Significado de `self` em Python

Adam Victor Nazareth Brandizzi ¡brandizzi@gmail.com¿

19 de setembro de 2007

Resumo

Um dos aspectos mais incompreendidos de Python é a declaração do argumento `self` para métodos. Esse texto tentará apresentar uma explicação baseada em funções. Pressupomos que o leitor já conheça a linguagem e saiba como definir funções em Python.

Sumário

1 Classes, funções e métodos

Para compreender o sentido do parâmetro `self`, temos de entender o que é uma classe. Uma *classe* é um "modelo" para diversos valores, que são chamados *objetos*. Definir classes é bem simples, e nem precisamos fazer algo em sua definição. Por exemplo, a classe `Person` abaixo, que usaremos representar pessoas, é definida sem nenhum comando significante dentro dela, exceto o comando `pass`, que informa que nada será feito além de definir a classe.

```
class Person(object):
    pass
```

Figura 1: Definição simples de classe

A definição de classe da figura 1 não faz nada além de dizer que existe a classe `Person`. Para executarmos operações nessa classe, precisamos definir *funções* que atuem sobre ela. Por exemplo, funções que armazenem o nome da pessoa:

```
class Person(object):
    pass

def set_name(person, name):
    # Não aceita nomes com menos de duas letras
    if len(name) >= 2:
        person.name = name

woman = Person()
set_name(woman, 'Juliana')

# Vai imprimir "Juliana"
print woman.name

# Não vai mudar o valor, pois len('J') < 2
set_name(woman, 'J')

# Vai imprimir "Juliana" de novo, pois não mudou o valor
print woman.name
```

Figura 2: Definição de classe e função

Na figura 2, definimos a função `set_name` que armazena o nome da pessoa num objeto da classe `Person`.

Por uma questão de praticidade, é tradição declarar as funções que operam sobre uma classe dentro da própria classe. Uma vez que a função foi declarada dentro da classe, toda vez que a função for chamada o nome da classe deve vir antes; isto é, chamar uma função declarada dentro de uma classe é como chamá-la de um módulo.

Na figura 3, declaramos a função dentro da classe, ao invés de ficar do lado de fora. A vantagem de fazer isso é que o código que altera a classe fica mais próximo da definição, ficando mais separado e legível. Depois, é só chamar o nome da classe seguido do nome da função que ele executa. É como se a função fosse um valor da classe.

Entretanto, embora essa notação possa ser muito útil, ficar digitando o nome da classe pode ser bem entediante. Certamente é redundante, pois todo objeto sabe a qual classe pertence. Desse modo, tiveram a idéia de, ao invés de preceder o nome da função com o nome da classe, precedê-lo com o objeto que é o primeiro parâmetro. Obviamente, não faz sentido usar o nome do objeto antes do nome da função e depois como parâmetro, como `woman.set_name(woman, 'Juliana')`. Se o nome do objeto já está lá antes do nome da função, ele deve ser retirado da lista de parâmetros, como na figura 4.

Essas "funções dentro de classes" são chamadas de *métodos*. Para chamar métodos, tanto faz chamá-los como em `Classe.metodo(objeto, parametros)`

```
class Person(object):
    def set_name(person, name):
        if len(name) >= 2:
            person.name = name

woman = Person()

# Nome da classe deve vir antes do uso da função, pois
# a função está dentro da classe.
Person.set_name(woman, 'Juliana')

# Vai imprimir "Juliana"
print woman.name
```

Figura 3: Função dentro da classe

quanto chamá-los como em `objeto.metodo(parametros)`

As formas são equivalentes, com apenas uma ressalva: `object` *deve ser um objeto da classe* `Classe`. Por sinal, mesmo a primeira notação `Classe.metodo(objeto, parametros)` resultaria em erro se `objeto` não fosse um objeto da classe `Classe`.

```
class Person(object):
    def set_name(person, name):
        if len(name) >= 2:
            person.name = name

woman = Person()
woman.set_name('Juliana')
print woman.name
```

Figura 4: Objeto antes da função

2 O nome `self`

No método `Person.set_name` acima, o nome do primeiro parâmetro do método era `person`. Entretanto, é tradição chamar esse primeiro parâmetro de `self`. Por quê?

Bem, não há nenhuma obrigatoriedade de se fazer assim – tanto o é que em nosso método usamos outro nome para o parâmetro. Essa tradição é mantida porque a maioria dos programadores Python já reconhece esse nome como o nome do objeto a ser invocado no método; ademais, esse é o padrão especificado pela PEP-8 [vReBW]. Por isso mesmo, via de regra é melhor utilizar `self` como o nome do primeiro parâmetro dos métodos.

```
class Person(object):
    def set_name(self, name):
        if len(name) >= 2:
            self.name = name

woman = Person()
woman.set_name('Juliana')
print woman.name
```

Figura 5: Uso do nome `self`

Em outras linguagens orientadas a objetos, o parâmetro correspondente ao `self` geralmente não é declarado na lista de parâmetros, mas sim passado implicitamente. Nessas linguagens, a assinatura do método `set_name` seria mais como
`def set_name(nome)`
que como
`def set_name(self, nome)`.

3 Aplicações

O `self` explícito é uma característica polêmica da linguagem Python [Eck], mas há diversas razões em favor do seu uso.

O argumento mais comum – e provavelmente o mais subjetivo – é que o `self` explícito está mais de acordo com o espírito da linguagem que um `self` implícito, já que, como diria o *Zen of Python*, *explícito é melhor que implícito* [Pet]. Também argumenta-se que a declaração explícita do `self` é, hoje em dia, tão intrínseca à linguagem que seria praticamente inviável abandoná-la [vR].

O `self` explícito tem, também, várias vantagens na prática quotidiana. Uma das mais notáveis é a habilidade de chamar métodos de classes ancestrais sobre objetos de classes herdadas. Por exemplo, voltando ao nosso exemplo com pessoas, suponhamos que definamos uma nova classe `CapitalizedPerson`, herdeira de `Person`, na qual o nome da pessoa deve começar com uma letra maiúscula. Nessa classe, também queremos que todas as restrições que o método `set_name` da classe `Person` ainda sejam seguidas. Como fazer?

Em Python, é bem simples: basta usar a notação de chamada de método na forma `Classe.metodo(self, parametros)` ou, mais especificamente, `Person.set_name(self, name)`, conforme demostrado na figura 6. Note que, se fizermos uma outra classe herdando de `CapitalizedPerson` e não queiramos usar o método `set_name` de `CapitalizedPerson`, podemos chamar diretamente o método `set_name` da classe `Person`. Essa é, provavelmente, uma solução bem mais elegante que o famoso objeto `super` de várias linguagens; certamente, é mais explícita e flexível.

Como Python suporta herança múltipla, essa funcionalidade pode se tornar

```
import string

class CapitalizedPerson(Person):
    def set_name(self, name):
        if name[0] in string.uppercase:
            Person.set_name(self, name)

woman = Person()
woman.set_name('Juliana')
print woman.name
```

Figura 6: Chamando métodos de classes ancestrais

indispensável. Suponha uma classe que herde, ao mesmo tempo, de duas classes. Para inicializar essa classe com o método inicializador de cada uma de suas classes-mães, basta chamar o método `__init__` de cada uma das classes, como na figura 7.

```
class ButtonPerson(Button, Person):
    """Uma pessoa que é um botão(?!)"""
    def __init__(self, name):
        Person.__init__(self)
        Button.__init__(self, "Pessoa: " + name)
        self.set_name(name)
```

Figura 7: Múltipla inicialização

4 Conclusão

O `self` explícito de Python é uma decisão de projeto polêmica. Entretanto, há inúmeras vantagens oriundas dela, como o caráter explícito da decisão e a maior flexibilidade na chamada de métodos. Suspeito até que o `self` explícito torne mais fácil a apreensão do conceito dos objetos `self`/`this`: se, por um lado, programadores oriundos de linguagens nas quais os objetos `self` ou `this` são implícitos sentem-se confusos com o `self` explícito, programadores que não estão acostumados com esse conceito parecem apreender melhor a idéia quando apresentados à "maneira *pythônica*" de se tratar do conceito.

Referências

[Eck] Bruce Eckel. Python 3.0 or Python 2.9. Publicado em `http://www.artima.com/weblogs/viewpost.jsp?thread=214112`.

[Pet] Tim Peters. PEP-20: The Zen of Python. Publicado em `http://www.python.org/dev/peps/pep-0020/`.

[vR] Guido van Rossum. A Response to Bruce Eckel. Publicado em `http://www.artima.com/weblogs/viewpost.jsp?thread=214325`.

[vReBW] Guido van Rossum e Barry Warsaw. PEP-8: Style Guide for Python Code. Publicado em `http://www.python.org/dev/peps/pep-0008/`.