Métodos especiais

Como usar bem os métodos __dunder__

Luciano Ramalho
ramalho@python.pro.br

PythonPro


PythonPro

Primeiro exemplo

O baralho polimórfico

O baralho polimórfico da palestra
“OO em Python sem sotaque”

PythonPro

Fazendo maço

```
>>> baralho = Baralho()
>>> len(baralho)
52
>>> baralho[0]
Carta(valor='2', naipe='paus')
>>> baralho[-1]
>>> from random import choice
>>> choice(baralho)
Carta(valor='4', naipe='paus')
>>> choice(baralho)
Carta(valor='A', naipe='espadas')
```

acesso por índice

sorteio

PythonPro

Fazendo maço

fatiamento!

```
>>> baralho[:5]
[Carta(valor='2', naipe='paus'),
 Carta(valor='3', naipe='paus'),
 Carta(valor='4', naipe='paus'),
 Carta(valor='5', naipe='paus'),
 Carta(valor='6', naipe='paus')]
>>> baralho[-3:]
[Carta(valor='Q', naipe='espadas'),
 Carta(valor='K', naipe='espadas'),
 Carta(valor='A', naipe='espadas')]
>>>
```

PythonPro

Fazendo maço (2)

iteração!!

```
>>> for carta in baralho:
...     print(carta)
...
Carta(valor='2', naipe='paus')
Carta(valor='3', naipe='paus')
Carta(valor='4', naipe='paus')
...
Carta(valor='Q', naipe='espadas')
Carta(valor='K', naipe='espadas')
Carta(valor='A', naipe='espadas')
>>>
```

PythonPro

Fazendo maço (3)

iteração reversa!!!

```
>>> for carta in reversed(baralho):
...     print(carta)
...
Carta(valor='A', naipe='espadas')
Carta(valor='K', naipe='espadas')
Carta(valor='Q', naipe='espadas')
...
Carta(valor='4', naipe='paus')
Carta(valor='3', naipe='paus')
Carta(valor='2', naipe='paus')
>>>
```

PythonPro

Fazendo maço (4)

enumeração!!!!

```
>>> for n, carta in enumerate(baralho, 1):
...     print(format(n, '2'), card)
...
 1 Carta(valor='2', naipe='paus')
 2 Carta(valor='3', naipe='paus')
 3 Carta(valor='4', naipe='paus')
...
50 Carta(valor='Q', naipe='espadas')
51 Carta(valor='K', naipe='espadas')
52 Carta(valor='A', naipe='espadas')
>>>
```

PythonPro

Tudo isso, quanto custa?

11 linhas!

```
import collections

Carta = collections.namedtuple('Carta', ['valor', 'naipe'])

class Baralho:
    valores = [str(n) for n in range(2,11)] + list('JQKA')
    naipes = 'paus ouros copas espadas'.split()
    def __init__(self):
        self.cartas = [Carta(v, n) for n in self.naipes for v in self.valores]
    def __len__(self):
        return len(self.cartas)
    def __getitem__(self, posicao):
        return self.cartas[posicao]
```

PythonPro

note: Baralho herda de object

Protocolo de sequência

- Protocolo é uma interface definida por convenção, e não formalmente verificada pelo compilador
  - Pode ser implementada parcialmente
- Em Python, o protocolo de sequência tem apenas dois métodos:
  - **s.__getitem__(chave) ↔ s[chave]**
  - **s.__len__() ↔ len(s)**

Documentação oficial

- **Language Reference > Data model**
  - http://docs.python.org/dev/reference/datamodel.html
- Seção **3.2 - The standard type hierarchy**
- Seção **3.3 - Special method names**

PythonPro

Table Of Contents

3. Data model

3.1. Objects, values and types

Objects are Python's abstraction for data. All data in a Python program is represented by objects or by relations between objects. (In a sense, and in conformance to Von Neumann's model of a "stored program computer," code is also represented by objects.)

Every object has an identity, a type and a value. An object's *identity* never changes once it has been created; you may think of it as the object's address in memory. The '`is`' operator compares the identity of two objects; the `id()` function returns an integer representing its identity.

> **CPython implementation detail:** For CPython, `id(x)` is the memory address where `x` is stored.

An object's type determines the operations that the object supports (e.g., "does it have a length?") and also defines the possible values for objects of that type. The `type()` function returns an object's type (which is an object itself). Like its identity, an object's *type* is also unchangeable. [1]

3.2. The standard type hierarchy

Below is a list of the types that are built into Python. Extension modules (written in C, Java, or other languages, depending on the implementation) can define additional types. Future versions of Python may add types to the type hierarchy (e.g., rational numbers, efficiently stored arrays of integers, etc.), although such additions will often be provided via the standard library instead.

Some of the type descriptions below contain a paragraph listing 'special attributes.' These are attributes that provide access to the implementation and are not intended for general use. Their definition may change in the future.

None

This type has a single value. There is a single object with this value. This object is accessed through the built-in name `None`. It is used to signify the absence of a value in many situations, e.g., it is returned from functions that don't explicitly return anything. Its truth value is false.

NotImplemented

This type has a single value. There is a single object with this value. This object is accessed through the built-in name `NotImplemented`. Numeric methods and rich comparison methods may return this value if they do not implement the operation for the operands provided. (The interpreter will then try the reflected operation, or some other fallback, depending on the operator.) Its truth value is true.

Ellipsis

Sequences

These represent finite ordered sets indexed by non-negative numbers. The built-in function `len()` returns the number of items of a sequence. When the length of a sequence is *n*, the index set contains the numbers 0, 1, ..., *n*-1. Item *i* of sequence *a* is selected by `a[i]`.

Sequences also support slicing: `a[i:j]` selects all items with index *k* such that *i* `<=` *k* `<` *j*. When used as an expression, a slice is a sequence of the same type. This implies that the index set is renumbered so that it starts at 0.

Some sequences also support "extended slicing" with a third "step" parameter: `a[i:j:k]` selects all items of *a* with index *x* where `x = i + n*k`, *n* `>=` `0` and *i* `<=` *x* `<` *j*.

Sequences are distinguished according to their mutability:

Immutable sequences

An object of an immutable sequence type cannot change once it is created. (If the object contains references to other objects, these other objects may be mutable and may be changed; however, the collection of objects directly referenced by an immutable object cannot change.)

The following types are immutable sequences:

Strings

A string is a sequence of values that represent Unicode codepoints. All the codepoints in range `U+0000 - U+10FFFF` can be represented in a

3.3. Special method names

A class can implement certain operations that are invoked by special syntax (such as arithmetic operations or subscripting and slicing) by defining methods with special names. This is Python's approach to *operator overloading*, allowing classes to define their own behavior with respect to language operators. For instance, if a class defines a method named `__getitem__()`, and `x` is an instance of this class, then `x[i]` is roughly equivalent to `type(x).__getitem__(x, i)`. Except where mentioned, attempts to execute an operation raise an exception when no appropriate method is defined (typically `AttributeError` or `TypeError`).

When implementing a class that emulates any built-in type, it is important that the emulation only be implemented to the degree that it makes sense for the object being modelled. For example, some sequences may work well with retrieval of individual elements, but extracting a slice may not make sense. (One example of this is the `NodeList` interface in the W3C's Document Object Model.)

3.3.1. Basic customization

`object.`**`__new__`**`(`*`cls`*`[, ...])`

Called to create a new instance of class *cls*. `__new__()` is a static method (special-cased so you need not declare it as such) that takes the class of which an instance was requested as its first argument. The remaining arguments are those passed to the object constructor expression (the call to the class). The return value of `__new__()` should be the new object instance (usually an

3.3. Special method names

[...]
When implementing a class that emulates any built-in type, it is important that the emulation only be implemented to the degree that it makes sense for the object being modelled. For example, some sequences may work well with retrieval of individual elements, but extracting a slice may not make sense.

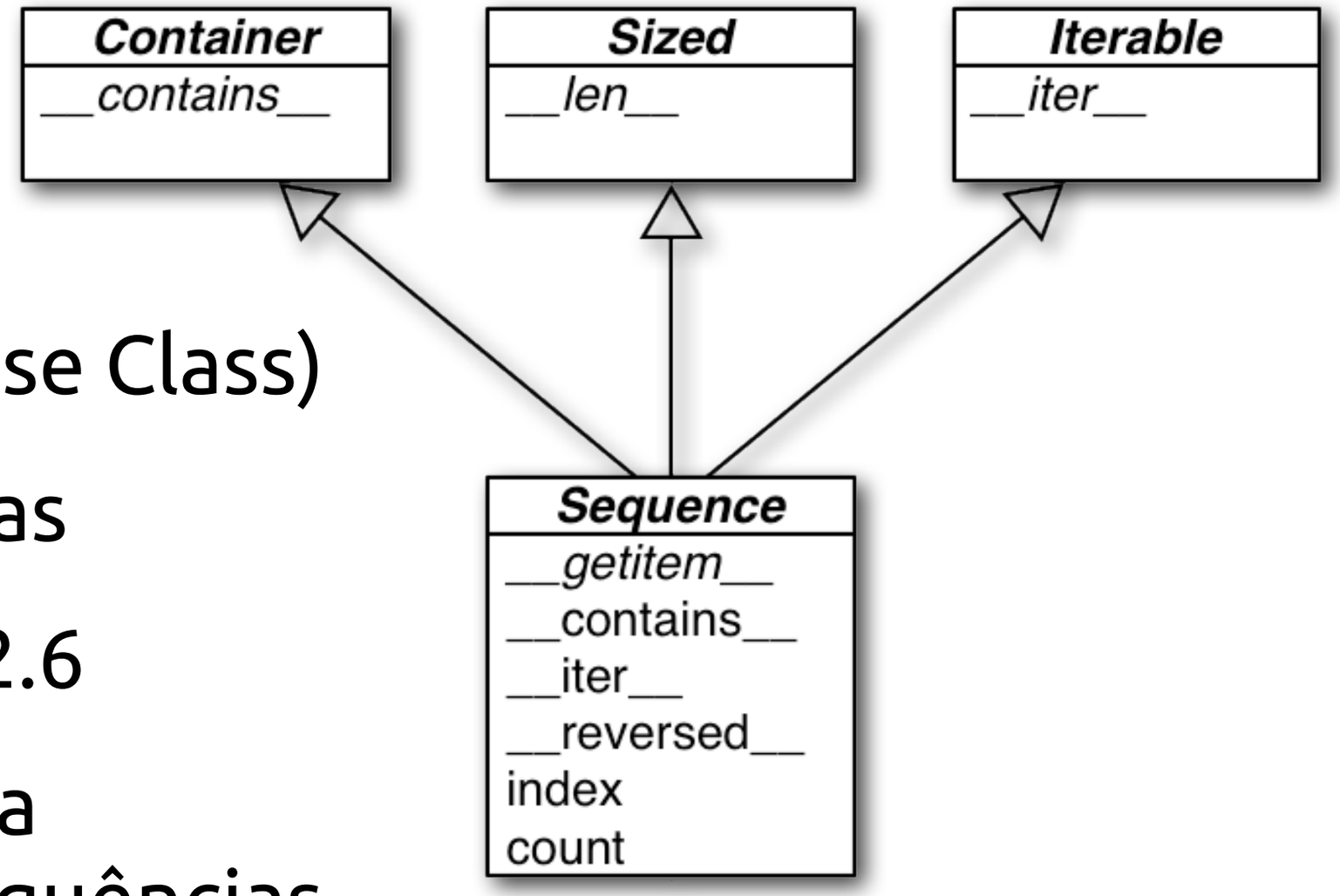
3.3. Nomes de métodos especiais

[...]
Ao implementar uma classe que emula qualquer tipo embutido (*built-in*), é importante que a emulação seja feita apenas na medida em que faz sentido para o objeto que está sendo modelado. Por exemplo, algumas sequências podem funcionar bem com a recuperação de elementos individuais, mas extrair uma fatia pode não fazer sentido.

Interfaces × protocolos × ABC

- **Interfaces** verificadas pelo compilador: não temos
- **Protocolos**: definidos pela documentação, não declarados e muito menos verificados
  - Duck typing: o que interessa é o comportamento, não o DNA do bicho
- **ABC**: Abstract Base Classes
  - Permitem especificar interfaces e declarar classes que as implementam

collections.abc.Sequence



- ABC (Abstract Base Class)
  - classes abstratas
  - desde Python 2.6
  - torna explícita a interface de sequências (e outras coleções)

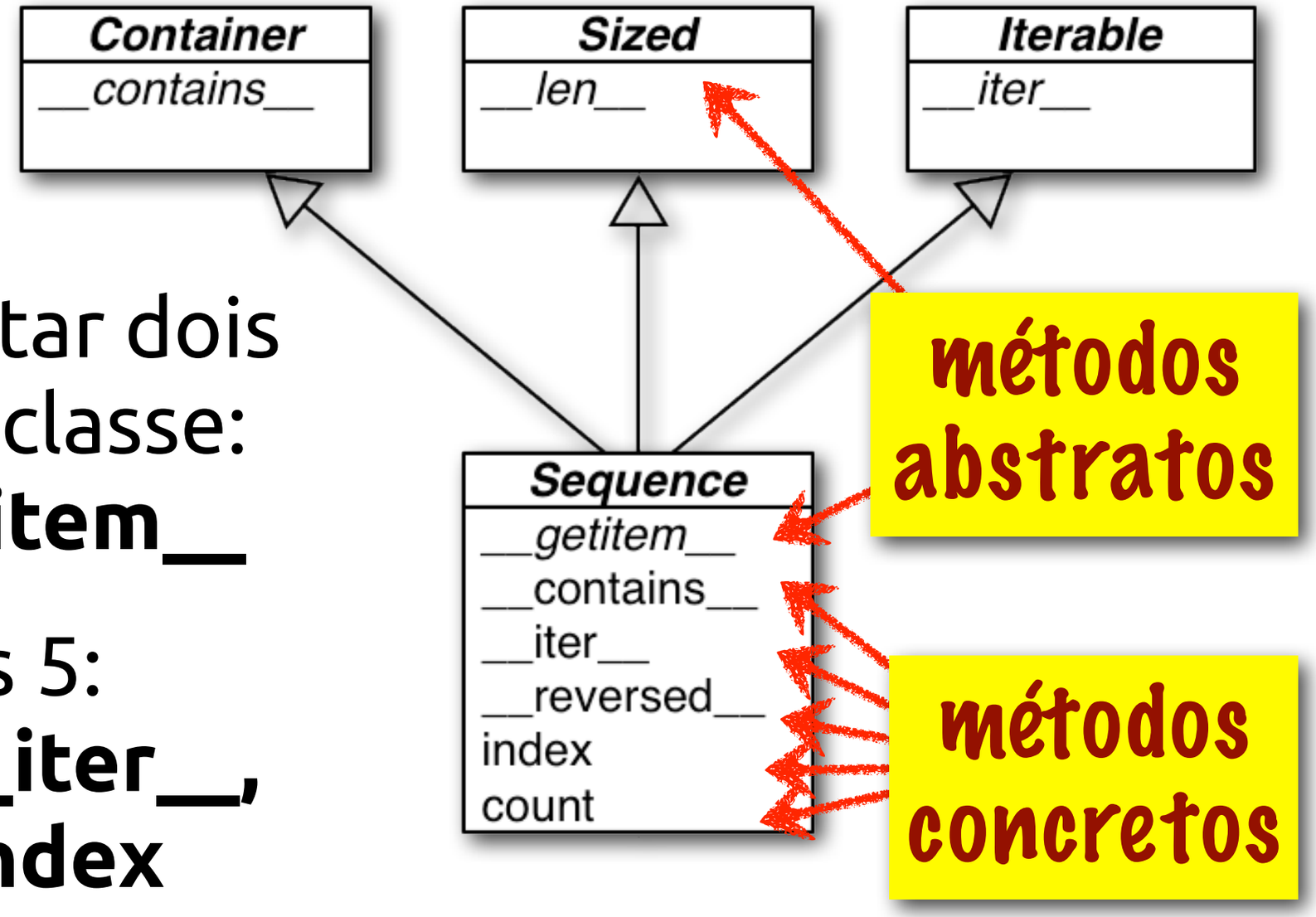
Baralho2: herdando de abc.Sequence

collections.abc.Sequence

```
import collections

Carta = collections.namedtuple('Carta', ['valor', 'naipe'])

class Baralho2(collections.abc.Sequence):
    valores = [str(n) for n in range(2,11)] + list('JQKA')
    naipes = 'paus ouros copas espadas'.split()
    def __init__(self):
        self.cartas = [Carta(v, n) for n in self.naipes for v in self.valores]
    def __len__(self):
        return len(self.cartas)
    def __getitem__(self, posicao):
        return self.cartas[posicao]
```

PythonPro

collections.abc.Sequence



- Basta implementar dois métodos na sua classe: **__len__** e **__getitem__**
- Você ganha mais 5: **__contains__, __iter__, __reversed__, index** e **count**

Baralho2 em ação

```
>>> baralho = Baralho2()
>>> zape = Carta(valor='4', naipe='paus')
>>> zape in baralho
True
>>> baralho.count(zape)
1
>>> baralho.index(zape)
2
>>> baralho[:3]
[Carta(valor='2', naipe='paus'),
 Carta(valor='3', naipe='paus'),
 Carta(valor='4', naipe='paus')]
```

__contains__

count

index

PythonPro

Verificação da implementação

- Python só permite que você crie instâncias de classes concretas (que implementam todos os métodos abstratos)

```
>>> baralho = BaralhoX()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class
BaralhoX with abstract methods __len__
```

Verificação da implementação

- Quando você herda de uma classe abstrata, Python não verifica se a sua classe derivada implementa todos os métodos abstratos, pois sua classe pode ser abstrata também
- Por isso a verificação é feita somente no momento da instanciação

Embaralhar com random

- Python vem com as pilhas incluídas!

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

Embaralhar com random?

- Mas a função shuffle não funciona com um baralho...

```
>>> shuffle(b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File ".../python3.3/random.py", line 265, in shuffle
    x[i], x[j] = x[j], x[i]
TypeError: 'Baralho' object does not support item assignment
>>>
```

'Baralho' não suporta atribuição a item

PythonPro

Embaralhar com random!

```
>>> def enfiar(baralho, pos, carta):
...     baralho.cartas[pos] = carta
...
>>> Baralho.__setitem__ = enfiar
>>> shuffle(b)
>>> b[:5]
[Carta(valor='K', naipe='espadas'),
 Carta(valor='2', naipe='espadas'),
 Carta(valor='3', naipe='copas'),
 Carta(valor='9', naipe='paus'),
 Carta(valor='Q', naipe='copas')]
>>>
```

monkey patch

agora funciona!

PythonPro

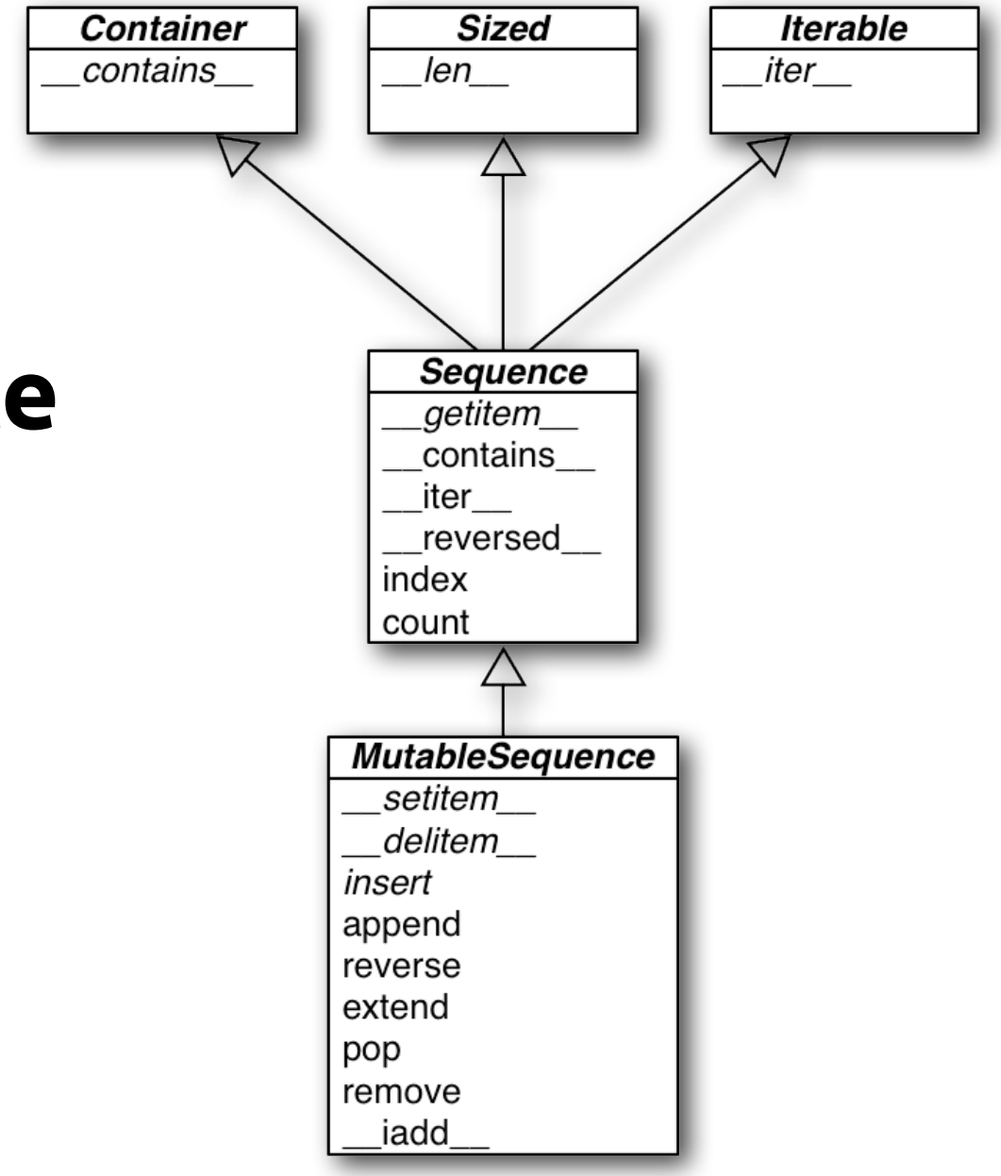
Baralho mutável

```
import collections

Carta = collections.namedtuple('Carta', ['valor', 'naipe'])

class Baralho:
    valores = [str(n) for n in range(2,11)] + list('JQKA')
    naipes = 'paus ouros copas espadas'.split()
    def __init__(self):
        self.cartas = [Carta(v, n) for n in self.naipes for v in self.valores]
    def __len__(self):
        return len(self.cartas)
    def __getitem__(self, posicao):
        return self.cartas[posicao]
    def __setitem__(self, posicao, carta):
        self.cartas[posicao] = carta
```

__setitem__

.abc.MutableSequence

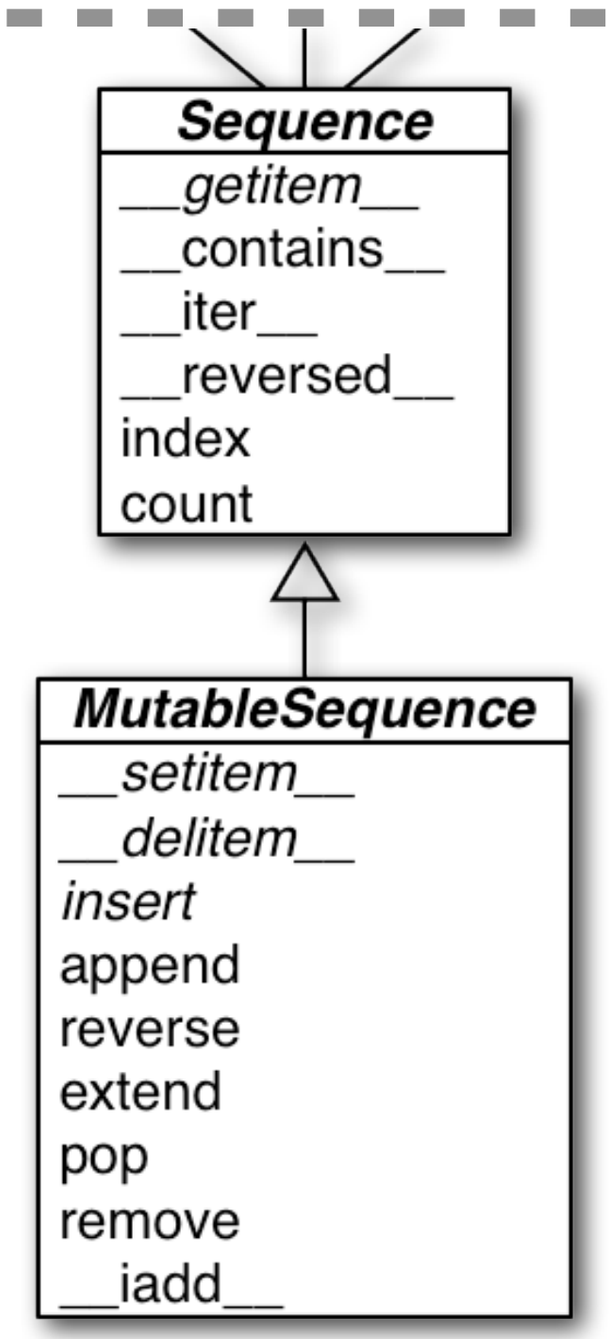
- Subclasse de **collections.abc.Sequence**
- Principal diferença: método abstrato ***__setitem__***



PythonPro

.abc.MutableSequence

- Aplicações simples do protocolo implementam só **__setitem__**
- Uma subclasse concreta tem que implementar **__setitem__**, **__delitem__** e **insert**
- A classe abstrata implementa outros 6 métodos concretos



Métodos especiais: regras básicas

Sintaxe e semântica

- Nomes no formato **__dunder__**
- Definem os protocolos fundamentais da linguagem, com suporte sintático
  - mecanismos básicos: iteração, inicialização de objetos, acesso a atributos, formatação, invocação, hash, contextos (**with**)
  - operadores aritméticos e lógicos
  - tipos numéricos, emulação de coleções

Como usar

- Normalmente não é o seu código que chama os métodos especiais, e sim o **interpretador Python**
  - Quase sempre a chamada é implícita
    - ex: laço **for i in x** → **iter(x)** → **x.__iter__()**
- Se precisar usar estes serviços, evite invocar diretamente o método especial
  - Use **iter(x)** em vez de **x.__iter__()**

Iteração: a relação entre `for`, `iter(x)` e `x.__iter__()`

oficinas Turing

python™

como já vimos na aula 3

De iteradores a geradores:

evolução de um *design pattern* em Python

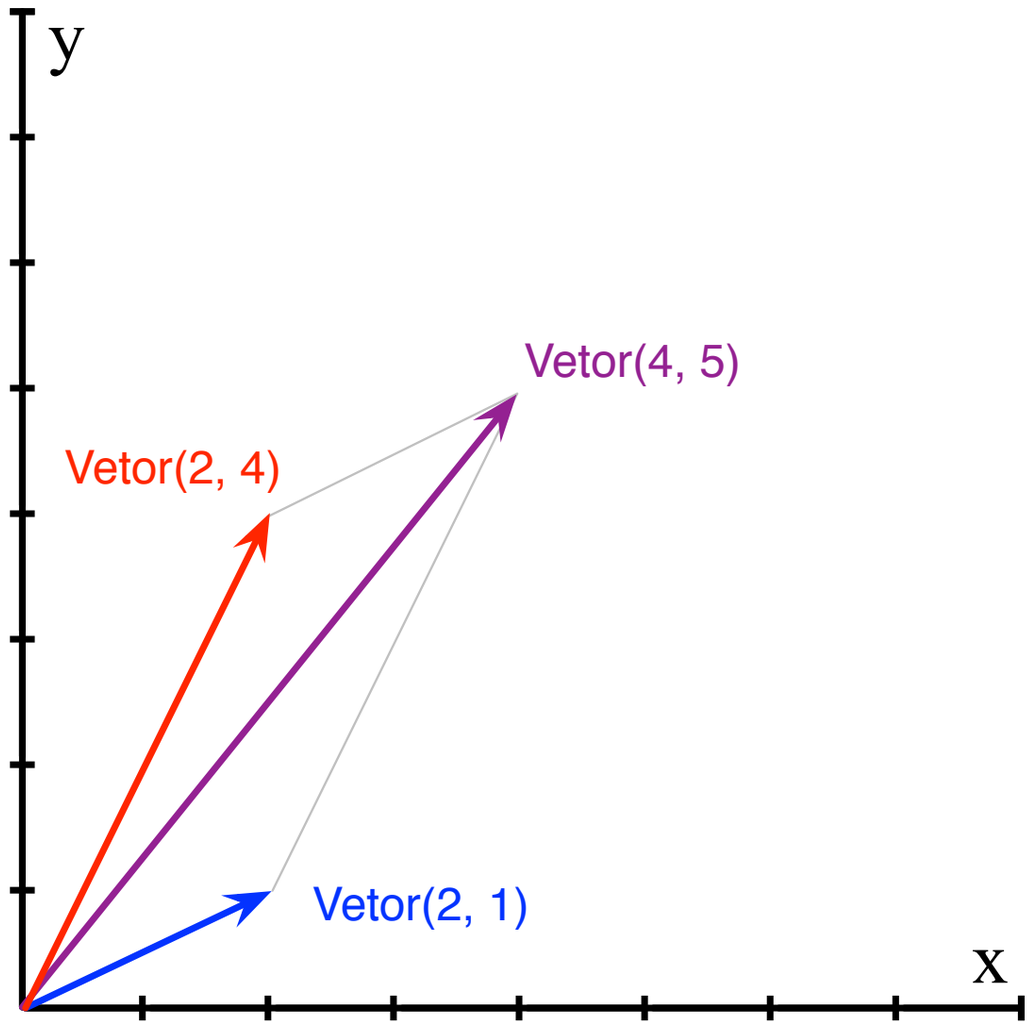
Luciano Ramalho
luciano@ramalho.org
@ramalhoorg

maio/2013

http://bit.ly/py-geradores

Sobrecarga de operadores aritméticos

Exemplo: Vetor 2d



- Campos: x, y
- Métodos:
  - distancia
  - abs (distância até 0,0)
  - + (__add__)
  - * (__mul__) escalar

PythonPro

Vetor

```python
from math import sqrt

class Vetor:

    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):
        x = self.x + v2.x
        y = self.y + v2.y
        return Vetor(x, y)

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

```
>>> 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)
```

PythonPro

Operadores reversos

Um problema

```python
from math import sqrt

class Vetor:

    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(

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

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

```
>>> vel = Vetor(3, 4)
>>> vel * 3
Vetor(9, 12)
>>> 3 * vel
Traceback (most recent call last):
    ...
TypeError: unsupported operand
type(s) for *: 'int' and 'Vetor'
```

Isso viola a propriedade comutativa da multiplicação!

PythonPro

Como funciona x * y

- O interpretador invoca **x.__mul__(y)**
- Se **x.__mul__** não existe ou retorna o valor especial **NotImplemented:**
  - O interpretador invoca **y.__rmul__(x)**
  - Se **y.__rmul__** não existe ou retorna o valor **NotImplemented**, o interpretador levanta a exceção **TypeError**

Esse padrão chama-se double-dispatch. Fonte: http://bit.ly/st-double-dispatch

Solução

```python
from math import sqrt

class Vetor:

    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):
        x = self.x + v2.x
        y = self.y + v2.y
        return Vetor(x, y)

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

    def __rmul__(self, n):
        return self * n
```

```
>>> vel = Vetor(3, 4)
>>> vel * 3
Vetor(9, 12)
>>> 3 * vel
Vetor(9, 12)
>>>
```

operador reverso

Outro problema

```python
from math import sqrt

class Vetor:

    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):
        x = self.x + v2.x
        y = self.y + v2.y
        return Vetor(x, y)

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

    def __rmul__(self, n):
        return self * n
```

```
>>> Vetor(1, 2) * Vetor(3, 4)
Vetor(Vetor(3, 4), Vetor(6, 8))
```

resultado sem sentido!

Solução 2

```python
from math import sqrt
from numbers import Real

class Vetor:

    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):
        x = self.x + v2.x
        y = self.y + v2.y
        return Vetor(x, y)

    def __mul__(self, n):
        if isinstance(n, Real):
            return Vetor(self.x*n, self.y*n)
        else:
            return NotImplemented

    def __rmul__(self, n):
        return self * n
```

```
>>> Vetor(1, 2) * Vetor(3, 4)
Traceback (most recent call last):
  ...
TypeError: unsupported operand
type(s) for *: 'Vetor' and 'Vetor'
>>>
```

verifica se é escalar

se não, delega para o interpretador

Atribuição aumentada

Atribuição aumentada

- Operadores: `+=` `-=` `*=` `/=` `//=` `**=`
  `%=` `&=` `^=` `|=` `<<=` `>>=`
  - Para tipos imutáveis, **a += b** ↔ **a = a + b**
    - Portanto um novo objeto (**a+b**) é criado
  - Para tipos mutáveis, a implementação de métodos como **__iadd__** permite a modificação do objeto alvo da atribuição

Sem __imul__

```python
from math import sqrt
from numbers import Real

class Vetor:

    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):
        x = self.x + v2.x
        y = self.y + v2.y
        return Vetor(x, y)

    def __mul__(self, n):
        if isinstance(n, Real):
            return Vetor(self.x*n, self.y*n)
        else:
            return NotImplemented

    def __rmul__(self, n):
        return self * n
```

```
>>> vel = Vetor(3, 4)
>>> id(vel)
4313270928
>>> id_vel = id(vel)
>>> vel *= 5
>>> vel
Vetor(15, 20)
>>> id(vel) == id_vel
False
```

*= cria um novo objeto porque Vetor não tem __imul__

Com __imul__

```python
from math import sqrt
from numbers import Real

class Vetor:

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

    def __repr__(self):

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

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

    def __mul__(self, n):
        if isinstance(n, Real):
            return Vetor(self.x*n, self.y*n)
        else:
            return NotImplemented

    def __rmul__(self, n):
        return self * n

    def __imul__(self, n):
        self.x *= n
        self.y *= n
        return self
```

```
>>> vel = Vetor(3, 4)
>>> id(vel)
4313270928
>>> id_vel = id(vel)
>>> vel *= 5
>>> vel
Vetor(15, 20)
>>> id(vel) == id_vel
True
```

ainda o mesmo vetor

modifica e devolve self

Contextos gerenciados

Blocos with

o objeto-arquivo devolvido por open é o gerenciador de contexto

```
with open('exemplo.txt', encoding='utf-8') as arq:
    for linha in arq:
        linha = linha.rstrip()
        if linha:
            print(linha)
```

- Suporte sintático a gerenciadores de contexto: objetos que implementam **__enter__** e **__exit__**

PythonPro

Métodos do gerenciador de contexto

```
with open('exemplo.txt', encoding='utf-8') as arq:
    for linha in arq:
        linha = linha.rstrip()
        if linha:
            print(linha)
```

o __enter__ de file devolve self

- **object.__enter__(self)**: invocado no início do bloco, devolve um objeto que é atribuído à variável alvo do **with/as**

PythonPro

Métodos do gerenciador de contexto

```
with open('exemplo.txt', encoding='utf-8') as arq:
    for linha in arq:
        linha = linha.rstrip()
        if linha:
            print(linha)
```

o __exit__ de file invoca self.close()

- **object.__exit__(self, exc_type, exc_value, traceback)**: invocado no fim do bloco, recebe informações sobre exceções ou **None, None, None**; se devolver **True**, suprime a exceção levantada

Controle de atributos

Controle de atributos: métodos básicos

- Acionado quando **obj** não possui atributo **a**:
  - **obj.a ↔ obj.__getattr__('a')**

equivale ao methodMissing de Ruby

- Acionados sempre:
  - **obj.a = x ↔ obj.__setattr__('a', x)**
  - **del obj.a ↔ obj.__delattr__('a')**

PythonPro

Proxy: demonstração

```
>>> t = Treco(5, 'azul', 80)
>>> t.cor
'azul'
>>> t._valor
80.0
>>> t.preciosidade()
16.0
>>> pr = Proxy(t)
>>> pr.cor
'azul'
>>> pr.preciosidade()
16.0
>>> pr._valor
Traceback (most recent call last):
  ...
AttributeError: Atributo inexistente ou protegido: '_valor'
>>>
```

Proxy

```python
class Treco:

    def __init__(self, peso, cor, valor):
        self.peso = peso
        self.cor = cor
        self._valor = float(valor)

    def preciosidade(self):
        return self._valor / self.peso

class Proxy:

    def __init__(self, embrulhado):
        self._embrulhado = embrulhado

    def __getattr__(self, nome_atr):
        if nome_atr.startswith('_'):
            msg = 'Atributo inexistente ou protegido: %r'
            raise AttributeError(msg % nome_atr)
        else:
            return getattr(self._embrulhado, nome_atr)
```

```
>>> t = Treco(5, 'azu
>>> t.cor
'azul'
>>> t._valor
80.0
>>> t.preciosidade()
16.0
>>> pr = Proxy(t)
>>> pr.cor
'azul'
>>> pr.preciosidade()
16.0
>>> pr._valor
Traceback (most recent call last):
  ...
AttributeError: Atributo inexistente ou protegido: '_valor'
>>>
```

Controle de atributos: métodos avançados

- Acionado sempre:
  - **obj.a ↔ obj.__getattribute__('a')**

poderoso e difícil de implementar corretamente

- Para definição de **descritores**:
  - **__get__**
  - **__set__**
  - **__delete__**

como vimos na aula 1

PythonPro