

Python: Classes

Claudio Esperança

Orientação a Objetos

- É uma disciplina de programação assim como a Programação Estruturada
- Tenta unificar as idéias de algoritmos e estruturas de dados através do conceito de *Objeto*
 - Um objeto é uma unidade de software que encapsula algoritmos e os dados sobre o qual os algoritmos atuam
- Os seguintes conceitos são importantes quando falamos de orientação a objetos:
 - Polimorfismo
 - Abstração
 - Herança

Polimorfismo

- É o que permite que dois objetos diferentes possam ser usados de forma semelhante
 - Por exemplo, tanto listas quanto tuplas ou strings podem ser indexadas por um número entre colchetes e suportam o método `len`
 - Assim, se escrevemos ...

```
for i in range(len(X)): print i, X[i]
```
 - ...não é possível saber de antemão se `X` é uma tupla, uma lista ou uma string
- Desta forma, se escrevemos um algoritmo para ser aplicado um objeto `X`, então também pode ser aplicado a um objeto `Y` desde que `Y` seja suficientemente polimórfico a `X`

Abstração (ou encapsulamento)

- É o que permite que um objeto seja utilizado sabendo-se sobre ele apenas a sua interface
 - Em particular, não precisamos conhecer a implementação dos seus *métodos*
- Em OO a abstração tem mais alcance pois um objeto encapsula tanto dados como algoritmos
 - Assim, podemos atribuir objetos ou passar objetos como argumentos, sem necessariamente saber como o objeto está implementado

Herança

- É o que permite construir objetos que são especializações de outro objeto
 - Isso permite o reuso de software já que objetos especializados *herdam* dos objetos genéricos uma série de atributos comuns
- Por exemplo, considere um objeto que representa uma forma geométrica. Então, ele pode ter características tais como área, perímetro, centróide, etc.
 - Um polígono é uma forma geométrica,
 - Portanto, herda todas as características de formas geométricas
 - Deve suportar também características específicas como número de lados e comprimento de arestas

Objetos em Python

- Python suporta OO através de **classes**
- Uma classe pode ser entendida como uma *fábrica* de objetos, todos com as mesmas características
 - Diz-se que objeto fabricado por uma classe é uma *instância* da classe
- A rigor, uma classe é também um objeto
 - Encapsula dados e algoritmos
 - Entretanto, não é normalmente um objeto *fabricado* por uma classe, mas um objeto criado pela construção `class`
- Um objeto encapsula dados e algoritmos sob a forma de variáveis e métodos
 - É comum chamar esses elementos constituintes dos objetos de *atributos*

Declaração de uma classe

- A maneira mais simples é:

```
class nome:  
    var = valor  
    ...  
    var = valor  
    def metodo (self, ... arg):  
        ...  
    def metodo (self, ... arg):  
        ...
```

- As variáveis e os métodos são escritos precedidos pelo nome da classe e por um ponto (.)
 - Assim, uma variável *v* definida numa classe *C* é escrita *C.v*
- Os métodos sempre têm `self` como primeiro argumento
 - `self` se refere a uma instância da classe
- Uma nova instância da classe é criada usando `nome ()`

Exemplo

```
>>> class C:  
    a = 2  
    b = 3  
    def f(self,x):  
        return C.a*x+C.b
```

```
>>> C.a = 9
```

```
9
```

```
>>> C.b
```

```
3
```

```
>>> obj=C()
```

```
>>> obj.f(7)
```

```
17
```


Atributos de instâncias

- No exemplo anterior, `a` e `b` eram atributos da classe `C` e portanto usáveis por qualquer instância de `C`
- Mais frequentemente, precisamos de atributos associados a instâncias individuais
- Um atributo `attr` associado a uma instância `obj` tem nome `obj.attr`
- Se queremos nos referir a um atributo `attr` de um objeto dentro de algum de seus métodos, usamos o nome `self.attr`

Exemplo

```
>>> class C:
    def init(self,a=2,b=3):
        self.a = a
        self.b = b
    def f(self,x):
        return self.a*x+self.b
```

```
>>> obj1 = C()
>>> obj1.init(2,3)
>>> obj2 = C()
>>> obj2.init(8,1)
>>> obj1.f(7)
17
>>> obj2.f(7)
57
```

Atributos herdados da classe

- Se uma classe define atributos de classe, as instâncias herdam esses atributos da classe como atributos de instância
- Ex.:

```
>>> class C:  
    a = 1  
    def f(self,x):  
        self.a += x
```

```
>>> c = C()
```

```
>>> c.f(2)
```

```
>>> c.a
```

```
3
```

```
>>> C.a
```

```
1
```

Construtores

- Um método como `init` do exemplo anterior é bastante útil para inicializar atributos da instância e é conhecido como *construtor* da classe
- Na verdade, Python suporta construtores que podem ser chamados automaticamente na criação de instâncias
 - Basta definir na classe um método chamado `__init__`
 - Este método é chamado automaticamente durante a criação de um nova instância da classe, sendo que os argumentos são passados entre parênteses após o nome da classe
- Obs.: o método `__init__` é apenas um exemplo de “método mágico” que é invocado de maneira não padrão (veremos outros adiante)

Exemplo

```
>>> class C:  
    def __init__(self,a=2,b=3):  
        self.a = a  
        self.b = b  
    def f(self,x):  
        return self.a*x+self.b
```

```
>>> obj1 = C()  
>>> obj2 = C(8,1)  
>>> obj1.f(7)  
17  
>>> obj2.f(7)  
57
```

Especialização de classes

- Para fazer uma classe C herdar de outra B , basta declarar C como:

```
class C(B):
```

```
    ...
```

- Diz-se que C é *sub-classe* (ou *derivada*) de B ou que B é *super-classe* (ou *base*) de C
- C herda todos os atributos de B
- A especialização de C se dá acrescentando-se novos atributos (variáveis e métodos) ou alterando-se métodos
- Se, um método de C , precisa invocar um método m de B , pode-se utilizar a notação $B.m$ para diferenciar do m de C , referido como $C.m$

Exemplo

```
>>> class B:
    n = 2
    def f(self,x): return B.n*x
>>> class C(B):
    def f(self,x): return B.f(self,x)**2
    def g(self,x): return self.f(x)+1
>>> b = B()
>>> c = C()
>>> b.f(3)
6
>>> c.f(3)
36
>>> c.g(3)
37
>>> B.n = 5
>>> c.f(3)
225
```

Unbound Method

- O parâmetro self não pode ser removido da chamada da função f de B, na classe C, do exemplo anterior:

```
>>> class C(B):
    def f(self,x): return B.f(x)**2
    def g(self,x): return self.f(x)+1
```

```
>>> c=C()
```

```
>>> print c.f(3)
```

Traceback (most recent call last):

```
File "<stdin>", line 1, in <module>
```

```
File "<stdin>", line 2, in f
```

```
TypeError: unbound method f() must be called with B instance
as first argument (got int instance instead)
```


Construtores de classes derivadas

- O construtor de uma classe D derivada de C precisa chamar o construtor de C
 - A chamada do construtor de C não é feita por default
 - Permite inicializar os elementos de C que não são específicos de D
 - Usa-se a notação C.__init__(self, ...)

Construtores de classes derivadas

■ Exemplo:

```
>>> class C:
...     def __init__(self):
...         print "Construtor de C"
...         self.x = 1
...
>>> class D(C):
...     def __init__(self):
...         print "Construtor de D"
...         C.__init__(self)
...         self.y = 2
...
>>> d=D()
Construtor de D
Construtor de C
>>> d.x
1
>>> d.y
2
```

Classes no “novo estilo”

- A partir do Python 2.2, classes podem também ser declaradas no chamado “novo estilo”:
 - Se uma classe não é derivada de nenhuma outra, ela deve ser declarada como derivada da classe especial chamada **object**. Ex.:

```
class C(object):
```
- Há várias diferenças entre o comportamento das classes no “novo estilo” e as do “velho estilo”
 - Permite derivar tipos primitivos
 - Descritores para propriedades, métodos estáticos, métodos de classe, etc
 - Essas diferenças são pouco significativas para o iniciante

Herança múltipla

- É possível construir uma classe que herda de duas ou mais outras. Ex.:
 - `class C(A,B): ...`
- Nesse caso, a classe derivada herda todos os atributos de ambas as classes-base
- Se ambas as classes base possuem um atributo com mesmo nome, aquela citada primeiro prevalece
 - No exemplo acima, se A e B possuem um atributo X, então C.X se refere ao que foi herdado de A

Exemplo

```
>>> class C:
    def __init__(self,a,b):
        self.a, self.b = a,b
    def f(self,x):
        return self.a*x+self.b
>>> class D:
    def __init__(self,legenda):
        self.legenda = legenda
    def escreve(self,valor):
        print self.legenda, '=', valor
>>> class E(C,D):
    def __init__(self,legenda,a,b):
        C.__init__(self,a,b)
        D.__init__(self,legenda)
    def escreve(self,x):
        D.escreve(self,self.f(x))
>>> e = E("f",10,3)
>>> e.escreve(4)
f = 43
```

Atributos privados

- Em princípio, todos os atributos de um objeto podem ser acessados tanto dentro de métodos da classe como de fora
- Quando um determinado atributo deve ser acessado apenas para implementação da classe, ele não deveria ser acessível de fora
 - Em princípio tais atributos não fazem parte da interface “pública” da classe
- Atributos assim são ditos *privados*
- Em Python, atributos privados têm nomes iniciados por dois caracteres “traço-embaixo”, isto é, __

Exemplo

```
>>> class C:
    def __init__(self,x): self.__x = x
    def incr(self): self.__x += 1
    def x(self): return self.__x
```

```
>>> a = C(5)
```

```
>>> a.x()
```

```
5
```

```
>>> a.incr()
```

```
>>> a.x()
```

```
6
```

```
>>> a.__x
```

```
Traceback (most recent call last):
```

```
  File "<pysHELL#13>", line 1, in -toplevel-
```

```
    a.__x
```

```
AttributeError: C instance has no attribute '__x'
```

Métodos mágicos

- São métodos que são invocados usando operadores sobre o objeto ao invés de por nome
- Já vimos um método desses: o construtor `__init__`
- Alguns outros são:
 - Adição: `__add__`
 - Chamado usando '+'
 - Subtração: `__sub__`
 - Chamado usando '-'
 - Representação: `__repr__`
 - Chamado quando objeto é impresso
 - Conversão para string: `__str__`
 - Chamado quando o objeto é argumento do construtor da classe `str`
 - Se não especificado, a função `__repr__` é usada

Exemplo

```
>>> class vetor:
    def __init__(self,x,y):
        self.x, self.y = x,y
    def __add__(self,v):
        return vetor(self.x+v.x, self.y+v.y)
    def __sub__(self,v):
        return vetor(self.x-v.x, self.y-v.y)
    def __repr__(self):
        return "vetor("+str(self.x)+", "+str(self.y)+")"

>>> a=vetor(1,2)
>>> a += vetor(3,5)
>>> a-vetor(2,2)
vetor(2,5)
>>> print a
vetor(4,7)
```

Protocolos

- Diferentemente de outras linguagens, não há necessidade de classes serem relacionadas para haver polimorfismo entre elas, basta que implementem métodos semelhantes
- Um protocolo é uma especificação de polimorfismo informal
- Por exemplo, listas, strings e tuplas possuem em comum o fato de poderem iterar sobre uma coleção de elementos
 - Todas implementam o protocolo para seqüências
 - Métodos “mágicos” para indexar, alterar, etc.

Protocolo para seqüências

- `__len__(self)` retorna o comprimento da seqüência
 - Chamada: `len(objeto)`
- `__getitem__(self, key)` retorna o elemento na posição `key` da seqüência
 - Chamada: `objeto[key]`
 - Deve-se implementar também chaves negativas!
- `__setitem__(self, key, value)`
 - Chamada: `objeto[key]=value`
 - Apenas para seqüências mutáveis
- `__del__(self, key)`
 - Chamada por `del objeto[key]`
 - Apenas para (algumas) seqüências mutáveis

Exemplo

```
>>> class ProgressaoAritmetica:
    def __init__(self,a1,incr):
        self.a1,self.incr=a1,incr
    def __getitem__(self,key):
        if not isinstance(key,(int,long)):
            raise TypeError
        if key<=0: raise IndexError
        return self.a1+(key-1)*self.incr
    def soma(self,n):
        return (self[1]+self[n])*n/2
>>> pa = ProgressaoAritmetica(1,2)
>>> pa[1]
1
>>> pa[10]
19
>>> pa.soma(100)
10000
```

Atributos, Getters e Setters

- Muitas vezes queremos que determinados atributos possam ser acessados de forma controlada, isto é, vigiados por métodos
- Os métodos que controlam o acesso a tais atributos são conhecidos como *getters* e *setters*, referindo-se a métodos de leitura e escrita, respectivamente
- Os atributos controlados são chamados de *propriedades*
- Na verdade, podemos ter propriedades abstratas que não correspondem 1 para 1 com atributos da classe

Exemplo

```
>>> class Retangulo:
    def __init__(self,tamanho):
        self.setTamanho(tamanho)
    def setTamanho(self,tamanho):
        if min(tamanho)<0: raise ValueError
        self.__tamx,self.__tamy = tamanho
    def getTamanho(self):
        return (self.__tamx,self.__tamy)
```

```
>>> r = Retangulo((20,30))
```

```
>>> r.getTamanho()
```

```
(20, 30)
```

```
>>> r.setTamanho((-1,0))
```

```
Traceback (most recent call last):
```

```
...
```

```
ValueError
```

A função *property*

- A função *property* pode ser usada para consubstanciar uma propriedade implementada por métodos de tal maneira que ela pareça um atributo da classe
- Ela é usada no corpo de uma declaração de classe com a forma:
atributo = `property(fget, fset, fdel, doc)`
- ...onde
 - *fget, fset, fdel* são métodos para ler, escrever e remover o *atributo*
 - *doc* é uma docstring para o atributo

Exemplo

```
>>> class Retangulo:
    def __init__(self,tamanho):
        self.setTamanho(tamanho)
    def setTamanho(self,tamanho):
        if min(tamanho)<0: raise ValueError
        self.__tamx,self.__tamy = tamanho
    def getTamanho(self):
        return (self.__tamx,self.__tamy)
    tamanho = property(getTamanho,setTamanho)
```

```
>>> r = Retangulo((20,30))
```

```
>>> r.tamanho
```

```
(20, 30)
```

```
>>> r.tamanho = (30,30)
```

```
>>> r.tamanho
```

```
(30, 30)
```


Dicas para uso de OO

- Agrupe funções e dados que se referem a um mesmo problema
 - Por exemplo, se uma função manipula uma variável global, é melhor que ambas sejam definidas numa classe como atributo e método
- Não permita promiscuidade entre classes e instâncias de classe
 - Por exemplo, se há necessidade de um objeto manipular um atributo de outro, escreva um método com essa manipulação e chame-o
 - Não escreva métodos extensos
 - Em geral, um método deve ser o mais simples possível