Herança e polimorfismo

Logo no primeiro capítulo deste material mencionamos 4 pilares básicos que devemos considerar na hora de planejar nossas classes: encapsulamento, abstração, herança e polimorfismo.

Vimos como reforçar os dois primeiros pilares através de recursos como níveis de acesso aos atributos, propriedades (getters e setters) e métodos mágicos.

Agora iremos estudar os últimos dois pilares: herança e polimorfismo.

## 1. Herança

### 1.1. Classe base e classe derivada

A herança tem como objetivo maximizar a reutilização de código. A ideia básica é que uma classe pode reaproveitar atributos e métodos de outra classe, sem a necessidade de copiar e colar seu código. Mudanças feitas na classe base serão reaproveitadas pela classe derivada.

Em materiais diversos você verá termos como classe base, superclasse e (mais informal) classe pai/mãe se referindo à classe original, e termos como classe derivada, subclasse e (mais informal) classe filha se referindo à nova classe que irá herdar o conteúdo da classe original.

Para indicar que uma classe herdará de outra classe em Python, basta colocar o nome da classe base entre parênteses ao criar a classe derivada, como no exemplo abaixo.

class Base:

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

self.x = 0

self.y = 1

def metodo\_base(self):

print(f'Oi, estou na classe Base e meus atributos valem {self.x} e {self.y}')

class Derivada(Base):

pass # vamos deixar essa classe vazia por enquanto...

Note que ainda não colocamos nada em nossa classe Derivada, apenas a criamos e indicando a herança. Vamos instanciar um objeto e testar seu comportamento:

obj\_derivado = Derivada()

print(obj\_derivado.x)

print(obj\_derivado.y)

obj\_derivado.metodo\_base()

Saída:

0

1

Oi, estou na classe Base e meus atributos valem 0 e 1

Note que o construtor da classe Base foi utilizado pelo nosso objeto. Afinal, os atributos x e y foram criados e ainda receberam os valores especificados naquele método.

Além disso, o método metodo\_base também estava disponível para nosso objeto da classe Derivada.

A ideia é que a classe Derivada seja uma especialização da classe Base, portanto não faz sentido simplesmente herdarmos tudo e não criarmos mais nada. Vamos criar um método novo em nossa subclasse e realizar alguns testes:

class Derivada(Base):

def metodo\_derivado(self):

print(f'Oi, estou na classe Derivada e meus atributos valem {self.x} e {self.y}')

# Criando um objeto de cada classe:

obj\_base = Base()

obj\_derivado = Derivada()

# Acessando os atributos:

print(obj\_base.x, obj\_base.y)

print(obj\_derivado.x, obj\_derivado.y)

# Acessando os métodos:

obj\_base.metodo\_base()

try:

obj\_base.metodo\_derivado()

except:

print('A classe Base não possui o método metodo\_derivado!')

obj\_derivado.metodo\_base()

obj\_derivado.metodo\_derivado()

Saída:

0 1

0 1

Oi, estou na classe Base e meus atributos valem 0 e 1

A classe Base não possui o método metodo\_derivado!

Oi, estou na classe Base e meus atributos valem 0 e 1

Oi, estou na classe Derivada e meus atributos valem 0 e 1

Note que nosso objeto da subclasse possui todos os métodos: tanto os métodos implementados em sua própria classe quanto os métodos da superclasse.

O oposto não é verdade: o objeto da superclasse não possui métodos da subclasse.

### 1.2. Sobrecarregando métodos

Muitas vezes um objeto de uma classe derivada pode até realizar as mesmas ações que o objeto da classe base, mas ele pode realizar essas ações de maneira diferente.

Neste caso, podemos sobrecarregar os seus métodos. Sobrecarregar um método consiste em criar mais de uma versão dele em diferentes classes.

Quando criamos nossa classe, o Python primeiro irá adotar todos os atributos e métodos determinados naquela classe. Em seguida, ele irá adotar os métodos e atributos de sua superclasse caso eles não existam. Ou seja, se houver uma versão "nova" de um método na subclasse, é esse método que valerá para os objetos dessa classe.

Vamos atualizar nossas classes para verificar esse comportamento:

class Base:

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

self.x = 0

self.y = 1

def metodo\_base(self):

print(f'Oi, estou na classe Base e meus atributos valem {self.x} e {self.y}')

class Derivada(Base):

# Sobrecarregando um método herdado:

def metodo\_base(self):

print(f'Oi, estou na classe Derivada e sobrecarreguei meu metodo\_base!')

def metodo\_derivado(self):

print(f'Oi, estou na classe Derivada e meus atributos valem {self.x} e {self.y}')

obj\_base = Base()

obj\_derivado = Derivada()

obj\_base.metodo\_base()

obj\_derivado.metodo\_base()

Saída:

Oi, estou na classe Base e meus atributos valem 0 e 1

Oi, estou na classe Derivada e sobrecarreguei meu metodo\_base!

### 1.3. Acessando métodos da classe base: super()

Bom, podemos observar um problema. Imagine que a gente gostaria de acrescentar uma funcionalidade nova em um método na nossa classe derivada, mas de resto gostaríamos que toda a lógica deste mesmo método na classe original fosse reaproveitado. Se estamos sobrecarregando o método, teremos que copiar novamente toda a lógica, certo?

A resposta nunca é copiar código. Lembre-se que um dos objetivos da programação orientada a objeto é melhorar a reutilização de código, e a herança em particular serve especificamente para isso. Naturalmente, existe uma ferramenta para evitar essa cópia de código: a função super.

Essa função serve para permitir que, dentro de um método de uma classe, você acesse métodos de sua classe base. Sendo assim, você pode fazer sua lógica nova e também chamar a lógica do método original. Vejamos com um exemplo:

class Base:

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

self.x = 0

self.y = 1

def metodo\_base(self):

print(f'Oi, estou na classe Base e meus atributos valem {self.x} e {self.y}')

class Derivada(Base):

# Sobrecarregando um método herdado:

def metodo\_base(self):

super().metodo\_base() # chamando o metodo\_base original

print(f'Oi, TAMBÉM estou na classe Derivada e sobrecarreguei meu metodo\_base!')

def metodo\_derivado(self):

print(f'Oi, estou na classe Derivada e meus atributos valem {self.x} e {self.y}')

obj\_derivado = Derivada()

obj\_derivado.metodo\_base()

Saída:

Oi, estou na classe Base e meus atributos valem 0 e 1

Oi, TAMBÉM estou na classe Derivada e sobrecarreguei meu metodo\_base!

Note que até o momento falamos apenas sobre métodos: criar métodos, sobrecarregar métodos, acessar métodos da classe base... E os atributos?

Uma estratégia para criar alguns atributos novos e reaproveitar os da classe base sem precisar redigitar código é sobrecarregando o construtor da classe derivada, adicionando lógica nova e usando o super() para chamar o construtor da classe original.

No exemplo abaixo, a classe Animal exige que todo bichinho tenha um nome. Especificamente os membros de Cachorro também deverão ter uma raça. Note como a sobrecarga do construtor permite isso:

class Animal:

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

self.nome = nome

def fala(self):

print(self.nome, 'faz barulho.')

class Gato(Animal):

# Sem construtor - herda de Animal

# Sobrecarrega a fala:

def fala(self):

print(self.nome, 'faz miau.')

class Cachorro(Animal):

# Sobrecarrega o construtor:

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

self.raca = raca # adiciona o atributo diferente

super().\_\_init\_\_(nome) # chama o método da superclasse para lidar com o resto

# Sobrecarrega a fala:

def fala(self):

print(f'{self.nome}, um {self.raca}, faz au au.')

class Dinossauro(Animal):

...

# não sobrecarrega nem cria nada - herda tudo de Animal

doguinho = Cachorro('Bidu', 'schnauzer')

gatinho = Gato('Mingau')

dino = Dinossauro('Horácio')

doguinho.fala() # note que neste método usamos o atributo "raca"

gatinho.fala()

dino.fala()

Saída:

Bidu, um schnauzer, faz au au.

Mingau faz miau.

Horácio faz barulho.

### 1.4. Herança múltipla

Há diferentes maneiras de uma classe ser herdeira de múltiplas classes. A mais óbvia, e que é bem aceita em todas as linguagens orientadas a objetos populares, é a herança vertical: a classe C, herda da classe B, que por sua vez herda da classe A.

De fato, todas as classes de Python são por padrão herdeiras de uma classe chamada de object - portanto, quando seu Gato herda de Animal, ele por tabela também está herdando de object.

A ordem de prioridade é a mesma que vimos para herança simples: sempre irá prevalecer a versão mais "recente".

class Avo:

def metodo\_teste(self):

print('Estou na classe Avo')

class Pai(Avo):

def metodo\_teste(self):

print('Estou na classe Pai')

class Filho(Pai):

def metodo\_teste(self):

print('Estou na classe Filho')

a = Avo()

p = Pai()

f = Filho()

a.metodo\_teste()

p.metodo\_teste()

f.metodo\_teste()

Saída:

Estou na classe Avo

Estou na classe Pai

Estou na classe Filho

Os problemas começam a surgir quando temos herança múltipla horizontal - ou seja, uma mesma classe herdando simultaneamente de duas ou mais classes. Nestes casos, a ordem do Python é a seguinte:

* Atributos e métodos na própria classe primeiro
* Busca da esquerda para a direita
* Busca em profundidade antes da largura

O que isso significa? Considere o seguinte esquema de herança:

A

^

|

|

|

B C

^ ^

| |

|------

|

D

Considere que D herda de B e C, e B herda de A. Em qual ordem o Python irá procurar por um certo método?

* D
* B (primeira superclasse da esquerda para a direita)
* A (busca em profundidade a partir de B)
* C (segunda da esquerda para a direita)

O fato de existir uma ordem específica ajuda a evitar o problema do diamante:

A

^

/ \

/ \

B C

^ ^

\ /

\ /

D

Algumas linguagens podem apresentar comportamento indefinido nesse caso. Outras chegam a proibir herança múltipla "horizontal" justamente por conta de problemas do tipo. Por isso é importante que haja essa padronização. Assim fica claro a ordem de prioridade para um objeto de D adotar um método: D, B, A, C.

Note como no exemplo abaixo a classe Filha herda todo o conteúdo de Pai primeiro (primeira classe da esquerda para a direita na herança), e herda o conteúdo de Mae que Pai não possui:

class Pai:

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

self.x = 0

print('Pai')

class Mae:

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

self.x = 1

print('Mãe')

def teste(self):

print('método único da classe mãe')

class Filha(Pai, Mae):

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

super().\_\_init\_\_()

f = Filha()

print(f.x)

f.teste()

Saída:

Pai

0

método único da classe mãe

## 2. Polimorfismo

A palavra polimorfismo vem do grego e significa "muitas formas". A ideia é que um objeto pertencente a uma classe pode ser tratado como se pertencesse a outras classes.

### 2.1. Polimorfismo por herança

A primeira forma de polimorfismo que devemos considerar é justamente pela herança. O Python reconhece os membros de uma classe como sendo também membros de sua classe base:

print('doguinho é Cachorro:', isinstance(doguinho, Cachorro))

print('gatinho é Gato:', isinstance(gatinho, Gato))

print('gatinho é Cachorro:', isinstance(gatinho, Cachorro))

print('doguinho é Animal:', isinstance(doguinho, Animal))

print('gatinho é Animal:', isinstance(gatinho, Animal))

Saída:

doguinho é Cachorro: True

gatinho é Gato: True

gatinho é Cachorro: False

doguinho é Animal: True

gatinho é Animal: True

Mas por que isso funciona bem? Em um conjunto bem planejado de classes, uma classe derivada deve ter todos os atributos e métodos da classe base, mesmo que alguns dos métodos tenham sido sobrecarregados. Sendo assim, qualquer pedaço de código (como uma função) esperando um objeto da classe base conseguirá lidar com objetos de suas classes derivadas sem dificuldades.

### 2.2. Duck Typing

Há diversas formas diferentes de se implementar polimorfismo em diferentes linguagens. Em linguagens como o Java, por exemplo, existe o conceito de interfaces, onde algumas classes podem assumir o compromisso de implementar certos métodos, e em troca, funções podem aceitar qualquer objeto de qualquer classe que implemente a interface.

No Python, como de costume, as coisas são um pouco mais flexíveis e livres de regra. O Python utiliza o conceito de duck typing:

If it walks like a duck and it quacks like a duck, it's a duck.

(Se anda como um pato e grasna como um pato, é um pato.)

Ou seja, não é necessário qualquer grande formalismo ou declaração: se uma função recebe um objeto como parâmetro e chama um método específico daquele objeto, qualquer objeto que implemente esse método irá funcionar com essa função.

Veja como exemplo a função abaixo:

def somatorio(\*numeros):

soma = numeros[0]

for n in numeros[1:]:

soma += n

return soma

Ela claramente é uma função para somar números, correto? Provavelmente podemos passar int e float para ela.

print(somatorio(1, 3, 5, 7, 9))

print(somatorio(2.718, 3.1415))

E se passarmos str?

print(somatorio('String ', 'possui ', 'o ', 'metodo ', '\_\_add\_\_, ', 'portanto, ', 'irá ', 'funcionar!'))

Bom, a resposta já está dada no trecho acima: a operação sendo feita sobre os parâmetros da função é a soma. Portanto, objetos de qualquer classe que implemente o método \_\_add\_\_ irão funcionar. O "andar do pato" da nossa função é conseguir realizar a operação de soma. 🦆

Essa função irá funcionar, inclusive, com objetos da classe Fracao que desenvolvemos no capítulo de métodos mágicos.

Você pode consultar uma explicação um pouco mais profunda sobre o sistema de tipagem de Python no [Towards Data Science](https://towardsdatascience.com/duck-typing-python-7aeac97e11f8) e sobre herança e polimorfismo [aqui](https://levelup.gitconnected.com/oop-inheritance-and-polymorphism-in-python-fd04878e4ffa) e [aqui](https://www.tutorialspoint.com/object_oriented_python/object_oriented_python_inheritance_and_ploymorphism.htm).