List Comprehension

List Comprehension

▼ Definição + Sintaxe

> ? **List Comprehension** é uma maneira rápida e simples de criar listas. Permite gerar uma nova lista a partir de qualquer coleção de dados (como listas, ranges ou strings) usando um único comando. Além disso, é possível transformar os itens dessa coleção ou selecionar apenas os que atendem a uma condição

- Sintaxe: **[expressão** for item in iterável if condição **]**
  - **expressão**: Define o que será adicionado à lista
  - **item**: Representa o elemento atual no iterável
  - **iterável**: O objeto de onde os itens serão extraído
  - **condição (opcional)**: Um filtro que determina quais itens serão processados
- Exemplo de criação de lista:

```
lista = [ numero for numero in range(10) ]
print(lista)  # Exibe: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

▼ Mapeamento de dados (map)

❓ Em **list comprehension**, o mapeamento de dados é a transformação de cada item da coleção original em um novo valor, de acordo com a operação definida. Isso é feito diretamente dentro da **expressão**, que aplica a transformação desejada a cada item, gerando uma nova lista.

- Digamos que você queira criar uma lista com o dobro dos números de outra lista. Isso seria um mapeamento de cada número para seu dobro. Exemplo:

```
numeros = [1, 2, 3, 4, 5]
dobros = [x * 2 for x in numeros]
print(dobros)  # Exibe: [2, 4, 6, 8, 10]
```

- Exemplo de mapeamento com dicionários:

```
produtos = [
    {'nome': 'p1', 'preco': 20, },
    {'nome': 'p2', 'preco': 10, },
    {'nome': 'p3', 'preco': 30 }
]
novos_produtos = [
    {**produto, 'preco': produto['preco'] * 1.05}
    if produto['preco'] > 20 else {**produto}
    for produto in produtos
]

print(*novos_produtos, sep='\n')
''' Exibe:
{'nome': 'p1', 'preco': 20}
{'nome': 'p2', 'preco': 10}
{'nome': 'p3', 'preco': 31.5} '''
```

- O código exemplifica um cenário em que produtos receberão 5% de aumento em seus preços, porém apenas se já custarem um valor maior do que 20.

▼ Filtro de dados (filter)

❓ **Filtros em List Comprehension** são usados para selecionar elementos de uma coleção que atendem a uma condição específica. Ao contrário do mapeamento, que transforma os valores, o filtro apenas **inclui** ou **exclui** os elementos com base em um critério

- Filtrar números pares:

```
pares = [numero for numero in range(10) if numero % 2 == 0]
print(pares)  # Exibe: [0, 2, 4, 6, 8]
```

- Filtrar vogais de uma palavra:

```
vogais = [letra for letra in "python" if letra in "aeiou"]
print(vogais)  # Exibe: ['o']
```

▼ Exemplos mapeamentos e filtros

‼️

- **O que vem à esquerda do** `for` pode ser tanto um **mapeamento** quanto uma **transformação** do item, dependendo do que você faz com o item. Se você alterar o valor do item (como elevar ao quadrado ou mudar para maiúscula), isso é um **mapeamento**. Se você simplesmente incluir o item sem alteração, isso não é um mapeamento, mas ainda pode ser parte de um filtro.
- **O que vem à direita do** `for` (após o `if`) é sempre um **filtro**, pois a condição (`if`) serve para selecionar quais elementos são incluídos na nova lista. A condição não altera o valor do item; ela apenas decide se o item será adicionado ou não.

1. Criar uma lista de números ao quadrado:

```
quadrados = [x ** 2 for x in range(5)]
print(quadrados)  # Exibe: [0, 1, 4, 9, 16]
```

   - Aqui, estamos aplicando uma transformação (mapeamento) em cada item da lista: estamos elevando cada número ao quadrado. Portanto, é um mapeamento.

2. Filtrar números pares de uma lista:

```
pares = [numero for numero in range(10) if numero % 2 == 0]
print(pares)  # Exibe: [0, 2, 4, 6, 8]
```

   - Esse exemplo é, na verdade, um filtro em vez de um mapeamento puro, pois estamos selecionando apenas os números pares e não transformando os valores de maneira significativa. Não há uma alteração no valor do item; ele é apenas incluído ou excluído com base na condição. Portanto, este não é um mapeamento no sentido de transformação dos dados, mas sim um filtro.

3. Transformar letras em maiúsculas:

```
letras_maiusculas = [letra.upper() for letra in "python"]
print(letras_maiusculas)  # Exibe: ['P', 'Y', 'T', 'H', 'O', 'N']
```

   - Este exemplo é um mapeamento, pois estamos aplicando uma transformação (convertendo cada letra para maiúscula) a cada item da lista. Cada letra é modificada e mapeada para seu valor em maiúscula.

4. Criar uma lista a partir de elementos condicionais:

```
par_impar = [
    "par" if x % 2 == 0 else "ímpar"
    for numero in range(5)
    ]
print(par_impar)  # Exibe: ['par', 'ímpar', 'par', 'ímpar', 'par']
```

- Esse também pode ser considerado um tipo de mapeamento, pois estamos criando uma nova lista com base em uma transformação condicional. Se o número for par, ele é mantido; se for ímpar, substituímos por `"ímpar"`. Portanto, estamos transformando os valores de acordo com a condição.

▼ Listas aninhadas / matrizes (mais de um **for**)

❓ Uma lista aninhada é uma lista dentro de outra lista. Isso permite representar estruturas bidimensionais, como matrizes.

- Acessar elemento:

```
matriz = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
print(matriz[0])      # Exibe: [1, 2, 3]  (primeira linha)
print(matriz[1][2])   # Exibe: 6  (linha 2, coluna 3)
```

- Criar uma lista bidimensional utilizando for e append:
  - Utilizando dois **for**:

```
lista = []
for i in range(3):
    for j in range(3):
        lista.append((i, j))

 print(lista)
 # [(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]
```

  - Utilizando List Comprehension:

```
lista2 = [(i, j) for i in range(3) for j in range(3)]
print(lista)
# [(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]
```

▼ List Comprehension com mais de um **for**

- Permite criar listas a partir de duas iterações aninhadas, funcionando como um **loop duplo** dentro de uma única expressão
  - Sintaxe: **[expressão** for item1 in iterável1 for item2 in iterável1 **]**
  - O segundo `for` percorre todos os valores do `iterável2` para cada valor de `iterável1`, funcionando de maneira similar a loops `for` aninhados
  - Exemplo de criação de lista bidimensional:

```
lista2 = [(i, j) for i in range(3) for j in range(3)]
print(lista)
# [(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]
```

  - Outra forma de escrever:

```
lista2 = [(i, j)
for i in range(3)
for j in range(3)
]
```

- List comprehension dentro de outra list comprehension:

```
lista = [[j for j in range(3)] for i in range(3)]
print(lista)  # Exibe: [[0, 1, 2], [0, 1, 2], [0, 1, 2]]
```

  - `for` interno cria uma lista `[0, 1, 2]`, pois `j` percorre `range ( 3 )`
  - `for` externo repete a list comprehension interna 3 vezes (para cada valor de `i`).

▼ Dictionary Comprehension

- Sintaxe: { **chave: valor** for item in iterável }
- Exemplo 1 - criar um dicionário de quadrados:

```
quadrados = {x: x**2 for x in range(5)}
print(quadrados)  # Exibe: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

- Exemplo 2 - inverter chaves com valores:

```
dic = {'a': 1, 'b': 2, 'c': 3}
dic_invertido = {v: k for k, v in dic.items()}
print(invertido)
# Exibe: {1: 'a', 2: 'b', 3: 'c'}
```

▼ Set Comprehension

- Sintaxe: { **expressão** for item in iterável }
- Exemplo 1 - criar um conjunto de quadrados:

```
quadrados = {x**2 for x in [1, 2, 3, 2, 1, 4]}
print(quadrados)  # Exibe: {16, 1, 4, 9}
```

- Exemplo 2 - criar um conjunto de vogais únicas em uma frase:

```
frase = "Python é legal"
vogais = {letra for letra in frase if letra.lower() in "aeiou"}
print(vogais)  # Exibe: {"o", "e", "a"}
```