05-FP-Listas

April 22, 2025

0.1 Listas em Python

Bem-vindo a mais uma aula de **Fundamentos de Programação em Python**!

Objetivos de Aprendizagem

Ao final desta aula, você será capaz de:

- Compreender o que são listas e para que servem em Python.
- Aprender a criar, acessar, modificar e percorrer listas.
- Praticar operações básicas e intermediárias com listas.
- Resolver problemas utilizando listas em programas.

0.2 O que é uma Lista?

Em Python, uma **lista** é uma estrutura de dados que armazena uma sequência de elementos, podendo conter diferentes tipos (números, strings, outras listas etc.).

Os valores em uma lista são chamados de *elementos*, ou, algumas vezes, de itens.

0.2.1 Sintaxe:

```
minha_lista = [10, 20, 30, 40]
```

Lista homogênea (elementos do mesmo tipo de dados):

[ ]:
```
[10, 20, 30, 40]
```

Lista heterogênea (elementos de tipo de dados diferentes):

[ ]:
```
['item', 2, 3.0, True, 'texto']
```

Lista vazia: Uma lista que não contém elementos é chamada de lista vazia; você pode criar uma com colchetes vazios [].

[ ]:
```
lista = []
print(lista)
```

0.3 Acessando Elementos

- Assim como strings, listas são indexadas a partir do índice 0.
- Qualquer expressão de números inteiros pode ser usada como índice.

- Se tentar ler ou escrever um elemento que não existe, você recebe um IndexError.
- Se um índice tiver um valor negativo, ele conta de trás para a frente, a partir do final da lista.

[ ]:
```
minha_lista = [10, 20, 30, 40]
print(minha_lista[0])  # Primeiro elemento
print(minha_lista[-1]) # Último elemento
```

[ ]:
```
colors = ['red', 'green', 'blue', 'yellow', 'white', 'black']
print(colors)
```

A lista acima pode ser representada da seguinte forma:

```
                +-------+ +-------+ +-------+ +-------+ +-------+ +-------+
elementos ->    | Red   | | Green | | Blue  | | Yellow| | White | | Black |
                +-------+ +-------+ +-------+ +-------+ +-------+ +-------+
índices   ->    | 0     | | 1     | | 2     | | 3     | | 4     | | 5     |
                +-------+ +-------+ +-------+ +-------+ +-------+ +-------+
```

Slicing (Fatiamento)

Assim como em strings, slicing permite que você extraia uma parte de uma lista sem modificar a lista original. É como se você estivesse "fatiando" a lista em pedaços menores.

Sintaxe básica:

`nome da variável lista [início:fim:passo]`

- **início**: Índice onde o slice começa (inclusive). Se não for especificado, o slice começa do início da lista. (Posição 0)
- **fim**: Índice onde o slice termina (exclusivo). Se não for especificado, o slice vai até o final da lista.
- **passo**: O intervalo entre os elementos a serem selecionados. Se não for especificado, o valor padrão é 1, ou seja, todos os elementos serão selecionados.

Portanto, o padrão do slicing (fatiamento) é `[início (default é 0) : fim (valor-1) : passo (default é 1)]`

[ ]:
```
colors[0:2]
```

[ ]:
```
colors[1:1]
```

[ ]:
```
colors[4:]
```

[ ]:
```
colors[1::2]
```

[ ]:
```
colors[0::2] # Seleciona os itens da posição 0, até a última posição (que é␣
↪default quando não tem informação) com o step 2
```

[ ]:
```
colors[0:6:2] # Seleciona os itens da posição 0, até a última posição, pois vai␣
↪ler até a posição 5 de 2 em 2
```

O operador `in` também funciona com listas:

[ ]:
```
"blue" in colors
```

[ ]:
```
"purple" in colors
```

0.4 Percorrendo uma lista com loops

É possível usar um loop `for` para percorrer os elementos de uma lista:

[ ]:
```
colors = ['red', 'green', 'blue', 'yellow', 'white', 'black']

for color in colors:
    print(color)
```

Você também pode usar os índices para percorrer a lista no loop:

[ ]:
```
for i in range(len(colors)):
    print(f"Posição {i}: {colors[i]}")
```

Note que um loop sobre uma lista fazia numa executa o código em seu escopo:

[ ]:
```
for x in []:
    print('Nunca vai executar.')
```

Exercício:

Dada a lista `[4, 7, 2, 8, 1]`, escreva um código que conte quantos números são maiores que 5.

0.5 Listas são Mutáveis

Diferente das strings, listas são mutáveis. Ou seja, podemos modificar um elemento da lista acessando sua posição via índice.

[ ]:
```
minha_lista = [10, 20, 30, 40]
minha_lista[2] = 99 # Modifica o terceiro elemento
print(minha_lista)
```

Use `del` para deletar elementos da lista:

[ ]:
```
numeros = [1, 2, 3, 4, 5]
del numeros[0] # Deleta o elemento na posição 0
print(numeros)
```

Para remover mais de um elemento por vez, você pode usar `del` com um índice de fatia:

[ ]:
```
numeros = [1, 2, 3, 4, 5]
del numeros[1:4] # Deleta os elementos nas posições 1, 2 e 3
print(numeros)
```

Se souber o índice do elemento que quer deletar, você pode usar `pop`. O método `pop` altera a lista e retorna o elemento que foi deletado:

[ ]:
```
numeros = [1, 2, 3, 4, 5]
x = numeros.pop(2)
print(x)
print(numeros)
```

Exercício:

Dada a lista `numeros = [1, 2, 3, 4, 5]`: - a) Adicione um novo número ao final. - b) Substitua o terceiro número. - c) Remova o primeiro e o segundo número.

0.5.1 Operações básicas

[ ]:
```
# Novo exemplo de lista
languages = ['C', 'C++', 'Python', 'Java']
print(languages)
```

[ ]:
```
# Tamanho da lista (quantos itens ela contém)
len(languages)
```

[ ]:
```
# Adicionar um elemento ao final da lista
# append()
languages.append("Último")
print(languages)
```

[ ]:
```
# Adicione um elemento no começo da lista
# insert()
languages.insert(0, "Javascript") #(posição onde será adicionado, novo elemento)
print(languages)
```

[ ]:
```
# Ordenar os elementos de uma lista
sorted(languages)
# Em uma lista de strings ordena em ordem alfabética
```

[ ]:
```
# A função sorted não modifica a lista original. Ela cria uma nova lista␣
 ↪ordenada.
print(languages)
```

[ ]:
```
# A lista ordenada pode ser assinada para outra variável
languages_ordenada = sorted(languages)
print(languages_ordenada)
```

[ ]:
```
# Ordenar elementos por comprimento (key=len)
languages = ['Python', 'Java', 'C', 'C++', 'Javascript']
sorted(languages,key=len) # Para ordenar pelo número de caracteres dos elementos
```

Exemplos com lista de **números**

[ ]:
```
numeros = [4, 7, 1, 9]
print(len(numeros))    # Tamanho da lista
```

```
print(sum(numeros))    # Soma dos elementos
print(max(numeros))    # Maior valor
print(min(numeros))    # Menor valor
```

O operador + concatena listas:

[ ]:
```
numbers_part_1 = [1, 2, 3]
numbers_part_2 = [4, 5, 6]

print(numbers_part_1)
print(numbers_part_2)

print("Nova lista:", numbers_part_1 + numbers_part_2) # Nova lista
```

O méotodo `extend` toma uma lista como argumento e adiciona todos os elementos:

[ ]:
```
# Para adicionar os elementos de uma lista na outra
numbers_part_1 = [1, 2, 3]
numbers_part_2 = [4, 5, 6]

numbers_part_1.extend(numbers_part_2) # Não criou uma nova lista, adicionou a␣
 ↪segunda na primeira

print(numbers_part_1)
print(numbers_part_2)
```

O operador * repete a lista um dado número de vezes:

[ ]:
```
[0] * 4
```

0.6 List Comprehensions

List comprehensions em Python são uma forma concisa e elegante de criar listas a partir de outras coleções de dados iteráveis (como listas, tuplas, ranges, etc.). Pense nelas como um atalho para escrever loops `for` que criam listas de forma rápida.

Sintaxe básica:

`nova_lista = [expressão for item in iterável if condição]`

- expressão: O que você quer colocar em cada elemento da nova lista (pode ser uma variável, uma operação matemática, etc.).
- item: A variável que representa cada elemento do iterável.
- iterável: A sequência de elementos a partir da qual você está criando a nova lista (lista, tupla, range, etc.).
- condição (opcional): Um filtro. Só os itens que satisfazem essa condição serão incluídos na nova lista.

Por que usar?

- **Facilidade:** Reduzem significativamente o código em comparação com loops `for` tradicionais.
- **Eficiência:** Em muitos casos, são mais rápidas do que loops `for` equivalentes

Exemplo Simples:

[ ]:
```python
numeros = [1, 2, 3, 4, 5]

# Cria uma lista com os quadrados de cada número
quadrados = [x**2 for x in numeros]

print(quadrados)
```

Exemplo com Condição:

[ ]:
```python
numeros = [1, 2, 3, 4, 5]

# Cria uma lista com os quadrados dos números pares
pares_ao_quadrado = [x**2 for x in numeros if x % 2 == 0]

print(pares_ao_quadrado)
```

Exercício:

Dada a lista `frutas = ["abacaxi", "banana", "morango", "ameixa", "uva", "acerola"]`, escreva uma *list comprehension* que crie uma nova lista chamada frutas_com_a contendo apenas as frutas da lista frutas que começam com a letra "a" (maiúscula), e converta cada fruta para maiúsculas antes de adicioná-la à nova lista.

0.7 Exercícios para Praticar:

1. Escreva uma função chamada `is_sorted` que tome uma lista como parâmetro e retorne `True` se a lista estiver classificada em ordem ascendente, e `False` se não for o caso. Por exemplo:

   ```
   is_sorted([1, 2, 2]) # Saída: True
   is_sorted(['b', 'a']) # Saída: False
   ```

2. Escreva uma função chamada `soma_cumulativa` que receba uma lista de números e retorne a soma cumulativa; isto é, uma nova lista onde o i-ésimo elemento é a soma dos primeiros `i+1` elementos da lista original. Por exemplo:

   ```
   t = [1, 2, 3]
   print(soma_cumulativa(t)) # Saída: [1, 3, 6]
   ```

3. Escreva uma função chamada `tem_duplicados` que tome uma lista e retorne `True` se houver algum elemento que apareça mais de uma vez, caso contrário retorna `False`. Ela não deve modificar a lista original. `python tem_duplicados([1, 2, 2]) # Saída: True tem_duplicados(['b', 'a']) # Saída: False`

0.8 Conclusão

Parabéns por concluir esta aula!

0.8.1 O que aprendemos hoje?

- Listas são fundamentais na programação com Python.
- Permitem armazenar, acessar e manipular coleções de dados.
- Operações como percorrer, fatiar e transformar listas são muito comuns no dia a dia de quem programa.

0.8.2 Próximos Passos

- Resolva os problemas na seção "Exercícios para Praticar".
- Verificar a documentação oficial do tipo list.

0.8.3 Parabéns pela dedicação! Nos vemos na próxima aula!