Atributos e métodos estáticos

Vamos fazer uma breve revisão de alguns conceitos dos capítulos iniciais:

* Classes definem as regrinhas de como devem ser os nossos objetos.
* Os objetos são instâncias concretas da classe.
* Os atributos, definidos na classe, serão particulares de cada objeto - por exemplo, cada objeto Fracao terá o seu próprio atributo "numerador" e o seu próprio atributo "denominador".
* Métodos são funções feitas para agir sobre um objeto. Seu primeiro parâmetro sempre será self, o objeto executando a ação, e o método poderá alterar o estado do objeto.

Neste capítulo iremos subverter um pouco essas regras.

# 1. Atributos estáticos

Estamos há bastante tempo reforçando a ideia de que atributo é algo particular de cada objeto, e geralmente esse é o caso. Mas vez ou outra pode existir o interesse em armazenar um dado que diga respeito à classe como um todo - uma constante que possa ser ajustada para afetar o comportamento de todos os objetos, ou uma informação de controle sobre a classe coletivamente.

Um exemplo bastante simples seria contar os objetos já instanciados daquela classe - uma espécie de contagem de população. Ela poderia ser útil para gerarmos números de matrícula sequenciais. Sempre que criarmos um objeto novo, vamos incrementar o contador de objetos já instanciados e utilizar esse contador como número de matrícula do objeto.

Para isso, vamos declarar uma variável diretamente dentro da classe, e não nos objetos. Para acessá-la podemos utilizar a sintaxe nome\_da\_classe.nome\_da\_variavel sem nem mesmo termos objetos instanciados na classe:

class Aluno:

total\_matriculados = 0

def \_\_init\_\_(self, nome, curso):

self.nome = nome

self.curso = curso

Aluno.total\_matriculados += 1

self.matricula = Aluno.total\_matriculados

print(Aluno.total\_matriculados)

aluno1 = Aluno('Rey', 'Data Science')

aluno2 = Aluno('Finn', 'Data Science')

aluno3 = Aluno('Poe', 'Data Science')

aluno4 = Aluno('Kylo', 'Web Full Stack')

print(Aluno.total\_matriculados)

Esse tipo de atributo não vinculado a nenhum objeto específico é conhecido de maneira geral em programação orientada a objeto como atributo estático, e em Python em particular ele frequentemente é chamado de atributo de classe ou mesmo variável de classe.

Note que é tecnicamente possível acessar o valor do atributo a partir de objetos da classe também:

print(aluno2.total\_matriculados) # resultado: 4

Mas tome cuidado: se você alterar a variável de classe a partir de um objeto, o objeto irá criar em si mesmo uma cópia do atributo e alterará a cópia:

aluno2.total\_matriculados = 10

print(aluno2.total\_matriculados)

print(Aluno.total\_matriculados)

Saída:

10

4

## 2. Métodos de classe e métodos estáticos

Um termo bastante frequente em programação orientada a objeto é método estático. Ele é o análogo de função dos nossos atributos estáticos: uma função declarada dentro da classe e que pode ser chamada diretamente a partir da classe, e não necessariamente por um objeto.

Em Python, porém, temos 2 conceitos diferentes: métodos de classe e métodos estáticos. A diferença entre eles é um pouco sutil, mas é importante para fazer uma boa modelagem.

### 2.1. Métodos de classe

Voltando ao nosso exemplo da classe Aluno acima, você pode ter identificado um problema. Dentro do construtor, estamos referenciando diretamente o nome da classe Aluno para poder acessar a variável de classe. De maneira geral, é indesejável dentro da classe referenciarmos o nome dela. Imagine que no futuro surja uma necessidade de refatorar o código e alterar o nome da classe. Você precisará atualizar todas as referências ao nome da classe. Seria ideal poder alterar o estado da classe sem "conhecer" o nome dela.

Nós já fazemos isso rotineiramente com objetos: nossos métodos convencionais, métodos "de objeto", recebem um parâmetro especial (geralmente chamado de self) que serve para referenciar o objeto dentro do método. Vamos adotar uma estratégia parecida: criar métodos que recebam a própria classe como parâmetro. Assim poderemos alterar o estado da classe (atualizar uma variável de classe, por exemplo) sem utilizar o nome da classe diretamente.

Para criar esse tipo de método, utilizaremos o decorator @classmethod. O primeiro parâmetro funcionará de maneira muito parecida com o self, mas ele irá representar a classe, não o objeto. Esses métodos podem ser chamados tanto a partir de classes quanto de objetos, e o Python irá se encarregar de passar necessariamente a classe naquele parâmetro. Assim como no caso do self, podemos utilizar qualquer nome para o parâmetro, mas convenciona-se utilizar cls (abreviação de "class").

class Aluno:

total\_matriculados = 0

def \_\_init\_\_(self, nome, curso):

self.nome = nome

self.curso = curso

self.incrementa\_matriculados() # será passado como parâmetro a CLASSE de self

self.matricula = Aluno.total\_matriculados

@classmethod

def incrementa\_matriculados(cls):

cls.total\_matriculados += 1 # equivale a: Aluno.total\_matriculados += 1

@classmethod

def visualizar\_matriculados(cls):

return cls.total\_matriculados

print(Aluno.visualizar\_matriculados())

aluno1 = Aluno('Rey', 'Data Science')

aluno2 = Aluno('Finn', 'Data Science')

aluno3 = Aluno('Poe', 'Data Science')

aluno4 = Aluno('Kylo', 'Web Full Stack')

print(Aluno.visualizar\_matriculados())

print(aluno3.visualizar\_matriculados()) # mesmo resultado da linha anterior!

Outra utilidade bem legal dos métodos é na implementação do padrão fábrica. Algumas linguagens orientadas a objeto permitem a criação de múltiplos construtores diferentes com parâmetros diferentes, o que nos ajuda a economizar na passagem de parâmetros mais frequentes para o construtor.

O Python não permite a multiplicidade de construtores, mas podemos criar diversos métodos de classe para funcionar como construtores. No exemplo abaixo, temos métodos específicos para retornar estudantes de Data Science e de Web Full Stack sem precisar repetidamente digitar o nome do curso:

class Aluno:

total\_matriculados = 0

def \_\_init\_\_(self, nome, curso):

self.nome = nome

self.curso = curso

self.incrementa\_matriculados() # será passado como parâmetro a CLASSE de self

self.matricula = Aluno.total\_matriculados

@classmethod

def incrementa\_matriculados(cls):

cls.total\_matriculados += 1 # equivale a: Aluno.total\_matriculados += 1

@classmethod

def visualizar\_matriculados(cls):

return cls.total\_matriculados

@classmethod

def ds(cls, nome):

return cls(nome, 'Data Science')

@classmethod

def web(cls, nome):

return cls(nome, 'Web Full Stack')

a1 = Aluno.ds('Smeagol')

a2 = Aluno.web('Gollum')

print(a1.nome, a1.curso, a1.matricula)

print(a2.nome, a2.curso, a2.matricula)

### 2.2. Métodos estáticos

Métodos estáticos em Python, assim como métodos de classe, não servem para alterar o estado de um objeto em particular. Eles não possuem self. Porém, além de não possuir self eles também não possuem cls. Ele recebe o decorator @staticmethod.

Um método estático não deve afetar o estado de um objeto, tampouco da classe. Ele poderia tranquilamente ser removido de dentro da classe e implementado como uma função "avulsa" e seu funcionamento não seria prejudicado.

Neste caso, por que eles existem? A ideia está relacionada à noção de namespace: implementamos um método estático quando temos uma função que está relacionada ao assunto de nossa classe, e por conta disso gostaríamos de "agrupá-la" com a classe e ter seu nome vinculado ao nome da própria classe.

Em nossa boa e velha classe Fracao de capítulos anteriores, por exemplo, pode ser útil ter uma função para calcular o maior divisor comum (MDC) entre diferentes números. Esse método poderia ser utilizado tanto por outros métodos, por exemplo, para simplificar frações quanto por programadores utilizando a classe que precisem resolver outros problemas relacionados. Vejamos como ela pode ser implementada:

class Fracao:

def \_\_init\_\_(self, num, den):

self.num = num

self.den = den

def \_\_str\_\_(self):

return f'{self.num}/{self.den}'

...

def simplificar(self):

mdc = self.mdc(self.num, self.den) # passamos os parâmetros e não self, pois ela não recebe self!

self.num /= mdc

self.den /= mdc

@staticmethod

def mdc(num1, num2):

while num1 != num2:

if num1 > num2:

num1 -= num2

else:

num2 -= num1

return num1

print(Fracao.mdc(48, 24)) # note que podemos usar em 2 números quaisquer, sem ter objeto Fracao

frac = Fracao(13, 39)

frac.simplificar() # esse método internamente também usa o método estático

print(frac)

### 2.3. Considerações sobre os métodos estático e de classe

Um ponto que já foi amplamente discutido ao longo deste material é que o Python frequentemente permite muita coisa, mas isso nem sempre significa que você deveria fazer: por exemplo, alterar diretamente o valor de um atributo protected ou, pior ainda, abusar do name mangling para alterar diretamente um atributo private.

Alguns conceitos similares se aplicam aqui. Por exemplo:

* É possível um método estático alterar o estado de uma classe (acessando diretamente através do nome da classe).
* É possível criar métodos sem self, sem cls e sem os decorators sinalizando que se são métodos de classe ou estáticos.

Aqui vai valer um pouco a questão de bom senso: apesar de tudo isso ser possível, você não deve fazer isso.

* Ao utilizarmos o decorator @classmethod nós estamos sinalizando para outros desenvolvedores que aquele método irá lidar com conteúdo interno da classe. Além disso, o @classmethod irá sempre receber no primeiro parâmetro a classe. Isso evita a repetição do nome da classe (que já discutimos o motivo de ser indesejável) e evita que caso o método seja chamado a partir de um objeto ele receba o objeto como parâmetro ao invés da classe (ou seja, que o cls seja tratado como self).
* Ao utilizarmos o decorator @staticmethod, além de sinalizar para outros desenvolvedores que o método não utilizará o conteúdo interno da classe, nós também estamos sinalizando para o Python que ele não deve utilizar "quem chamou" o método como parâmetro. No exemplo da fração, por exemplo, apesar de termos feito Fracao.mdc, note que a classe Fracao não foi passada como parâmetro para o método. Sem o decorator, ela seria o parâmetro num1 (que se comportaria como um cls), o número 48 seria o parâmetro num2, e a chamada daria erro por ter um terceiro parâmetro (o número 24). Já dentro do método simplificar, como fizemos self.mdc, o primeiro parâmetro seria o próprio objeto - e a situação toda seria semelhante.

Você pode se aprofundar mais na discussão sobre os diferentes tipos de métodos e ver outros exemplos nestes artigos do [Real Python](https://realpython.com/instance-class-and-static-methods-demystified/), [Geeks for Geeks](https://www.geeksforgeeks.org/class-method-vs-static-method-python/#:~:text=The%20difference%20between%20the%20Class,t%20access%20or%20modify%20it) e [Towards Data Science](https://towardsdatascience.com/whats-the-difference-between-static-and-class-methods-in-python-1ef581de4351) .