



Introdução à Orientação a Objetos em Python (sem sotaque)

Luciano Ramalho

luciano@ramalho.org



Saiu em uma revista...



```
#!/usr/bin/env python
#Definição da classe Animal que herda da classe Object.
class Animal(object):

    #Declaração de atributos
    _nome = "";
    _idade = 0;
    _sexo = 'M';

    #Método construtor da classe
    def __init__(self):
        print("este é um novo animal\n");

    #Declaração de métodos da classe
    def setNome(self, _nome):
        self._nome = _nome;

    def getNome(self):
        return self._nome;

    def setIdade(self, _idade):
        self._idade = _idade;

    def getIdade(self):
        return self._idade;

    def setSexo(self, _sexo):
        self._sexo = _sexo;

    def getSexo(self):
        return self._sexo;

#Tmprime em vídeo os atributos do Animal
```

Python com sotaque javanês



academia
python

```
#!/usr/bin/env python
#Definição da classe Animal que herda da classe Object.
class Animal(object):

    #Declaração de atributos
    _nome = "";
    _idade = 0;
    _sexo = 'M';

    #Método construtor da classe
    def __init__(self):
        print("este é um novo animal\n");

    #Declaração de métodos da classe
    def setNome(self, _nome):
        self._nome = _nome;

    def getNome(self):
        return self._nome;

    def setIdade(self, _idade):
        self._idade = _idade;

    def getIdade(self):
        return self._idade;

    def setSexo(self, _sexo):
        self._sexo = _sexo;

    def getSexo(self):
        return self._sexo;

#Tmprime em vídeo os atributos do Animal
```

Não use
; no final
das linhas

Não é um erro
sintático, mas é
desnecessário
e portanto
deselegante

```

#!/usr/bin/env python
#Definição da classe Animal que herda da classe Object.
class Animal(object):

    #Declaração de atributos
    _nome = "";
    _idade = 0;
    _sexo = 'M';

    #Método construtor da classe
    def __init__(self):
        print("este é um novo animal\n");

    #Declaração de métodos da classe
    def setNome(self, _nome):
        self._nome = _nome;

    def getNome(self):
        return self._nome;

    def setIdade(self, _idade):
        self._idade = _idade;

    def getIdade(self):
        return self._idade;

    def setSexo(self, _sexo):
        self._sexo = _sexo;

    def getSexo(self):
        return self._sexo;

#Tmprime em vídeo os atributos do Animal

```

Esses
não são
métodos
da classe

Esse método agem sobre as instâncias (note o self). Métodos de classe são decorados com @classmethod.

```
#!/usr/bin/env python
#Definição da classe Animal que herda da classe Object.
class Animal(object):

    #Declaração de atributos
    _nome = "";
    _idade = 0;
    _sexo = 'M';

    #Método construtor da classe
    def __init__(self):
        print("este é um novo animal\n");

    #Declaração de métodos da classe
    def setNome(self, _nome): ←
        self._nome = _nome;

    def getName(self): ←
        return self._nome;

    def setIdade(self, _idade): ←
        self._idade = _idade;

    def getIdade(self): ←
        return self._idade;

    def setSexo(self, _sexo): ←
        self._sexo = _sexo;

    def getSexo(self): ←
        return self._sexo;

#Tmprime em vídeo os atributos do Animal
```

Não
abuse de
getters e
setters

Em Python não usamos muitos getters e setters dessa forma.
Para controlar acesso usamos properties.

Características básicas

OO em Python se parece com...

- herança múltipla, como C++
- sobrecarga de operadores, como C++
- não obriga a criar classes, como C++
- tipagem dinâmica, como Smalltalk e Ruby
- tipagem dinâmica, mas não tipagem fraca

O que é tipagem fraca?

- conversão automática entre tipos
- comum em linguagens de scripting (JavaScript, Perl, PHP)
- uma fonte de bugs difíceis de localizar e tratar

```
< "9" + 10  
▶ "910"  
--  
< "9" * 10  
▶ 90  
--  
< "9" - 10  
▶ -1  
--  
< "9" + (-10)  
▶ "9-10"
```

Exemplos reais digitados
no console JavaScript do
Firefox 6.0

Atenção: Python
não é assim!



Tipagem dinâmica

- Variáveis (e parâmetros) não têm tipos declarados e podem ser associados a objetos de qualquer tipo em tempo de execução
- Também conhecida como “duck typing” (tipagem pato) nas comunidades Python, Ruby e Smalltalk

```
>>> def dobro(n):
...     '''devolve duas vezes n'''
...     return n + n
...
>>> dobro(7)
14
>>> dobro('Spam')
'SpamSpam'
>>> dobro([10, 20, 30])
[10, 20, 30, 10, 20, 30]
```

Tipagem dinâmica forte

- Python não faz conversão automática de tipos
 - exceções, por praticidade:
 - int → long → float
 - str → unicode

```
>>> "9" + 10
TypeError: cannot concatenate 'str' and 'int' objects
>>> "9" * 10
'9999999999'
>>> "9" - 10
TypeError: unsupported operand type(s) for -: 'str' and 'int'
>>> "9" + (-10)
TypeError: cannot concatenate 'str' and 'int' objects
```

Para quem conhece Java

- Python não tem interfaces
 - mas tem herança múltipla e classes abstratas
- Python não tem sobrecarga de métodos
 - mas tem sobrecarga de operadores e passagem de argumentos flexível
- Python não tem tipos primitivos
 - tudo é objeto (desde Python 2.2, dez/2001)

```
>>> 5 .__add__(3)  
8
```



Ex: 5 é uma instância de int

```
>>> 5.__add__(3)
8
>>> type(5)
<type 'int'>
>>> dir(5)
['__abs__', '__add__', '__and__', '__class__', '__cmp__',
 '__coerce__', '__delattr__', '__div__', '__divmod__',
 '__doc__', '__float__', '__floordiv__', '__format__',
 '__getattribute__', '__getnewargs__', '__hash__', '__hex__',
 '__index__', '__init__', '__int__', '__invert__',
 '__long__', '__lshift__', '__mod__', '__mul__', '__neg__',
 '__new__', '__nonzero__', '__oct__', '__or__', '__pos__',
 '__pow__', '__radd__', '__rand__', '__rdiv__',
 '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__',
 '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__',
 '__ror__', '__rpow__', '__rrshift__', '__rshift__',
 '__rsub__', '__rtruediv__', '__rxor__', '__setattr__',
 '__sizeof__', '__str__', '__sub__', '__subclasshook__',
 '__truediv__', '__trunc__', '__xor__', 'conjugate',
 'denominator', 'imag', 'numerator', 'real']
```

atributos de *int*

Sintaxe de classes

Classe com 3 métodos

```
class Contador(object):
    def __init__(this):
        this.contagem = {}

    def incluir(this, item):
        qtd = this.contagem.get(item, 0) + 1
        this.contagem[item] = qtd

    def contar(this, item):
        return this.contagem[item]
```

Classe com 3 métodos

```
class Contador(object):
    def __init__(this):
        this.contagem = {}

    def incluir(this, item):
        qtd = this.contagem.get(item, 0) + 1
        this.contagem[item] = qtd

    def contar(this, item):
        return this.contagem[item]
```

não usamos
this

Classe com 3 métodos

```
class Contador(object):
    def __init__(self):
        self.contagem = {}

    def incluir(self, item):
        qtd = self.contagem.get(item, 0) + 1
        self.contagem[item] = qtd

    def contar(self, item):
        return self.contagem[item]
```

usamos
self

Peculiaridade: **self** explícito

- Todos os métodos de instâncias devem declarar o **self** como primeiro parâmetro
- Todos os acessos a atributos (inclusive métodos) das instâncias devem ser feitos via referência explícita a **self**

```
class Contador(object):  
    def __init__(self):  
        self.contagem = {}  
  
    def incluir(self, item):  
        qtd = self.contagem.get(item, 0) + 1  
        self.contagem[item] = qtd
```

Ex: uso da classe Contador

```
>>> cont = Contador()  
>>> palavra = 'abacaxi'  
>>> for letra in palavra:  
...     cont.incluir(letra)  
...  
>>> for letra in sorted(set(palavra)):  
...     print letra, cont.contar(letra)  
...  
a 3  
b 1  
c 1  
i 1  
x 1
```

não existe o
operador *new*

Convenções

- classes devem herdar de `object` ou de outras classes que herdam de `object`
 - classes antigas ('old style') não seguem essa regra
 - não existem mais classes antigas em Python 3
- construtor deve se chamar `__new__` (uso raro)
- inicializador deve se chamar `__init__` (uso comum)
- o `__init__` faz o papel do que chamamos de construtor em outras linguagens



__init__ versus __new__

- __init__ recebe uma instância já construída, e sua função é inicializar os atributos da instância
 - isso é o mesmo que acontece em Java!
- Em Python temos mais controle: podemos sobrescrever o método __new__ da classe para interferir no processo de construção da instância
 - na prática é bem raro implementarmos __new__
 - técnica avançada de meta-programação

Instâncias abertas

- instâncias podem receber atributos dinamicamente
- por isso às vezes é útil criar classes vazias
 - não muito comum

Exemplo: classe vazia

```
>>> class Animal(object):
...     pass
...
>>> baleia = Animal()
>>> baleia.nome
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Animal' object has no attribute 'nome'
>>> baleia.nome = 'Moby Dick'
>>> baleia.peso = 1200
>>> print '{0.nome} ({0.peso:.1f})'.format(baleia)
Moby Dick (1200.0)
```

pass serve para
criar blocos vazios



Classes abertas?

- Em Ruby as classes são “abertas”, a sintaxe comum permite que um módulo redefina uma classe e adicione atributos a uma classe definida em outro módulo
- É uma violação do princípio “Open Closed” (SOLID)
 - “entidades de software (classes, módulos, funções etc.) devem ser abertas para extensão mas fechadas para modificação” (Bertrand Meyer, OO Sw. Construction)
- Em Python isso é chamado de “monkey patching”, usa uma sintaxe de reflexão explícita e não é considerada uma boa prática (mas acontece)



academia
python

Atributos de classe x de instância

```
>>> class Animal(object):
...     nome = 'Rex'
...
>>> cao = Animal()
>>> cao.nome
'Rex'
>>> cao.nome = 'Fido'
>>> cao.nome
'Fido'
>>> Animal.nome
'Rex'
>>> dino = Animal()
>>> dino.nome
'Rex'
```

atributo da classe

atributo encontrado
na classe

atributo criado
na instancia

nada mudou na classe

Métodos de classe/estáticos

- Indicados por decoradores de função

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

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



Herança

- no exemplo abaixo, `ContadorTolerante` extende `Contador`
- o método `contar` está sendo sobrescrito
- os métodos `__init__` e `incluir` são herdados

```
class Contador(object):
    def __init__(self):
        self.contagem = {}

    def incluir(self, item):
        qtd = self.contagem.get(item, 0) + 1
        self.contagem[item] = qtd

    def contar(self, item):
        return self.contagem[item]
```

```
class ContadorTolerante(Contador):
```

```
    def contar(self, item):
        return self.contagem.get(item, 0)
```



academia
python

Invocar método de super-classe

- A forma mais simples:

```
class ContadorTotalizador(Contador):  
    def __init__(self):  
        Contador.__init__(self)  
        self.total = 0  
  
    def incluir(self, item):  
        Contador.incluir(self, item)  
        self.total += 1
```

Invocar método de super-classe

- A forma mais correta:

```
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
```

Herança múltipla

- classe que totaliza e não levanta exceções:

```
class ContadorTT(ContadorTotalizador, ContadorTolerante):  
    pass
```

- como funciona:
 - MRO = ordem de resolução de métodos

```
>>> ContadorTT.__mro__  
(<class '__main__.ContadorTT'>,  
<class '__main__.ContadorTotalizador'>,  
<class '__main__.ContadorTolerante'>,  
<class '__main__.Contador'>,  
<type 'object'>)
```

Uso de herança múltipla

```
>>> from contadores import *
>>> class ContadorTT(ContadorTotalizador,
...     ContadorTolerante):
...     pass
...
>>> ctt = ContadorTT()
>>> for letra in 'abacaxi':
...     ctt.incluir(letra)
...
>>> ctt.total
7
>>> ctt.contar('a')
3
>>> ctt.contar('z')
0
```



academia
python

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
```

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  

```



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
```



Exemplo: baralho polimórfico

- começamos com uma classe bem simples:

```
class Carta(object):  
    def __init__(self, valor, naipe):  
        self.valor = valor  
        self.naipe = naipe  
    def __repr__(self):  
        return '<%s de %s>' % (self.valor, self.naipe)
```

Baralho polimórfico 2

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

```
class Baralho(object):  
    naipes = 'paus copas espadas ouros'.split()  
    valores = 'A 2 3 4 5 6 7 8 9 10 J Q K'.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, 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!



academia
python

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]  
>>> shuffle(b)
```

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...



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):  
    naipes = 'paus copas espadas ouros'.split()  
    valores = 'A 2 3 4 5 6 7 8 9 10 J Q K'.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, pos):  
        return self.cartas[pos]  
    def __setitem__(self, pos, item):  
        self.cartas[pos] = item
```

Academia Python

- instrutor: Luciano Ramalho
- 112 horas/aula, 3½ meses
- 5 módulos
 - Introdução à linguagem
 - OO e frameworks
 - Django + Jquery
 - Django profissional
 - Cloud, NoSQL etc.



A **Academia Python** dá uma visão acessível e prática da linguagem: principais bibliotecas, desenvolvimento Web com **Django**, receitas para tarefas comuns, programação Orientada a Objetos e multi-paradigma e testes automatizados.

As academias da **Globalcode** são formações completas compostas por vários módulos com muito mais tempo para os alunos interagirem com os instrutores.

A **Academia Python** tem cinco módulos totalizando 112 horas aula. É fruto da união entre a qualidade e metodologia da **Globalcode** e a experiência e conhecimento do **Luciano Ramalho**.



Módulos da Academia Python: **112 H**

PY1 - Introdução à linguagem Python

PY2 - Orientação a Objetos e frameworks

PY3 - Desenvolvimento Web com Django e JQuery

PY4 - Django profissional

PY5 - Cloud, NoSQL e novas arquiteturas

python.globalcode.com.br



Mais informações:
python@globalcode.com.br

(11) 3145-2230