02

Orientação a objetos e o conceito de classe

Python e Paradigma OO

No capítulo anterior, aprendemos a criar um pequeno programa de cadastro. De acordo com a opção escolhida pelo usuário determinada tarefa era executada. Mas será que a maneira que organizamos nosso código tem futuro?

Perfil e suas características: como representá-las em Python?

Bem, sabemos que um perfil de usuário, além do nome, possui outras características como telefone e empresa:

```
>>> perfil1_nome = 'Flávio Almeida'
>>> perfil1_telefone = 'não informado'
>>> perfil1_empresa = 'Caelum'
```

E se tivermos mais um perfil criado em nosso programa? Isso é muito fácil, não? Basta pedirmos ajuda do CONTROL + C e seu inseparável amigo CONTROL + V para em seguida mudarmos os nomes das variáveis:

```
>>> perfil1_nome = 'Flávio Almeida'
>>> perfil1_telefone = 'não informado'
>>> perfil1_empresa = 'Caelum'

>>> perfil2_nome = 'Nico Steppat'
>>> perfil2_telefone = 'segredo'
>>> perfil2_empresa = 'Caelum'
```

Problema resolvido, não? Inclusive eu tenho o sonho de um dia criar uma rede social para concorrer com as já existentes, quem sabe tendo uns 10.000 perfis criados. Opa, espere um pouco! Eu disse 10.000? Isso significa quantos CONTROL + C e CONTROL + V? Sem falar que teremos que renomear milhares de variáveis! Cruzes, nem pensar: solução descartada.

Que tal usarmos um `dictionary` ? Não custa nada tentarmos:

```
>>> perfil = {'nome' : 'Flávio Almeida' , 'telefone' : 'não informado' , 'empresa': 'Caelum'}
>>> perfil['nome']
'Flávio Almeida'
>>> perfil['empresa']
'Caelum'
```

Bem melhor! No lugar de declararmos várias variáveis, declaramos apenas uma que guarda todos os dados do perfil. Hum, mas se quisermos criar outro perfil?

```
>>> perfil2 = {'nome' : 'Nico Steppat' , 'telefone' : 'segredo' , 'empresa': 'Caelum'}
>>> perfil2['nome']
'Nico Steppat'
>>> perfil2['empresa']
'Caelum'
```

Hum, essa solução não está boa. Para cada perfil criado, teremos que lembrar quais são as características (nome, telefone, etc.) que todo perfil possui. Você conseguirá lembrar sempre delas, inclusive não errará seus nomes quando for defini-las para 10.000 perfis? Se entrar uma nova característica, lembrará de adicioná-la em todos os perfis criados? É, nem eu!

O problema é que não temos uma estrutura, um molde, algo que sirva como especificação de como um perfil deve ser criado. Queremos algo como se fosse uma fôrma de bolo quadrada. A partir dela podemos criar bolos de sabores e cores diferentes, inclusive receitas, porém todos terão a mesma estrutura quadrada. Impossível termos um bolo redondo!

Agora, se tivéssemos uma "fôrma de perfis", poderíamos criar os mais diversos perfis e mesmo que cada um guarde informações diferentes, todos terão a mesma estrutura: nome, telefone e empresa. Legal essa ideia, pois estaríamos jogando a responsabilidade da estrutura para a fôrma, nos livrando desse pepino.

Para que possamos resolver o problema que vimos antes, precisamos mudar a maneira pela qual pensávamos sobre eles, isto é, precisamos mudar nosso paradigma. Abra sua mente e seja bem-vindo ao paradigma da **Orientação a Objetos**.

Classes

O paradigma da Orientação a Objetos (O.O) parte do princípio no qual dados e os comportamentos que operam nestes dados caminham juntos. Há diversos benefícios dessa xipofagia entre dado e comportamento, porém, no estágio em que estamos, o conceito de classe é o primeiro que vem a calhar.

Uma classe é uma especificação, isto é, aquela nossa "fôrma de bolo". A partir de uma classe, criamos objetos que seguem a estrutura da especificação, aqueles "bolos" de diversos sabores, porém todos possuem a mesma estrutura.

A boa notícia é que Python suporta bastante coisa do paradigma O.O, inclusive nos permite criar **classes**, nome bonito para a especificação que estamos debatendo. Nossa tarefa agora é aprender como criar uma classe que represente a especificação de um Perfil, inclusive como criar objetos a partir desta classe.

Lembra que utilizávamos a palavra reservada `def` para criarmos funções? A partir de agora ela será utilizada para definirmos os comportamentos de nossa classe. No mundo OO, chamamos essas funções de **métodos**.

Mas nem criamos nossa classe ainda e já estamos pensando em criar métodos! Primeiro, criaremos o arquivo `models.py`. É neste arquivo que declararemos nossa classe **Perfil** e qualquer outra que seja necessária. Utilizamos a palavra mágica **class** seguida do nome da classe:

```
# -*- coding: UTF-8 -*-
class Perfil():
   'Classe padrão para perfis de usuários'
```

Criamos uma classe que possui uma breve descrição. Porém, não especificamos quais características ela define.

Função construtora e self

Precisamos definir quais serão as características de nossa classe, fazemos isso através do construtor `__init__` que recebe como parâmetro um `self`. Utilizamos o termo construtor porque ele é chamado toda vez que criamos um objeto a partir de uma classe. O parâmetro `self` é surreal. Gostou da explicação? Eu também não, mas sempre precisamos recebê-lo como parâmetro no construtor. Por enquanto, aceite essa explicação simplória.

```
# -*- coding: UTF-8 -*-
class Perfil():
   'Classe padrão para perfis de usuários'
   def __init__(self):
```

Tudo bem, criamos o tal construtor que recebe o misterioso `self`, mas onde está o nome, telefone e empresa nesta especificação, nesta "fôrma"? Bem, você lembra do que acabei de dizer sobre `__init__`? Ela é uma função construtora, mas precisamente construtora de perfis! Imediatamente após o `self`, ela pode receber quantos parâmetros quisermos. Não queremos zilhões de parâmetros, apenas aqueles que receberão o nome, telefone e empresa:

```
# -*- coding: UTF-8 -*-
class Perfil():
   'Classe padrão para perfis de usuários'
   def __init__(self, nome, telefone, empresa):
```

Nossa classe não está pronta, mas se fôssemos criar um perfil a partir dela utilizaríamos a sintaxe:

```
>>> perfil1 = Perfil('Flávio Almeida', 'não informado', 'Caelum')
>>> perfil2 = Perfil('Nico Steppat', 'segredo', 'Caelum')
```

Repare que não passamos para o construtor o `self`, nem poderíamos neste caso, porque ele é passado por debaixo dos panos através do Python. Sabe o motivo de ele ser passado misteriosamente pelo Python? É que o `self` é uma referência ao objeto que criamos a partir da classe! Logo, o `self` do objeto `prefil1` não é o mesmo `self` do objeto `perfil2`! Cada objeto criado a partir de nossa classe terá seu próprio `self` criado pelo Python! Mistério explicado.

Tudo bem, entendemos que cada objeto criado a partir de nossa classe terá seu próprio `self`, mas para quê ele realmente serve? Lembre-se que cada perfil criado deve ter suas próprias características.

Atributos de instância

Se cada objeto perfil criado tem seu próprio `self`, não seria interessante pendurarmos nele as informações que recebemos através do construtor `__init__`?:

```
# -*- coding: UTF-8 -*-
class Perfil():
   'Classe padrão para perfis de usuários'
   def __init__(self, nome, telefone, empresa):
      self.nome = nome
      self.telefone = telefone
      self.empresa = empresa
```

Quando usamos a sintaxe `self.nome_qualquer` significa estamos criando um **atributo de instância** que estará presente em todos os objetos criados a partir daquela classe, porém o valor desses atributos são específicos para cada objeto criado. Tanto isso é verdade que será possível acessarmos esses atributos através do perfis que criarmos.

Instanciando objetos

Desta vez, vamos criar um teste real pelo console do Python, só não se esqueça de importar nossa classe utilizando a mesma sintaxe que já aprendemos, porém no lugar de * (tudo), indicaremos o nome da classe:

```
>>> from models import Perfil

>>> perfil1 = Perfil('Flavio Almeida', 'não informado', 'Caelum')
>>> perfil1.nome
'Flavio almeida'

>>> perfil2 = Perfil('Nico Steppat', 'segredo', 'Caelum')
>>> perfil2.nome
'Nico Steppat'
```

Que emoção, funciona! Porém, esqueci de colocar o acento no nome. Será que podemos alterar um objeto depois de criado? Com certeza, vazemos isso através do operador 'ponto':

```
>>> perfil1.nome = 'Flávio Almeida'
>>> perfil1.nome
'Flávio Almeida'
```

Criamos nossa primeira classe e nosso primeiro objeto! Porém, alguém cético pode ainda duvidar de nossa capacidade e pedir para que imprimamos no console todos os atributos do objeto:

```
>>> from models import Perfil

>>> perfil1 = Perfil('Flávio Almeida', 'não informado', 'Caelum')
>>> perfil1.nome
>>> perfil1.telefone
>>> perfil1.empresa

>>> perfil2 = Perfil('Nico Steppat', 'segredo', 'Caelum')
>>> perfil2.nome
>>> perfil2.telefone
>>> perfil2.empresa
```

Pronto, menos um cético no mundo!

Nosso primeiro método

É muito comum queremos exibir as informações de um perfil, imagine termos que repetir o código que vimos toda vez? E se esquecermos de incluir algum atributo? Lembra do paradigma O.O? Nele, dado e comportamento caminham juntos. Que tal adicionarmos em nossa classe um comportamento que será responsável pela impressão de seus dados?

```
# -*- coding: UTF-8 -*-
class Perfil():
   'Classe padrão para perfis de usuários'
   def __init__(self, nome, telefone, empresa):
      self.nome = nome
      self.telefone = telefone
      self.empresa = empresa
```

```
    def imprimir(self):
        print "Nome : %s, Telefone: %s, Empresa %s" % (self.nome, self.telefone, self.empresa)
```

Desta vez, criamos uma função que não é construtora, mais um método de nossa classe. Quando este método for chamado, precisaremos acessar o `self` daquele objeto que estamos manipulando porque é ele que contem os atributos que desejamos imprimir. É por isso que ele também recebe `self` como parâmetro. Assim como na função construtora `__init__` , quem for chamar nosso método não precisará passar o `self` , pois o Python se encarregará automaticamente disso para nós. Pronto, agora, vamos chamar este método em nossos objetos (só não esqueça de fechar o terminal, importar nosso Perfil e toda aquela bagaça que o faz odiar o console do Python):

```
>>> from models import Perfil

>>> perfil1 = Perfil('Flávio Almeida', 'não informado', 'Caelum')
>>> perfil1.imprimir()
Nome : Flávio Almeida, Telefone: não informado, Empresa Caelum

>>> perfil2 = Perfil('Nico Steppat', 'segredo', 'Caelum')
>>> perfil2.imprimir()
Nome : Nico Steppat, Telefone: segredo, Empresa Caelum
```

Parâmetros nomeados

Imagine agora que você chamou aquele seu amigo curioso que sempre sonhou criar um objeto com Python. Você explica para ele como criar um objeto e o deixa praticando no console. Depois de você beber e voltar, no console há a seguinte instrução:

```
>>> perfil = Perfil('Caelum', 'não informado', 'Fulaninho Ciclano')
```

Consegue perceber um problema? Ele inventou a ordem dos parâmetros! Será que a responsabilidade disso é dele ou de nosso código? Será que também não poderíamos cair no mesmo problema? É por isso que o Python permite passarmos parâmetros nomeados. Funciona da seguinte forma:

```
>>> perfil = Perfil(empresa='Caelum', telefone='não informado', nome='Fulaninho Ciclano')
```

Dessa maneira, não importa a ordem dos parâmetro passados no construtor, o Python saberá associar o valor passado para o atributo indicado. Inclusive podemos utilizar este recurso com métodos ou funções avulsas mesmo.

Um pouco mais sobre classes: Old-style X New-style

Aprendemos a declarar classes em Python garantindo assim que todos os nossos perfis tivessem a mesma estrutura. Inclusive aprendemos a criar métodos e entendemos como o Python lida com atributos privados.

Porém, há duas maneiras de declararmos classes em Python. A primeira, chamada **old-style** é a que utilizamos. Mas qual a razão de terem criado uma segunda forma? Para entendermos isso mais facilmente, faremos um teste. Podemos perguntar para a instância de uma classe qual é seu tipo. Porém, há duas maneiras de fazer isso. A primeira é através da função **type** e a segunda é através da propriedade **class** da própria instância:

```
>>> perfil = Perfil('Flávio Almeida', 'não informado', 'Caelum')
>>> type(perfil)
<type 'instance'>
>>> perfil.__class__
<class models.Perfil at 0x10f3b4870>
```

Hum, o resultado da função `type` é `instance` dizendo muito pouco sobre qual o tipo do objeto. Porém, o atributo `__class__` nos diz exatamente seu tipo. Essa discrepância foi um dos motivos para o novo estilo de declaração de classes, o **new-style**:

```
# -*- coding: UTF-8 -*-
class Perfil(object):
   'Classe padrão para perfis de usuários'
   def __init__(self, nome, telefone, empresa):
      self.nome = nome
      self.telefone = telefone
      self.empresa = empresa

   def imprimir(self):
      print "Nome : %s, Telefone: %s, Empresa %s" % (self.nome, self.telefone, self.empresa)
```

Repare que o nome da classe agora recebe um parâmetro, um `object` . Pronto, estamos no novo estilo de declaração de variáveis do Python, fácil assim? Você deve estar se perguntando o que ganhamos com uma alteração tão ínfima como essa, certo? Vamos recarregar no arquivo .py e realizar o mesmo teste que fizemos antes:

```
>>> perfil = Perfil('Flávio Almeida', 'não informado', 'Caelum')
>>> type(perfil)
<class 'models.Perfil'>
>>> perfil.__class__
<class 'models.Perfil'>
```

Repare que a função type e a propriedade **class** retornam agora um mesmo valor, o nome da classe. Esta é apenas uma das diferenças, pois o novo estilo de declaração de classes adicionou novos recursos na linguagem. Veremos alguns deles durante o treinamento.

Muito informação, não é mesmo? É por isso que esta é a hora de praticarmos o que aprendemos.