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 = 'atlanta'
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: Atlanta - 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!

AULA 2

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 convenção, 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!

Ainda precisamos resolver a duplicação de que tratamos anteriormente. Nossas classes `Filme` e `Serie` deram origem à classe `Programa`, por meio da generalização. Isto é, pegamos duas classes mais específicas e criamos algo mais genérico, o que poderia ser feito de outra maneira: poderíamos ter o oposto, ou seja, a partir de uma classe mais genérica, fazer algo mais específico, uma especialização.

Por exemplo, se quiséssemos criar uma classe `Documentario`, ela também seria do tipo `Programa`, e herdaria as informações desta classe. A questão é que ainda não resolvemos tudo que queríamos no que concerne à herança.
Em nosso código, temos um construtor para `Programa`, e também para cada classe. Isto porque cada classe filha possui especificidades, como `temporada` e `duracao`. O ideal é não termos que replicar o código, como acontece com `nome`, que está sendo setado pois ainda há um ponto de falha diferente em locais distintos do código. Se houver um bug, teremos que corrigi-lo em todos estes lugares, e isso não é o que queremos.

Existe uma função que pode nos auxiliar muito neste caso, que chama um método, uma funcionalidade, na nossa classe mãe, e se chama superclasse. Quando formos criar nosso objeto do tipo Filme, na verdade já teremos criado um do tipo Programa. O que faremos é modificá-lo para que ele se pareça mais com Filme.
O código referente a _nome, _likes e ano não será mais necessário, uma vez que serão feitos na classe mãe, já que trata-se de informações comuns. Manteremos apenas duracao, e chamaremos a classe mãe, a partir da qual queremos chamar seu inicializador, com __init__. A única diferença aqui é que não estamos declarando-o, então não é necessário passar o self. O código ficará da seguinte forma:

```
class Filme(Programa):
    def __init__(self, nome, ano, duracao):
        super().__init__(nome, ano)
        self.duracao = duracao
```

Com isto, o nosso objeto receberá esta instância criada pela classe mãe. Por fim, definiremos a duração. Em Serie, faremos praticamente o mesmo, neste caso mantendo-se apenas temporadas:

```
class Serie(Programa):
    def __init__(self, nome, ano, temporadas):
        super().__init__(nome, ano)
        self.temporadas = temporadas
```

Agora, temos somente um ponto de falha para alguns códigos da nossa classe. Ao acessarmos o __init__ da classe mãe utilizando a tecla "Ctrl", poderemos ver onde ele se localiza, em outros pontos do código.
Conseguimos fazer um pouco mais de código e resolvemos nosso problema de duplicação. Ainda falta deixar nosso código mais "pythônico", como costumamos dizer, mais bem feito, com design melhor, mais bem estruturado e expressivo.

Veremos algumas regras e boas condutas no decorrer do curso, bem como mais conceitos de orientação a objetos em cima destes modelos que criamos.

Vamos recapitular o conhecimento adquirido em relação às heranças!

Conseguimos reduzir a quantidade de código repetido na nossa aplicação, e deixamos de ter pontos de falhas. No entanto, no PyCharm, o que acontece é que tivemos que passar Programa - a classe mãe - como parâmetro da classe filha, no momento de definirmos esta classe.

> O Python suporta heranças de mais de uma classe.

A herança possui alguns conceitos, e um deles, bastante importante, é que recebemos da classe mãe todas as suas funcionalidades, isto é, métodos, atributos, e demais informações. Deste modo, conseguimos usá-las.

Por exemplo, na classe Filme, não precisamos reimplementar os nossos getters e setters, ou seja, as properties. Isso significa que eles já se encontram prontos em algum lugar, no caso, em Programa. Assim, conseguimos recuperar, acessar e reutilizar informações da classe mãe.
Outro conceito é que temos a opção de estendermos a classe mãe, ou seja, agregar mais àquela classe. O __init__() de Filme é utilizado também na classe Serie, sendo que já temos um __init__() genérico na nossa classe mãe.

Na classe Filme, chamamos o super(), que por sua vez chama qualquer método ou atributo da classe mãe. Podemos testar isto digitando super(). e observando o que o PyCharm nos sugere como autocomplete. Todos os métodos ou propriedades exibidos são provenientes da classe mãe, evidenciado por Programa, ao lado. Podemos acessá-los direta ou indiretamente.
Sendo assim, por que é que não usamos apenas este método, em vez de o chamarmos dentro de um método nosso? O que fizemos foi uma extensão, pois estamos sobrescrevendo o método __init__(). Se não queremos algo que a classe mãe utiliza, ou se queremos usar algo além daquilo que a classe mãe usa, para que seja adicionado ao comportamento, chamamos o método que a classe mãe disponibiliza, como em super().__init__(nome, ano), e adicionaremos algo específico, neste caso, self.duracao = duracao.
Neste exemplo, sobrescrevemos o método __init__() para que possamos utilizar o __init__() da classe mãe, e além disto ter algo mais específico.
Há ainda o conceito de não precisarmos mais ficar focados em simplesmente reaproveitar o código da classe mãe, uma vez que podemos pegar uma classe mais específica, como Filme ou Serie, e dizer que precisamos de algo exclusivo daquela classe, algo que não faça sentido para a classe mãe. Da mesma forma, ela não necessariamente precisa ter algo específico para que isto seja compartilhado com as classes filhas.
Se quiséssemos criar algo específico para Filme, como um cadastro diferenciado, por exemplo, uma função para o retorno deste cadastro seria:

```
def retorna_cadastro_diferenciado(self):
    pass
```

> O pass é usado quando não queremos definir uma implementação para um método ou uma classe, fazendo com que o código rode sem erros.

Isso é possibilitado pela flexibilidade da herança para fazermos algo exclusivo.
É interessante notarmos como funciona cada pedaço da herança para entendermos melhor o nosso código como um todo. Mas para onde vamos com tudo isso? Já entendemos como funciona a herança, e criamos um trecho que exibe informações de alguns filmes e séries:

```
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}')
```

Tornaremos este código mais enxuto, para passarmos ao próximo passo, que consiste em tentarmos criar uma playlist contendo filmes e séries. Deletaremos Nome:, Ano: e Duração:, e manteremos apenas o traço - entre eles. Removeremos Likes: também. Como o código ainda fica grande, deletaremos também as informações acerca do ano, por enquanto. Faremos o mesmo com Serie:

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

```
print(f'{vingadores.nome} - {vingadores.duracao}: {vingadores.likes}')
atlanta = Serie('atlanta', 2018, 2)
atlanta.dar_like()
atlanta.dar_like()
print(f'{atlanta.nome} - {atlanta.temporadas}: {atlanta.likes}')
```

Precisaremos realizar uma playlist que irá agregar tanto séries quanto filmes. Isto é, armazenaremos diversos programas de TV. Para isso, é necessário uma estrutura de dados, um objeto... Pensaremos direito sobre a melhor opção. Por ora, vimos como as heranças funcionam. Adiante teremos uma noção de como começar a montar uma listagem, ou um grupo de objetos de tipos distintos, ou não.

Da mesma forma que temos alguns atributos diretamente da classe, e que são acessíveis sem necessidade de criar uma instância, conseguimos também criar métodos ligados à classe.

Há duas formas de criar estes métodos:

Métodos de classe

São métodos declarados com `@classmethod`. Quando criamos um método de classe, temos acesso aos atributos da classe. Da mesma forma com os atributos de classe, podemos acessar estes métodos de dentro dos métodos de instância, a partir de `__class__`, se desejarmos:

```
class Funcionario:
    prefixo = 'Instrutor'

    @classmethod
    def info(cls):
        return f'Esse é um {cls.prefixo}'
```

Perceba que, ao invés de `self`, passamos `cls` para o método, já que neste caso sempre recebemos uma instância da classe como primeiro argumento. O nome `cls` é uma convenção, assim como `self`.

Métodos estáticos

A outra forma de criar métodos ligados à classe é com a declaração `@staticmethod`. Veja abaixo:

```
class FolhaDePagamento:
    @staticmethod
    def log():
        return f'Isso é um log qualquer'
```

Note que, no caso acima, não precisamos passar nenhum primeiro argumento fixo para o método estático. Nesse caso, não é possível acessar as informações da classe, porém o método estático é acessível a partir da classe e também da instância.

Cuidados a tomar...
Sempre que você usar métodos estáticos em classes que contém herança, observe se não está tentando acessar alguma informação da classe a partir do método estático, pois isso pode dar algumas dores de cabeça pra entender o motivo dos problemas.

Alguns pythonistas não aconselham o uso do `@staticmethod`, já que poderia ser substituído por uma simples função no corpo do módulo. Outros mais puristas entendem que os métodos estáticos fazem sentido, sim, e que devem ser vistos como responsabilidade das classes.

Nesta aula, conseguimos fazer com que o código duplicado fosse eliminado. Aprendemos os seguintes conceitos:

•Herança
•Generalização/especialização
•Método `super()`

Até breve!

AULA 3

Precisamos criar uma playlist para, de alguma forma, armazenarmos o nosso programa de TV. Ela terá um nome específico, temático ou não, uma lista de forma ordenada e seu tamanho (`tamanho()`).

Para tornar isto realidade, uma das opções mais simples é tentar colocar todos os programas - filmes, séries - em uma estrutura específica de dados, que possa contê-los.

Uma lista no Python é uma estrutura ordenada - faz sentido para uma playlist, com ordem de execução - que permite a verificação de seu tamanho (a duração), então podemos tentar implementá-la. Começaremos colocando tudo em uma lista, e depois pensaremos no objeto, com a classe `Playlist`, e tudo o mais.
Da maneira como nosso código se encontra no momento, na classe já há uma forma de mostrarmos os objetos existentes, no caso, `vingadores` e `atlanta`; um deles é um filme, o outro, uma série. E ao lidarmos com heranças, podemos dizer que as classes filhas são do mesmo tipo que o da classe mãe.
Para criarmos uma lista no Python, definiremos uma variável qualquer, como `filmes_e_series`, que guardará a lista, representada por colchetes ([]):

```
filmes_e_series = [vingadores, atlanta]
```

No entanto, trata-se de objetos de tipos diferentes... Isso é possível? Claro! O Python é uma linguagem dinâmica, que não se apega aos tipos que se encontram dentro de uma lista ou estrutura específica de dados.
Levando em conta tudo isso, podemos dizer que `Filme` e `Serie` possuem um relacionamento com a classe mãe, por serem classes filhas. Este relacionamento é chamado de é um - uma série é um programa, da mesma forma que um filme também é um programa. O filme, no entanto, não é uma série, já que este relacionamento só se dá entre uma classe genérica e uma específica, no caso de heranças.
Como vantagens, temos que podemos simplesmente imprimir cada item da lista que implementamos anteriormente por meio do for in:

```
filmes_e_series = [vingadores, atlanta]
```

```
for programa in filmes_e_series:
    print(f'{programa.nome} - {programa.temporadas}: {programa.likes}')
```

Vamos executar o código para ver se funciona? Inicialmente rodaremos a classe clicando com o lado direito do mouse e selecionando "Run 'modelo'". Mas isto nos leva a um problema que indica que o objeto Filme não possui um atributo temporadas, algo de que já sabíamos, pois programa.temporadas não existe.
Atualmente, imprimimos dados sobre filmes e séries, cada qual com suas especificidades. É necessário encontrarmos uma forma mais genérica de imprimir, ou então de verificarmos o tipo antes da impressão. A forma mais rápida é apagar o trecho que está dando erro; vamos fazer assim, e veremos que desta vez o código roda sem nenhum problema.

Após as duas vezes em que rodamos o código, obtemos:

```
Vingadores - Guerra Infinita - 160: 1
Atlanta - 2: 2

Vingadores - Guerra Infinita - 1

Atlanta - 2
```

Aqui, vemos a vantagem de trabalharmos com heranças, chamada de polimorfismo. Desta forma, conseguimos percorrer uma lista qualquer e, sendo elas do mesmo supertipo, no caso, de Programa, e com uma estrutura similar, com nome e likes, conseguimos usar o for independentemente do tipo que existe ali.

> Não conseguiríamos fazer o mesmo em linguagens estaticamente tipadas, já que tipo em uso importa.

A questão é que ainda falta algo, pois recebemos um erro. Como resolveremos aquele problema?

Podemos verificar em tempo de execução que tipo de atributo o filme ou série em questão possui. Cada tipo possui detalhes próprios, sendo assim criaremos uma variável denominada detalhes. Nela, definiremos o que ela contém, e verificaremos o atributo contido no objeto por meio da função hasattr(), ou has attribute.
Para implementarmos o if em uma única linha, começaremos com o valor que queremos exibir, no caso, programa.duracao, apenas se a duração existir. E então usaremos o hasattr() passando como parâmetros o objeto (o) e o nome que queremos saber se o objeto possui (name).
O if, então, retornará duracao, caso houver. Caso contrário (else), será devolvido programa.temporadas.

```
filmes_e_series = [vingadores, atlanta]


for programa in filmes_e_series:
    detalhes = programa.duracao if hasattr(programa, 'duracao') else programa.temporadas
    print(f'{programa.nome} - {detalhes} D - {programa.likes}')
```

Vamos executar o programa novamente para verificar se o detalhes da linha de print() será o esperado. Ao rodarmos a aplicação, será exibido:

```
Vingadores - Guerra Infinita - 160: 1
```

```
Atlanta - 2: 2

Vingadores - Guerra Infinita - 160 D - 1

Atlanta - 2 D - 2
```

Com isso, resolvemos o problema verificando-se em tempo de execução se há um atributo naquele objeto que estamos exibindo. Mas será que esta é a melhor forma de fazê-lo? E se tivéssemos mais atributos diferentes na classe Filme, ou mesmo em Series? Teríamos que fazer a verificação de um em um, separadamente.
Nossa aplicação está funcionando, mas podemos melhorá-lo!

Mostramos os dados em ordem, bem como em uma lista, imprimindo-os um por um na iteração. Temos um código com um if ternário que, apesar de ocupar apenas uma linha, ainda é um problema, porque se tivéssemos mais atributos diferentes o if seria obviamente maior. Poderíamos tentar um if não ternário, e sim padrão, com detalhes recebendo valores distintos, como em:

```
for programa in filmes_e_series:
    if hasattr(programa, 'duracao'):
        detalhes = programa.duracao
    else:
        detalhes = programa.temporadas
```

Entretanto, isto ainda faz com que o código fique maior e o problema permaneça. Queremos imprimir informações acerca dos objetos, mas estamos nos aprofundando demais nele sem necessidade. Queremos saber se o objeto possui duracao ou temporadas e estamos lidando com Programa, um tipo de objeto.
Não precisamos saber se programas têm temporadas ou durações, eles simplesmente precisam ser exibidos. De certa forma, o filme e a série deveriam ser responsáveis pelas suas impressões. Quando vamos modelar classes e objetos, devemos levar em consideração suas responsabilidades.
Portanto, o ideal seria que cada classe tivesse sua responsabilidade com clareza e, quando isto ocorre, é possível chamá-la de classe coesa, ou seja, quando ela sabe qual é sua responsabilidade e não faz mais do que aquilo a que se propõe a fazer.
Neste caso, tanto Filme quanto Series devem saber se imprimir de forma usual. Ao imprimirmos uma string, ela sabe como fazê-lo. Mas como é que isso funciona?
Como a partir de agora só precisaremos criar os objetos, e não imprimi-los, deletaremos os print() do trecho do código que lida com as variáveis vingadores e atlanta, deixando-o assim:

```
vingadores = Filme('vingadores - guerra infinita', 2018, 160)
vingadores.dar_like()
atlanta = Serie('atlanta', 2018, 2)
atlanta.dar_like()
atlanta.dar_like()
```

No Python, existem várias maneiras de imprimir o valor de um objeto. Para fins de exemplo, podemos fazer uma definição em Programa:

```
def imprime(self):
    print(f'{self._nome} - {self.ano} - {self._likes} Likes')
```

Com isso, mudaremos o fim do código para não precisarmos mais do `if` que estávamos usando anteriormente, já que usaremos o método `imprime()`:

```
for programa in filmes_e_series:
    programa.imprime()
```

Executaremos a aplicação, e será impresso:

```
Vingadores - Guerra Infinita - 2018 - 1 Likes
Atlanta - 2018 - 2 Likes

Process finished with exit code 0
```

Resolvemos uma parte do problema! Agora, precisaremos não só imprimir o dado de um programa genérico, e sim para cada tipo diferente existente. No entanto, não poderemos herdar este comportamento de `imprime()` do `Programa`. Teremos que definir um método para cada uma das classes filhas.
Neste caso, faremos uma sobrescrita de `imprime()` da classe mãe, porque não queremos utilizá-lo.
No caso de `Filme`, incluiremos portanto a `duracao`:

```
class Filme(Programa):
    def __init__(self, nome, ano, duracao):
        super().__init__(nome, ano)
        self.duracao = duracao


    def imprime(self):
        print(f'{self._nome} - {self.ano} - {self.duracao} min - {self._likes} Likes')
```

E para `Serie`, aplicaremos o mesmo, apenas substituindo as informações sobre a duração para temporadas.

```
class Serie(Programa):
    def __init__(self, nome, ano, temporadas):
        super().__init__(nome, ano)
        self.temporadas = temporadas


    def imprime(self):
        print(f'{self._nome} - {self.ano} - {self.temporadas} temporadas - {self._likes} Likes')
```

O trecho abaixo poderá ser deletado do código, pois ele não nos serve para nada:

```
def retorna_cadastro_diferenciado(self):
    pass
```

Desta vez, temos um `imprime()` diferente para cada uma das classes filhas. Esta é a ideia do polimorfismo: por meio de um `for`, conseguimos mostrar que, a cada iteração a ser executada,

estaremos com Filme ou Serie, e quando o imprime() for chamado, não importa qual o tipo deste objeto, o método será chamado de acordo com quem o tiver.
Vamos fazer um teste executando o código. Veremos:

```
Vingadores - Guerra Infinita - 2018 - 160 min - 1 Likes
Atlanta - 2018 - 2 temporadas - 2 Likes

Process finished with exit code 0
```

De forma mais clara, conseguimos ter uma função que representa textualmente o nosso objeto. Usando-se o conceito de polimorfismo, reduzimos a quantidade de ifs do nosso código, ou seja, a complexidade do código. Assim, não importa a forma, o tipo que estamos passando ao for, o método imprime() correspondente será executado corretamente.
Esta, no entanto, não é a melhor forma de executarmos este método. Seria melhor utilizarmos algo proveniente da linguagem Python, cujas bibliotecas são escritas de outra maneira. No Python, quando temos um objeto qualquer, podemos representá-lo de forma textual. E uma das principais maneiras de se fazê-lo é por strings, que é o que o método print() faz.
Pressionando-se "Ctrl" e clicando em imprime(), navegaremos à sua definição, e veremos que print() é uma função built-in, ou seja, nativa do Python. Esta função nativa está pronta, e tenta exibir o objeto que recebe com uma string, imprimindo-o.
Na prática, se tentarmos rodar a impressão de um número qualquer, como print(124124), veremos no console a impressão deste número, que não é uma string. Foi feita a execução de string neste número, e o mecanismo por trás disto é que a função chama algo como print(str(124124)), isto é, o str() do objeto 124124.
Se rodarmos o print() com o str(), então, funcionaria da mesma maneira, apenas estaríamos forçando-a. O que esta string, este str(), quer dizer?
Quer dizer que ele está tentando fazer uma amostra textual para o usuário final. Assim, quando o executamos, ele buscará dentro do objeto 124124 um método especial, de que não falaremos por enquanto, mas veremos adiante como ele funciona, e tentaremos aplicá-lo nas classes que já temos. Com isso teremos um código mais "pythônico", sem muitas amarras ao imprime() que criamos.

Falamos anteriormente que existe uma maneira mais pythônica de representarmos strings ou textos - ou objetos como textos -, e sabemos que a função str() com print() fazem com que o objeto seja mostrado como uma string.

Alguns métodos no Python são especiais, ou dunder methods, como costumam chamar. Dunder vem de double underscore, isto é, "dois underlines". Um exemplo de método especial é o nosso __init__() que, ao ser definido, o Python sabe, por convenção, que ele é o inicializador de uma classe na criação de um objeto.
Neste caso, com __init__(), é necessário termos em uma classe, mesmo que não seja obrigatório, um método especial capaz de representar um objeto textualmente. Um destes, que não é o imprime(), é chamado de __str__(), ou dunder str, ou ainda "str especial".
Quando definimos esta função, não é possível simplesmente fazermos um print() nela, pois é esperado que se retorne um valor como string, que represente o objeto desejado. O trecho de código corresponde seria, portanto:

```
def __str__(self):
    return f'{self._nome} - {self.ano} - {self._likes} Likes'
```

Assim, f'{self._nome} - {self.ano} - {self._likes} Likes' representa o objeto programa na classe Programa em forma de texto. Vamos substituir os imprime(self) por __str__(self) e, como deixamos de ter a função imprime(), no fim do mesmo arquivo deletaremos print(str(124124)) e descomentaremos o trecho que havíamos comentado anteriormente, deixando-o desta forma:

```
for programa in filmes_e_series:
    print(programa)
```

Se executarmos o código da maneira em que ele está, teremos o seguinte erro:

```
TypeError: str returned non-string (type NoneType)
```

Isto significa que o __str__() retorna algo que não é uma string, e sim um NoneType. Vamos voltar ao código para verificar o que houve na impressão. No caso do Filme, esquecemos de remover o print() da função. Ou seja, em todos os lugares em que usamos print(), trocaremos por return.
Percebam que no Python não há muita validação de tipos, mas temos algumas com verificação e um aviso no caso de erro. Podemos reexecutar a aplicação, e desta vez a impressão será exatamente como gostaríamos:

```
Vingadores - Guerra Infinita - 2018 - 160 min - 1 Likes
Atlanta - 2018 - 2 temporadas - 2 Likes

Process finished with exit code 0
```

Outra forma de representação do objeto é o repr(), que não é de tão fácil leitura para o usuário, já que é para o próprio computador, a linguagem Python.
O que normalmente acontece quando imprimimos um valor, ou melhor, o retornamos, no nosso console do Python, é que temos não a chamada da função str(), e sim a repr(), a qual mostra o objeto de forma diferente, e falaremos sobre ela mais adiante.
Resolvemos um grande problema, e estamos com o código mais pythônico. Quando visto por alguém que é mais experiente na linguagem, a pessoa saberá que no código, com o __str__(), estamos definindo uma representação textual para o nosso objeto.
Voltando à playlist que desejamos implementar, não iremos simplesmente pegar Programa, que é um tipo genérico, e colocá-lo em uma lista. Lembra que nossa playlist possui nome, programas e um tamanho? No caso, ela precisará ser representada de forma a ser uma classe, um objeto. E para isso é necessário criarmos objetos deste tipo, definindo estes métodos que temos.
Como faremos isso?

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

---

1) Comece a armazenar as informações de uma playlist. Para começar, coloque os filmes e séries numa lista Python:

```
class Programa:
    def __init__(self, nome, ano):
        self._nome = nome.title()
        self.ano = ano
        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 Filme(Programa):
    def __init__(self, nome, ano, duracao):
        super().__init__(nome, ano)
        self.duracao = duracao

class Serie(Programa):
    def __init__(self, nome, ano, temporadas):
        super().__init__(nome, ano)
        self.temporadas = temporadas

vingadores = Filme('vingadores - guerra infinita', 2018, 160)
atlanta = Serie('atlanta', 2018, 2)
vingadores.dar_likes()
vingadores.dar_likes()
vingadores.dar_likes()

atlanta.dar_likes()
atlanta.dar_likes()
```

```
listinha = [atlanta, vingadores]


for programa in listinha:
    print(f'Nome: {programa.nome} - Likes: {programa.likes}')
```

As informações de filmes e séries são mostradas em um único `for in`, já que são tipos derivados de `Programa`, você consegue imprimir os seu valores sem precisar diferenciar cada tipo.
2) Há um problema pra resolver pois `Filme` e `Serie` são diferentes. O primeiro tem o atributo "duração" e o último tem o atributo "temporadas". Antes de imprimir, implemente uma forma de verificar qual dos dois atributos deve ser impresso a fim de mostrar informações mais completas sobre o que está sendo impresso.
Na hora de imprimir, faça o seguinte:

```
for programa in listinha:
    if hasattr(programa, 'duracao'):
        detalhe = f'{programa.duracao} min'
    else:
        detalhe = f'{programa.temporadas} temporadas'


    print(f'Nome: {programa.nome} - {detalhe} - Likes: {programa.likes}')
```

Com este código, você verifica se tem o atributo `duracao` (atributo que um filme tem), caso contrário, será uma série.
3) O problema foi resolvido, mas tem uma outra forma de fazer o mesmo sem precisar desse `if` intrujão. Crie um método que imprime o programa como string.
Na classe `Programa`:

```
def imprime(self):
    print(f'Nome: {self.nome}  Likes: {self.likes}')
```

Na classe `Filme`:

```
def imprime(self):
    print(f'Nome: {self.nome} - {self.duracao} min - Likes: {self.likes}')
```

E na classe `Serie`:

```
def imprime(self):
    print(f'Nome: {self.nome} - {self.temporadas} temporadas - Likes: {self.likes}')
```

4) Agora, remova o `if` da parte da impressão, deixando assim:

```
for programa in listinha:
    programa.imprime()
```

Ficou bem mais enxuto né? Desta forma, você está usando o polimorfismo para executar o método `imprime` de qualquer programa, seja filme ou série, tendo um resultado diferente para cada um.

5) Melhorou o código, mas ainda não é o jeito mais pythonico de fazer isto. Para apresentar objetos como string de forma pythonica, é preciso implementar o método __str__. Substitua o método imprime pela implementação abaixo nas 3 classes.
Na classe Programa:

```
def __str__(self):
    return f'Nome: {self.nome} Likes: {self.likes}'
```

Na classe Filme:

```
def __str__(self):
    return f'Nome: {self.nome} - {self.duracao} min - Likes: {self.likes}'
```

Na classe Serie:

```
def __str__(self):
    return f'Nome: {self.nome} - {self.temporadas} temporadas - Likes: {self.likes}'
```

6) Note que agora é preciso retornar uma string ao invés de imprimir diretamente. Feito isso, modifique também a impressão, para deixar somente o objeto sendo impresso:

```
for programa in listinha:
    print(programa)
```

Nesta aula, aprendemos sobre outros benefícios de usar herança:

•Polimorfismo
•Relacionamento é um
•Representação textual de um objeto

Até a próxima!

AULA 4

A partir de agora criaremos nossa playlist! Em modelo.py, temos uma listagem contida em uma estrutura de dados List, identificada pelo uso de colchetes, e que representa filmes_e_series. Para criarmos uma playlist com nome, é preciso algo mais abstrato, de algo como o tipo Playlist.

Assim, criaremos uma classe com este nome, que terá, além de nome e programa, o tamanho(), função que retornará a quantidade de itens que existem em uma determinada lista de programas, por meio do len().

```
class Playlist:
    def __init__(self, nome, programas):
        self.nome = nome
        self.programas = programas

    def tamanho(self):
```

```
        return len(self.programas)
```

E para que a playlist seja representada e tenha os itens percorridos, vamos aproveitar filmes_e_series, que criamos anteriormente. Antes disso, alteraremos e reordenaremos o trecho com as variáveis vingadores e atlanta, usando o atalho "Ctrl + Shift" e a tecla com a seta para cima ou para baixo, no PyCharm.
Acrescentaremos mais um filme e uma série, e com "Ctrl + D" duplicaremos vingadores.dar_like() para distribuir os likes entre eles. O código ficará assim:

```
vingadores = Filme('vingadores - guerra infinita', 2018, 160)
atlanta = Serie('atlanta', 2018, 2)
tmep = Filme('Todo mundo em pânico', 1999, 100)
demolidor = Serie('Demolidor', 2016, 2)

vingadores.dar_like()
tmep.dar_like()
tmep.dar_like()
tmep.dar_like()
tmep.dar_like()
demolidor.dar_like()
demolidor.dar_like()
atlanta.dar_like()
atlanta.dar_like()
atlanta.dar_like()

filmes_e_series = [vingadores, atlanta, demolidor, tmep]
playlist_fim_de_semana = Playlist('fim de semana', filmes_e_series)

for programa in filmes_e_series:
    print(programa)
```

Lembrem-se que, no Python, a ordenação dos itens em uma lista importa!
E então, criaremos uma playlist chamada de playlist_fim_de_semana, que receberá um objeto do tipo Playlist, que por sua vez tem como parâmetros o nome e a referência de filmes_e_series. Já temos um for rodando em cima de filmes_e_series, e poderíamos mantê-lo assim, no entanto queremos percorrer a listagem que está na playlist.
Sendo assim, poderíamos usar algo como for programa in playlist_fim_de_semana, mas se tentarmos percorrer a playlist, receberemos o seguinte erro:

TypeError: 'Playlist' object is not iterable

Isto é, Playlist não é iterável, o que é incompatível ao uso do loop (o for in). Da forma como a playlist está modelada no momento, temos acesso a programas e, quando formos percorrer, não o

faremos em playlist_fim_de_semana, e sim em playlist_fim_de_semana.programas, pois é necessário conhecermos a estrutura da playlist.
Estamos passando para dentro da playlist a variável programas, que está entrando no inicializador, mas este nome não importa muito, podendo ser substituído simplesmente por playlist ou filmes_e_series. A questão é que quando temos programas, precisamos conhecer a parte interna da playlist e saber que ela possui dentro de si uma variável com nome programas. Caso não soubéssemos este nome, ou se programas fosse privado, como faríamos? Seria estranho, não? Conseguimos fazer a playlist ser percorrida, o que pode ser confirmado ao executarmos a aplicação, mas o código não está da melhor maneira possível, pois temos que chamar programas, que está dentro da playlist.
Em Orientação a Objetos, não é preciso conhecer toda a estrutura interna de um objeto - chamamos este conceito de deixarmos para fora apenas o que queremos que interaja com o mundo externo de encapsulamento. Ou seja, em um objeto, precisamos expor apenas aquilo que queremos que os seus outros clientes acessem, e que faça sentido na interface do objeto ou classe.
Neste caso estamos ferindo um pouco este princípio, pois estamos acessando programas sem nem sabermos direito sobre a sua estrutura interna. Como saberíamos que trata-se de uma lista? E se tivéssemos que utilizar um objeto Playlist que foi criado por outra pessoa... Seria um problema, não concorda? Já que, idealmente, acessaríamos playlist_fim_de_semana da maneira como fizemos, sem erros.
E se por exemplo usássemos herança para resolvermos este problema? Como faríamos para que o objeto Playlist seja reconhecido também como um tipo de objeto iterável, ou algum outro tipo compatível com esta notação do for, que possa ser incluído após o in?
Poderíamos usar um list, um set, ou qualquer outro objeto iterável, a partir do qual poderíamos herdar os comportamentos, fazendo com que a playlist se passe por algum deles... Veremos tudo isso a seguir!

Da maneira como deixamos nosso código, entendemos que a playlist não está tão boa quanto gostaríamos, pois precisamos conhecer minimamente a sua estrutura interna, como ela funciona, se há uma lista de programas lá dentro, qual o nome da variável, e por aí vai.

Talvez esta não seja a melhor forma de apresentar essa playlist. Até pensamos em aplicarmos uma herança para que a playlist acesse as informações de um list, utilizado internamente. Como fazemos uma herança?
Primeiramente, incluímos os parênteses na classe Playlist para indicar de qual classe queremos herdar, no caso, list. Com isso, "magicamente" temos tudo o que o list possui, sendo assim, não é mais necessário definirmos tamanho(), uma vez que há uma forma específica de fazermos isto. Com len() que chamamos em return len(self.programas), conseguimos o tamanho de uma lista. Faremos o mesmo com Playlist. Outro ponto é que, da forma como estávamos utilizando o inicializador, sobrescrevemos o inicializador do list, de quem estamos herdando.
O que é melhor, usarmos o nosso inicializador, ou do list, neste caso? O de list recebe uma lista preparada para poder ser inicializada por aquele objeto. O nosso, no entanto, possui um nome, porém se o usarmos deixaremos de ter os benefícios provenientes do list.
Há uma maneira de utilizarmos os dois: podemos chamar o método inicializador da nossa superclasse, mantendo-se o nome, e definimos a lista de programas que está naquela list. Então, com super() chamaremos o construtor (o inicializador de list), passando simplesmente uma

lista qualquer para dentro dele. Neste caso, como recebemos uma lista de programas, é isto que será usado.
Com esta chamada, criamos um objeto que terá nome e, dentro de sua estrutura, será setada uma lista de programas:

```
class Playlist(list):

    def __init__(self, nome, programas):

        self.nome = nome

        super().__init__(programas)
```

Continuamos não sabendo como list funciona internamente, no entanto sabemos que ele nos provê uma série de facilidades, por estar pronto para ser iterado, ser uma sequência Python, e tudo o mais.
Vamos testar para verificar o que houve com a nossa iteração da playlist. Mais abaixo no código, removeremos .programas após playlist_fim_de_semana no loop do for, pois não precisamos mais conhecer sua estrutura interna.
Desta vez, é importante saber que a playlist também é do tipo list, por herdar seus comportamentos e propriedades. Iremos executar o código para verificar se conseguimos percorrer, por meio do for, a nossa playlist_fim_de_semana, um objeto do tipo Playlist que está sendo criada dentro da variável playlist_fim_de_semana.
Ao executarmos, veremos que tudo funciona conforme esperado, e recebemos:

```
Vingadores - Guerra Infinita - 2018 - 160 min - 1 Likes
Atlanta - 2018 - 2 temporadas - 3 Likes

Demolidor - 2016 - 2 temporadas - 2 Likes

Todo Mundo Em Pânico - 1999 - 100 min - 4 Likes
```

Conseguimos fazer uma execução que passa pelo código e pelo for in com playlist_fim_de_semana, o que indica que nossa playlist poderá ser utilizada como um list em qualquer lugar, e esta é uma grande vantagem, pois não precisamos escrever nenhuma linha de código a mais.
Lembrando que com o len() acessamos o tamanho da lista, de que forma imprimiremos o tamanho de playlist_fim_de_semana? Vamos tentar incluir um print() passando len() como parâmetro.
Reforçando que não é estritamente necessário sabermos como ele faz esta impressão, pois quem é responsável por isto é o list, e isso já está implementado em algum lugar.

```
filmes_e_series = [vingadores, atlanta, demolidor, tmep]

playlist_fim_de_semana = Playlist('fim de semana', filmes_e_series)


print(f'Tamanho do playlist: {len(playlist_fim_de_semana)}')
```

Vamos executar a aplicação novamente. Como retorno, teremos, antes das informações do conteúdo da playlist:

```
Tamanho do playlist: 4
```

Isto é, fizemos uma melhoria incluindo a classe list como nossa classe mãe em relação à Playlist.
Percebemos que a herança nos permite economizar código, por reutilizarmos muito do que já foi feito.

Porém, na aula sobre polimorfismo, vimos que conseguimos usar uma Playlist como se ela fosse um list, sem precisarmos saber do tipo. Quando fazemos o for programa in playlist_fim_de_semana, o in espera um iterável qualquer, já que ele itera sobre alguma listagem ou iterável, o objeto específico.
Como herança, também ganhamos a forma de chamar o len(), o que não poderia ser feito anteriormente, uma vez que playlist_fim_de_semana não era iterável. Na verdade, o len() é um sizeable - a ideia é que, a partir de então, temos um objeto que consegue informar seu próprio tamanho, por meio desta função.
Além disso, se quiséssemos verificar a existência de algum objeto na lista, como por exemplo, se em playlist_fim_de_semana existe o filme "Demolidor", é possível pegarmos a variável demolidor e perguntar à lista se ela o contém. Isso funcionaria da seguinte forma:

```
print(f'Tá ou não tá? {demolidor in playlist_fim_de_semana}')
```

Assim, serão exibidas a string e a verificação booleana de acordo com a existência ou não de demolidor dentro de playlist_fim_de_semana. Ou seja, perguntamos se algum objeto está contido (in) em um objeto do tipo playlist_fim_de_semana.
Ao rodarmos este código, o retorno no console será, juntamente com as demais informações:

```
Tá ou não tá? True
```

Caso decidíssemos remover "Demolidor" da nossa lista que a variável filmes_e_series recebe, e executar a aplicação novamente, teríamos:

```
Tá ou não tá? False
```

Isto nos demonstra que a herança garante muitos benefícios.

Só que em vários momentos deste vídeo falamos sobre não sabermos muito bem sobre a interface do list, que é de onde vem a herança. Será que isso é realmente vantajoso? Quão grande será esta interface, quantos métodos será que ele possui? Todos eles farão sentido para aquilo que estamos implementando, ou seja, a playlist?

Tínhamos feito a herança de list na nossa classe Playlist, e com isso podemos dizer Playlist é um list, e que absorvemos todas as suas funcionalidades para a Playlist.

No entanto, list é uma classe conhecida como built-in, embutida no Python e pronta para uso. E tivemos o benefício de reaproveitarmos vários códigos desta classe, mesmo não conhecendo todas as suas funcionalidades ou extensões. Precisamos tomar cuidado ao fazermos este tipo de herança!
No código, em relação a Playlist, utilizamos o __init__() para fazermos a chamada usando nome e programas. Sendo assim, trata-se de um __init__ diferente de um inicializador de um list.
Quando alguém for criar um objeto de Playlist, terá que utilizar este inicializador, o que será um tanto confuso, pois por ser um list, deveria receber apenas a listagem de itens, ou de repente somente um item, para começar uma lista, a qual possui um protocolo, uma interface, distintos.
Um ponto positivo é que estamos criando uma playlist mas não nos limitaremos a um list só - a playlist tem mais significado para o nosso sistema, talvez possamos fazer uma busca por nome, uma filtragem para encontrar elementos que façam parte de uma categoria, como "série", por exemplo.
Estamos começando a perceber as particularidades de list e Playlist, e já notamos que é um tanto complexo utilizarmos um tipo built-in, uma classe que já está pronta no sistema, porque não sabemos as exceções que ela possui.

Por exemplo, list pode conter um método que permita o acesso a algum de seus itens, e ele pode ser protegido, por algum motivo. Muito em Python é implementado em CPython, então, pode ser que haja alguma função de list que seja protegida e não permita a sobrescrita. Mas para descobrirmos isto, teríamos que ler toda a documentação do list, e talvez isso nem nos ajude em nada no fim das contas.

No crescimento da nossa classe, teremos que nos preocupar com esses detalhes o tempo todo: será que esta função que vamos criar já existe em list? Será que estamos sobrescrevendo uma função preexistente, resultando em erro? Ou impedindo meu objeto de ter um len(), ou até de ser iterável?

Anteriormente tínhamos um design, uma visão muito mais simples e clara do que queríamos por meio de Playlist - que fosse menos complexo, contendo nome, programas e um tamanho. Vimos que list seria interno, possível de ser percorrido por algo, isto é, seria iterável.

Por algum tempo focamos nisso, mas acabamos criando complexidades ao optarmos pela herança. Também ficamos nos perguntando se valia a pena termos acesso a tudo que vem de list. Nesta situação, o ideal seria conseguirmos fazer algo parcialmente, aproveitando o melhor dos mundos, que seria Playlist não ser do tipo list e, de alguma maneira, ter vantagens que ele tem.

Para evitarmos as complexidades adquiridas da herança de list, olharemos no nosso código para entendermos o que pode ser feito. Se estamos herdando algo que é mais complexo do que esperávamos, com uma interface gigantesca, e que talvez não esteja preparado para fazer com que a nossa Playlist se adapte a ela, vamos desfazer a herança.

> Precisamos ter controle do que estamos fazendo nas nossas classes, o que implica em uma boa prática de programação, ainda mais em se tratando de Orientação a Objetos.

Na classe Playlist, então, faremos algumas modificações, deletaremos o super(), voltando à necessidade de termos uma lista de programas, os quais não queremos que sejam acessíveis. Para isso, podemos incluir o _ (underscore) antes de programas:

```
class Playlist:
    def __init__(self, nome, programas):
        self.nome = nome
        self._programas = programas
```

O underline acaba servindo de aviso também, pois é algo de que a pessoa externa não poderá depender. E também indica que podemos alterar programas a qualquer momento. Quando estávamos herdando de list, imagine que em vez de um list ordenado de programas, quiséssemos passar como parâmetro um dicionário, ou um set, qualquer outra estrutura do Python - como faríamos?

Com a herança, estávamos muito presos ao list e, no caso de mudanças, teríamos problemas ainda maiores, pois tudo que usasse a classe Playlist seria alterado também. Já que retiramos a herança e resolvemos incluir um programas que não tem livre acesso, teremos que criar alguma censura, como uma property, que chamará a definição de listagem.

Esta função retornará os programas, que ficarão protegidos, e representará a nossa listagem. Com ela, conseguimos exibir dados, e tudo o mais. Falta demonstrarmos o tamanho da listagem criando outra property, o que garante que agimos de acordo com as boas práticas, para representar o tamanho (exatamente o len() de programas da listagem), já que sabemos que, por enquanto e internamente, ela é um list:

```
class Playlist:
    def __init__(self, nome, programas):
```

```
        self.nome = nome
        self._programas = programas

    @property
    def listagem(self):
        return self._programas

    @property
    def tamanho(self):
        return len(self._programas)
```

No caso de usarmos um dicionário, mesmo com o `len()` funcionando neste caso, não importa ao usuário (externo), ao cliente daquela classe, como a nossa listagem de programas está sendo implementada internamente. A partir de agora temos a flexibilidade de fazermos alterações, caso desejemos.
Em `list`, até poderíamos sobrescrever os métodos, mas isto seria confuso, pois criaríamos um `list` com comportamentos estranhos.
Conseguimos resolver a questão da complexidade, porém nos deparamos com outro problema: quebramos a parte inferior do código, e não conseguimos mais fazer a execução em cima de `playlist_fim_de_semana`, sendo necessário chamar `listagem` após o `.`:

```
print(f'Tamanho do playlist: {len(playlist_fim_de_semana.listagem)}')


for programa in playlist_fim_de_semana.listagem:
    print(programa)
```

O `len()` que colocamos mais acima também deixa de existir, assim como o `in`. É melhor ou pior dessa forma?
Se executarmos o código acima, ele funcionará, mas vocês viram que o código está bem esquisito? É porque ainda não estamos usando a parte legal do Python, de que falaremos muito em breve!

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

---

1) Comece a abstrair mais a ideia da playlist. Não dá pra colocar em uma `list` e somente depois dar nome, por exemplo. Você precisa criar uma classe pra representar as diferentes playlists.
Então, crie a classe logo abaixo da classe `Serie`:

```
class Playlist():
    def __init__(self, nome, programas):
        self.nome = nome
        self.programas = programas
```

```
    def tamanho(self):
        return len(self.programas)
`
```

Com esta classe, você consegue representar melhor a playlist.

2) Mude o código de impressão e verifique se vai dar certo:

```
vingadores = Filme('vingadores - guerra infinita', 2018, 160)
atlanta = Serie('atlanta', 2018, 2)
tmep = Filme('todo mundo em panico', 1999, 100)
demolidor = Serie('demolidor', 2016, 2)

vingadores.dar_likes()
vingadores.dar_likes()
vingadores.dar_likes()
atlanta.dar_likes()
atlanta.dar_likes()
tmep.dar_likes()
tmep.dar_likes()
demolidor.dar_likes()
demolidor.dar_likes()

listinha = [atlanta, vingadores, demolidor, tmep]
minha_playlist = Playlist('fim de semana', listinha)

for programa in minha_playlist:
    print(programa)
```

Com essa alteração, o código gerou um erro, sinalizou que a playlist não é um iterable, um iterável.
3) Realmente não é. Por agora, corrija isto apenas pegando a lista interna da `Playlist`:

```
for programa in minha_playlist.programas:
    print(programa)
```

Funcionou! Só que desse jeito, você está acessando uma informação interna da `Playlist`. E se você fizer uma herança da classe `list`?
4) Tente isso para facilitar a iteração. Se você definir que `Playlist` herda de `list`, pode dizer que `Playlist` é um `list`, e como consequência disso, já que `list` é `Iterable`, `Playlist` também será.
O código fica assim:

```
class Playlist(list):
```

```
    def __init__(self, nome, programas):
        self.nome = nome
        super().__init__(programas)
```

5) E a parte da impressão muda também, para percorrer diretamente a Playist:

```
for programa in minha_playlist:
    print(programa)
```

6) Perceba que quando você herda de list, você reutiliza o init da superclasse, que pode receber os itens em forma de list. Outro detalhe importante é que não é necessário mais o método tamanho, já que list suporta o len do Python.
Se você quiser verificar o tamanho, pode fazer o seguinte:

```
print(f'Tamanho: {len(minha_playlist)}')
```

7) Com a herança do list, como foi falado na aula, você começa a usar muitas funcionalidades desta classe que já estavam prontas. Isso é muito bom, pois você não precisa reinventar a roda, porém tem alguns problemas nessa abordagem, como acoplamento e aumento da complexidade da sua classe. Para tentar diminuir esta complexidade, dê alguns passos para trás nesse momento, para tentar aproveitar apenas as vantagens. Deixe a classe Playlist assim:

```
class Playlist():
    def __init__(self, nome, programas):
        self.nome = nome
        self._programas = programas

    @property
    def listagem(self):
        return self._programas

    @property
    def tamanho(self):
        return len(self._programas)
```

8) E o código de impressão deste jeito:

```
for programa in minha_playlist.listagem:
    print(programa)

print(f'Tamanho: {len(minha_playlist.listagem)}')
```

Essas alterações removeram a complexidade da herança, e encapsula a listagem pra ficar claro pro usuário que existe uma propriedade representando os programas listados e também o tamanho (já que você perdeu a vantagem de herdar list).
Deu uma sensação que o código andou para trás agora, que você perdeu algumas vantagens de usar a herança. Mas na próxima aula, você dará um jeito pythônico nisso!

Nesta aula, aprendemos sobre:

•Herança de um tipo built-in (nativo)
•Vantagens da herança de um iterável
•Desvantagem de fazer herança

Bons estudos e até a próxima aula!

AULA 5

Vamos tentar entender o porquê de termos removido a herança da nossa classe. Neste caso, estávamos herdando de list, e esta ideia de que Playlist é um list não era tão verdade, já que o list possui vários comportamentos que desconhecíamos. Além disso, estávamos adicionando acoplamentos ao nosso modelo de classes.

Por meio de uma herança, acessamos todos os comportamentos que se encontram na classe superior, isto é, a superclasse. Nem sempre queremos tais comportamentos e, além disso, se tivermos que alterar algo nela, tudo que estiver interligado será afetado também, o que não é tão interessante.

Queremos encontrar uma alternativa para um encapsulamento de list sem depender tanto da outra classe, que mal conhecemos. No nosso código, temos que acessar a listagem de playlist_fim_de_semana para conseguirmos percorrê-la:

```
for programa in playlist_fim_de_semana.listagem:
    print(programa)
```

Em outro ponto, também acessamos listagem para conseguirmos ver a contagem de itens dentro dela:

```
print (f'Tamanho do playlist: {len(playlist_fim_de_semana.listagem)}')
```

O que não parece muito correto é que, anteriormente, estávamos usando Playlist como algo que pudéssemos iterar, e essa capacidade foi perdida. Se removermos .listagem no loop do for e rodarmos o código, receberemos o erro indicando que Playlist não é um objeto iterável.
Para nós, é desvantajoso perder esta funcionalidade. De que maneira poderemos fazer isso, então, sem termos que recorrer à herança?

No Python, existe um jeito de fazer com que a classe seja considerada iterável, sem a necessidade de usarmos a herança. É preciso, porém, entendermos o que algo iterável faz. No nosso caso, temos as definições de listagem() e tamanho(), valores que estão sendo retornados como propriedades.
Há um método mágico - um dunder method - que, ao ser implementado, permite que a classe seja considerada um objeto iterável: o __getitem__(). Este método define algo que é iterável e, no caso, precisaremos receber um item para que este seja repassado à lista interna que estamos utilizando, isto é, programas.

```
def __getitem__(self, item):
    return self._programas[item]
```

Assim, repassamos um item para a nossa lista interna de programas e, se rodarmos o código, já conseguimos iterar por todos os itens da playlist. Além disso, conseguimos fazer outras operações, como o seguinte trecho:

```
print(vingadores in playlist_fim_de_semana)
```

Isso nos permite sabermos se vingadores está em playlist_fim_de_semana. Ao executarmos, receberemos True. Ou seja, ganhamos o in e o for in. Também é possível exibirmos o primeiro item da lista ([0]) com o código abaixo:

```
print(playlist_fim_de_semana[0])
```

Na verdade, ao retirarmos a herança, tivemos uma melhoria de prática. Para tentar entendermos o porquê da herança ser boa ou ruim dependendo do seu propósito, vamos pensar nos motivos para seu uso:

•Interface, quando queremos resolver questões relativas a polimorfismo;
•Reuso do código, ou remoção de duplicações.

Quando estávamos herdando de uma lista, tínhamos apenas um motivo - a classe Playlist não precisa ser uma lista efetivamente, e sim ter algumas de suas funcionalidades. Ao mesmo tempo, queríamos alguns códigos que faziam sentido de estarem na nossa classe, como uma solução para mensurar quantidades na lista, sem termos que implementar ou verificar isto manualmente.
Ou então no caso de precisarmos iterar a lista sem termos que implementar tudo isso na classe Playlist, reaproveitando este código de algum outro lugar. Porém, isto se tornou inviável, pois estávamos pensando somente no reuso, e a interface não era tão importante. Da mesma forma, se outra classe quisesse usar a nossa como sendo um list, talvez não fizesse sentido, pela sua especificidade e por não ser exatamente um list, como uma classe built-in, nativa, do Python.
É importante levarmos estes dois motivos em conta ao decidirmos pela herança, isto é, que optemos por ela tanto para termos interface quanto para reuso de código, para que o polimorfismo seja absorvido e também para não termos código duplicado.

No caso do reuso, com a herança (que chamamos de extensão), estamos estendendo a classe, usando uma mais genérica e estendendo-a e, assim, somos capazes de usar ambas da mesma forma. Significa que teremos acesso a todo o código referente à superclasse.
Se quisermos algo diferente, teremos que, forçadamente, sobrescrever o código da superclasse. O outro jeito de fazermos reuso, que não é tão nocivo, nem envolve tanto acoplamento (evitando assim que quaisquer alterações feitas na superclasse não interfiram negativamente na subclasse), é a composição.
Em vez de termos um relacionamento é um, teremos Playlist tem um list, assim sendo, ninguém precisa saber como a lista interna funciona, mesmo se quisermos fazer uma implementação de lista diferente da que está sendo usada. Isso porque a nossa interface é mais simples, e não precisamos expor todos os métodos de list.
Com isso, não precisamos de herança para obtermos as vantagens que queríamos, e então nos questionamos quanto à sua real necessidade. Então, se você tiver apenas um dos motivos apresentados, é possível pensar um pouco melhor em uma solução.

No caso, temos uma playlist que se comporta como um(a) sequência iterável, e testamos a implementação do método mágico __getitem__(), que indica ao Python que a classe poderia ser usada para um for in, ou um in, para verificar se o item está contido em uma determinada lista, e também poderíamos acessar um item específico por meio do seu índice.

Isto porque o Python, e estes aspectos mais idiomáticos da sintaxe da linguagem, funciona bem quando utilizamos estes métodos mágicos, que possuem o double underscore (underscores duplos), com os quais passamos ao Python uma ideia de maneira mais clara.
O nome desta característica da linguagem é Duck typing, termo famoso por causa de uma fala do Alex Martelli. Este termo remete à "tipagem de pato", dando a ideia de que não precisamos necessariamente identificar uma ave para saber se trata-se de um pato ou não, basta sabermos se ela emite o mesmo som que o pato, voa ou anda como ele.
Não precisamos nos preocupar com isto, pois o próprio Python verificará se ele se comporta como tal. Então, se sabemos que ele se comporta como um iterável, não precisamos nos preocupar com a sua tipagem. Essa é uma grande vantagem da linguagem, e agora temos uma noção melhor de como estes métodos mágicos podem nos ajudar.

Percebam que ficou faltando a saída de listagem no código, uma vez que ela se tornou desnecessária. Além disso, deveríamos ter, também, uma forma de demonstrarmos o tamanho da playlist como se ela fosse um sized, uma classe que sabe informar seu tamanho.
Não temos um método size() porque o len() funciona de uma forma esperada pela linguagem Python, a qual utilizará aquilo que o objeto que está sendo passado para ele tiver. Se nosso objeto souber indicar seu próprio tamanho, ele mesmo irá conseguir resolver isso.
Por exemplo, se executarmos o código abaixo, receberemos no console que não há nenhum len() para ser executado em Playlist:

```
print(f'Tamanho do playlist: {len(playlist_fim_de_semana)}')


for programa in playlist_fim_de_semana:
    print(programa)
```

Vamos resolver isso a seguir!

Já entendemos como funciona o duck typing, e agora aprofundaremos um pouco esta ideia. Além dele, já que uma lista está sendo usada internamente, também vimos que podemos substituir a herança por um pouco de composição. Sendo assim, podemos fazer a nossa playlist se passar por outro tipo de objeto específico, sem precisarmos da herança; e não chamamos isso de polimorfismo, e sim de duck typing.

Criada a nossa playlist, voltaremos ao código, em que implementamos o método mágico __getitem__(), o qual representa o comportamento não de uma lista - que tem mais de um -, mas de uma sequência capaz de ser iterável, neste caso.
A partir deste dunder method conseguimos iterar para uma lista, e verificar se existe algum item lá dentro por meio do operador in. Neste caso, obtivemos muitos ganhos simplesmente implementando um comportamento - basicamente, o duck typing consiste nisso. Não precisaremos herdar de uma classe, ou informar que somos de um determinado tipo, mas teremos que nos comportar como um objeto daquele tipo.
Com base nisso, começaremos a criar alguma melhoria nesta classe para podermos compreender melhor a ideia do duck typing e deixar o código ainda mais pythônico. Tínhamos visto anteriormente que a forma de lermos o tamanho de uma listagem, ou algum iterável, ocorre verificando-se o len(). No entanto, fomos "forçados" a colocar o tamanho() de volta, com que internamente chamamos o len() de _programas, que é a lista.

Como indicamos que temos uma forma diferente de implementar este len(), juntamente a programa, no loop for in?
Já que playlist_fim_de_semana representa uma lista, poderíamos implementar algo deste tipo:

```
len(playlist_fim_de_semana)
```

Ao rodarmos o código, receberemos o erro

```
TypeError: object of type 'Playlist' has no len()
```

Ou seja, não existe len() em Playlist. Isto porque há um dunder method que poderá ser implementado para que a lista se comporte como um sized, uma ideia de algo que possui tamanho, e que então precisará implementar um __len__() para que o len() externo possa funcionar em nossa classe.
No trecho abaixo, removeremos o @property, pois a partir de então ele é um método mágico:

```
@property
def listagem(self):
    return self._programas
```

E o deixaremos da seguinte maneira:

```
def __len__(self):
    return len(self._programas)
```

Deste modo, sempre que o len() externo for chamado, conseguiremos obter o __len__() da nossa listagem interna. Executaremos o código novamente, mas antes disso deletaremos .listagem do trecho que indica a impressão:

```
print(f'Tamanho do playlist: {len(playlist_fim_de_semana)}')


print(playlist_fim_de_semana[0])
```

O código rodará sem nenhum problema, mas será que existem outros métodos que podemos implementar para ganharmos mais funcionalidades? Basicamente, o que os métodos com dois underscores na frente e outros na frente fazem?

Existe um conceito chamado Python Data Model. Anteriormente, dissemos que nossa Playlist se comporta como uma sequência (__getitem__()) que possibilita o uso de diversos recursos da linguagem.
Por exemplo, o for funciona com a listagem, assim como o in, e desta vez implementamos o __len__(), que também suporta a classe Playlist. Com isso, conseguimos fazer muito com a nossa classe, daquilo que é compatível com a estrutura e os protocolos da linguagem.
No Python Data Model, todo objeto em Python pode se comportar de forma a ser compatível e mais próximo à linguagem, e de toda a ideia idiomática dela. O len() do Python, por exemplo, se diferencia um pouco de outras linguagens.
Há outras formas:

Para quê?	Método
Inicialização	__init__
Representação	__str__, __repr__
Container, sequência	__contains__, __iter__, __len__, __getitem__
Numéricos	__add__, __sub__, __mul__, __mod__

Chamamos o objeto utilizando parênteses quando queremos inicializá-lo, junto ao nome da classe. O que define isso também é um método com dois underscores, que faz parte deste Data Model. Como já vimos anteriormente, quando vamos representar ou converter um objeto como uma string, há dois métodos que podem ser implementados.
O `__repr__()` é utilizado para demonstrar como o objeto foi criado, útil mais para o compilador do Python do que para o usuário final. Dentre os métodos que servem para sequências - que são contêineres, para iterações -, há o `__contains__()`, que também fará o `in` funcionar. Também é possível mudarmos a forma como ele funciona, e implementá-lo de forma mais performática.
O `__iter__()` define protocolos de iteração, então, estamos criando o iterador a ser retornado.
O `__len__()`, por sua vez, retorna o tamanho da lista, e o `__getitem__()`, que também já vimos, nos ajuda a percorrermos a lista e pegarmos um item específico dela.
Não apresentamos todos os dunder methods do Python Data Model, apenas os mais usados e famosos, pois são muitos. Ainda, poderemos definir como funcionam os operadores do Python com o nosso objeto.
Por exemplo, imaginem que queremos adicionar um item à playlist - neste caso, não poderíamos utilizar um `append()`, porque não o herdamos da estrutura de lista. No entanto, poderíamos definir que o operador `+` fará uma adição de um item na nossa lista interna. Para isto, é necessário implementarmos `__add__()`.
Assim, conseguimos implementar o funcionamento dos operadores aritméticos com o objeto, ou com a forma como eles se interagem, o que é bem interessante. Para entendermos como tudo isso funciona no Python após a implementação destes métodos, temos o seguinte:

Para quê?	Como?
Inicialização	obj = Novo()
Representação	print(obj), str(obj), repr(obj)
Container, sequência	len(obj), item in obj, for i in obj, obj[2:3]
Numéricos	obj + outro_obj, obj * obj

No caso da inicialização, significa que estamos chamando o `__init__()` do objeto, e no caso de obj[2:3], conseguimos fazer o slice da lista acessando somente um trecho da mesma, por meio de um "fatiamento" dela.
Isso tudo nos dá base para definirmos uma interface que seria para uso por convenção. Quem já trabalhou com Java ou outras linguagens talvez tenha visto a ideia de interface de um jeito um pouco diferente. Vocês já devem ter ouvido falar em interface como sendo uma palavra reservada da linguagem, que representa uma noção de contrato que define que é necessário implementar vários métodos e, dependendo da linguagem, ela não possui implementação.
No Python, é algo muito próximo a algumas linguagens do tipo Smalltalk, isto é, de protocolo. Isto quer dizer que nosso objeto precisa se comportar daquele modo específico, sendo necessária a implementação de métodos que comportem segundo um protocolo específico.
Por exemplo, digamos que temos um protocolo de sequência, ou seja, temos que fazer com que nosso objeto se comporte como uma sequência por meio do uso destes métodos mágicos. Assim, o objeto fica mais preparado para trabalhar com estes aspectos idiomáticos da linguagem.

Além disso, há uma grande vantagem: implementamos apenas o `__getitem__()` e, se tivéssemos uma interface como no Java, teríamos que implementar todos os métodos. Neste caso, com o `__getitem__()`, ele funcionará perfeitamente pois temos como fazer isso parcialmente.

Então, o duck typing nos permite que sigamos o protocolo de forma parcial, o que acaba sendo bem interessante. Veremos mais aspectos sobre este tipo na linguagem durante todos os cursos de Python.
Mas... E a questão da interface? Será que existe uma maneira de garantir que em outra linguagem que não o Python tenhamos que implementar um método abstrato, por exemplo, para que a subclasse (no caso de uma herança) precise implementar um método dunder? Como faremos isso no Python?

Vamos falar sobre interfaces - será que no Python é possível fazê-las? No caso do Java, por exemplo, a interface consiste em uma estrutura da linguagem que garante que todos que a implementarem terão a obrigação de implementar todos os métodos abstratos nela.

No Python, queremos que uma subclasse de alguma classe qualquer seja obrigada a implementar uma estrutura de métodos. Por exemplo, se tivermos que criar uma classe que é uma lista mutável, queremos garantir que todas as suas filhas sejam obrigadas a implementar o `getitem()`, pois caso contrário não será uma lista mutável.
Neste caso, é preciso alguma estrutura específica do Python, pois trata-se de uma linguagem dinâmica. Existem algumas classes abstratas denominadas ABC, acrônimo para Abstract Base Classes.
No Python, não é muito aconselhável criar nossas próprias classes abstratas, por ser um aspecto mais avançado da linguagem, já que lidaremos com metaprogramação, e afins. O Python recomenda que, caso queiramos criar uma classe que dependa de outra, precisamos checar se já não existe uma destas classes base prontas para uso.

Se importarmos um pacote `collections.abc`, por exemplo, que contém diversas classes que podem ser utilizadas, inclusive o `MutableSequence`, uma sequência mutável cujos valores podem ser alterados:

```
from collections.abc import MutableSequence
```

Digamos que iremos criar nossa `Playlist` novamente, e ela não contém nada. Se queremos herdar de `MutableSequence`, que não possui apenas métodos abstratos, mas também comportamentos, os quais queremos absorver, teremos algo como:

```
class Playlist(MutableSequence):
    pass
```

E se executarmos o código acima, teoricamente o programa acusaria um erro, certo?

No entanto, o que acontece é que ele roda sem problemas, pois o Python, de novo, é uma linguagem compilada e dinâmica, e permite a execuções diretas. Sua tipagem é dinâmica e, neste caso, não temos como validar nada.

Para isso, já que a classe está implementando o `MutableSequence` corretamente, precisaremos criar o objeto desta classe:

```
filmes = Playlist()
```

Feito isso, ao executarmos novamente, o erro que recebemos é o seguinte:

```
TypeError: Can't instantiate abstract class Playlist with abstract
methods __delitem__, __getitem__, __len__, __setitem__, insert
```

Ou seja, são indicados todos os métodos que precisam ser implementados para podermos herdar o comportamento ou funcionamento da MutableSequence. Deste modo, entendemos que, tendo um pacote collections.abc, precisamos saber se há alguma coleção a ser implementada, ou algo a ser reforçado para que a classe implemente corretamente, o que será buscado no pacote de coleções.
No caso de tentarmos implementar MutableSequence, seremos avisados das funções e métodos que também precisam ser implementados na nossa classe para que ela se comporte perfeitamente, como uma classe ABC.
Este conceito surgiu para complementar a ideia do duck typing, pois existem algumas validações que precisamos fazer dependendo daquilo que estivermos criando em nosso sistema, que o duck typing ainda não nos garante.
Pode até ser que tenhamos o comportamento do __getitem__(), por exemplo, mas pode ser que queiramos outros, e validar se aquele objeto realmente os possui. E se herdarmos diretamente, teremos os comportamentos, mas tampouco teremos a garantia do tipo específico do objeto de que estamos herdando, isto é, da superclasse.
Neste caso, estamos reforçando a validação e inspecionando a classe de alguma maneira, por meio da abstract base class para saber se ela implementa tais métodos. Inclusive, elas não existem apenas em collections.abc, e sim, no caso, de from numbers import Complex também.
Assim, se criarmos uma classe Numero que herda de Complex, quando quisermos criar filmes = Numero(), receberemos o mesmo erro que houve com Playlist. Isso nos indica que podemos ter o mesmo funcionamento de uma classe abstrata do Java, para garantirmos uma ideia de comportamento a ser seguido.
No caso de termos uma classe abstrata, poderíamos criar um __getitem__() que recebe um item, como em:

```
from numbers import Complex

class Numero(Complex):
    def __getitem__(self, item):
        super().__getitem__()
```

Também poderemos usar o super(), mas... Ele não era abstrato? Sendo assim, somos forçados a implementá-lo? Em super(), existe implementação?
Diferentemente do Java, no Python, pode ter uma implementação na classe que possui o método abstrato. Ou seja, se este for o caso, poderemos aproveitar esta implementação; seria um início para podermos cuidar de algo. Neste exemplo, uma parte de __getitem__() já está implementada, e podemos continuar implementando, ou modificá-la.
No entanto, para isto é preciso consultar a documentação destas classes. O PyCharm possui uma facilidade, que é clicar no nome da classe, o que nos levará à sua documentação, a doc string, uma string de documentação.
Veremos inclusive que algumas abstract base classes herdam de mais de uma classe, aspecto do Python que também não costumamos conhecer quando estamos mais familiarizados com outra linguagem de programação.
Imaginem que temos outro problema agora, em relação a funcionários e a herança no Python. No nosso sistema, temos o Funcionario, que pode ser do tipo Alura ou Caelum. Se tivéssemos funcionários que são de ambos os tipos, como lidaremos com isso?
Verificamos que é possível usar classes abstratas do sistema, e conseguimos alguns benefícios de bandeja, por exemplo:

•Tenho um erro que me diz, em tempo de instanciação, se eu esqueci de implementar algum método da superclasse
•E também sou impedido de instanciar um objeto do tipo da superclasse, pra não ter problema com os métodos abstratos.
•Ainda posso aproveitar código dos meus métodos abstratos (que podem ter implementação na classe mãe)
Se você ficou com curiosidade sobre como criar uma classe abstrata, vamos pensar no seguinte caso: imagine que não queremos permitir que ninguém instancie um objeto da classe Programa, e queremos garantir que todo mundo implemente o __str__ nas suas subclasses. Parece uma boa ideia.
Para fazer isso, o nosso código ficaria assim:

```
from abc import ABCMeta, abstractmethod
class Programa(metaclass = ABCMeta):
    @abstractmethod
    def __str__(self):
        pass
```

Apenas com isso, podemos garantir que o __str__ será implementado nas nossas subclasses, se não for implementado em alguma, será avisado em tempo de instanciação (não vai conseguir criar instâncias).

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

Na última aula, o código acabou ficando sem herança, mas perdeu algumas funcionalidades do Python, como poder usar o for in, in e o len.
1) Para conseguir continuar com as vantagens do polimorfismo sem precisar herdar a classe list, faça o seguinte, altere a classe Playlist para:

```
class Playlist():
    def __init__(self, nome, programas):
        self.nome = nome
        self._programas = programas

    def __getitem__(self, item):
        return self._programas[item]

    @property
    def listagem(self):
        return self._programas
```

```
    @property
    def tamanho(self):
        return len(self._programas)
```

2) Apenas com esta alteração, você consegue agora iterar sobre a Playlist, sem acessar a listagem como antes:

```
for programa in minha_playlist:
    print(programa)
```

No Python, não é preciso herdar de uma classe específica pra ter polimorfismo. O que é importante no Python é: se você quer um iterável, devo se preocupar com o que um iterável deve fazer.

O nome dessa característica é duck typing.

3) Será que dá para alterar a classe de outra forma para suportar também o len, ao invés de ter que acessar o método/property tamanho? Sim! Faça o seguinte:

```
class Playlist():
    def __init__(self, nome, programas):
        self.nome = nome
        self._programas = programas

    def __getitem__(self, item)
        return self._programas[item]

    def __len__(self):
        return len(self._programas)
```

Desta forma, você consegue chamar len(minha_playlist), já que Playlist implementa __len__. Isso torna a Playlist um Sized (alguém que implementa __len__).

Observação: Você não precisa mais dos métodos/propriedades listagem e tamanho, então pode apagar :)

4) Com duck typing, você ganha muita flexibilidade ao não precisar ficar preso aos tipos dos objetos, só que em alguns momentos você pode querer ter restrições, como de uma garantia que uma classe implemente os métodos que você quer.

Em outras linguagens, há a ideia de classes e métodos abstratos, que forçam as classes filhas a implementar alguns métodos.

Para fazer o mesmo no Python, é possível usar classes abstratas, as chamadas ABC (A*bstract *Base Classes). Existem classes já prontas que ajudam nessa ideia.

Para exemplificar, crie um novo arquivo com o nome testeAbc.py, para teste, com o seguinte conteúdo e execute-o:

```
from collections.abc import MutableSequence


class MinhaListinhaMutavel(MutableSequence):
    pass
```

Você vai perceber que não acontece nada quando apenas este código é executado.

5) A classe MinhaListinhaMutavel herda de MutableSequence, ou seja, é desejável que o Python avise que tem que implementar todos os métodos abstratos de uma MutableSequence. Só que parece que não funcionou.
Como Python é uma linguagem de tipagem dinâmica, não dá pra ter essa garantia só percorrendo o código de definição da classe. O que dá pra fazer é validar os métodos em tempo de instanciação.
Adicione o código abaixo e teste novamente:

```
objetoValidado = MinhaListinhaMutavel()

print(objetoValidado)
```

Agora dá um erro, dizendo que você esqueceu de implementar todos os métodos necessários para tornar a classe uma MutableSequence.
Sempre que você quiser garantir a implementação de alguns métodos, pode recorrer às classes já existentes em collections.abc e outros pacotes também.

Nesta aula, aprendemos sobre:

•Duck typing
•Python data (object) model
•Dunder methods
•Uso de ABCs

AULA 6

Vamos resolver o problema apresentado anteriormente, em relação a Funcionario, para ilustrar uma nova forma de herança. Temos três classes criadas: Funcionario, Alura e Caelum, que são representações de funcionários na empresa. Uma nova regra implica na necessidade de especificarmos melhor como ficam os funcionários junior, pleno e sênior.

Sendo assim, teremos também Junior, Pleno e Senior, e há uma relação entre eles. Por exemplo, Junior só poderá acessar informações de Alura, enquanto o Pleno e o Senior têm mais responsabilidades, e poderão acessar os dois tipos.
Verificaremos como trabalharíamos se tivéssemos que herdar comportamentos destas classes superiores. Pleno e Senior, portanto, estarão fazendo uma herança múltipla, de dois tipos diferentes, tanto de Alura quanto Caelum.
Na prática, temos uma classe com um arquivo pronto (contendo Funcionario, Caelum e Alura). Em Funcionario, há registra_horas(), e a impressão da quantidade de horas registradas, bem como uma forma de mostrar as tarefas realizadas, por meio de mostrar_tarefas().

```
class Funcionario:
    def registra_horas(self, horas):
        print('Horas registradas...')


    def mostrar_tarefas(self):
```

```
        print('Fez muita coisa...')

class Caelum(Funcionario):
    def mostrar_tarefas(self):
        print('Fez muita coisa, Caelumer')

    def busca_cursos_do_mes(self, mes=None):
        print(f'Mostrando cursos - {mes}' if mes else 'Mostrando cursos desse mês')

class Alura(Funcionario):
    def mostrar_tarefas(self):
        print('Fez muita coisa, Alurete!')

    def busca_perguntas_sem_resposta(self):
        print('Mostrando perguntas não respondidas do fórum')
```

Caelum e Alura possuem formas distintas de lidar com as tarefas. Isto significa que estamos sobrescrevendo mostrar_tarefas() de Funcionario para ambos, cada qual de um jeito. No entanto, temos particularidades, como busca_cursos_do_mes() e busca_perguntas_sem_resposta(), métodos específicos de cada classe.

Portanto, precisaremos fazer com que os funcionários - que serão de Junior, Pleno, ou Senior - herdem comportamentos adequadamente. No Python, como faríamos esta herança múltipla?

Vamos começar pela criação da classe Junior, a mais simples por herdar de apenas uma classe, Alura. Não precisaremos incluir a implementação, pois testaremos o comportamento da classe mãe.

Criaremos a outra classe também, Pleno, e colocamos as definições de quais classes queremos herdar entre parênteses e separados por vírgula. Para testarmos o código, vamos criar um funcionário, jose, com que testaremos acessando o método busca_perguntas_sem_resposta(). Também testaremos o funcionamento de busca_cursos_do_mes():

```
class Junior(Alura):
    pass

class Pleno(Alura, Caelum):
    pass

jose = Junior()
jose.busca_perguntas_sem_resposta()
jose.busca_cursos_do_mes()
```

Ao executarmos o código, receberemos o seguinte erro:

```
AttributeError: 'Junior' object has no attribute 'busca_cursos_do_mes'
```

É indicado "erro de atributo" pois no Python tudo é atributo, inclusive métodos. Era esperado que recebessemos este erro, porque este método faz parte de Caelum, e não de Alura. Sendo jose Junior, ele não poderia ter acesso a busca_cursos_do_mes().

Vamos testar novamente, desta vez sem a linha jose.busca_cursos_do_mes(). Além disso, incluiremos luan, que será Pleno, tipo que possui duas mães, Caelum e Alura, o que permite acesso a todos os métodos:

```
jose = Junior()

jose.busca_perguntas_sem_resposta()


luan = Pleno()

luan.busca_perguntas_sem_resposta()

luan.busca_cursos_do_mes()


luan.mostrar_tarefas()
```

Ao executarmos, obteremos:

```
Mostrando perguntas não respondidas do fórum
Mostrando perguntas não respondidas do fórum

Mostrando cursos desse mês

Fez muita coisa, Alurete!
```

Basicamente, a herança múltipla funciona desta maneira, e muitos se perguntam o porquê de não existir herança múltipla em outras linguagens, e o motivo é que isso pode acabar ficando muito complexo. Neste exemplo, foi impresso "Fez muita coisa, Alurete!", porém, se a herança vem de Caelum e Alura, por que esta mensagem, e não "Fez muita coisa, Caelumer"?

Já vimos a estrutura para a definição da herança múltipla, mas falta entendermos o motivo de termos recebido esta mensagem, neste caso. Entenderemos melhor esse aspecto a seguir!

Chamar mostrar_tarefas() acabou deixando nosso código um tanto confuso. Para entendermos melhor o que houve, vamos executá-lo novamente. A linha "Fez muita coisa, Alurete!" está executando este método, e serve tanto para quem é Pleno quanto para quem é Junior.

O método mostrar_tarefas() está implementado para os dois tipos e, em algum momento, o programa tem que decidir qual delas utilizar. Por que a escolha foi sempre referente à classe Alura? Quando definimos Pleno, não designamos qual seria considerado primeiro, se Alura ou Caelum. A ordem normalmente segue da esquerda para a direita, como em uma leitura usual. Assim, a primeira opção é Alura.

Para a tomada de decisão sobre qual método deverá ser executado quando temos diversas superclasses que o possuem, internamente, a versão 3 do Python usa um algoritmo chamado MRO (Method Resolution Order), com um funcionamento que começa a busca pela classe atual, que é a própria classe.

Por exemplo, em Pleno, a primeira classe que será buscada neste caso é o próprio Pleno. Em seguida, o acesso será em sua classe mãe. No caso, ela possui duas mães, Alura e Caelum, e é aí que ocorre a primeira distinção - a primeira classe mãe é Alura, portanto ela será consultada primeiro. Além disto, o cálculo do algoritmo acessará não apenas esta classe, mas todas as classes mães de Alura, e assim por diante, hierarquicamente:

```
Pleno > Alura > Funcionario > Caelum > Funcionario
```

Assim, é feita a verificação de qual método será chamado. Antes de executarmos o código novamente para confirmar se é isso mesmo, modificaremos a classe Alura comentando o trecho com o método mostrar_tarefas(), para vermos de onde será feita a chamada:

```
class Alura(Funcionario):

    # def mostrar_tarefas(self):

    #     print('Fez muita coisa, Alurete!')
```

Receberemos "Fez muita coisa..." e, mais abaixo, "Fez muita coisa, Caelumer". A Alura não tem mais a implementação de "Fez muita coisa...", e como executamos para Junior e para Pleno, o primeiro chama "Fez muita coisa..." , que vem de Alura (que não possui implementação), portanto seu ancestral foi acessado, Funcionario.

No caso de Pleno, deveria ser feito o mesmo, ou seja, começado por Alura, que não tem implementação, passando para Funcionario. No entanto, ele passou para Caelum. Por que isto aconteceu?

Há duas etapas para a verificação do algoritmo MRO, que exige uma noção de que há uma repetição nesta lista, como no caso de Funcionario. Esta repetição precisa ser removida para que seja possível encontrar o local de onde acessaremos o método que está sendo implementado.

No nosso caso, em vez de Funcionario, Caelum é que foi acessado, pois a parte da remoção da duplicidade verifica se Funcionario é "uma boa cabeça" (good head). Caso positivo, quer dizer que poderemos mantê-la. Como o primeiro Funcionario não é uma good head, iremos removê-la:

```
Pleno > Alura > Caelum > Funcionario
```

Mas o que isto quer dizer, exatamente?

"Boa cabeça" indica que não há nenhuma outra classe que seja da mesma hierarquia, ou seja, que esteja abaixo de Funcionario (neste caso), e que possa ser utilizada. Já que Caelum também herda de Funcionario, podemos utilizá-la no lugar desta.

Para entendermos melhor, vamos voltar ao código, na parte em que definimos Pleno, cuja hierarquia é Alura, seguida de Caelum. Sendo assim, não é muito intuitivo seguir de Alura para Funcionario, e é por isto que se vai para Caelum, de hierarquia mais específica.

Para tornar isso mais explícito, no Python 3, quando a duplicata é removida, é feita uma verificação da existência de alguma outra classe que herde desta. Por exemplo, será que tem alguma classe que herde de Funcionario na lista? Tem, Caelum. Assim, ela é removida, pois este não é seu lugar, já que existe uma classe em que ela pode caber melhor.

Caelum seria um good head. É por isto que houve aquela confusão, sobre termos recebido "Fez muita coisa, Caelumer!" quando deveria ser a mensagem referente a Alura. Mas isso não é tudo. Poderia acontecer de termos classes em que não conseguimos resolver a busca do método, porém não exploraremos isso neste curso. A herança múltipla, e a forma como lidaremos com ela aqui será mais focada para o uso, para quando trabalhamos com uma classe que pode ser provida como herança, mas não criaremos muitas classes, pois normalmente estes recursos um pouco mais

avançados são utilizados na criação de frameworks, ou de um conceito de aplicação a ser utilizada por muitas pessoas.
Neste caso, estamos simplesmente fazendo uso da herança múltipla - temos várias classes e queremos usá-las como herança, e então tomaremos cuidados adequados e aplicaremos boas práticas.

Neste exemplo visto aqui já temos uma solução que funciona: precisaremos saber como este cálculo de MRO funciona. Após Alura, o método que será chamado é Caelum. Vamos verificar nosso código para confirmar que não há mais nada a ser feito em relação a herança múltipla, até porque só vimos como fazemos para criá-la e qual é a ordem de execução.
Quando é mais vantajoso utilizar a herança múltipla?

A ideia é juntarmos comportamentos distintos, e que fazem sentido para uma classe única. Porém, há outros casos - Caelum e Alura fazem parte da mesma hierarquia, mas e se quiséssemos dizer que temos mais informações sobre Funcionarios, como Desenvolvedor ou UX, por exemplo? De que forma trabalharíamos com isso?

Tínhamos um problema em relação aos funcionários de uma empresa, e demos uma olhada em como funciona o cálculo de resolução do algoritmo mas, a partir deste momento, temos todos os funcionários herdando de alguma classe específica. E se quisermos simplesmente adicionar um comportamento para converter o Funcionario para string... Lembra daquele __str__()?

Queremos alguma forma de fazer isso sem interferir na classe mãe, e queremos incluir os métodos de forma a podermos compartilhá-los com qualquer classe, não somente com Funcionario, mas com Aluno também, se este vir a existir, por exemplo. Assim, se quisermos que aqueles que forem da classe Aluno tenham "Hipster" antes de seus nomes, poderemos compartilhar este código.
Imagine que tivéssemos uma classe denominada Hipster, que contém esta lógica ou algoritmo para a conversão de nome, e que poderíamos compartilhar este código com qualquer outra classe, baseando-se em herança.
Supondo que Senior tivesse este comportamento, ou seja, a relação com Hipster, como é que faríamos isso?
Vamos criar a classe Hipster para que ela contenha esta lógica, tendo apenas um método que retornará um texto com "Hipster" na frente do nome. Teremos que criar um nome para cada objeto:

```
class Hipster:
    def __str__(self):
        return f'Hipster, {self.nome}'
```

Reparem que nesta classe o inicializador é desnecessário. Estamos dependendo que a classe que herdará Hipster tenha self.nome, e não queremos que Hipster seja herdada. Também incluiremos Senior, que herdará Alura e Caelum:

```
class Senior(Alura, Caelum):
    pass
```

Indicaremos que todas as nossas classes Funcionario herdarão, além de Funcionario, um método inicializador apenas para a adição de um nome em cada um deles:

```
class Funcionario:
    def __init__(self, nome):
```

```
        self.nome = nome
```

Por ora não estamos nos preocupando com encapsulamento. Assim, poderemos criar um funcionário, como luan - que será sênior -, passando nome, e o exibiremos por meio de print().

```
luan = Senior('Luan')

print(luan)
```

Ao rodarmos a aplicação, será retornado o endereço de memória:

```
        <main.Senior object at 0x03AE29D0>
```

De que forma compartilharemos este código?

Se colocarmos que Senior passa a herdar também da classe Hipster, conseguiremos fazer com que esta forma de exibição da conversão para string seja implementada sem muito trabalho. Rodando o código, teremos:

```
        Hipster, Luan
```

Conseguimos este comportamento implementando uma simples herança, sem o perigo que tínhamos antes, pois não estamos herdando vários comportamentos. Hipster é uma classe muito mais concisa, e possui apenas este comportamento específico.

Com este tipo de classe, conseguimos compartilhar alguns comportamentos com outras classes, os quais talvez não sejam o comportamento core, principal, daquela classe. A conversão de nome para "Hipster" seguido do nome não é tão importante para a classe Senior, que precisa de regras de negócio de um funcionário sênior, isto é, características específicas.

No caso deste comportamento, ele poderá ser compartilhado com qualquer classe. Poderemos ter a classe chamada Aluno com este mesmo comportamento, que não necessariamente será um Funcionario. Por este motivo dizemos que talvez não seja uma boa ideia colocarmos o mesmo comportamento de Aluno em Alura, Caelum, ou Funcionario.

Chamamos estas classes pequenas, cujos objetos nem precisam ser instanciados, de Mixins. Elas são bastante utilizadas em Python no caso de precisarmos compartilhar algum comportamento que não é o mais importante desta classe.

Será que daria pra fazermos algo diferente?

Se quiséssemos, poderíamos implementar um log() para que toda vez que alguém for chamar registra_horas(), o log() do sistema seja feito em algum arquivo, e fosse feita uma auditoria. Para tal, criaríamos uma classe que faria um Logger, que disponibilizaria o log() para a nossa classe.

Desta forma, não precisaremos implementar o log() para cada classe que temos. Se tivéssemos que colocá-lo em Funcionario, por exemplo, não poderíamos reutilizar este código - do log() - em outras classes que precisassem deste comportamento também.

Neste caso, pode ser interessante utilizarmos este tipo de herança. Mas lembre-se de que o mixin envolve a ideia de não termos que instanciar um objeto desta classe. Em Hipster, isto nem funcionaria direito, porque ela está dependendo de um nome para que o nome, ou o texto deste objeto, seja alterado.

Assim, fechamos um pouco o assunto de classes com herança múltipla, e seus usos. Como dito anteriormente, não nos aprofundaremos tanto, mas queria que você tivesse esta visão para conseguir utilizar, de repente, um framework que trabalha com isso, ou que dependa do uso de herança múltipla, ou de mixins, que possuem ideias semelhantes.

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

---

1) Nesta aula, teste um pouco uma outra forma de herança, onde você pode reutilizar comportamentos de mais de uma classe mãe. Para isso, comece com este código:

```
class Funcionario:
    def registra_horas(self, horas):
        print('Horas registradas.')

    def mostrar_tarefas(self):
        print('Fez muita coisa...')

class Caelum(Funcionario):
    def mostrar_tarefas(self):
        print('Fez muita coisa, Caelumer')

    def busca_cursos_do_mes(self, mes=None):
        print(f'Mostrando cursos - {mes}' if mes else 'Mostrando cursos desse mês')

class Alura(Funcionario):
    def mostrar_tarefas(self):
        print('Fez muita coisa, Alurete!')

    def busca_perguntas_sem_resposta(self):
        print('Mostrando perguntas não respondidas do fórum')
```

2) Agora, crie as classes que herdam os comportamentos destas classes já criadas. Crie `Junior`, que herda de `Alura`, e `Pleno`, que herda de `Alura` e `Caelum`, pois trabalha nas duas frentes:

```
class Junior(Alura):
    pass

class Pleno(Alura, Caelum):
    pass
```

3) Para testar estas classes, adicione o código abaixo e execute:

```
jose = Junior()
jose.busca_perguntas_sem_resposta()
```

```
luan = Pleno()
luan.busca_perguntas_sem_resposta()
luan.busca_cursos_do_mes()
```

Com este código, dá para ver que foi possível herdar comportamento das duas classes no funcionário `Pleno`.
4) Agora, adicione o seguinte código na última linha:

```
luan.mostrar_tarefas()
```

Executando, você pode entender qual método `mostrar_tarefas` está sendo chamado.
5) Para visualizar de forma diferente, comente o método `mostrar_tarefas` da classe `Alura`:

```
class Alura(Funcionario):
    # def mostrar_tarefas(self):
    #     print('Fez muita coisa, Alurete!')
```

Execute novamente e veja como a ordem muda.

6) Outro uso comum de herança múltipla é quando você tem aspectos não-funcionais usados por múltiplas classes.

Para compartilhar estes comportamentos, você pode usar Mixins, que são apenas classes que podem ser herdadas mas que não devem ser instanciadas, já que servem para disponibilizar comportamentos para outras classes.
Crie a classe `Hipster`, abaixo da classe `Alura`:

```
class Hipster:
    def __str__(self):
        return f'Hipster,  {self.nome}'
```

7) Adicione um método `init` na classe `Funcionario`, para ter um nome no funcionário:

```
class Funcionario:
    def __init__(self, nome):
        self.nome = nome

    def registra_horas(self, horas):
        print('Horas registradas.')

    def mostrar_tarefas(self):
        print('Fez muita coisa...')
```

8) E em seguida, modifique a classe `Pleno` para herdar também de `Hipster`, no fim da lista, e execute para ver o que acontece ao imprimir o objeto da classe `Pleno`:

```
class Pleno(Alura, Caelum, Hipster):
    pass
```

```
# restante do código ...



jose = Junior('José')
jose.busca_perguntas_sem_resposta()

luan = Pleno('Luan')
luan.busca_perguntas_sem_resposta()
luan.busca_cursos_do_mes()

luan.mostrar_tarefas()

print(luan)
```

9) Com esse print, dá para concluir que agora que há um __str__ compartilhado pela classe Hipster, qualquer classe pode herdar este comportamento, se você desejar.
Os mixins são usados desta forma pois o seu comportamento normalmente não precisa ser de responsabilidade da classe filha, pois esta é mais específica.

Nesta aula, aprendemos sobre:

•Herança múltipla
•Resolução da ordem de chamada de métodos
•Mixins

Chegamos ao final do nosso aprendizado de Orientação a Objetos deste curso, espero que tenha gostado.

Falamos sobre alguns assuntos como herança, polimorfismo, duck typing e herança múltipla. Pratique bastante, faça os exercícios, e nos envie um feedback. Diga o que achou, se foi bom, ruim, pois tudo isso é importante e nos ajuda a melhorarmos cada vez mais, tanto em relação à plataforma Alura quanto aos cursos em si.

Boa sorte, nos vemos em um próximo curso!