Curso DSA – Análise de dados

Na pasta dos arquivos...

jupyter notebook

Interface gráfica do usuário, Texto, Aplicativo, Email

O conteúdo gerado por IA pode estar incorreto.

Interface gráfica do usuário, Texto, Aplicativo, Carta, Email

O conteúdo gerado por IA pode estar incorreto.

def buble\_sort(arr):

n = len(arr)

# para cada elemento i do array

for i in range(n):

# para cada elemento j do array

for j in range(0,n-i-1):

# se elemento i for maior que elemento j

if arr[j] > arr[j+1]:

# troque os elementos i e j

arr[j], arr[j+1] = arr[j+1], arr[j]

return arr

lista1=[10,2,3,5,18,32,154,52,1,7,58,30]

print (buble\_sort(lista1))

**Strings**

Lembre que Python atribui a string e indexa seus componentes... portanto, vc pode utilizar partes desta string imprimindo o índice...

Strings são objetos IMUTÁVEIS, logo não é possível alterar partes da string

A = ‘casa’

A[0]=’c’

A[i:f] 🡪 notação de fatiamento (slice) i=primeira posição f=última posição exclusiva (não conta este)

A[::n] 🡪 salto de posição n=valor do salto

Possível utilizar operações com strings com operadores aritméticos.

**Funções built-in de strings**

São acessados pelo “.” após a string pressionando a tecla <tab>, para o caso do jupyter

**Estrutura de dados**

**Lista** em python são limitadas por chaves [], sendo um objeto mutável.

Listas aninhadas são lista dentro de listas.

A=((1,2),(2,5))

B=A[0][0] 🡪 atribui a B o valor do item 0 da lista que está na posição 0 que é o valor (1)

**Dicionários** em python

Dicionários são criados com parêntesis () e dois pontos “:” separando o item dentro da lista

Isto possibilita a dependência dos dados.

Estudante = [“Livia”:21, “Marcelo”:13, “Pedro”:35]

Possível usar dicionários com listas...

## acesso de itens específicos

dict1 = {'chave':1230, 'chave2':[22,453,73.4], 'chave3':['picanha', 'fraldinha', 'alcatra']}

dict1['chave3'][0].upper()

var1 = dict1['chave2'][0]-2

var1

20

dict1['chave2'][0] -= 2 # fazendo várias operações com pouco código

dict1

{'chave': 1230,

'chave2': [20, 453, 73.4],

'chave3': ['picanha', 'fraldinha', 'alcatra']}

Possível aninhar dicionários...

**Tuplas** em python

Tuplas são feitas com atribuição de parênteses ()

Tupla1=(“teste”, “teste1”)

A tupla é uma estrutura imutável.

**List Comprehension** é uma maneira concisa e eficiente de criar listas em Python. Ela substitui o uso de loops for tradicionais, tornando o código mais curto e legível.

**Sintaxe Básica**

nova\_lista = [expressão for item in iterável if condição]

* **expressão** → O valor a ser adicionado à nova lista.
* **iterável** → Qualquer estrutura iterável (lista, tupla, string, etc.).
* **condição (opcional)** → Filtra os itens antes de adicioná-los à nova lista.

**Exemplos**

1️⃣ Criando uma lista com números ao quadrado:

numeros = [1, 2, 3, 4, 5]

quadrados = [x\*\*2 for x in numeros]

print(quadrados) # Saída: [1, 4, 9, 16, 25]

2️⃣ Filtrando números pares:

numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

pares = [x for x in numeros if x % 2 == 0]

print(pares) # Saída: [2, 4, 6, 8, 10]

3️⃣ Criando uma lista de palavras com mais de 3 letras:

palavras = ["oi", "python", "é", "legal"]

longas = [palavra for palavra in palavras if len(palavra) > 3]

print(longas) # Saída: ['python', 'legal']

4️⃣ Transformando strings:

frase = "Olá mundo"

letras\_maiusculas = [letra.upper() for letra in frase]

print(letras\_maiusculas) # Saída: ['O', 'L', 'Á', ' ', 'M', 'U', 'N', 'D', 'O']

5️⃣ **Usando if e else dentro da List Comprehension**:

numeros = [1, 2, 3, 4, 5]

par\_ou\_impar = ["par" if x % 2 == 0 else "ímpar" for x in numeros]

print(par\_ou\_impar) # Saída: ['ímpar', 'par', 'ímpar', 'par', 'ímpar']

**Vantagens do List Comprehension**

✅ Código mais curto e legível.  
✅ Melhor performance do que for loops convencionais.  
✅ Menos linhas de código, mantendo a clareza.

Se precisar de mais exemplos ou explicação, só perguntar! 🚀

Ótima observação! Isso acontece porque a variável x, usada dentro da **list comprehension**, **só existe dentro do seu escopo** e **não é acessível fora dela**.

Vamos analisar isso em detalhes:

**Exemplo com for comum (funciona normalmente)**

numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

pares = []

for x in numeros:

if x % 2 == 0:

pares.append(x)

print(x) # Isso funciona! Exibe o último valor de `x`

🔹 Como o for está no **escopo global**, a variável x ainda existe depois da execução do loop.

**Agora, com list comprehension (dá erro)**

numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

pares = [x for x in numeros if x % 2 == 0]

print(x) # Isso dá erro! NameError: name 'x' is not defined

🔹 **Motivo:**  
A variável x está **dentro do escopo da list comprehension**. Assim que a list comprehension termina, o Python **descarta** essa variável.

**Por que isso acontece?**

* O Python trata a list comprehension como uma **expressão isolada**.
* As variáveis usadas nela **não vazam para o escopo global**.
* Isso evita conflitos inesperados e melhora a eficiência.

**Como contornar isso?**

Se precisar acessar os valores individuais fora da list comprehension, pode usar um loop for comum:

numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

for x in numeros:

if x % 2 == 0:

print(x) # Aqui funciona normalmente

Ou, se quiser acessar todos os valores de uma vez, basta imprimir a própria lista:

print([x for x in numeros if x % 2 == 0])

Espero que isso esclareça! 🚀

Ótima reflexão! De fato, **list comprehensions** podem ser um desafio no começo, mas depois que você pega o jeito, elas aceleram bastante a escrita e leitura do código. 🚀

Agora, falando sobre **performance em programas grandes**, **list comprehension geralmente é mais rápida que um loop for tradicional**. Mas por quê? 🤔

**1️⃣ Performance: List Comprehension vs. Loop for**

O motivo pelo qual a **list comprehension é mais rápida** tem a ver com a maneira como o Python executa o código:

* **Loop for tradicional** → Executado linha por linha, interpretado a cada iteração.
* **List comprehension** → Executada internamente por funções otimizadas em **C**, sem precisar interagir com o interpretador Python a cada passo.

**Vamos testar na prática!** 🏎💨

**Testando a velocidade no Python**

Vamos medir o tempo de execução para criar uma lista com 10 milhões de números ao quadrado:

**🔴 Usando for tradicional**

import time

inicio = time.time()

numeros = []

for x in range(10\_000\_000):

numeros.append(x\*\*2)

fim = time.time()

print(f"Tempo com for: {fim - inicio:.4f} segundos")

**🟢 Usando List Comprehension**

inicio = time.time()

numeros = [x\*\*2 for x in range(10\_000\_000)]

fim = time.time()

print(f"Tempo com list comprehension: {fim - inicio:.4f} segundos")

🔹 **Resultado esperado**: A list comprehension geralmente é **2 a 5 vezes mais rápida**! 🚀

**2️⃣ Quando List Comprehension pode ser mais lenta?**

Embora seja geralmente mais rápida, a **list comprehension pode ser ineficiente** em alguns casos:

1. **Quando a lista gerada é muito grande** → A memória pode estourar, já que a lista precisa ser armazenada inteira de uma vez.
2. **Quando há muita lógica complexa** → Se a condição dentro da list comprehension for complicada, pode ficar **menos legível e até mais lenta** do que um loop tradicional.
3. **Quando não precisamos de uma lista** → Se a intenção é apenas iterar sobre os elementos sem armazená-los, um **generator** (yield ou () no lugar de []) pode ser mais eficiente.

**3️⃣ Alternativa ainda mais eficiente: Generator Expression**

Se você precisa apenas **iterar sobre os dados sem armazená-los na memória**, uma **generator expression** pode ser a solução mais eficiente:

numeros = (x\*\*2 for x in range(10\_000\_000))

📌 A diferença aqui é que os valores são **gerados sob demanda**, economizando MUITA memória! 💡

**Conclusão**

✅ **Para grandes quantidades de dados, list comprehension é mais rápida que um for, pois é executada internamente em C.**  
✅ **Se precisar economizar memória, considere usar um generator expression em vez de uma list comprehension.**  
✅ **Se o código ficar muito complexo dentro da list comprehension, pode ser melhor usar um for para legibilidade.**

🚀 Ou seja: **List comprehension é um foguete, mas só vale a pena se você souber quando usá-lo!**

Ótima ideia! Se você já pegou o jeito do **list comprehension**, vai ver que o **dict comprehension** segue a mesma lógica, mas aplicado a **dicionários** (dict). Ele permite criar dicionários de forma mais rápida e eficiente. 🚀

**🔹 Sintaxe do Dict Comprehension**

A estrutura básica é:

novo\_dicionario = {chave: valor for item in iterável if condição}

* **chave** → A chave do dicionário.
* **valor** → O valor correspondente à chave.
* **iterável** → Lista, dicionário, range, etc.
* **condição (opcional)** → Filtra quais itens serão incluídos.

**🔹 Exemplos Práticos**

**1️⃣ Criando um dicionário a partir de uma lista**

Se quisermos criar um dicionário onde cada número é a chave e seu quadrado é o valor:

numeros = [1, 2, 3, 4, 5]

quadrados = {x: x\*\*2 for x in numeros}

print(quadrados)

# Saída: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

**2️⃣ Filtrando dados**

Criando um dicionário **apenas com números pares**:

numeros = range(10)

pares = {x: x\*\*2 for x in numeros if x % 2 == 0}

print(pares)

# Saída: {0: 0, 2: 4, 4: 16, 6: 36, 8: 64}

**3️⃣ Invertendo chaves e valores de um dicionário**

original = {'a': 1, 'b': 2, 'c': 3}

invertido = {valor: chave for chave, valor in original.items()}

print(invertido)

# Saída: {1: 'a', 2: 'b', 3: 'c'}

**4️⃣ Criando um dicionário a partir de duas listas**

chaves = ['nome', 'idade', 'cidade']

valores = ['Alice', 25, 'São Paulo']

dicionario = {chaves[i]: valores[i] for i in range(len(chaves))}

print(dicionario)

# Saída: {'nome': 'Alice', 'idade': 25, 'cidade': 'São Paulo'}

**5️⃣ Usando if e else dentro do Dict Comprehension**

numeros = range(5)

status = {x: "par" if x % 2 == 0 else "ímpar" for x in numeros}

print(status)

# Saída: {0: 'par', 1: 'ímpar', 2: 'par', 3: 'ímpar', 4: 'par'}

**🔹 Vantagens do Dict Comprehension**

✅ **Mais rápido** que loops for tradicionais.  
✅ **Código mais compacto e legível.**  
✅ **Evita a necessidade de usar dict.update() ou loops for externos.**

**🔹 Cuidados ao Usar Dict Comprehension**

⚠ **Se o código ficar muito complexo, pode ser melhor usar um for tradicional** para melhorar a legibilidade.  
⚠ **Chaves duplicadas serão sobrescritas**, já que um dicionário não pode ter chaves repetidas.

**🔹 Conclusão**

O **dict comprehension** segue a mesma ideia do **list comprehension**, mas aplicado a dicionários. Ele é uma ferramenta muito útil para criar, transformar e filtrar dicionários de forma mais eficiente e elegante.

Curtiu? Se quiser mais exemplos ou tiver dúvidas, só perguntar! 🚀

# construindo uma calculadora melhor...

'''Selecione o número da operação desejada:

1 - Soma

2 - Subtração

3 - Multiplicação

4 - Divisão

Digite sua opção (1/2/3/4): 1

Digite o primeiro número: 23

Digite o segundo número: 45

23 + 45 = 68'''

Map () e a iteração... (carregamento em memória é iterado)

Perfeito! Sim, o map() retorna **um iterador** também — ele **aplica uma função a cada item de um iterável**, **preguiçosamente** (ou seja, só calcula quando você usa next() ou itera com for).

**🧪 Exemplo que confirma isso:**

python

CopiarEditar

nums = [1, 2, 3] dobros = map(lambda x: x \* 2, nums) print(dobros) *# <map object at 0x...> → é um iterador* print(next(dobros)) *# 2* print(next(dobros)) *# 4* print(next(dobros)) *# 6*

Se você tentar next(dobros) depois disso, dará erro (StopIteration), pois o iterador **não reinicia** — exatamente como explicamos.

Fico feliz que tenha entendido!  
Se quiser aprofundar no map(), filter(), zip() ou **funções lambda**, só dizer que continuo aqui. 🚀

**Você disse:**

interessante... por isso o professor usou um for para demonstrar... Quando vc fala "preguiçosamente" vc quer dizer que o iterável somente se mostra item a item, mas existem iteráveis que não sejam preguiçosos?

**O ChatGPT disse:**

Exatamente! Seu raciocínio está **muito bom** — e sua dúvida vai direto ao ponto das diferenças entre **iteráveis “preguiçosos” (lazy)** e **iteráveis “eager” (ansiosos ou imediatos)**.

**✅ O que é “preguiçoso” (lazy evaluation)?**

* Significa que os **valores só são gerados ou processados sob demanda**, ou seja, **um de cada vez**, quando você realmente precisa.
* Não carrega tudo na memória de uma vez.

**📌 Exemplos de iteráveis preguiçosos em Python:**

| **Função / Objeto** | **Tipo** |
| --- | --- |
| map() | Lazy |
| filter() | Lazy |
| zip() | Lazy |
| range() (em Python 3+) | Lazy |
| open('arquivo.txt') (linha por linha) | Lazy |
| itertools (como count(), cycle(), etc.) | Lazy |

**✅ O que é “eager” (imediato)?**

* São objetos que **já possuem todos os elementos armazenados**.
* Você pode percorrer, indexar, medir com len(), etc.
* **O conteúdo já está todo pronto na memória.**

**📌 Exemplos de iteráveis eager:**

| **Tipo** | **Características** |
| --- | --- |
| list, tuple, str | Todos os elementos já estão construídos |
| dict, set | Também armazenam todos os elementos prontos |

**🧠 Comparação rápida:**

| **Ação** | **Lazy (**map**,** filter**)** | **Eager (**list**,** tuple**)** |
| --- | --- | --- |
| Ocupa muita memória? | ❌ Não | ✅ Sim (se muito grande) |
| Permite indexação? | ❌ Não | ✅ Sim |
| Funciona com len()? | ❌ Não | ✅ Sim |
| Gera valores sob demanda? | ✅ Sim | ❌ Não |

**🧪 Exemplo prático:**

python

CopiarEditar

nums = [1, 2, 3, 4, 5] dobros\_lazy = map(lambda x: x \* 2, nums) dobros\_list = list(dobros\_lazy) *# força avaliação dos valores* print(dobros\_list) *# [2, 4, 6, 8, 10]*

**🎯 Conclusão:**

Sim! Existem **iteráveis preguiçosos e não preguiçosos**.  
Os preguiçosos **economizam memória e são mais eficientes para grandes volumes de dados**,  
enquanto os eager **são mais fáceis de manipular diretamente** (com índice, slicing, etc.).

Se quiser, posso montar um gráfico visual ou demonstrar com itertools. Deseja?

Você captou exatamente o **comportamento natural e importante** da função map() nesse tipo de uso: ela **para na menor lista**.

**🔍 O que está acontecendo no seu exemplo:**

python

CopiarEditar

a = [1, 2, 3, 4, 5] *# 5 elementos* b = [6, 7, 8, 9, 10] *# 5 elementos* c = [11, 12, 13, 14, 15, 18] *# 6 elementos* resultado = list(map(lambda x, y, z: x + y + z, a, b, c))

**🧠 O** map() **vai emparelhar os elementos posição a posição:**

* (a[0], b[0], c[0])
* (a[1], b[1], c[1])
* ...
* Isso só é possível **até que a menor lista acabe** (no caso, a e b com 5 elementos).

✅ Resultado: 5 somas são feitas  
❌ O 6º item de c (18) é ignorado porque a e b já acabaram.

**📌 Comportamento formal:**

map() **para na menor sequência** envolvida.  
Isso evita exceções como IndexError.

**✅ Se quiser lidar com listas de tamanhos diferentes...**

Você pode usar itertools.zip\_longest() do módulo itertools, e definir um valor padrão para os elementos faltantes.

**🔧 Exemplo com** zip\_longest()**:**

python

CopiarEditar

from itertools import zip\_longest a = [1, 2, 3, 4, 5] b = [6, 7, 8, 9, 10] c = [11, 12, 13, 14, 15, 18] somas = [x + y + z for x, y, z in zip\_longest(a, b, c, fillvalue=0)] print(somas)

**✅ Resultado:**

python

CopiarEditar

[18, 21, 24, 27, 30, 18]

✔ Aqui, o elemento extra (18 de c) foi somado com 0 + 0.

**🎯 Conclusão:**

O map() **trabalha apenas até o fim da menor sequência**.  
Para tratar listas de comprimentos diferentes, use itertools.zip\_longest() com fillvalue.

Observe que o código abaixo está trazendo somente a palavra 'Science' como resultado... import re texto = 'A Data Science Academy oferece os melhores cursos de análise de dados do Brasil.' # Regex para encontrar palavras após "Data" e "Science" padrao = r'\b(?:Data|Science)\s+(\w+)\b' resultado = re.findall(padrao, texto) print(resultado)

**O ChatGPT disse:**

Perfeito, agora entendi o que está acontecendo — e sua observação está **100% correta**.

**🧠 O que está acontecendo?**

Você está usando o regex:

python

CopiarEditar

r'\b(?:Data|Science)\s+(\w+)\b'

Esse padrão encontra **"Data" ou "Science"** seguidos por uma palavra — **mas ele não é “reavaliado” sobrepostas vezes** porque o re.findall() faz **avanço linear**, então:

* Ele casa Data → captura Science
* Mas depois não casa Science de novo → pois ela já foi parte do match anterior (Data Science)
* Por isso **a palavra após** Science **("Academy") não é capturada**

**✅ Como resolver:**

Usamos o recurso de **lookbehind** para capturar a palavra **que vem depois de "Data "** e **a palavra que vem depois de "Science "** — de forma independente:

**✅ Regex com lookbehind:**

python

CopiarEditar

import re texto = 'A Data Science Academy oferece os melhores cursos de análise de dados do Brasil.' *# Regex: pega a palavra que vem depois de "Data" ou "Science"* padrao = r'(?<=\bData\s)\w+|(?<=\bScience\s)\w+' resultado = re.findall(padrao, texto) print(resultado)

**✅ Saída correta:**

css

CopiarEditar

['Science', 'Academy']

**🔍 Explicação:**

* (?<=\bData\s)\w+ → procura palavra **após** "Data"
* | → ou
* (?<=\bScience\s)\w+ → procura palavra **após** "Science"
* re.findall() retorna as palavras **capturadas por ambos os padrões**

Se quiser, posso adaptar essa abordagem para considerar frases com pontuação, acentuação, ou aplicar diretamente a um corpus de texto. Deseja expandir?

**Você disse:**

o que seria o recurso de lookbehind?

**O ChatGPT disse:**

Ótima pergunta! Vamos entender com clareza.

**🔍 O que é lookbehind em regex?**

**Lookbehind** é um recurso de expressões regulares que permite **verificar o que vem *antes* de um determinado padrão**, **sem incluir isso no resultado final**.

**📌 Sintaxe:**

regex

CopiarEditar

(?<=padrão)

* (?<=...) → significa: "procure algo **que venha imediatamente depois de** ..., mas **sem incluir** ... na seleção."

**✅ Exemplo prático:**

Frase:

python

CopiarEditar

texto = "Meu e-mail é contato@example.com e o de suporte é suporte@example.com"

Se quisermos **capturar apenas os endereços de e-mail que aparecem após "é "**, usamos:

python

CopiarEditar

import re re.findall(r'(?<=é\s)[\w\.-]+@[\w\.-]+', texto)

**Saída:**

python

CopiarEditar

['contato@example.com', 'suporte@example.com']

Observe:

* O regex **“olha para trás”** e só captura o e-mail se ele estiver depois de "é ", **mas não inclui "é "** na captura.
* Isso é o **lookbehind** funcionando.

**🧠 Diferença entre Lookbehind e Lookahead**

| **Tipo** | **Verifica...** | **Exemplo** | **Sintaxe** |
| --- | --- | --- | --- |
| Lookbehind | Antes | (?<=foo)bar | “bar” depois de “foo” |
| Lookahead | Depois | foo(?=bar) | “foo” antes de “bar” |

**⚠️ Observação:**

Nem todos os motores de regex suportam **lookbehind**, mas o Python (com re) **suporta totalmente** lookbehind **com comprimento fixo**.

Se quiser, posso te mostrar como usar lookahead, negative lookbehind, e variações. Deseja?