Atributos privados e métodos de acesso

1. Violação de encapsulamento

No primeiro capítulo, um dos princípios mais importantes que citamos na programação orientada a objeto foi o encapsulamento. Segundo este princípio, uma classe deveria ser a única responsável por manipular seus próprios atributos.

Mas considere por um minuto as classes que fizemos até agora. Sempre foi possível acessar e manipular diretamente seus atributos:

import random

class Usuario:

# Método construtor

def \_\_init\_\_(self, nome, cpf, email):

self.nome = nome

self.cpf = cpf

self.login = email

self.senha = str(random.randint(100000, 999999))

def fazer\_login(self, login, senha):

if login == self.login and senha == self.senha:

print(self.nome, 'logado com sucesso!')

else:

print('Erro! Login ou senha incorretos!')

user01 = Usuario('Rafael', 13579024681, 'rafael@ada.com')

print(user01.nome) # note que estamos acessando diretamente o atributo aqui

user01.senha = '123456' # note que estamos alterando um atributo diretamente

print(user01.senha)

user01.fazer\_login('rafael@ada.com', '123456')

Saída na tela:

Rafael

123456

Rafael logado com sucesso!

Isso não ocorre apenas no programa principal. Tecnicamente, é possível que objetos de outras classes também manipulem os atributos de objetos de nossa classe.

Considere o seguinte exemplo envolvendo duas classes:

Televisor

ControleRemoto

Pense na realidade como esses dois dispositivos funcionam. Um controle remoto está vinculado a um televisor. Ele possui, entre outras coisas, um botão para aumentar volume e outro para diminuir volume. Porém, um controle remoto não possui um volume. Quem possui volume é o televisor, e quando pedimos para o controle remoto aumentar o volume, tudo o que ele faz é repassar essa mensagem para o televisor, e este sim terá o seu volume aumentado.

Vamos esboçar nossas classes tendo isso em mente:

class Televisor:

def \_\_init\_\_(self, marca, modelo, volume, canal):

self.marca = marca

self.modelo = modelo

self.volume = volume

self.canal = canal

...

def aumentar\_volume(self):

if self.volume < 100:

self.volume += 1

else:

print('Volume já está no máximo!')

...

class ControleRemoto:

def \_\_init\_\_(self, tv):

self.tv = tv

...

def aumentar\_volume(self):

self.tv.volume += 1

televisor = Televisor('Samsung', 'UHD55SMART', 98, 'HBO')

controle = ControleRemoto(televisor)

# Aumentando o volume via tv:

televisor.aumentar\_volume() # funciona, foi pra 99

print(televisor.volume)

televisor.aumentar\_volume() # funciona, foi pra 100

print(televisor.volume)

televisor.aumentar\_volume() # não funciona, fica em 100

print(televisor.volume)

# Aumentando via controle:

controle.aumentar\_volume() # funciona, foi pra 101

print(televisor.volume)

Observe que um objeto ControleRemoto possui 1 único atributo: um objeto da classe Televisor. Seu método aumentar\_volume altera o volume do objeto Televisor.

Ao projetar a classe Televisor, previmos uma regra: volume não deveria passar de 100. Quando acionamos o método de volume do próprio Televisor, ele faz a checagem para evitar um valor inválido.

Porém, como o atributo é livremente acessível, outras classes podem alterá-lo sem passar pelo método. É o que ocorreu na classe ControleRemoto: o programador, por distração ou desconhecimento, alterou diretamente um atributo da classe Televisor dentro da classe ControleRemoto sem fazer qualquer tipo de verificação. Com isso, o ControleRemoto criou uma brecha para que tivéssemos objetos Televisor com volume superior a 100.

Isso é conhecido como furar o encapsulamento da classe Televisor, e geralmente é ruim.

2. Níveis de acesso dos atributos

Diversas linguagens orientadas a objeto oferecem uma ferramenta para ajudar a proteger o encapsulamento da classe: restringir o acesso aos seus atributos. Na maioria dessas linguagens teremos 3 níveis de acesso, e seu significado varia pouco de linguagem para linguagem. Eles tipicamente são:

Private (privado): apenas objetos da própria classe possuem acesso ao atributo.

Protected (protegido): apenas objetos da própria classe ou de classes herdeiras possuem acesso ao atributo.

Public (público): os atributos podem ser acessados livremente em qualquer ponto do código.

Em Python, porém, não temos as palavras reservadas private, protected ou public. Ao invés disso, utilizaremos underline (o símbolo \_) no início de nossos atributos:

self.atributoA = 'teste público' # sem underline: atributo público

self.\_atributoB = 'teste protegido' # 1 underline: atributo protegido

self.\_\_atributoC = 'teste privado' # 2 underlines: atributo privado

Vejamos um exemplo baseado na nossa antiga classe Televisor:

class Televisor:

def \_\_init\_\_(self, marca, modelo, volume, canal):

self.marca = marca # público

self.\_modelo = modelo # protegido

self.\_\_volume = volume # privado

self.canal = canal

# Vamos aos testes:

televisor = Televisor('Samsung', 'UHD55SMART', 98, 'HBO')

print(televisor.marca)

print(televisor.\_modelo)

print(televisor.\_\_volume) # essa linha vai provocar erro!

Como era de se esperar, ao tentarmos acessar o atributo \_\_volume, nos deparamos com um erro. Estávamos fora da classe e tentamos acessar um atributo privado. O Python age como se o atributo não existisse, afinal, fora da classe ninguém deveria mexer com ele.

A surpresa está no atributo \_modelo. Ele não deveria ser protegido? Sim. Porém, o Python possui uma política chamada "we're all consenting adults". Isso significa que não existe qualquer grau real de privacidade nessa linguagem. Se nós realmente quisermos e soubermos o que estamos fazendo, podemos acessar e modificar qualquer tipo de atributo, inclusive os privados:

A sintaxe de argumento protegido (1 underline) é apenas uma convenção. Ou seja, ela não possui efeito prático no código. Porém, outros programadores saberão que ela não deveria ser modificada fora da "família" da classe ao ver o símbolo. Da mesma forma, alguns debuggers e linters poderão alertar o desenvolvedor de que ele está usando de maneira indevida aquele atributo.

A sintaxe de argumento privado (2 underlines) de fato esconde o argumento. Porém, ela o faz utilizando uma estratégia chamada de "name mangling" (algo como "deturpação de nome"). Isso significa que ela "bagunça" o nome do atributo, de modo que o nome "oficial" não vale mais. Porém, a estratégia utilizada para o mangling é bem conhecida:

\_NomeDaClasse\_\_NomeDoAtributo

Observe o que acontece quando utilizamos esse padrão para acessar um dos atributos privados do Televisor:

print(televisor.\_Televisor\_\_volume) # imprime '98' na tela sem erro

3. Métodos de acesso

Supondo que todos os programadores sejam aderentes às boas práticas de programação e não pretendam explorar a política de consenting adults do Python para bagunçar nossos atributos, como faremos para permitir que eles possam ler ou mesmo alterar (respeitando as regras de negócio) nossos atributos?

3.1. Getters e setters

Aqui entram os métodos de acesso. Programadores de diversas linguagens orientadas a objeto os usam. O padrão é sempre mantermos nossos atributos privados e criarmos 2 métodos para cada um deles, um conhecido como getter e outro como setter. Eles funcionam da seguinte maneira:

O getter normalmente recebe o nome getNomeDoAtributo, não possui parâmetros e simplesmente retorna o valor atual do atributo.

O setter normalmente recebe o nome setNomeDoAtributo e recebe o novo valor como parâmetro. Ele deve validar o valor de acordo com as regras da classe, e caso o valor seja válido, o setter irá copiá-lo para o atributo. Setters tipicamente não retornam nada.

Sendo assim, vamos mais uma vez atualizar a nossa classe Televisor:

class Televisor:

# Note que podemos usar get/set até mesmo no construtor!

# Isso vai deixar nossa classe mais segura, evitando que objetos sejam CRIADOS com valores indevidos:

def \_\_init\_\_(self, marca, modelo, volume, canal, lista\_canais):

self.set\_lista\_canais(lista\_canais)

self.set\_marca(marca)

self.set\_modelo(modelo)

self.set\_volume(volume)

self.set\_canal(canal)

def get\_marca(self):

return self.\_\_marca

def set\_marca(self, marca):

self.\_\_marca = marca

def get\_modelo(self):

return self.\_\_modelo

def set\_modelo(self, modelo):

self.\_\_modelo = modelo

def get\_volume(self):

return self.\_\_volume

def set\_volume(self, volume):

if volume > 100:

self.\_\_volume = 100

elif volume < 0:

self.\_\_volume = 0

else:

self.\_\_volume = volume

def get\_canal(self):

return self.\_\_canal

def set\_canal(self, canal):

if canal in self.get\_lista\_canais(): # note que podemos fazer get de uma atributo mesmo dentro da classe!

self.\_\_canal = canal

elif len(self.get\_lista\_canais()) > 0:

self.\_\_canal = self.get\_lista\_canais()[0]

else:

self.\_\_canal = None

def get\_lista\_canais(self):

return self.\_\_lista\_canais

def set\_lista\_canais(self, lista):

if type(lista) == list:

self.\_\_lista\_canais = lista

else:

self.\_\_lista\_canais = []

# criando um objeto para testar:

televisor = Televisor('Samsung', 'UHD55SMART', 98, 'HBO', ['Globo', 'SBT', 'Manchete'])

# note que tentamos colocar lá em cima um canal inexistente (HBO)

# porém, o construtor usou os setters para definir os atributos

# o setter de canal prevê o seguinte comportamento: se o canal não está na lista, pega o primeiro da lista

# logo, qual será o canal sintonizado no momento?

print(televisor.get\_canal())

Note que agora mesmo nem sequer tenhamos implementado ainda um método de aumentar/diminuir volume na classe Televisor, qualquer outra classe que queira mexer no volume terá que passar pelo setter do volume, e este setter já evita valores superiores a 100 ou inferiores a 0.

Portanto, nosso amigo programador porco distraído que irá implementar a classe ControleRemoto não precisa fazer verificações adicionais e ainda assim nosso objeto está seguro:

class ControleRemoto:

def \_\_init\_\_(self, tv):

self.tv = tv

...

def aumentar\_volume(self):

atual = self.tv.get\_volume()

self.tv.set\_volume(atual+1)

controle = ControleRemoto(televisor)

# Aumentando via controle:

controle.aumentar\_volume() # Funciona, foi pra 99

print(televisor.get\_volume()) # note que sempre que queremos visualizar, utilizamos o getter

controle.aumentar\_volume() # Funciona, foi pra 100

print(televisor.get\_volume())

controle.aumentar\_volume() # Não funciona, fica em 100

print(televisor.get\_volume())

3.2. Getters e setters mais pythonicos

As vantagens de utilizar getters e setters tradicionais estão evidentes. O encapsulamento da classe Televisor está muito mais robusto. Seus atributos estão bem protegidos, e mesmo programadores pouco cuidadosos dificilmente conseguirão colocar um objeto Televisor em um estado inválido (por exemplo, com volume = 101).

Porém, houve um custo: a sintaxe ficou mais carregada. Trocamos código intuitivo por código um pouco menos intuitivo e mais carregado:

tv.volume = 10 virou tv.set\_volume(10)

vol = tv.volume virou vol = tv.get\_volume()

Felizmente, o Python fornece meios para combinarmos o melhor dos dois mundos: a sintaxe limpa do acesso direto aos atributos com a segurança da checagem de valores através de métodos. Temos duas formas de fazer isso, e abordaremos ambas.

3.2.1. 1ª forma: @property + @atributo.setter

Para nosso novo código, utilizaremos algo que ainda não apareceu. Acima de alguns de nossos métodos, colocaremos algumas palavrinhas precedidas por @. Isso é o que chamamos de decorator, e serve para sinalizar para o Python que um certo método possui algumas propriedades especiais.

O decorator @property sinaliza para o Python que o método abaixo será o getter de algum atributo. Já o @atributo.setter indica que o método abaixo servirá como setter para o atributo chamado "atributo".

Qual a vantagem deste método? Nós, que estamos desenvolvendo a classe, iremos enxergá-los como métodos. Porém, quem utiliza a classe, irá enxergá-los como um atributo. Ou seja, eles poderão ler o atributo diretamente:

valor = objeto.atributo

print(objeto.atributo)

# etc

objeto.atributo = 'novo valor'

Porém, ele não acessará o atributo. O Python irá substituir esses supostos acessos ao atributo por chamadas para os métodos automaticamente.

Vamos mais uma ver manipular nosso Televisor (ou uma versão simplificada dele):

class Televisor:

def \_\_init\_\_(self, volume):

# IMPORTANTE:

# não existe atributo volume!

# aqui estamos implicitamente chamando o setter definido lá embaixo!

# é dentro do próprio setter que o atributo \_\_volume será criado ao receber um valor pela primeira vez

self.volume = volume

# getter para volume

# note que utilizamos o nome que gostaríamos que aparecesse como atributo para os outros programadores (volume)

@property

def volume(self):

return self.\_\_volume

# setter para volume

# note que utilizamos o nome que gostaríamos que aparecesse como atributo para os outros programadores

@volume.setter

def volume(self, valor):

if valor > 100:

self.\_\_volume = 100

elif valor < 0:

self.\_\_volume = 0

else:

self.\_\_volume = valor

tv = Televisor(102) # tentando passar um volume inválido...

print(tv.volume) # note qual valor foi parar lá...

tv.volume = 10 # passando um volume "bem comportado"

print(tv.volume) #... e funciona!

tv.volume = -5 # outro volume estranho...

print(tv.volume) # e o setter nos salvou de novo

tv.volume += 1

tv.volume = tv.volume + 1

print(tv.volume)

Saída na tela:

100

10

0

2

Note que o trabalho do programador da classe ControleRemoto tornou a ficar mais fácil. O código abaixo, apesar de mais simples, segue seguro, já que self.tv.volume não irá mais acessar diretamente um atributo de Televisor, mas passará por um setter:

class ControleRemoto:

def \_\_init\_\_(self, tv):

self.tv = tv

def aumentar\_volume(self):

self.tv.volume += 1

3.2.2. 2ª forma: função property

Como pudemos observar nos exemplos acima, o uso dos decorators melhorou significativamente nosso código. O excesso de "get" e "set" para todos os lados deixava nosso código verboso e com mais carinha de Java do que de Python. Ao aplicarmos os decorators, voltamos a ter carinha de Python.

Porém, há quem ache burocrático escrever diversas arrobas dentro da classe. Felizmente, existe uma segunda estratégia:

Criar métodos get e set sem os decorators.

(Opcional) Tornar esses métodos privados, para não confundir outros programadores.

Criar um "atalho" passando esses métodos para a função property().

Vamos mexer com a nossa classe Televisor pela última vez:

class Televisor:

def \_\_init\_\_(self, volume):

self.volume = volume

# Note o \_\_ no início. Esses métodos são privados. Isso é opcional.

def \_\_get\_volume(self):

return self.\_\_volume

def \_\_set\_volume(self, valor):

if valor > 100:

self.\_\_volume = 100

elif valor < 0:

self.\_\_volume = 0

else:

self.\_\_volume = valor

# Tendo o getter e o setter, criamos a propriedade:

volume = property(\_\_get\_volume, \_\_set\_volume)

# Note que passamos as funções SEM parênteses

# Não estamos chamando a função para passar seus retornos. Estamos passando as funções em si.

Note que a classe acima se comportará EXATAMENTE como a da primeira forma. Inclusive iremos repetir os mesmos exemplos e chegar nos mesmos resultados:

tv = Televisor(102) # tentando passar um volume inválido...

print(tv.volume) # note qual valor foi parar lá...

tv.volume = 10 # passando um volume "bem comportado"

print(tv.volume) #... e funciona!

tv.volume = -5 # outro volume estranho...

print(tv.volume) # e o setter nos salvou de novo

Saída em tela:

100

10

0

Saiba mais: decorators

Isso não pertence ao deste módulo, mas fizemos uma simplificação ao explicar o que é decorator, e aqui está uma explicação mais completa sobre o que eles são, caso você tenha curiosidade.

Lá em cima, definimos decorators como um jeitinho de marcar uma função como possuindo algum tipo de propriedade especial.

Na realidade, decorators tem a ver com a ideia de clausuras, ou closures, uma técnica que envolve utilizarmos uma função para gerar outras funções com base em seus parâmetros.

A função abaixo funciona como uma "fábrica" de funções exponenciais: ela recebe um número e retorna uma função, e essa função retornada sempre irá elevar o seu parâmetro àquele número:

def fabrica\_expoentes(expoente):

def exponencial(base):

return base\*\*expoente

return exponencial

quadrado = fabrica\_expoentes(2)

cubo = fabrica\_expoentes(3)

print(f'Quadrado de 2 = {quadrado(2)}')

print(f'Quadrado de 3 = {quadrado(3)}')

print(f'Cubo de 2 = {cubo(2)}')

Primeiro: sim, é possível uma função retornar outra função! Segundo, uma função pode também ser passada como parâmetro, e essa vai ser a base para nosso decorator. No exemplo abaixo, a funcao\_externa é o nosso decorator, e podemos utilizar @funcao\_externa acima de qualquer função. Isso fará com que a função seja passada como parâmetro para funcao\_externa, que por sua vez executará a função interna, e esta possui uma chamada para a função decorada. Soou confuso? Observe o exemplo com atenção e execute para ver o resultado na tela.

def funcao\_externa(funcao\_parametro):

# Note que a função interna possui acesso ao parâmetro da função externa

def funcao\_interna():

print('Olá')

funcao\_parametro()

print('Mundo')

return funcao\_interna()

# Criando uma função nova e "decorando" com a funcao\_externa:

@funcao\_externa

def teste():

print('teste')

Note que nem sequer precisamos chamar a função teste. Ao declará-la com um decorator, automaticamente a função que recebe o nome do decorator foi chamada passando a função "decorada" como parâmetro. Especificamente em nosso exemplo, a função decorada acaba chamando a função passada, mas isso não é necessário.

A explicação ainda simplificada, mas factualmente correta para o uso dos decorators para criar getters e setters é a seguinte: nós estamos criando a lógica de cada getter e setter, e ao decorá-las, estamos passando essa lógica (nossas funções) para uma outra função que irá criar os getters/setters (a tal da property) padrões do Python para nós de maneira automática.

Leia mais sobre o assunto aqui. Você também pode consultar a PEP 318 que descreve os decorators em Python.