FGV EMAp

Linguagens de Programação - 2025.2

Resumo para a A2

Ana Beatriz Botacim Rodrigues

25/11/2025

Contents

1 Pandas

1.1 DataFrames

• **pd.DataFrame()** é o objeto construtor da classe DataFrame, que cria um df a partir do que passarmos, ele recebe qualquer iterável como parâmetro (listas, por exemplo). Podemos já definir os nomes das colunas fazendo **pd.DataFrame("dates" : dates, "temp" : temp)**.

• Sendo df o nome do dataframe, para selecionar uma coluna chamada nome, por exemplo, basta fazer **df["nome"]**. É possível selecionar mais de uma coluna fazendo **df[["nome", "idade"]]**.

• Para selecionar linhas, há algumas formas: se a linha tem nome, por exemplo, Ana, podemos fazer **df.loc["Ana"]** ou usar o índice implícito, **df.iloc[2]**, se "Ana" estiver na terceira linha.

• Por fim, para selecionar células específicas, linhas e colunas desejadas, basta **df.loc["Ana", "idade"]**, tal que o primeiro parâmetro sempre é a linha e o segundo sempre é a coluna. Conseguimos fazer isso usando também **df.iloc[3,1]**. Além destes dois, é possível usar **df.at["Ana", "idade"]**, que acaba sendo mais rápido. Se quisermos pegar mais de uma linha/coluna, basta passar listas ou slices como primeiro e/ou segundo argumento posicional nos comandos citados acima.

• **df.head(n)** devolve as n primeiras linhas de df, ao passo que **df.tail(n)** devolve as n últimas.

• Além disso, podemos fazer **df["nome"].unique()** para retornar os valores únicos da coluna selecionada, **df["nome"].nunique()** para contagem de valores únicos, **df["nome"].count()** para contagem de nomes e **df["nome"].value_counts()** para contagem de cada valor de nome.

• É possível usar **df["idade"].min()** para pegar a menor idade, **df["idade"].max()** para pegar a maior idade, **df["idade"].mean()** para a média das idades.

• **df.columns** retorna o nome das colunas e **df.index** o índice das linhas. Uma nova coluna pode ser facilmente adicionada usando **df["nova coluna"] = serie**. O **df.drop("coluna", axis = 1, inplace = True)** remove colunas (axis = 1), ou seja, percorre coluna a coluna[1]. O padrão de drop é axis = 0.

• É possível criar uma série comparando colunas, por exemplo **df["idade"] > 18**. Ainda, é fácil colocar isso num dataframe **df[df["idade"] > 18]**. Conseguimos colocar mais de uma condição usando o operador "e": **&** ou o operador "ou": |.

• Ainda, a função **df.isnull()** retorna o df com valores booleanos. Podemos usar comparações desse tipo para criar df's menores. A **df.dropna(thresh = 2, inplace = True)** apaga as linhas com menos de 2 valores não nulos. Por fim, **df.fillna(15, inplace = True)** completa as células nulas com o valor 15.

• Outro comando importante para pandas é o **pd.read_csv("arqv.csv", sep = ".")**, onde sep é o separador utilizado no arquivo. Podemos salvar os dados, em csv, por exemplo com **df.to_csv("dados.csv", encoding = "utf-8")**.

• Para converter uma coluna pro tipo data, basta fazer **df["data"] = df.to_datetime(df["data"])**

• O método **df.dtypes** mostra o datatype de cada coluna. Enquando a função **df.set_index("coluna")** define qual coluna vai ser usada como índice para o df. Assim, **df.sort_index()** ordena o dataframe com base na coluna de índice.

• **df.describe()** retorna um resumo dos dados do dataframe, como média de cada coluna, valor máximo, mínimo, desvio padrão.

• **df.groupby("coluna")["valor"].sum()** agrupa os valores da coluna "coluna" e soma os valores de "valor" de cada grupo. É possível aplicar mais de uma função: **df.groupby("categoria")["vendas"].agg(["sum", "mean", "max"])** ou agrupar por índice hierárquico: **df.groupby(["ano", "mes"])["vendas"].sum()** → cria níveis hierárquicos.

1.2 Gráficos

• Podemos fazer gráficos usando o pandas, **df.plot(x = "ano", y = "compras", title = "img")** é um exemplo de uso. Há alguns comandos importantes, suponha que ax seja o nome do plot. **ax.set_xlabel("ano")** define a legenda do eixo x, o eixo y é análogo[2].

• Podemos ainda usar **df.plot(kind = "bar", title = "grfc de barras")** para plotar gráfico de barras, o parâmetro kind define o tipo: "pie" → pizza; "hist" → histograma; "barh" → barras horizontais; "box" → gráfico de caixa (boxplot); "scatter" → gráfico de dispersão. Precisar usar o MatPlotLib.PyPlot para mostrar a figura, por meio de **plt.show()**.

[1] axis = 0 faz o contrário, remove linhas.
[2] Em vez de fazer assim, podemos definir as labels no momento de criação do plot.

• **pd.concat([df, df_2], ignore_index = True)** adiciona ao final de df a linhas de df_2, o ignore_index garante que os índices não serão repetidos: o concat cria novos.

2 NumPy

• A parte de NumPy que vimos na segunda parte curso (além dos arrays) foi o random. Quando vamos criar dados aleatórios, é uma boa prática fixar uma semente **np.random.seed(3)**, por exemplo, para que o resultado do código seja igual mesmo que executado outras vezes.

• **np.random.choice(list, size, p)** é uma ferramenta para criar uma série com size elementos de list aleatórios, de forma que cada um tem p chance de ser sorteado (p é uma lista com números de 0 a 1 que devem somar 1).

3 Orientação a Objetos

• Objetos têm estados, que são valores/características e comportamentos, que são os métodos.

• Uma classe diz para a linguagem de programação como criar um objeto, tendo como base determinados comportamentos e atributos

• Classes precisam de docstrings, que podem ter até doctest.

• O **init** é o primeiro método que criamos, ele é chamado automaticamente quando chamamos a classe. O init não cria um objeto, ele apenas inicializa com as características que definimos!

• Quando fazemos, por exemplo, **self.owner = owner** é verificado se o próprio objeto que chamou o atributo owner (self) tem um atributo owner: caso positivo, ele é sobrescrito com o valor passado, senão, ele é criado com o que disponibilizamos.

• Se fizemos, por exemplo, dentro da classe professor algo como **self.profissao = "Professor"**, todo objeto dessa classe terá o atributo profissão com o valor "Professor" associado.

• Ainda que não definido publicamente, toda classe tem um pai "por debaixo dos panos", da qual ela herda vários métodos, como __str__, __dict__: veja **dir(objeto)**.

• Um método é algo como **self.add_account()** e uma propriedade é **self.accounts**.

• Quando criamos uma classe que dentro dela usa outra classe (por exemplo, o objeto do tipo Guerreiro tem um espaço no qual faz referência a um outro objeto do tipo arma) aumentamos o **acomplamento** entre as classes, e quanto mais o acoplamento aumenta, pior é pra resolver bugs.

• Atributos de classe são aqueles compartilhados entre todos os objetos da classe, são definidos fora do escopo de qualquer função, mas dentro de determinada classe. Para usá-lo no init fazemos algo como Classe.atributo = "Atributo de classe"

• Quando queremos usar atributos de classe, criamos um **@classmethod**, que não recebe self como parâmetro, mas sim a classe: **cls**.

• Seja Fruta uma classe e banana um objeto da mesma. **banana.envelhecer()** é o mesmo que fazer **Fruta.envelhecer(banana)**

• O init coloca tudo o que passamos na criação de um objeto dentro de seu dicionário: **.__dict__**

• Podemos criar novos atributos para objetos fazendo **professor.poliglota = True** porque é como se estivessémos colocando isso no dicionário do objeto. Todavia, os outros objetos da mesma classe não terão os mesmos atributos.

• Se eu não colocar init numa função, ou qualquer outro dos seus métodos, e chamarmos esse em específico, será usado o método de seu pai - caso existir - ou de algum de seus descendentes. Caso contrário, dará erro.

• Sempre que estamos dentro de uma representação (**.__repr__** de um objeto imprimindo atributos ou outros objetos (self.raca, self.nome, ...) usamos sua representação por meio do !r. É importante que a string é para o usuário ler, internamente, o python trabalha com a representação entre os programas.

•O **__new__** olha para a instância de uma classe, ele controla o "momento de concepção" de um objeto, o new chama quem cria o objeto A lógica é: faço o que é só do meu new e chamo o new do meu pai, e isso segue em cadeia até que seja chamado o new de object (todos precisam ser objetos)

• O **new** CRIA, o **init** INICIALIZA os objetos!!!

• Uma classe asbtrata é uma classe que não pode ser instanciada. A única maneira de fazer isso em python é usando o ABC (Abstract Base Class) do módulo abc, daí, todos os filhos herdarão isto.

• Usamos uma classe abstrata quando as propriedades de uma classe permitem que ela assuma estados (sacável, depositável, ...). Quando queremos definir como estados e comportamentos de um objeto vão ser dados, definimos uma classe asbtrata. Ao criar uma Classe abstrata, definimos estados (variaveis) e comportamentos para todos os filhos. A classe abstrata teoricamente é abstrata ao herdar de ABC, mas o python só reconhece como abstrata se tem um método abstrato. Queremos uma classe abstrata quando pensamos "não quero que esta classe seja instanciada pq não faz sentido".

• **@abstractmethod** é um método que não tem corpo, ele está aqui para que os filhos dele implementem. Aquele que implementar, será o primeiro filho concreto da classe Abstrata, os que não implementam não podem ser instanciados. Uma classe abstrata pode ter métodos concretos: um método abstrato a torna abstrata

• MOCK/STUB é um método que não serve pra nada, é um cotoco no seu código. Se isso existe no seu código, você sabe que sua modelagem OO está ruim.

• Usamos a classe **Utils** para juntar métodos importantes, ela não tem init nem self, porque só serve para juntar funções que ficariam jogadas pelo código.

3.1 Polimorfismo

Polimorfismo: "irmãos", que, em tese fazem a mesma coisa mas de formas diferentes Por exemplo, duas classes filhas de um mesmo pai que tem a mesma ação mas pensam diferente na hora de fazê-la: um tipo de conta libera crédito até que ele acabe e a outra só libera se tiver crédito disponível.

Filhos diferentes de uma mesma classe se comportam de maneira diferente no código, mas conseguimos usar métodos unificados para todas elas: por exemplo os métodos de BankUtils funcionam para todas elas, mesmo sendo diferentes.

3.2 Sobrecarga

Há dois tipos de sobrecarga: de operadores e de métodos herdados.

Sobrecarregar um método é refazer o seu comportamento, por exemplo: toda classe herda o método __**str**__, mas normalmente mudamos seu retorno para algo personalizado.

Podemos, por exemplo, definir a soma entre dois objetos por meio da sobrecarga de __**add**__.

Alguns operadores que podem ser sobrecarregados (ter outro significado dentro da nossa classe):

1. __lt__: lesser than
2. __le__: lesser qual
3. __gt__: greather than
4. __ge__: greather equal
5. __eq__: equal
6. __ne__: not equal
7. __add__: addition
8. __sub__: subtraction
9. __mul__: multiplication
10. __floordiv__: floor of division
11. __truediv__: divisão de ponto flutuante
12. __neg(other)__: negative

e muitos mais!

3.3 Associação

Dois objetos estão associados se eles se relacionam de alguma forma no código do programa.

Suponha classes Guerreiro, Feiticeiro e GrupoAventureiros, tais que Guerreiro e Feiticeiro não se relacionam diretamente mas fazem parte de um GrupoAventureiros: Guerreiro e Feiticeiro não têm exatamente nenhuma associação, até que se crie um grupo que os contenha, a partir daí, eles estão fracamente associados, e (grupo - feiticeiro) e (grupo - guerreiro) estão agregados.

Na associação, não há dependência, eles se encontram no código mas não são dependentes um do outro.

3.4 Composição

Na composição, quando um objeto é deletado, os outros criados a partir dele também o são. Isso é o oposto de agregação.

3.5 Agregação

Os objetos são independentes nesse caso: um vive sem o outro. A ideia de agregação é fazer de forma que se um objeto morre, ninguém depende dele então todo mundo continua ok.

Os objetos se relacionam mas não são dependentes!

3.6 Encapsulamento

Encapsulamento é proteger nossos atributos. Usar um underscore no nome de um atributo (**self._balance**, por exemplo) significa que o atributo é protegido e só deve ser modificado dentro da classe (e não no driver code) as classes filhas herdam. Apesar de mutável (em python), chamamos balance de membro de dados PROTEGIDO. Em outras linguagens de programação, apenas a classe em que um membro de dados privado foi criado pode mudá-lo. Se usamos dunder (**self.__cpf**, por exemplo), criamos uma camada de dificuldade mas ainda é possível mudar o atributo de fora da classe (em python), ele se torna privado, as classes filhas não herdam.

Não deveria ser possível modificar métodos protegidos ou privados, mas o python é permissivo, ou seja, o underscore é apenas uma convenção. Em linguagens como Java e C++, de fato não é possível modificar variáveis protegidas.

Para acessar um dado privado/protegido, criamos um ponto de acesso (método) público para acessá-lo.

Quando fazemos algo como **self.created_at**, por exemplo, chamamos de membro de dados público.

Usar um método no filho para acessar protegidos do pai não é erro semântico (se fosse método privado, seria, porque na teoria isso é errado) porque eu escolho como e quando o nosso objeto filho pode usar os métodos do pai Controlamos o que é de dentro mostrando apenas o que queremos. Essa é a ideia do getter e do setter.

---

self.public = "Public": a mensagem desse atributo é que ele pode ser modificado a vontade através da instância.

self._protected = "Protected": a mensagem desse é que ele não deve ser acessado, mas, se alguém quiser, é facil fazê-lo. Todas as classes filhas herdam esse atributo.

self.__private = "Private": também consegue ser acessado, mas tem um grau a mais de dificuldade. Este não é herdado pelas classes filhas[3].

O mesmo pode ser usado para definir métodos (funções), com os exatos mesmos significados.

3.6.1 Getter e Setter

Quando usamos o Encapsulamento, privamos nossas variáveis de alteração externa, mas precisamos de algum meio de acessá-las: tanto para visualizar quanto para modificar. O getter, normalmente com mesmo nome do atributo que desejamos acessar (sem _), é decorado com **@property**, de forma que ao chamarmos, por exemplo, **self.balance** nos retorne o valor encapsulado. Ainda, para modificar, criamos o setter, este, é decorado com algo como **@balance.setter** e seu uso é algo como **self.balance = self.balance + dif**: a primeira chamada de **self.balance** é o setter, pois recebe algo a mais que apenas o self. Já a segunda é o getter, tendo em vista que a unica coisa que recebe como parâmetro é o self.

Note que em python é possível alterar um atributo fazendo **self._balance = amount**, mas estamos trabalhando com o encapsulamento, e mesmo que seja permitido, faremos como se não fosse.

3.7 SRP: Princípio de Responsabilidade Única

O princípio de responsabilidade única diz que cada método deve ter o mínimo de responsabilidades para si, então, por exemplo, métodos que retornam strings explicando o que foi feito, devem retornar apenas o resultado gerado na sua execução e a string vai ser trabalho de outro método.

3.8 OCP: Open Closed Principle

Quando você faz uma classe bem feita, você tem que permitir que as pessoas consigam adicionar comportamento aos objetos. Ainda, uma classe tem que estar fechada para modificação, não podemos usar uma subclasse para afetar uma superclasse, forçando ela a mudar, por exemplo.

[3] No python podemos acessar usando filho.Pai.__private

No momento em que criamos um método que nenhum dos irmãos tem, significa que a classe pai é aberta para extensão. Se não foi necessário modificar a classe pai para isso, ela é fechada para modificação e segue o OCP!

3.9 LSP: Princípio de Substituição de Liskov

Em qualquer programa, devemos conseguir substituir um objeto da classe pai por um objeto da classe filho sem problemas. Se no seu programa um objeto de conta genérico ao se tornar um objeto de conta poupança quebra o programa, você quebrou o princípio e essa não é uma boa modelagem de OO. O objeto filho também tem que poder ser substituído pelo pai.

Se uma função recebe mais argumentos no filho do que no pai, por exemplo, o LSP é quberado.

3.10 ISP: Princípio de Segregação das Interfaces

Quando temos uma interface enorme, temos vários métodos que seremos obrigados a implementar, segregando, teremos vários derivados que poderemos escolher.

Veja o exemplo: Toda a vez que eu crio uma classe de conta nova no banco, eu defino que coisas essa classe terá. Dizemos que a classe será "sacável", "depositável", etc. Quando as "classes" viram adjetivos, chamamos de INTERFACE (como o depositable, withdrawable, etc)