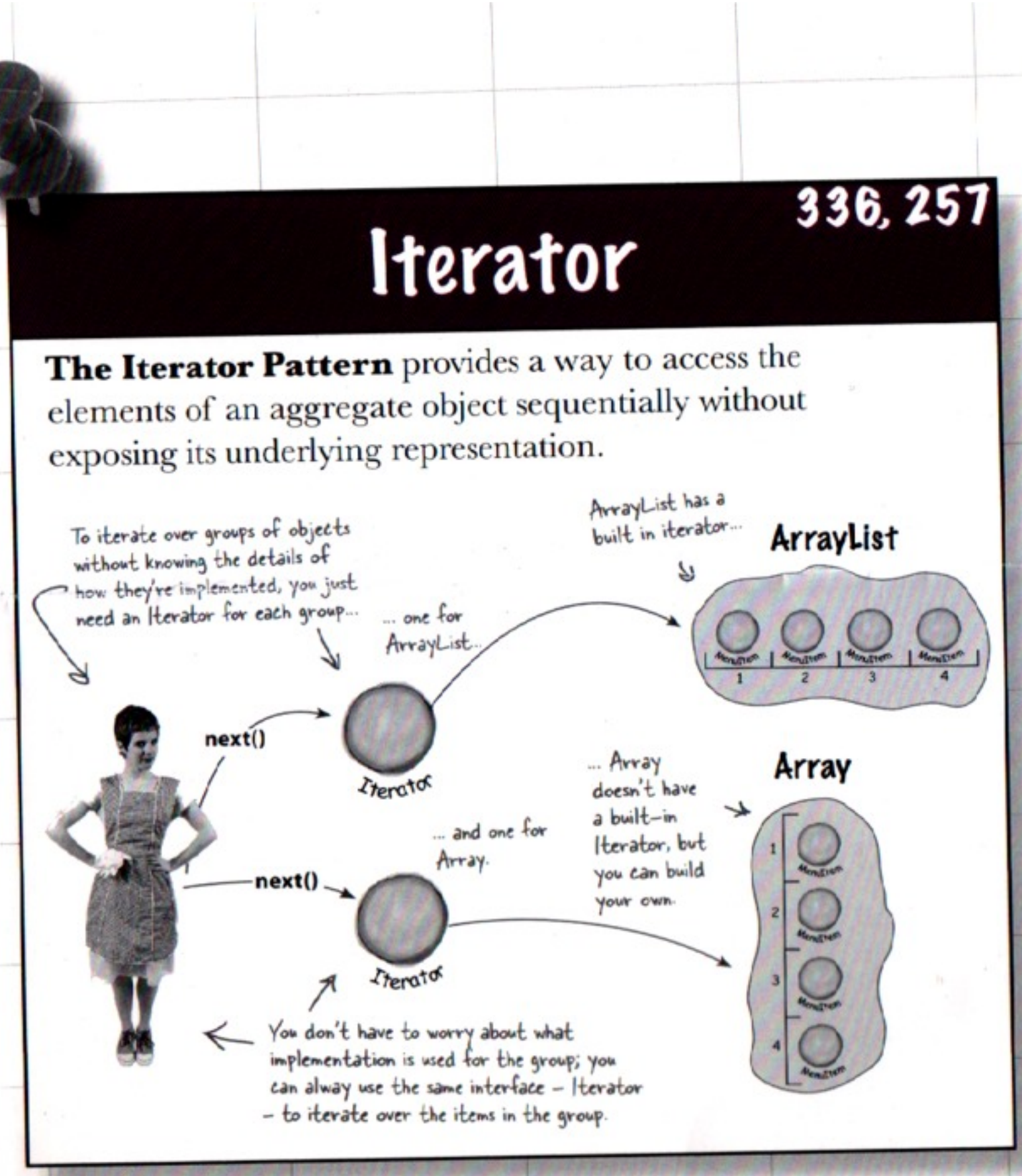
O'Reilly,
ISBN 0-596-10214-3

@pythonprobr



O padrão **Iterator** permite acessar os itens de uma coleção sequencialmente, isolando o cliente da implementação da coleção.

Head First Design Patterns Poster
O'Reilly,
ISBN 0-596-10214-3



Trem com iterator

```
class Trem(object):
    def __init__(self, num_vagoes):
        self.num_vagoes = num_vagoes
    def __iter__(self):
        return IteradorTrem(self.num_vagoes)

class IteradorTrem(object):
    def __init__(self, num_vagoes):
        self.atual = 0
        self.ultimo_vagao = num_vagoes - 1
    def next(self):
        if self.atual <= self.ultimo_vagao:
            self.atual += 1
            return 'vagao #%s' % (self.atual)
        else:
            raise StopIteration()
```

iter(t)

```
>>> t = Trem(4)
>>> for vagao in t:
...     print(vagao)
vagao #1
vagao #2
vagao #3
vagao #4
```

- for vagao in t:
  - invoca iter(t)
    - devolve IteradorTrem
  - invoca itrem.next() até que ele levante StopIteration

Em Python, um iterável é...

- Um objeto a partir do qual a função **iter** consegue obter um iterador.
- A chamada iter(x):
  - invoca x.__iter__() para obter um iterador
  - **ou**, se x.__iter__ não existe:
    - fabrica um iterador que acessa os itens de x sequenciamente fazendo x[0], x[1], x[2] etc.

interface Iterable

protocolo de sequência

Iteração em C (exemplo 2)

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
        int i;
        for(i = 0; i < argc; i++)
                printf("%d : %s\n", i, argv[i]);
        return 0;
}
```

```
$ ./args2 alfa bravo charlie
0 : ./args2
1 : alfa
2 : bravo
3 : charlie
```

Iteração em Python (ex. 2)

```
import sys

for i, arg in enumerate(sys.argv):
    print i, ':', arg
```

```
$ python args2.py alfa bravo charlie
0 : args2.py
1 : alfa
2 : bravo
3 : charlie
```

Iterator x generator

- Gerador é uma generalização do iterador
  - assunto para a aula 4...