Python 3 – Orientações a Objetos

Curso 1 – Introdução a Orientação a Objetos:

1. Aula 1 – Aprenda o Paradigma OO com Python:

1.1. Ao criar uma função, para contas de um banco, por exemplo, podemos colocar como parâmetros o que desejamos que essa conta tenha, como titular, número, saldo, dentre outros.

1.1.1. Dentro da função, criamos uma variável conta que recebe um dicionário: conta = {"titular": titular, "numero": numero, por exemplo, e retorna a conta no final.

1.2. Para criar uma nova conta, criamos uma variável "conta" que recebe a função "cria_conta(titular, numero, saldo...)", com os parâmetros já definidos, desse modo, quando chamarmos essa "conta["titular"]", por exemplo, receberemos o nome que foi dado como parâmetro "titular" logo aqui acima.

1.3. A ideia central do paradigma orientação a objeto é: Dado e funcionalidade (comportamentos) andam juntos.

1.4. O que aprendemos:

1.4.1. Dicionário;

1.4.2. Funções;

1.4.3. Encapsulamento de código.

2. Aula 2 – Classes e Objetos:

2.1. Classes são como receitas de bolo, ou formas, que servem para que o projeto final seja um só.

2.2. Class: Determina uma classe.

2.3. Nomenclatura em classes: NomeDeUmaClasseSeEscreveAssim.

2.4. Pass: Faz com que uma classe ou função passe mesmo que não tenha nada escrito ainda. Python entende que anda será escrito algo lá e não dá erro durante a execução do código.

2.5. Para chamar uma classe no console, é só colocar o nome dela (com a inicial Maiúscula) seguida de (), como uma função.

2.6. Quando guardamos uma classe dentro de uma variável (ex.: conta = Conta()), chamamos essa variável de referência. Ele guarda o endereço da memória onde essa classe foi alocada para saber onde encontrar esse objeto.

2.7. O parâmetro *self* é o que indica a referência, a localização daquela classe.

2.8. Def __init__(self):: É uma função construtora, responsável por criar um objeto.

2.8.1. Funciona basicamente como a função *cria_conta* que falamos lá acima.

2.8.2. Logo depois do *self* colocamos outros parâmetros, como: numero, titular, saldo, dentre outros.

2.8.3. Utilizamos o self para nos referir a locação da referência, ao local onde esse objeto está armazenado.

2.8.4. O código final fica assim:

```
def __init__(self, numero, titular, saldo, limite):
    self.numero = numero
    self.titular = titular
    self.saldo = saldo
    self.limite = limite
```

2.8.5. Colocando os parâmetros ao invés de os números e nomes direto, torna esse molde interativo, fazendo com que possamos criar várias contas utilizando o mesmo construtor, apenas com parâmetros diferentes.

2.9. Podemos criar atributos pré definidos. Caso queiramos que o limite de todas as contas seja o mesmo, menos para contas especiais, basta colocar, junto com os atributos, limite = 1000.0, por exemplo. Nesse caso, durante a criação das contas só colocamos o atributo limite naquelas que terão limites especiais, caso contrário, só precisamos atribuir até o saldo, pois o limite será adicionado automaticamente. Ex.:

```
def __init__(self, numero, titular, saldo, limite = 1000.0):
    self.numero = numero
    self.titular = titular
    self.saldo = saldo
    self.limite = limite
```

2.10. Através da referência podemos acessar diretamente os atributos do objeto criado. Ex.:

```
>>> conta = Conta(123, "Bruno", 55.0, 1000.0)
>>> conta
<conta.Conta object at 0x000001A8A21E7220>
>>> conta.saldo
55.0
```

2.10.1. Primeira linha: variável conta recebe a classe Conta com os atributos.

2.10.2. Segunda e Terceira linha: chama a referência e diz que a variável conta é um objeto no endereço 0x00...

2.10.3. Quarta e Quinta linha: acesso direto ao atributo saldo da conta através da referência/variável conta e exibição do mesmo.

2.11. O que aprendemos:

2.11.1. Classes;

2.11.2. Objetos;

2.11.3. Função construtora;

2.11.4. Endereço e referência de objetos;

2.11.5. Atributos de classe;

2.11.6. Acesso aos atributos através do objeto.

3. Aula 3 – Implementando Métodos:

3.1. Métodos são comportamentos que os objetos tem.

3.2. Métodos são funções que criamos dentro de uma classe para executar coisas.

3.3. Passando como parâmetro o *self* no início da função, sempre estaremos nos referindo à referência que escolhemos no console, seja conta 1, 2, 3... Ex.:

```
def extrato (self):
    print(f'Saldo {self.saldo} do titular {self.titular}')
```

3.4. Para utilizar/chamar o método, precisamos colocar a referência "." E o método que quer executar. Ex.:

```
>>> conta.extrato()
Saldo 55.0 do titular Bruno
>>> conta2.extrato()
Saldo 100.0 do titular Marco
```

3.5. Caso queiramos depositar ou sacar, a lógica é a mesma para as funções criadas lá no começo, com a diferença de que, ao invés de passar o parâmetro "conta" na hora de criar o método, passamos o parâmetro "self", para que a modificação seja feita na conta em que estamos mexendo, seguido do parâmetro "valor", que será adicionado ou tirado. Ex.:

```
def deposita(self, valor):
    self.saldo += valor

def saca(self, valor):
    self.saldo -= valor
```

3.6. No console fica assim:

```
>>> conta.deposita(15.0)
>>> conta.extrato()
Saldo de 70.0 do titular Bruno
>>> conta.saca(10.0)
>>> conta.extrato()
Saldo de 60.0 do titular Bruno
```

3.7. No console não precisamos passar a referência nos () do método pois ela já foi chamada no início.

3.8. Por enquanto estamos fazendo métodos de apenas algumas linhas, mas esses métodos podem conter mais 100 linhas de código, o que para o usuário não importa, pois está chamando o método que quer usar e pronto.

3.9. Quando você cria um objeto conta dentro de uma variável conta, você está dizendo que sempre que mencionar a variável conta, está se referindo ao objeto conta, agora, se você criar um outro objeto conta dentro da mesma variável conta que já tinha um objeto conta, a nossa variável conta irá começar a se referir e ir somente para esse novo objeto conta criado, o antigo objeto conta será totalmente desvinculado apesar de ainda existir, ou seja, ele ainda ocupa memória, mas se torna inalcançável.

3.10. No python existe um processo que recicla esses objetos perdidos/inutilizados, eliminando-os e dando espaço para novos objetos ocuparem.

3.11. None: faz com que uma referência que antes apontava para algo, deixar de apontar, ou seja, se eu criar uma variável chamada outraRef que recebe a variável conta (outraRef = conta) agora temos 2 variáveis referências que nos levam para o mesmo endereço, o objeto conta. Caso eu queira que a variável/referência outraRef deixe de apontar para esse objeto conta, preciso fazer com que ela receba *none* (outraRef = none).

3.12. O que aprendemos:

3.12.1. Métodos, que definem o comportamento de uma classe;

3.12.2. Criação de métodos;

3.12.3. Como chamar métodos através do objeto;

3.12.4. Acesso aos atributos através do self;

3.12.5. Garbage Collector;

3.12.6. O tipo None.

4. Aula 4 – Encapsulamento:

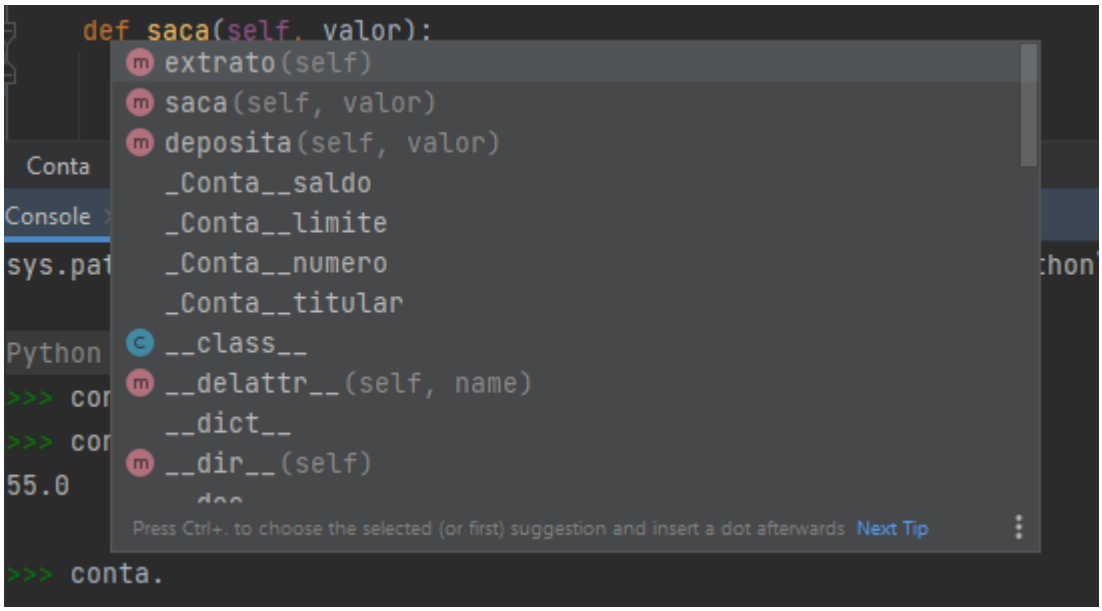
4.1. Para deixar um atributo privado, basta colocar dois "__" antes de cada atributo (em python. Outras linguagens usam palavras como "private"). Ex.:

```
def __init__(self, numero, titular, saldo, limite):
    self.__numero = numero
    self.__titular = titular
    self.__saldo = saldo
    self.__limite = limite
```

4.1.1. Quando fazemos isso esses atributos "somem" quando tentamos acessar os métodos e atributos. Ex.:

4.1.1.1. Sem ser privado:

4.1.1.2.Sendo privado:



4.1.2. Quando está como privado, as primeiras coisas que aparecem são os métodos, em seguida os atributos, ao contrário de quando não estava privado. Porém, nos atributos quando privados, podemos ver que sua aparência está diferente de quando eram públicos, aparecendo *"_Conta__saldo"*, *"_Conta__limite"*, *"_Conta__numero"*...

4.1.3. Apesar de o nome ter mudado, ainda temos acesso direto aos atributos se selecionarmos eles, porém, como um desenvolvedor experiente você sabe que isso não é uma boa prática e se trata de um

aviso do python, dizendo que apesar de poder, você não deve utilizar o acesso direto, sempre pelos métodos.

4.2. Podemos criar métodos que utilizem outros métodos. Caso queiramos transferir dinheiro de uma conta para a outra, precisamos sacar um valor da origem e depositar no destino, certo!? Tendo isso em mente, podemos criar um método transfere() que faça isso automaticamente. Ex.:

```
def transfere(self, destino, valor):
    self.saca(valor)
    destino.deposita(valor)
```

4.2.1. Nesse caso, o self é a origem e não precisa ser declarado novamente, uma vez que o método será chamado a partir da conta que desejamos fazer a transferência, ficando assim no console:

```
>>> conta2.transfere(conta, 10.0)
>>> conta.extrato()
Saldo de 65.0 do titular Bruno
>>> conta2.extrato()
Saldo de 90.0 do titular Marco
```

4.3. Isso se chama encapsulamento, pois colocamos dois métodos encapsulados dentro de um outro, facilitando/encurtando o caminho para a execução do que queremos, desse modo, ao invés de ter que usar 2 métodos separadamente, um para sacar o dinheiro de uma conta e outro para depositar esse dinheiro em outra conta, criamos um método que encapsula esses dois e faz esse processo todo automaticamente.

4.4. Nossos códigos precisam ser coesos, ou seja, precisamos colocar métodos dentro de classes que fazem sentido eles estarem, não faz sentido, por exemplo, criar um método que verifica se o cliente é inadimplente ou não, na classe Conta(), esse método deveria estar na classe Cliente(). Precisamos ficar atentos pois esse tipo de coisa acontece o tempo todo e é normal, já que faz parte do dia a dia modificarmos códigos para que fiquem cada vez melhores e fazendo mais sentido.

4.5. Uma classe deve ter responsabilidade única, quando isso não ocorre, chamamos ela de sem coesão, pois está fazendo mais coisas do que deveria.

4.5.1. Esse é o princípio de *responsabilidade única*.

4.6. Além desse princípio, no início dos anos 2000 Robert C. Martin definiu outros princípios que foram denominados **SOLID:**

4.6.1. **S** - Single responsibility principle;

4.6.2. **O** - Open/closed principle;

4.6.3. **L** - Liskov substitution principle;

4.6.4. **I** - Interface segregation principle;

4.6.5. **D** - Dependency inversion principle.

4.7. O que aprendemos:

4.7.1. Atributos privados;

4.7.2. Encapsulamento de código;

4.7.3. Encapsulamento da manipulação dos atributos nos métodos;

4.7.4. Coesão do código.

5. Aula 5 – Usando Propriedades:

5.1. Getter: basicamente uma nomenclatura para métodos que só devolvem informações, como somente o número do saldo (sem o extrato), somente o titular, limite etc. Quando quiser um método que faça essas coisas, basta nomear ele como get_saldo(), get_titular(). Ex.:

```
def get_saldo(self):
    return self.__saldo

def get_titular(self):
    return self.__titular

def get_limite(self):
    return self.__limite
```

5.2. Setters: basicamente nomenclatura dada para métodos que definem coisas, como por exemplo, um novo limite. Setters nunca retornam nada, apenas settam coisas novas. Ex.:

```
def set_limite(self, limite):
    self.__limite = limite
```

```
>>> conta.get_limite()
1000.0
>>> conta.set_limite(2000.0)
>>> conta.get_limite()
2000.0
```

5.3. Sempre usar getters e setters só quando forem necessários/realmente fazerem sentido.

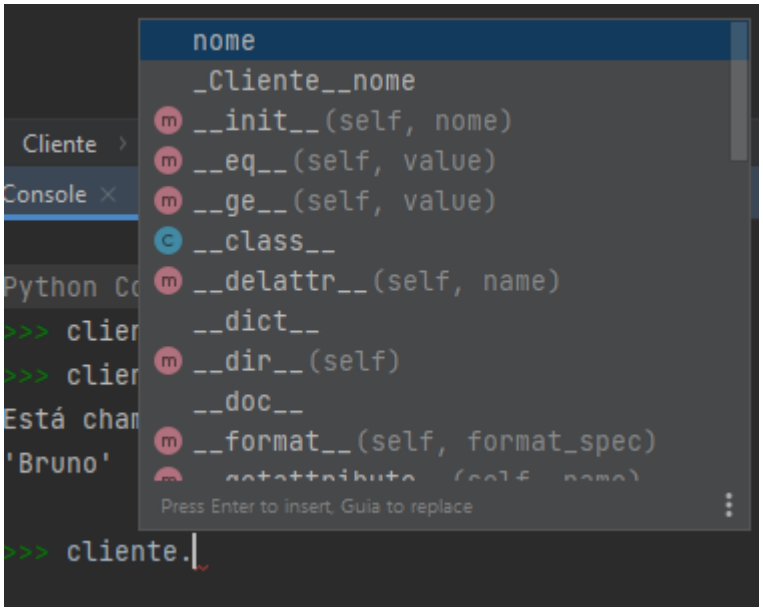
5.4. @property: Quando colocado antes de um método, mascara ele e faz com que se pareça com um atributo na hr de chamar no console. Ex.:

```
@property
def nome(self):
    print("Está chamando um getter")
    return self.__nome.title()
```

5.4.1. Usando isso podemos fazer com que ao invés de acessar o atributo direto, o usuário esteja tendo acesso à um método getter, mas sem o nome get_nome, que não precisa usar nem mesmo os () para ser executado. Ex. no console:

```
>>> cliente = Cliente("bruno")
>>> cliente.nome
Chamando um @property como se fosse um atributo que na verdade é um método getter
'Bruno'
```

5.4.2. Outra vantagem, como podemos ver, é que o nome já retorna tratado, ou seja, se na hora do cadastro você digitar o nome tudo em minúsculas, na hora de acessar o "atributo" (que na verdade é a nossa propriedade) ele retorna o nome tratado em *title()*.



5.4.3. Na imagem acima podemos ver a propriedade “nome” aparecendo em primeiro e o atributo real logo abaixo como “_Cliente__nome”, como sugestão do próprio python.

5.5. Podemos fazer a mesma coisa com os setters, mas utilizando uma sintax um pouco diferente:

```
@nome.setter
def nome(self, nome):
    print("Chamando um @property como se fosse um atributo que na verdade é um método setter")
    self.__nome = nome
```

5.5.1. Como podemos ver, ao invés de usar o @property, colocamos o nome que queremos dar para o método seguido de um “.” E a função que ele executará, sendo, nesse caso, um setter.

5.5.2. O resto do programa é basicamente o método que faz a alteração do que queremos, assim como foi feito no limite da conta lá em cima.

5.5.3. No console fica assim:

```
>>> cliente = Cliente("bruno")
>>> cliente.nome
Chamando um @property como se fosse um atributo que na verdade é um método getter
'Bruno'
>>> cliente.nome = "Adriano"
Chamando um @property como se fosse um atributo que na verdade é um método setter
>>> cliente.nome
Chamando um @property como se fosse um atributo que na verdade é um método getter
'Adriano'
```

5.5.4. Se quiser chamar o getter, basta dar enter ao colocar .propriedade, se quiser chamar o setter, faça uma nova atribuição de valor, como se estivesse fazendo acesso direto ao atributo.

5.6. O que aprendemos:

5.6.1. Métodos de leitura dos atributos, os getters;

5.6.2. Métodos de modifição dos atributos, os setters;

5.6.3. Propriedades.

6. Aula 6 – Métodos Privados e Estáticos:

6.1. Podemos criar condições usando métodos.

6.2. É possível retornar uma condição de um método direto (true/false), sem ter a necessidade de criar ela em uma variável antes do retorno.

6.3. Para indicar que um método é privado e que não foi feito para ser utilizado fora da classe, colocamos "__" antes do nome do método, assim como é feito com os atributos.

6.4. Métodos estáticos são os métodos da classe.

6.4.1. Digamos que a gente queira que o código do banco aparece mesma que não tenhamos criado uma conta, ao tentar ver qual é o código isso não funciona pois o python automaticamente vai criar esse objeto e ainda assim não retornará o código do banco, para isso temos que criar esses métodos estáticos.

6.4.2. @staticmethod: diz para o python que é um método estático. Usa da mesma maneira que os outros @'s.

```
@staticmethod
def codigo_banco():
    return "001"
```

6.4.3. No console:

```
>>> Conta.codigo_banco()
'001'
```

6.4.4. Lembrando que para isso precisamos chamar a classe Conta antes do ".".

6.5. Podemos retornar dicionários diretamente, sem a necessidade de criar uma variável/atributo para colocá-lo dentro:

```
@staticmethod
def codigos_bancos():
    return {'BB': '001', 'Caixa': '104', 'Bradesco': '237'}
```

6.6. Métodos estáticos também são chamados de métodos da classe, já que precisamos usar a classe como referência na hora de utilizar o método.

6.7. Quando métodos estáticos muito simples serão criados, como no caso de retornar um valor único, não precisamos criar nem sequer um método para ele, basta colocar um atributo dentro da classe e fazer ele receber o valor, desse modo, se tornando um método estático direto. Ex.:

```
class Circulo:

    PI = 3.14
```

6.8. No console:

```
Circulo.PI
3.14
```

6.9. O que aprendemos:

6.9.1. Métodos privados;

6.9.2. Métodos da classe, os métodos estáticos.

Curso 2 – Avançando na Orientação a Objetos:

1. Aula 1 – Relembrando Classes e Objetos:

1.1. Nada novo, apenas relembramos:

1.1.1. Criação da classe;

1.1.2. Definição de métodos assessores;

1.1.3. @property;

1.1.4. Name.

2. Aula 2 – Removendo Duplicação com Herança:

2.1. Utilizando o método de herança, com uma classe mais genérica, podemos fazer com que outras classes utilizem as mesmas características sem ficar copiando e colando códigos.

2.1.1. Temos como exemplo o projeto desse curso, onde estamos fazendo um código que exibe o nome, ano, likes, duração (filme) e temporadas (séries), para uma classe de filmes e uma classe de séries.

2.1.2. O código para essas duas classes se repete muito, distinguindo-se somente na questão de duração/temporada.

2.1.3. Levando isso em consideração, podemos desenvolver uma classe de "programa" (de TV), que unifica as características entre as duas, sendo elas nome, ano e likes.

2.1.4. Usando essa metodologia, fazemos com que o programa fique mais dinâmico, fácil de ler, coeso e melhor escrito, sendo essa uma boa prática da programação.

2.2. Para dizer que a classe Filme e Series são filhas da classe Programa, precisamos colocar *class Classe (Nome_Da_Classe_Mae): Código*. Ex.:

```
class Filme(Programa):
    def __init__(self, nome, ano, duracao):
        self.__nome = nome
        self.ano = ano
        self.duracao = duracao
        self.__likes = 0

class Series(Programa):
    def __init__(self, nome, ano, temporadas):
        self.__nome = nome.title()
        self.ano = ano
        self.temporadas = temporadas
        self.__likes = 0
```

2.2.1. Uma classe pode ter várias classes mães de onde herdam características.

2.3. Sempre herdamos todos os comportamentos da classe mãe quando fazemos isso, menos alguns.

2.3.1. Sempre que temos atributos privados na classe mãe, ele dará um erro nas classes filhas que tentam utilizar esse atributo, já que ele altera o nome do atributo.

2.3.2. No caso do __nome, na classe Programa, ficará _Programa__nome, para a classe filha, porém, isso é totalmente contra produtivo, pois teremos que alterar todos os atributos das filhas para o nome novo.

2.3.3. O python, porém, tem uma solução mais fácil para isso, sendo ela simplesmente deixar todos os atributos privados em todas as classes com apenas 1 "_" ao invés de 2 "__", evitando o erro.

2.3.4. Isso fará com que o atributo não fique "privado", mas ainda assim, através da nomenclatura com _, dirá para outros programadores que estejam mexendo no nosso código, que essa classe é privada e não deve ser alterada diretamente, servindo apenas como nomenclatura.

2.4. A classe mãe é chamada de super classe.

2.5. Super(): Faz referência a classe mãe.

2.5.1. Podemos juntar ele com o .__init__(atributos que queremos para a classe). Ex.:

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


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

2.5.2. Utilizando esse método evitamos bugs e erros.

2.5.3. É outro método que utilizamos para poder replicar menos códigos e reutilizar mais.

2.5.4. Ao invés de repetir o nome, likes e ano das duas classes, reutilizamos isso da super classe com o super().__init__(), passando no init os parâmetros que queremos utilizar na classe filha e deixamos apenas o que difere da super classe que é a geral.

2.5.5. Fazendo isso nós tiramos os pontos de falha que existem no nosso código até agora.

2.5.6. Ele pode também chamar qualquer método que está na classe mãe.

2.5.7. Colocar o super init depois do def init da classe filha, basicamente serve para sobrescrever o init da classe filha pelo da classe mãe, e tudo o que vier abaixo dele, será conteúdo novo para acrescentar nessa classe.

2.6. Métodos de classe:

2.6.1. 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}'
```

2.6.2. 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.

2.7. Métodos estáticos:

2.7.1. 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'
```

2.7.2. 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.

2.8. Cuidados a tomar:

2.8.1. 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.

2.8.2. 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.

2.9. O que aprendemos:

2.9.1. Herança;

2.9.2. Generalização/especialização;

2.9.3. Método super().

3. Aula 3 – Reduzindo ifs Com Polimorfismo:

3.1. Quando usamos herança, nós podemos dizer que as classes filhas são do mesmo tipo que as da mãe.

3.2. Uma das vantagens da herança e poder utilizar o polimorfismo.

3.2.1. Para a utilização dele, podemos criar uma variável que possui um if atribuído.

3.2.2. No caso do nosso programa, nós temos filmes com duração e séries com temporadas, porém, na hora de imprimir isso no for que será nossa playlist de exibição, dará erro.

3.2.3. Para corrigir isso utilizamos o polimorfismo da seguinte forma:

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

3.2.4. Criamos uma variável chamada detalhe e utilizamos um if atribuído à ela para definir o que será exibido, duração ou temporadas.

3.2.5. Basicamente o código está dizendo que: exibe a duração se o programa tiver o atributo *(hasattr(has atribute))* ‘duracao’, se não, exibe a temporada.

3.2.6. Note que o *programa* está em minúsculo, ou seja, se referindo a uma variável criada especificamente para esse for, e não à Classe Programa().

3.2.7. O for está dizendo que para cada programa, ou seja, item, dentro da variável *filmes_e_series*, imprima o nome dele, os detalhes (que especificamos e explicamos nos tópicos acima) e a quantidade de likes de cada programa.

3.2.8. Essa foi a variável criada:

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

3.2.9. E esses são os programas de tv criados:

```
vingadores = Filme('Vingadores - guerra infinita', 2018, 160)
atlanta = Series('Atlanta', 2018, 2)
```

3.2.10. Nome, ano e duração, no caso do filme, e temporadas, no caso da série.

3.3. Para economizar ifs, podemos evitar fazer tudo o que foi dito ali em cima (para imprimir uma coisa diferente para cada objeto) e simplesmente criar um novo método chamado imprime para a super classe, mandando-a imprimir cada objeto do jeito certo.

3.3.1. Isso é bom porque além de poupar trabalho a classe também se torna coesa, fazendo nada mais do que o que deve fazer.

3.3.2. Os objetos devem saber se imprimir, assim como uma string sabe se imprimir quando você manda ela imprimir.

3.3.3. Exemplo da classe imprime:

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

3.3.4. O problema é que ainda não temos a especificação de cada objeto, cada classe filha dessa super classe com o método imprime.

3.3.5. Para corrigir isso, cada uma das classes terá que ter seu próprio método imprime com suas especificações, assim:

3.3.5.1. No filme:

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

3.3.5.2. Na série:

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

3.3.6. Além disso também precisamos mudar o nosso *for*, falando para ele que, a cada programa em *filmes_e_series* imprima o método *imprime()* de cada objeto:

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

3.3.7. O que vai acontecer é que, como colocamos um método imprime para cada objeto, *Filme* e *Serie*, esse método irá sobrescrever o da classe mãe, exibindo o método *imprime()* específico do *Filme*, quando for a vez do filme ser impresso, e o método específico da *Serie*, quando for a ver dela, ficando assim como resultado final no console:

```
Vingadores - Guerra Infinita - 2018 - 160 min - 1
Atlanta - 2018 - 2 temp. - 2
```

3.3.8. Onde é exibido o nome, ano, duração/temporada (a depender do objeto exibido) e a quantidade de likes cada um possui.

3.3.9. Isso também é um método de polimorfismo reduzindo os *if's* e a complexidade do nosso código, executando o método *imprime()* de cada tipo sem olhar de quem ele está imprimindo.

3.4. Existem métodos especiais que o python utiliza, geralmente sendo aqueles que possuem"__" antes e depois de seus nomes.

3.4.1. Um exemplo é o __init__, onde o python, por convenção, já identifica que é o inicializador da classe.

3.4.2. Além desse método, também é importante termos em uma classe um método que represente ela textualmente. Não é obrigatório, mas muito recomendável para aquelas em que se aplica.

3.4.3. Já existe um método que nos ajuda com isso, chamado de *def __str__(self):*.

3.4.4. Quando esse método é implementado, você não pode simplesmente implementar um *print()* nele, o que se espera é que você retorne um valor/string daquele objeto que você quer representar.

3.4.5. O nosso método, anteriormente sendo o *imprime()*, agora está assim:

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

3.4.6. Isso é extremamente bom e importante, pois agora, ao invés de no nosso for pedirmos para que execute o método *imprime()* de cada *programa* (variável) que está na lista *filmes_e_series*, podemos simplesmente pedir para que ele imprima o programa. Desse modo:

```
vingadores = Filme('Vingadores - guerra infinita', 2018, 160)
atlanta = Series('Atlanta', 2018, 2)
```

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

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

3.4.7. O que acontece por debaixo dos panos é que quando pedimos para que ele imprima o programa, o python irá rodar o método interno de cada objeto responsável por exibir o conteúdo dele em forma textual.

3.4.8. O método interno, como já mostrado anteriormente:

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

3.4.9. Exibindo isso no console:

```
Vingadores - Guerra Infinita - 2018 - 160 min - 1
Atlanta - 2018 - 2 temp. - 2
```

3.4.10. Apesar de o conteúdo exibido e o comportamento ser teoricamente o mesmo, na prática o nosso código ficou muito melhor e mais *pythonico*, utilizando as funções, métodos, atributos e tudo mais de maneira correta, mais organizada e coesa do que antes, ficando cada vez mais profissional.

3.5. __repr__:

3.5.1. Assim como o __str__, existe outro método especial chamado __repr__, que é usado para mostrar uma representação que ajude no debug ou log de um código.

3.5.2. Por exemplo, se você quiser entender como funciona seu objeto, ou se está válido, e imprimir o seu valor string, qual resultado abaixo facilita sua vida?

> *Filme(nome='vingadores', ano=2018, duracao=160)*

> *"Filme: Vingadores de 2018 - 160 min"*

3.5.3. A ideia da primeira versão é remover ambiguidade e permite, por exemplo, recriar o objeto, já que está mostrando todas as informações.

3.5.4. Outro exemplo, se chamarmos o *repr* de um objeto do tipo *list*, podemos ter uma ideia do que é esperado quando criarmos o nosso próprio com *__repr__*:

lista = [1, 2, 4, 5]

print(repr(lista))

3.5.5. Se pegarmos o resultado do print, conseguimos recriar o objeto lista.

3.6. O que aprendemos:

3.6.1. Polimorfismo;

3.6.2. Relacionamento é um;

3.6.3. Representação textual de um objeto.

4. Aula 4 – Quando Não Usar Herança:

4.1. Para podermos usar um objeto no *for* como um *list*, precisamos torna-lo *interable*, ou seja, interativo.

4.1.1. Uma forma de fazer isso é fazer a nossa *Playlist* herdar a *(list)*.

4.1.2. Herdar o *list* nos trás várias vantagens e funcionalidades como saber o tamanho de uma lista, se existe algo específico dentro de uma lista ou não, como um filme ou série no caso da nossa playlist, dentre outras coisas.

4.1.3. Tudo isso já implementado dentro da *list* que estamos herdando:

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

4.1.4. Para não sobrescrever, apenas incrementar o nome no nosso inicializador, colocamos o *super().__init__(programas ~~sendo esse um nome qualquer para a list~~)*, dizendo assim que queremos todo o inicializador da classe mãe.

4.1.5. A classe *list* é um *built-in*, ou seja, uma classe que já vem implementada nativamente no python. A desvantagem é que não sabemos/conhecemos essa classe *list*, não sabemos o quão grande e/ou tudo o que ela possuiu.

4.1.6. O problema é que criamos complexidade no nosso código ao herdar a classe *list*. Além de que precisamos conhecer o nosso código/classe, obviamente.

4.2. Para evitar esses problemas, existe um jeito de conseguir todas as funcionalidades de um *list*, mas sem "converter" o nosso objeto/classe em um *list*.

4.3. O que aprendemos:

4.3.1. Herança de um tipo built-in (nativo);

4.3.2. Vantagens da herança de um iterável;

4.3.3. Desvantagem de fazer herança.

5. <u>Aula 5 – Duck Typing e um Modelo de Dados:</u>

5.1. Para que nossas classes sejam interaveis não é necessário que ela herde isso de outra classe, basta sabermos o que um iteravel faz e encapsular isso na nossa classe.

5.1.1. Def __getitem__(self, item): Método utilizado para nossa classe se tornar um iteravel.

5.1.2. Basta retornar o seu atributo lista com *item* dentro para que funcione:

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

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

5.1.3. Além disso também ganhamos as funcionalidades *for in* e *in* de um *list*, ou seja, podemos criar loops sem utilizar um método para isso e

também podemos verificar se determinado item existe na nossa *list* ou não. Ex.:

```
print(vingadores in playlist_fim_de_semana)

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

5.1.4. No console:

```
Tamanho da minha playlist: 4
True
Vingadores - Guerra Infinita - 2018 - 160 min - 2
Atlanta - 2018 - 2 temp. - 3
Demolidor - 2016 - 2 temp. - 2
Todo Mundo Em Pânico - 1999 - 100 min - 4
```

5.1.5. O *"true"* está dizendo que *"sim, vingadores está na playlist_fim_de_semana"*.

5.1.6. Também podemos pedir para que print qual é determinado item na lista, que ele irá funcionar.

5.2. Esse método de trazer o comportamento de um iteravel para nossa classe fazendo com que ela se comporte como tal, se chama duck typing.

5.2.1. Isso porque o python não se importa com tipagem de classes, para ele só importa se a classe tá fazendo o que precisa, o resto não interessa.

5.2.2. Basicamente: se existe uma ave que faz quack, anda como pato, voa como pato, não me importa qual ave é aquela, pois, como se comporta como pato, pra mim é um pato.

5.3. __len__(self): Outro "método mágico" para utilizarmos. Um comportamento interno da nossa classe que retorna o tamanho da nossa lista, assim podemos utilizar o len(lista).

5.3.1. Ao invés de fazer isso:

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

5.3.2. Fazemos isso:

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

5.3.3. Dessa forma podemos utilizar o len() normalmente no nosso código, assim:

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

5.3.4. Ao invés de ter que colocar o .listagem, como estava antes:

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

5.4. Métodos existentes e suas funcionalidades:

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

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

5.5. O que aprendemos:

5.5.1. Duck typing;

5.5.2. Python data (object) model;

5.5.3. Dunder methods;

5.5.4. Uso de ABCs.

6. Aula 6 – Herança Múltipla:

6.1. Para herdar mais de uma classe, basta colocar "," e a próxima. Ex.:

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

6.1.1. Desse modo, essa nova classe pode utilizar tanto os métodos da Alura quanto da Caelum, enquanto um que herde apenas 1, só poderá usar os métodos dele.

6.1.2. Apesar de múltiplas heranças ser muito bom e facilitar vários processos, também cria complexidade para o nosso código, portanto, tenha cuidado ao usar.

6.2. Existe uma ordem de verificação quando se pede para executar um método.

6.2.1. Primeiramente o python procura na própria classe que pediu o método, em seguida na classe mãe.

6.2.2. Se a classe filha tiver mais de uma classe mãe, a ordem, geralmente, é da esquerda para a direita.

6.2.3. Mas além disso, o que acontece é que ele não vê só nas classes mãe da filha que está pedindo o método, ele irá ver na primeira classe mãe, da esquerda pra direita, aí nas classes mães dessa classe mãe e assim por diante até não ter mais onde ver, só depois disso tudo que ele irá verificar na próxima classe mãe da filha. Ex.:

6.2.3.1. Nossa classe filha que procura o método é a Pleno(), a classe mãe dela é a Alura() e a Caelum(), a classe mãe, tanto da Alura() quanto da Caelum(), é a Funcionario().

6.2.3.2. Basicamente ficando assim:

6.2.3.3. Pleno(Alura, Caelum) > Alura(Funcionario) > Funcionario() > Caelum(Funcionario) > Funcionario().

6.3. Existe, porém, outro meio de verificação de classes em busca do método...

6.3.1. O que acontece é que, antes de verificar a classe mãe da classe mãe, ou seja, Funcionario(), logo em seguida de Alura(), é a verificação se a próxima classe mãe de Pleno(), ou seja, Caelum(), possui a mesma classe mãe que Alura() ou não.

6.3.2. Se Ambas tiverem a mesma classe mãe, python altera a sequência, fazendo com que a verificação, ao invés de acontecer como no tópico acima, fique assim:

6.3.3. Pleno(Alura, Caelum) > Alura(Funcionario) > Caelum(Funcionario) > Funcionario().

6.3.4. Isso ocorre porque ele entende que alí não é o lugar dela ainda e pode ter outra classe, na sequência, que se encaixe melhor, deixando a classe mãe em comum das duas por último.

6.3.5. Chamamos isso de good head, ou seja, uma classe de “cabeça boa”, que pode se encaixar melhor antes de fazer a verificação da classe mãe comum entre elas.

6.4. Mixin: Classes que não são instanciadas e bem pequenas usadas para fazer o compartilhamento de um comportamento que não é o mais importante de uma classe.

6.4.1. É um tipo de herança muito útil, onde podemos utilizar um comportamento em várias outras classes sem instanciar ele.

6.5. O que aprendemos:

6.5.1. Herança múltipla;

6.5.2. Resolução da ordem de chamada de métodos;

6.5.3. Mixins.