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! 🚀