Transcrição curso python – avançando na orientação a objetos

Olá! Esta é a continuação do curso de Orientação a Objetos com Python. Nosso objetivo será reforçar o conhecimento sobre esse assunto, falaremos sobre alguns aspectos idiomáticos da linguagem, que nos auxiliarão a atuar com Python. Além disso, trataremos aspectos específicos da Orientação a Objeto e como eles são aplicados nesta linguagem.

Por exemplo, **herança, polimorfismo e Duck Typing**. Falaremos também sobre herança múltipla, um aspecto que pode assustar algumas pessoas, mas apresentaremos de forma sucinta, deixando claro o conceito.

Abordaremos o assunto **mixin**, falaremos de forma a garantir maior segurança quando trabalharmos com um código Python. Outra situação que podemos encontrar será o código no qual seja exigida uma interface mais ligada aos aspectos desta linguagem.

Tentaremos introduzir esse conhecimento para a linguagem Pyhton ligado a O.O., um tema que este será o foco nesta parte, mas que provavelmente será abordado em outros cursos.

Espero que você goste, vamos praticar bastante!

Começaremos o curso abordando o nosso primeiro assunto, no qual **relembraremos classes e objetos**. Nós pensaremos em trechos de uma aplicação para absorver melhor esses conceitos de O.O. no Python e tentar entendê-los de uma forma mais tranquila.

Primeiramente lidaremos com este trecho de aplicação, com a qual teremos o controle de playlist de programa de TV. Podemos lidar com filmes, séries, documentários e afins. Começaremos pelo filme - cujo modelo faremos no Python.

Um filme tem as seguintes características:

* Nome;
* ano;
* duração.

Uma série, por sua vez, possui:

* Nome;
* ano;
* temporadas.

A partir dessas informações, tentaremos construir um modelo usando a linguagem Python. Para isto, abriremos o ***PyCharm*** e criaremos um projeto que receberá o nome python3oo2. Com o número 2 indicamos que é um arquivo referente à segunda parte do curso de O.O..

No interpretador, deixaremos o arquivo configurado com "Python 3.6", por ele já estar instalado na máquina. Em seguida, criaremos o arquivo Modelo.py, nome geralmente usado ao criarmos conceitos de classe os quais **representarão um domínio no nosso sistema**. Criados o projeto e o modelo, começaremos a trabalhar com o código.

Inicialmente, adicionaremos class Filme - o que precisaremos fazer para criar uma classe? Podemos definir o que será necessário para criar um objeto deste tipo, e teríamos que começar do inicializador. Isso é necessário sempre?

O Python possui uma flexibilidade, que permite que criemos um objeto ao colocarmos pass e indicarmos que simplesmente estamos passando-o. Por exemplo, trabalharemos com o filme vingadores, em seguida verificaremos se o objeto está pronto, com print().

class Filme:

pass

vingadores = Filme()

print(vingadores)

Até o momento, vingadores não possui um atributo, a ideia é simplesmente demonstrar que conseguimos criar um objeto de uma classe de forma muito flexível. Se clicarmos com o botão direito e selecionarmos "Run 'modelo'", o PyCharm vai imprimir no console as informações, algumas da memória do Python:

C:\Users\Alura\Appdata\Local\Progrms\Python\Python36-32\pythosn.exe\luan.silva

<\_mains\_.Filme object at 0x050220F0>

Process finished with code 0

Assim, são exibidas as informações correspondentes ao tentarmos imprimir um objeto que seja do tipo **filme**. Agora vamos modelar essa classe, que na verdade não possui apenas pass, e sim atributos como def, com o qual definiremos o inicializador ( \_\_init\_\_):

class Filme:

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

self.nome = nome

self.ano = ano

self.duracao = duracao

vingadores = Filme()

print(vingadores.nome)

O inicializador \_\_init\_\_ sempre receberá self, e também passaremos outros atributos: nome, ano e duracao. Com estes três devemos preencher os três valores do objeto na nossa instancia. Por isso usamos self.nome, self.ano e self.duracao. A partir de agora podemos pedir para que se imprima especificamente o nome do filme, passando para print o vingadores.nome.

Se executarmos novamente o modelo, veremos uma mensagem de erro: TypeError: \_\_init\_\_() missing 3 required positional arguments - fomos avisados de que temos um inicializador com mais argumentos, e que faltou passá-los no código.

vingadores = Filme('vingadores - guerra infinita', 2018, 160)

print(vingadores.nome)

Se executarmos novamente com "Run 'modelo'", veremos impresso:

vingadores - guerra infinita

Em seguida, incluiremos class Serie, podendo-se inclusive aproveitar o código da primeira.

class Filme:

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

self.nome = nome

self.ano = ano

self.duracao = duracao

class Serie:

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

self.nome = nome

self.ano = ano

self.temporadas = temporadas

vingadores = Filme('vingadores - guerra infinita', 2018, 160)

print(vingadores.nome)

atlanta = Serie('atlanta', 2018, 2)

print(f'Nome: {atlanta.nome} - Ano: {atlanta.ano} - Temporadas: {atlanta.temporadas}')

Passaremos como série atlanta, isto é, será igual a Serie(), com os valores correspondentes em relação aos atributos nome, ano e temporadas. No caso, ela tem o total de 2 temporadas, e imprimiremos além do nome, a anotação de formatação do Python 3.6. Dentro de {} incluiremos o atributo que queremos imprimir. Observe que, desta vez, adicionamos outras informações na string.

Ao executarmos o código, veremos impresso no console:

vingadores - guerra infinita

Nome: atlanta - Ano: 2018 - Temporadas: 2

Vamos tornar esse informação mais compreensível agregando mais informações em Filme.

class Filme:

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

self.nome = nome

self.ano = ano

self.duracao = duracao

class Serie:

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

self.nome = nome

self.ano = ano

self.temporadas = temporadas

vingadores = Filme('vingadores - guerra infinita', 2018, 160)

print(f'Nome: {vingadores.nome} - Ano: {vingadores.ano} ' -

f'- Duração: {vingadores.duracao}')

atlanta = Serie('atlanta', 2018, 2)

print(f'Nome: {atlanta.nome} - Ano: {atlanta.ano} ' -

f'- Temporadas: {atlanta.temporadas}')

A impressão ficará da seguinte maneira:

Nome: vingadores - guerra infinita - Ano: 2018 - Duração: 160

Nome: atlanta - ano: 2018 - Temporadas: 2

Relembrando como é criar uma classe e um objeto da mesma... Para onde isso nos leva?

Por enquanto temos nossa aplicação funcionando, mas é normal termos que lidar com informações novas. As aplicações mudam, é normal. No caso, temos uma informação nova: além das informações já disponibilizadas, teremos a quantidade de ***likes***. Será necessário implementarmos uma funcionalidade que indique a popularidade daquela série.

Teremos também uma regra que nos dirá que, sempre que for um filme for inserido, colocaremos um nome específico. No momento em que imprimirmos filme.nome, devemos fazê-lo de forma que Meu Filme esteja com as primeiras letras em maiúsculo.

Nós ainda estamos imprimindo meu filme de forma diferente, mas foi algo dentro do código que conseguiu exibir o texto da forma modificada. Falta implementarmos esta regra de negócio tanto para filmes como para séries.

Vemos que já temos código para fazer, essa será nossa missão a seguir. Mas antes, recomendo que você faça os exercícios de fixação dos conceitos vistos até agora.

Vamos continuar a escrever os trechos de código referentes a Filme e Serie. Falamos anteriormente que acrescentaríamos algumas regras: ambos os tipos devem informar a quantidade de **likes** recebida, ou seja, eles passarão a ideia de sua popularidade para a classe.

Outro ponto será a inclusão de um comportamento ao exibirmos um nome de determinada maneira, quando criarmos um objeto do tipo Filme, ao incluirmos meu filme ou meu nome e o colocarmos em nome. Quando passarmos esse elemento pelo inicializador, imprimiremos um nome e este será exibido com as **primeiras letras de cada palavra em maiúsculo** como em Meu Filme. O mesmo deverá acontecer em relação às séries.

Resolveremos essa questão trabalhando no código, de volta à IDE:

class Filme:

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

self.nome = nome

self.ano = ano

self.duracao = duracao

class Serie:

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

self.nome = nome

self.ano = ano

self.temporadas = temporadas

vingadores = Filme('vingadores - guerra infinita', 2018, 160)

print(f'Nome: {vingadores.nome} - Ano: {vingadores.ano} ' -

f'- Duração: {vingadores.duracao}')

atlanta = Serie('atlanta', 2018, 2)

print(f'Nome: {atlanta.nome} - Ano: {atlanta.ano} ' -

f'- Temporadas: {atlanta.temporadas}')

O próximo passo será incluir a informação do nome e depois, transformá-lo. Como podemos fazer isso? Investigaremos como funciona a biblioteca de string do Python. Começaremos experimentando a função capitalize() para vermos como isso irá refletir no momento da impressão.

class Filme:

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

self.nome = nome.capitalize()

self.ano = ano

self.duracao = duracao

No console, o nome do filme será impresso da seguinte forma:

Nome: Vingadores - guerra infinita - Ano: 2018 - Duração: 160

Nome: atlanta - ano: 2018 - Temporadas: 2

A primeira letra do nome já foi impressa em maiúsculo, mas as demais não. Ao incluirmos capitalize(), podemos "capitalizar" a primeira letra para deixá-la em caixa alta, mas ainda não é bem o que queremos. Vamos testar outra função, title(), e ver se conseguimos modificar o comportamento. Faremos o mesmo na série.

class Filme:

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

self.nome = nome.title()

self.ano = ano

self.duracao = duracao

class Serie:

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

self.nome = nome.title()

self.ano = ano

self.temporadas = temporadas

Com o uso do title() teremos o resultado esperado:

Nome: Vingadores - Guerra Infinita - Ano: 2018 - Duração: 160

Nome: Atlanta - ano: 2018 - Temporadas: 2

Agora falta implementar os likes. Porém, não basta incluir o valor dos likes no momento em que criamos Filme, pois este será um elemento incremental. Como faremos para definir algo que será modificado posteriormente, e que não será definido no momento da criação do objeto? Devemos ter um método para inserir a informação e modificar seu estado recém criado.

Também definiremos um método com a utilidade de dar like, sem receber parâmetros e que, no fim, adicionará +1 à contagem de likes.

class Filme:

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

self.nome = nome.title()

self.ano = ano

self.duracao = duracao

def dar\_like(self):

self.likes += 1

Conseguimos dar likes, mas e a exibição desse valor? Para isto, podemos criar uma função, ou colocar likes dentro do construtor. Mas dessa vez sua utilidade não será para definir um valor específico, e sim, inicial. No caso, será igual a 0.

class Filme:

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

self.nome = nome.title()

self.ano = ano

self.duracao = duracao

self.likes = 0

def dar\_like(self):

self.likes += 1

O próximo passo será replicar o mesmo comportamento para Serie.

class Serie:

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

self.nome = nome.title()

self.ano = ano

self.temporadas = temporadas

self.likes = 0

def dar\_like(self):

self.likes += 1

Lembre-se que no construtor não iremos receber likes, ele sempre começará com o valor igual a 0, sem termos influência do usuário. Agora, adicionaremos o texto da string para exibirmos a popularidade.

vingadores = Filme('vingadores - guerra infinita', 2018, 160)

print(f'Nome: {vingadores.nome} - Ano: {vingadores.ano} ' -

f'- Duração: {vingadores.duracao}' - Likes: {vingadores.likes}')

atlanta = Serie('atlanta', 2018, 2)

print(f'Nome: {atlanta.nome} - Ano: {atlanta.ano} ' -

f'- Temporadas: {atlanta.temporadas}' - Likes: {atlanta.likes}')

Temos muitos trechos de código repetido, mas resolveremos isso mais adiante. Em seguida, executaremos o código:

Nome: Vingadores - Guerra Infinita - Ano: 2018 - Duração: 160 - Likes: 0

Nome: Atlanta - ano: 2018 - Temporadas: 2 - Likes: 0

Para adicionarmos um like, podemos usar a função dar\_like(), o que faremos duas vezes para atlanta e uma para vingadores.

vingadores = Filme('vingadores - guerra infinita', 2018, 160)

print(f'Nome: {vingadores.nome} - Ano: {vingadores.ano} ' -

f'- Duração: {vingadores.duracao}' - Likes: {vingadores.likes}')

vingadores.dar\_like()

atlanta = Serie('atlanta', 2018, 2)

print(f'Nome: {atlanta.nome} - Ano: {atlanta.ano} ' -

f'- Temporadas: {atlanta.temporadas}' - Likes: {atlanta.likes}')

atlanta.dar\_like()

atlanta.dar\_like()

Entretanto, se executarmos o código, o valor de Likes continuará igual a 0. Por que isso aconteceu? A questão é que incluímos o like após o print(), e resolveremos o assunto se mudarmos a ordem das linhas.

vingadores = Filme('vingadores - guerra infinita', 2018, 160)

vingadores.dar\_like()

print(f'Nome: {vingadores.nome} - Ano: {vingadores.ano} ' -

f'- Duração: {vingadores.duracao}' - Likes: {vingadores.likes}')

atlanta = Serie('atlanta', 2018, 2)

atlanta.dar\_like()

atlanta.dar\_like()

print(f'Nome: {atlanta.nome} - Ano: {atlanta.ano} ' -

f'- Temporadas: {atlanta.temporadas}' - Likes: {atlanta.likes}')

Com os print()s depois dos likes, teremos o seguinte resultado:

Nome: Vingadores - Guerra Infinita - Ano: 2018 - Duração: 160 - Likes: 1

Nome: Atlanta - ano: 2018 - Temporadas: 2 - Likes: 2

Já conseguimos dar likes, fizemos as tarefas propostas, mas temos algumas falhas. Por exemplo, definimos que o nome é titularizado, porém, se adicionarmos uma nova linha com atlanta.nome no código, logo abaixo de atlanta, ele deixará de fazer a alteração nas letras.

atlanta = Serie('atlanta', 2018, 2)

atlanta.nome = 'atlanda'

atlanta.dar\_like()

atlanta.dar\_like()

print(f'Nome: {atlanta.nome} - Ano: {atlanta.ano} ' -

f'- Temporadas: {atlanta.temporadas}' - Likes: {atlanta.likes}')

No console, veremos:

Nome: Vingadores - Guerra Infinita - Ano: 2018 - Duração: 160 - Likes: 1

Nome: atlanta - ano: 2018 - Temporadas: 2 - Likes: 2

O problema aconteceu porque **não estamos protegendo o nome do atributo**, este é alterado apenas no momento da criação, mas e depois? Como definiremos set do atributo, ou seja, seu valor? Veremos como proteger a criação do atributo mais adiante.

A seguir, precisaremos proteger o nosso atributo nome. Se analisarmos o código, devemos fazer o mesmo e proteger likes pois, da maneira como foi construído, é possível fazermos uma alteração, como vemos abaixo:

class Filme:

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

self.nome = nome.title()

self.ano = ano

self.duracao = duracao

self.likes = 0

def dar\_like(self):

self.likes += 1

Estes atributos estão públicos, totalmente disponíveis para alteração. Como podemos deixá-los mais restritos? Vimos no curso anterior que podemos definir esses atributos como privados e adicionar getter e setter para apresentar os valores que estão dentro dos atributos, incluindo-os antes deste \_\_ (dois underlines).

class Filme:

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

self.\_\_nome = nome.title()

self.ano = ano

self.duracao = duracao

self.\_\_likes = 0

def dar\_like(self):

self.likes += 1

Os atributos ano e duracao não são tão importantes quanto nome e a quantidade de likes. Os elementos que não necessitam de proteção, sem nenhuma lógica, ficarão públicos. Na linguagem Python, é raro deixarmos tudo como private, isto é mais recorrente em outras linguagens, que têm aspectos diferentes. No Python, o recomendável é mantermos nosso código simples para que ele seja de fácil compreensão e mais expressivo.

O próximo passo será repetir essa ação para a classe Serie:

class Serie:

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

self.\_\_nome = nome.title()

self.ano = ano

self.temporadas = temporadas

self.\_\_likes = 0

def dar\_like(self):

self.likes += 1

Em breve, deixaremos de copiar o código. Nós titularizamos o texto e protegemos o \_\_likes, que só pode ser alterado por meio do método dar\_like(). Porém, como não conseguimos mais acessar likes, nosso código vai quebrar, e veremos várias mensagens de erro no console.

Traceback (most recent call last):

File "D:/luan.silva/stages/python3oo2/modelo.py", line 26, in <module>

vingadores.dar\_like()

File "D:/luan.silva/stages/python3oo2/modelo.py", line 11, in dar\_like

self.likes += 1

AttributeError: 'Filme' object has no attribute 'likes'

Process finished with exit code 1

Somos avisados de que não foi encontrado o atributo likes. Desta forma, não é mais possível acessá-lo. Teremos que definir outra forma de acesso para esse atributo - por exemplo, criaremos algum atributo para pegar o like. Se tentássemos criar o método get\_likes(), teríamos que alterar todos os códigos que dependem de likes. Tem alguma forma de não deixarmos nosso código quebrar, por exemplo, criando um assessor para este atributo?

Quando trabalharmos com nome, teremos que pegar get\_nome, onde temos vingadores.nome e atlanta.nome, veremos nosso código quebrar. Todos os objetos que forem acessados de forma semelhante vão quebrar. Esta necessidade de alterar diferentes trechos de código não é uma boa prática. A melhor opção seria construirmos um código de forma que a classe não afete os dependentes.

Como faremos para diminuir o impacto? O Python nos oferece uma forma de fazermos isso. Em vez de usarmos getter, podemos usar @property. Com isso, criamos um elemento parecido com um atributo.

class Filme:

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

self.\_\_nome = nome.title()

self.ano = ano

self.duracao = duracao

self.\_\_likes = 0

@property

def likes(self):

return self.\_\_likes

def dar\_like(self):

self.\_\_likes += 1

O que vai acontecer ao definimos @property com nome likes()? Ele deverá retornar o valor que esperamos, por isso, incluímos return self.\_\_likes. Observe que utilizamos \_\_likes privado. Resolvemos o trecho referente a likes, e a seguir alteraremos a parte de \_\_nome.

Como faremos quando tivermos que alterar um código e formos definir um novo \_\_nome? A seguir, adicionaremos o @property e definiremos nome(self):

def dar\_likes(self):

self.\_\_likes += 1

@property

def nome(self):

return self.\_\_nome

No fim, retornaremos self.\_\_nome referente ao valor que queremos mostrar. Ainda está faltando incluirmos o setter, para isto, vamos incluir nome.setter. A partir disso, definiremos como será feita a inserção de valor dentro do nome. Como é feito isso?

A partir de nome.setter conseguiremos definir como colocar valor dentro de nome(), que vai receber novo\_nome, com o qual poderemos tratar self.\_\_nome, que receberá o valor novo\_nome.

@property

def nome(self):

return self.\_\_nome

@nome.setter

def nome(self, novo\_nome):

Com o elemento novo\_nome, poderemos editar o que ainda não tinha recebido tratamento, e receber novo\_nome titularizado, assim cuidaremos do bug que estava na nossa aplicação.

@nome.setter

def nome(self, novo\_nome):

self.\_\_nome = novo\_nome.title()

Nós temos os dois comportamentos, o que falta é incluir getters e settters na série. Colando-os, teremos um código bem similar em ambos trechos - o que pode ser um problema. Vamos verificar se nosso código está funcionando rodando a aplicação. Desta vez conseguiremos imprimir todos os valores corretamente.

Nome: Vinagdores - Guerra Infinita - Ano 2018 - Duração: 160 - Likes: 1

Nome: Atlanda - Ano: 2018 - Temporadas: 2 - Likes: 2

Process finished with exit code 0

O nome Atlanta foi impresso dessa forma, pois foi assim que escrevemos no código. Mas todos os termos foram exibidos com a primeira letra em maiúsculo. Podemos alterar a string para atlanta - de glover:

# ...

atlanta = Serie('atlanta', 2018, 2)

atlanta.nome = 'atlanta - de glover'

atlanta.dar\_like()

atlanta.dar\_like()

print(f'Nome: {atlanta.nome} - Ano: {atlanta.ano} ' -

f'- Temporadas: {atlanta.temporadas}' - Likes: {atlanta.likes}')

Essa alteração também será vista no console quando rodarmos a aplicação.

Nome: Vingadores - Guerra Infinita - Ano 2018 - Duração: 160 - Likes: 1

Nome: Atlanta - De Glover - Ano: 2018 - Temporadas: 2 - Likes: 2

Process finished with exit code 0

Vamos remover a linha atlanta.nome = 'atlanta - de glover' e deixar o código como estava. Nós conseguimos corrigir o bug e deixamos nossa aplicação mais segura, porque agora não conseguimos mais acessar like diretamente. Se tentarmos, cairemos em um erro por falta do atributo likes, e por um motivo semelhante, não poderemos acessar nomes.

Nosso código está funcionando, mas estamos copiando muitos trechos. Como faremos para mudar essa prática e reaproveitar um pouco do código?

Quando criamos atributos no inicializador, estamos definindo quais serão as características do objeto sendo definido. Mas esta não é a única forma de adicionar características ao objeto ou mesmo à classe.

O aspecto dinâmico da linguagem permite que seja possível adicionar atributos até sem precisar do \_\_init\_\_. Veja abaixo:

class Pessoa:

pass

pessoa = Pessoa()

pessoa.nome = 'Jade'

print(pessoa.nome)

Se você tentar executar este código, verá que funciona perfeitamente.

Optamos em usar o inicializador, primeiro para facilitar a criação de novos objetos e segundo para diminuir a confusão em saber o que a classe precisa para criar um objeto aceitável. Sem o init, não dá para saber facilmente quais atributos a classe possui.

Normalmente usamos o init para definir os atributos, mas o que fazer se precisarmos definir um valor padrão para todos os objetos? Ou até criar um atributo que será compartilhado para todas as instâncias?

Para isto, vai ser necessário criar um atributo ligado à classe, ao invés de ligado à instância (self). Por exemplo:

class Pessoa:

tamanho\_cpf = 11

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

self.cpf = cpf

self.nome = nome

def valida\_cpf(self):

return True if len(self.cpf) == \_\_class\_\_.tamanho\_cpf else False

pe = Pessoa('00000000001', 'Ruby')

print(pe.valida\_cpf())

pe = Pessoa('0000000000', 'Cristal')

print(pe.valida\_cpf())

Veja como o valor de tamanho\_cpf é usado por todas as instâncias.

Esse é um atributo de classe. É possível alterar o valor deste atributo, mudando seu estado e não é necessário criar uma instância para acessá-lo.

No trecho de código acima, precisamos usar o \_\_class\_\_ para definir que queremos o atributo de classe. Dentro do nosso método de instância precisamos fazer desta forma.

Se não fizermos deste jeito, podemos ter problemas, como no código abaixo. Faça um teste:

class Pessoa:

tamanho\_cpf = 11

p = Pessoa()

print(p.tamanho\_cpf)

p.tamanho\_cpf = 12

print(p.tamanho\_cpf)

print(Pessoa.tamanho\_cpf)

O que acontece é que, caso não exista o atributo tamanho\_cpf na instância, o Python busca o atributo na classe. Em seguida, adicionamos um atributo tamanho\_cpf na instância e quando dizemos que o valor é 12, o atributo da classe não é alterado, já que são atributos diferentes, um da classe e outro só da instância.

Chegou a hora de você pôr em prática o que foi visto na aula. Para isso, execute os passos listados abaixo.

1) Reveja a parte de criação de classes e objetos. Então crie uma classe primeiro:

class Filme:

pass

vingadores = Filme()

print(vingadores)

Execute o código para ver o que imprime.

2) Agora adicione o inicializador e execute abaixo:

class Filme:

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

self.nome = nome

self.ano = ano

self.duracao = duracao

vingadores = Filme('vingadores - guerra infinita', 2018, 160)

print(vingadores.nome)

É exibido o nome do filme.

3) Agora, adicione uma classe Serie e adicione mais funcionalidades, como o nome titulizado e também os likes. Vai ficar como o código abaixo:

class Filme:

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

self.\_\_nome = nome.title()

self.ano = ano

self.duracao = duracao

self.\_\_likes = 0

@property

def likes(self):

return self.\_\_likes

def dar\_likes(self):

self.\_\_likes += 1

@property

def nome(self):

return self.\_\_nome

@nome.setter

def nome(self, nome):

self.\_\_nome = nome

class Serie:

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

self.\_\_nome = nome.title()

self.ano = ano

self.temporadas = temporadas

self.\_\_likes = 0

@property

def likes(self):

return self.\_\_likes

def dar\_likes(self):

self.\_\_likes += 1

@property

def nome(self):

return self.\_\_nome

@nome.setter

def nome(self, nome):

self.\_\_nome = nome

vingadores = Filme('vingadores - guerra infinita', 2018, 160)

print(vingadores.nome)

atlanta = Serie('atlanta', 2018, 2)

print(f'Nome: {atlanta.nome} - Ano: {atlanta.ano}')

Com as alterações, dá para notar que agora estamos encapsulando o comportamento do nome iniciar com letra maiúscula, e também a forma de adicionar likes, que sempre adiciona apenas um like por vez.

Nesta aula, aprendemos sobre a construção de objetos e classes, utilizando encapsulamento. Vimos:

* Criação da classe
* Definição de métodos assessores
* @property
* name

Até a próxima aula!

Vamos resolver a questão de duplicação; por enquanto, estamos fazendo cópias de trechos de código para replicar os comportamentos que eram comuns entre as duas classes. Da forma como fizemos, o comportamento do nome para que as primeiras letras fiquem em maiúsculo foi incluído nas duas classes. Se tivermos um problema, a correção precisará ser feita nas duas classes.

Se tivéssemos uma nova classe, como documentario, teríamos mais um novo trecho de código duplicado. A seguir, mostraremos uma forma de fazermos o reuso deste código por meio da **herança**. Vamos abstrair. Estamos trabalhando com dois conceitos:

**Filme**

* nome
* ano
* duração
* likes

**Série**

* nome
* ano
* temporadas
* likes

Quando pensamos em herança, pensamos em uma pessoa mais velha que vai deixar bens ou transmitir dados genéticos. Usaremos essa ideia para classes, pois estamos pensando em Filme e Série, cujos comportamentos têm algumas semelhanças. Se tivéssemos uma classe mais genérica com estes mesmos comportamentos, as duas poderiam aproveitar os dados em comum.

Precisamos criar uma classe que representará a ideia genérica e podemos, por exemplo, chamá-la de Programa, em referência a programas de TV. Todo programa de TV terá nome, ano e likes. As duas classes também possuem informações mais específicas que pertencem somente a cada uma delas. A herança na verdade é uma ligação que essas classes terão, e que vão representar que Filme contém informações do programa, porque ele herdará Programa. Da mesma forma, Serie terá informações do mesmo.

Ainda assim, não conseguiremos ver isso. No PyCharm, veremos como resolvermos isso no código Python. Se esta abstração está criando uma classe, teremos que adicionar class Program com informações bastante parecidas com Filme e Serie. Para isto, analisaremos o que as duas têm em comum. Veremos que será necessário termos um construtor com um \_\_init\_\_ parecido por conter as propriedades like e nome, repetidas nas duas classes.

Vamos tentar reduzir o código colocando-as em um único lugar, no caso, na classe Programa.

class Programa:

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

self.\_\_nome = nome.title()

self.ano = ano

self.\_\_like = 0

@property

def likes(self):

return self.\_\_likes

def dar\_likes(self):

self.\_\_likes += 1

@property

def nome(self):

return self.\_\_nome

@nome.setter

def nome(self, novo\_nome):

self.\_\_nome = novo\_nome.title()

A diferença é que Programa só tem nome e ano, e a duração não existe no programa de TV, porque a classe terá informações genéricas. No entanto, isto deixa class Filme vazia. Os atributos likes e nome funcionarão de forma muito similar.

Filme não poderá ficar vazia, porque ela deve ter um construtor diferenciado, sendo necessário apresentar duracao. O que diferencia o filme de uma série ou de um programa de TV em geral é a duração. Por isso, vamos adicionar duracao novamente.

class Filme:

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

self.\_\_nome = nome.title()

self.ano = ano

self.duracao = duracao

self.\_\_likes = 0

Desta vez teremos a comparação, e falta limparmos as informações que não ficarão mais na série. Nós vamos compartilhar as informações entre programa e as classes filhas. Para isto, adicionaremos parênteses depois das mesmas e, dentro delas, especificaremos o nome da classe mãe Programa.

class Filme(Programa):

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

self.\_\_nome = nome.title()

self.ano = ano

self.duracao = duracao

self.\_\_likes = 0

class Serie(Programa):

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

self.\_\_nome = nome.title()

self.ano = ano

self.temporada = temporadas

self.\_\_likes = 0

Ao fazermos isso, todas as informações desde \_\_init\_\_ até nome foram herdadas em Filme, que as terá sem precisarmos escrevê-las. A classe Filme possui \_\_init\_\_, mas também tem os *properties*, e os likes. Porém, só poderemos garantir isso testando.

Rodaremos o nosso código para ver se tudo ocorre perfeitamente, e concluiremos que não funciona como esperávamos, pois é exibida a mensagem sobre AttributeError:

Traceback (most recent call last)

File "D:/luan.silva/stages/python3oo2/modelo.py", line 41, in <module>

vingadores.dar\_like()

File "D:/luan.silva/stages/python3oo2/modelo.py", line 12, in dar\_like

self.\_\_likes += 1

AttributeError: 'Filme' object has no attribute '\_Programa\_likes'

Process finished with exit code 1

De acordo com a mensagem, o objeto Filme não tem o atributo \_Programa\_\_likes. Este nome não foi criado por nós. Podemos clicar na linha em que ocorreu o erro, que nos levará até \_\_likes. Quando fazemos herança, herdamos a maioria dos comportamentos das classes mães, mas alguns ficam de fora.

Em nossos cursos anteriores de Pyhton, falamos que ao fazermos uso dos dois *underscores*, deixamos o atributo privado. Na realidade, o que acontece é que com isso \_\_ será transformado em uma outra variável, e esta ação recebe o nome de ***name mangling***.

Por exemplo, onde temos \_\_nome após a transformação, o resultado seria \_Programa\_\_nome. Com essa mudança de nome, as classes externas terão acesso ao atributo de forma mais fácil, deixando-o protegido. Isto significa que o atributo não está realmente privado, sendo acessível por meio de \_Programa\_\_nome. Mas fazê-lo não seria uma boa prática.

No caso, como colocamos a definição de privado (\_\_nome), este elemento não vai para a classe filha. Com isso, temos um problema, porque teríamos que fazer este *name mangling* manualmente, como no exemplo abaixo:

class Filme(Programa):

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

self.\_Programa\_\_nome = nome.title()

self.ano = ano

self.duracao = duracao

self.\_\_likes = 0

No entanto, essa abordagem acaba recebendo algumas ressalvas e podemos evitar deixar o atributo privado. Dependendo da situação, o caso pode ficar muito complexo; uma melhor opção é usar simplesmente um **\_** (*underscore*), assim oferecemos a ideia de protegido, sem fazermos o *name mangling*. Por conversão, quando criamos uma variável, esperamos que ela não seja alterada depois. Ela estará protegida - mas reforçando, **apenas por convenção**.

Nós podemos usar a variável \_nome da mesma forma que \_\_nome, mas agora deixaremos de receber o problema anterior ao executarmos. Faremos um teste removendo um dos *underscores* de todas as variáveis protegidas anteriormente:

@property

def nome(self):

return self.\_nome

@nome.setter

def nome(self, novo\_nome):

self.\_nome = novo\_nome.title()

Faremos isso em todos os pontos do código onde usamos os dois underscores.

class Filme(Programa):

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

self.\_nome = nome.title()

self.ano = ano

self.duracao = duracao

self.\_likes = 0

Alteramos o código, que não está mais fazendo *name mangling* e, mesmo assim, quando selecionarmos "Run 'modelo'", ele funcionará. Isto porque agora a classe filha tem os dados da classe mãe, e se quisermos acessar os atributos, tudo funcionará corretamente.

Porém, teremos um problema: estamos replicando a criação de diversos atributos no momento de inicializarmos Programa, Serie e Filme. Para resolvermos isso, sempre que fizermos a alteração em uma parte, teremos que fazer o mesmo em diferentes classes em que eles estiverem sendo usados.

Resolveremos essa duplicação mais adiante!