Listas

Listas

▼ Definição

> ❓ Uma **lista** é uma estrutura de dados que permite armazenar múltiplos valores em uma única variável. Ela é um tipo mutável, ou seja, os elementos podem ser adicionados, removidos ou modificados após a criação. É semelhante a um vetor em C, mas é mais flexível, pois pode armazenar diferentes tipos de dados e ser redimensionada dinamicamente.

▼ Sintaxe

- É declarado como: nome da variável = [...]. Exemplo:

```
lista = [ 123 , “nome” , True , 1.2 , [] ]
```

▼ Acessar índice na lista (**lista[i]**)

- Exemplo de como acessar um elemento em uma lista:

```
lista = [123, “nome”, True, 1.2, []]
print(lista[2])  # Exibe 'nome'
```

- Exemplo de como acessar um elemento de um lista dentro de outra lista:

```
lista = [123, [“abc”, “def”, “ghi”], True, 1.2, []]
print(lista[1][2])  # Exibe 'ghi'
```

▼ Deletar item (**del**)

- Método que serve para remover um item da lista
- Ao remover um elemento da lista, todos os outros são movidos, alterando os índices. Exemplo:

```
# Índice 0  1  2  3
lista = [1, 2, 3, 4]
del lista[2]
# Lista passa a ser [1, 3, 4]
# Move os índices:   0  1  2
```

- Evitar remover dessa forma quando for utilizar listas muito grandes, pois a movimentação de diversos elementos da lista pode tornar o código lento e menos eficiente.

▼ Adicionar item (**append( )**)

- Método que serve para adicionar um novo elemento ao final da lista

```
lista = [1, 2, 3, 4]
lista.append(5)
# Lista passa a ser [1, 2, 3, 4, 5]
```

▼ Remover ultimo elemento (**pop( )**)

- Método que serve para remover o último elemento da lista

```
lista = [1, 2, 3, 4]
lista.pop()
# Lista passa a ser [1, 2, 3]
```

- Também pode ser utilizado para remover um elemento em um índice específico da lista, de modo que ele seja indicado como argumento para o método

```
lista = [1, 2, 3, 4]
lista.pop(1)
# Lista passa a ser [1, 3, 4]
```

    - Com essa remoção, ocorre uma movimentação de elementos e de índices, não sendo propícia para listas muito grandes

▼ Limpar a lista (**clear( )**)

- Método que serve para limpar os elemento da lista

```
lista = [1, 2, 3, 4]
lista.clear()
print(lista)  # Exibe []
```

▼ Inserir em uma posição específica (**insert( )**)

- Esse método serve para inserir um novo elemento em uma posição desejada da lista, sendo necessário passar um primeiro argumento para o índice e um segundo argumento para o valor

```
lista = [1, 2, 3, 4]
lista.insert(2, "X")
print(lista)  # Exibe [1, 2, 'X', 3, 4]
```

- Caso seja passado como argumento um índice muito grande e que não corresponde a uma posição da lista, o interpretador considera como a posição final, mesmo que o número não corresponda. Exemplo:

```
lista = [1, 2, 3, 4]
lista.insert(10000, "X")
print(lista)  # Exibe [1, 2, 3, 4, 'X']
```

▼ Juntar duas listas (**extend( )** ou **+**)

- Método que serve para juntar duas listas, de modo que a lista na qual está sendo utilizado o método se transforme na fusão das duas

```
lista_a = [1, 2]
lista_b = [3, 4]
lista_a.extend(lista_b)
print(lista_a)  # Exibe [1, 2, 3, 4]
```

- Esse método modifica apenas o objeto em que está sendo utilizado. Assim, o exemplo a seguir retornaria um valor None

```
lista_a = [1, 2]
lista_b = [3, 4]
lista_c = lista_a.extend(lista_b)
print(lista_c)  # Retorna None
```

    - Isso ocorre pois extend é um método sem retorno de valores, apenas modificando o objeto em que é utilizado. Desse modo, o exemplo mostra que a lista_a foi modificada pelo método, mas nenhum valor foi atribuído a lista_c

- Forma alternativa de justar duas lista, utilizando o sinal de adição (+)

```
lista_a = [1, 2]
lista_b = [3, 4]
lista_c = lista_a + lista_b
print(lista_c)  # Retorna [1, 2, 3, 4]
```

▼ Diferença de atribuição em dados mutáveis

- Como as listas apresentam dados mutáveis, deve-se tomar cuidado ao fazer atribuições. Caso lista_b = lista_a, o valor de uma não é copiado para a outra, pois o que acontece é que ambas as listas apontarão para o mesmo lugar da memória. Dessa forma, uma mudança na lista_a alteraria as duas listas. Exemplo:

```
lista_a = [1, 2, 3]
lista_b = lista_a
lista_a[0] = "X"
```

```
print(lista_a)  # Exibe ['X', 2, 3]
print(lista_b)  # Exibe ['X', 2, 3]
```

- Comparação com dados imutáveis:

```
a = 1
b = a
a = "X"
print(a)  # Exibe 'X'
print(b)  # Exibe 1
```

▼ Copiar lista (**copy( )**)

- Método que serve para copiar uma lista

```
lista_a = [1, 2, 3]
lista_b = lista_a.copy()
lista_a[0] = "X"
print(lista_a)  # Exibe ['X', 2, 3]
print(lista_b)  # Exibe [1, 2, 3]
```

- Nesse caso, a utilização de **copy( )** é útil para evitar erros ao alterar o valor de uma das listas, posto que se fosse utilizado apenas o sinal de atribuição, elas apontariam para o mesmo local da memória, por serem um tipo de dado mutável. Assim, fazer uma cópia garante que ambas não serão modificadas juntas.

▼ Ordenar lista (**sort( )**, **sorted( )** e ***key***)

- **sort( )** é um método usado diretamente em listas para organizá-las em ordem crescente por padrão.
  - Modifica a lista original (opera **in-place**) e não retorna nada (`None`).
  - Aceita parâmetros opcionais:
    - `reverse = True`: ordena em ordem decrescente
    - `key`: permite definir uma função personalizada para o critério de ordenação
  - Exemplo:

```
numeros = [3, 1, 4, 1, 5, 9]
numeros.sort()  # Ordena em ordem crescente
print(numeros)  # Exibe: [1, 1, 3, 4, 5, 9]

# Ordenação decrescente
numeros.sort(reverse=True)
print(numeros)  # Exibe: [9, 5, 4, 3, 1, 1]
```

- **sorted( )** é uma função que retorna uma nova lista ordenada, sem alterar a lista original. Assim, essa função cria uma cópia rasa de uma lista
  - Funciona com qualquer iterável (listas, tuplas, sets, etc)
  - Também aceita os mesmos parâmetros opcionais: `reverse` e `key`
  - Exemplo:

```
palavras = ["banana", "maçã", "laranja"]
ordenadas = sorted(palavras)  # Ordena em ordem alfabética
print(ordenadas)  # Exibe: ['banana', 'laranja', 'maçã']
print(palavras)   # A lista original permanece inalterada

# Ordenação decrescente
ordenadas = sorted(palavras, reverse=True)
print(ordenadas)  # Exibe: ['maçã', 'laranja', 'banana']
```

- O parâmetro `key` em métodos de ordenação como `sort()` e `sorted()` permite definir uma função personalizada que será usada como critério para ordenar os elementos. Essa função é aplicada a cada elemento da lista (ou iterável) e o resultado é usado para determinar a ordem
  - A função passada no parâmetro `key` transforma cada elemento da lista em um "valor chave"
  - Os elementos são ordenados com base nesses valores-chave, mas o conteúdo original da lista permanece o mesmo
  - Exemplo com função existente:

```
palavras = ["banana", "maçã", "laranja", "kiwi"]
# Ordenar com base no tamanho das palavras
palavras.sort(key=len)
print(palavras)  # Exibe: ['kiwi', 'maçã', 'banana', 'laranja']
```

- Exemplo com uma lista de dicionários:

```
pessoas = [
    {"nome": "João", "idade": 25},
    {"nome": "Maria", "idade": 30},
    {"nome": "Ana", "idade": 20},
]
# Ordenar pela idade
pessoas.sort(key=lambda pessoa: pessoa["idade"])
print(pessoas)
# Exibe: [{'nome': 'Ana', 'idade': 20}, {'nome': 'João', 'idade': 25}, {'nor
```