

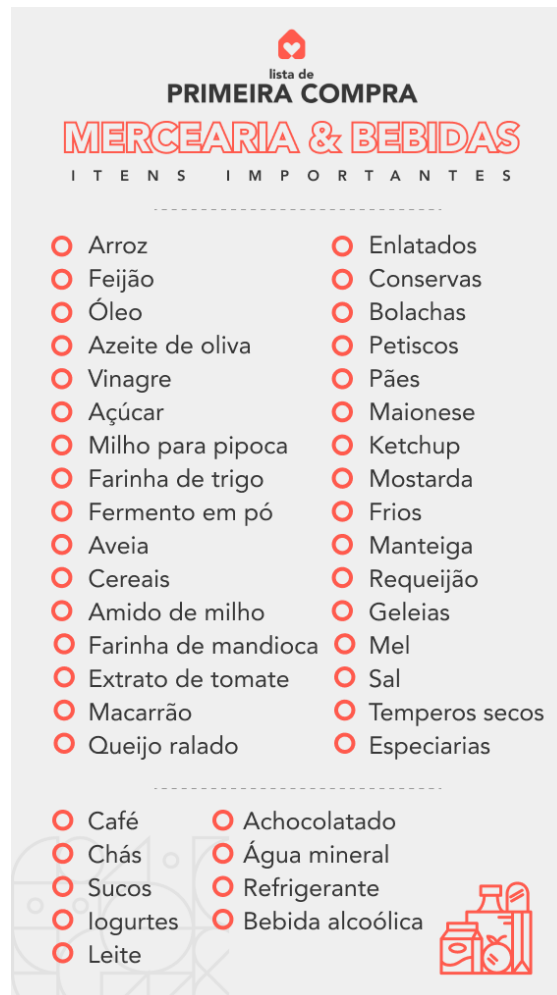
COMPUTAÇÃO 1 – AULA 5

Listas e Tuplas

Prof. Cesar Raitz

1. Introdução

- Até agora, cada variável identificava um número, uma string ou um booleano → um valor singular
- Mas como podemos guardar uma sequência de dados?



- Para isso, inventaram a **lista**.
- Conhecer e manipular listas em Python é essencial para criar algoritmos avançados.

2. Definição

- **Lista** é uma sequência de **itens** ou **elementos**.
- Um **item** pode ser qualquer estrutura de dados do Python: número, string, booleano, lista *etc*.
- Pode ter tantos itens quanto necessários, separados por vírgula.
- Sua definição começa e termina com **colchetes** [].
- Veja alguns exemplos:

1.	<code>fibo = [1, 1, 2, 3, 5, 6, 13, 21, 34]</code>
2.	<code>polos = [1+1j, 1-1j, 3+4j, 3-4j]</code>
3.	<code>personagens = ["Luluzinha", "Bolinha"]</code>

- Neste código:
 - `fibo` é uma lista de números **int**
 - `polos` é uma lista de números **complex**
 - `personagens` é uma lista de **str**
- Mas os elementos *não precisam ser do mesmo tipo*, podemos ter uma lista assim:
`[111, "João", 2.25, "roubou", 7+8j, "pão", False]`
- Também podemos criar listas de apenas um elemento ou nenhum:

1.	<code>lista_vazia = []</code>
2.	<code>lista_um = [1]</code>
3.	<code>print(lista_vazia)</code>
4.	<code>print(lista_um)</code>

- O tipo de variável para listas é **list**.

3. Indexação e Fatiamento

- A maioria das operações com strings também foram definidas para listas, inclusive **indexação** e **fatiamento**.
- Assim como strings, *imagine uma lista como uma sequência de caixinhas*. Só que, ao invés de caracteres, teremos itens em cada caixinha. Por exemplo, uma lista de pintores:

pintores

"Picasso"	"Van Gogh"	"Cézanne"	"Da Vinci"
0	1	2	3

1.	pintores = ["Picasso", "Van Gogh",
2.	"Cézanne", "Da Vinci"]

- Assim, cada índice tem um elemento correspondente na lista:
 - pintores[0] é a caixinha de "Picasso"
 - pintores[1] é a caixinha de "Van Gogh"
 - pintores[2] é a caixinha de _____
 - pintores[3] é a caixinha de _____
- Também vale índices negativos:
 - pintores[-1] é a caixinha de "Da Vinci"
 - pintores[-2] é a caixinha de "Cézanne"
 - *etc.*
- Também vale fatiamento e suas regras de omissão de índices:
 - pintores[1:3] equivale a ["Van Gogh", "Cézanne"]
 - pintores[:2] equivale a ["Picasso", "Cézanne"]

pintores[::2]

"Picasso"	"Van Gogh"	"Cézanne"	"Da Vinci"
0	1	2	3

- Assim como strings, podemos imprimir
 - O resultado de indexação é um dado (**str**, **int**, **float**, *etc.*)
 - ⚠ Mas o resultado de um fatiamento é *outra lista*! ⚠

```
3. print(pintores[0], "é string")
4. print(pintores[-1], "é string")
5. print(pintores[1:2], "é lista!")
```



Comprimento da lista

- Para obter o comprimento de uma string (número de caracteres), usamos **len()**.
- Para obter o comprimento de uma lista (número de elementos), também usamos **len()**.

Exercício 1. Lista de tarefas.

- a) Com a sintaxe aprendida, crie uma lista com algumas poucas tarefas de casa. É claro, cada tarefa será uma string, como "Arrumar a cama".
- b) Depois imprima cada tarefa enumerando-as a partir de 1.
- c) Usando **len()**, imprima quantas tarefas tem na lista.


4. Modificando uma lista

- Listas podem ser **modificadas**, por isso dizemos que elas são **mutáveis**.
- Por exemplo, criamos uma lista de notas, mas vamos corrigir duas:

1.	<code>notas = [6.5, 9.8, 7.7, 8.5]</code>
2.	<code># Corrigindo notas</code>
3.	<code>notas[0] = 7.2</code>
4.	<code>notas[-1] = 8.2</code>

notas

6.5	9.8	7.7	8.5
0	1	2	3



7.2	9.8	7.7	8.2
0	1	2	3

- Vamos **atualizar** os valores:

5.	<code># Mais pontos de participação</code>
6.	<code>notas[1] = notas[1] + 0.1</code>
7.	<code>notas[2] = notas[2] + 0.1</code>

- Explicando linha 6 tim-tim por tim-tim:
 - Um novo valor será **atribuído** à posição 1 da lista (`notas[1]`)
 - Mas antes, é preciso calcular o que está do lado direito
 - O Python pega o valor na posição 1 (9.8) e soma 0.1
 - Esse novo valor (9.9) será colocado na posição 1

notas

7.2	9.8	7.7	8.2
0	1	2	3

 →

7.2	9.9	7.8	8.2
0	1	2	3

- Quando a lista é numérica, podemos somá-la com **sum()**

```
10. # Calcula a média
11. media = sum(notas)/len(notas)
12. print(media)
```



Isso é notável!

- Strings podem ser indexadas. O resultado é uma string com apenas um caractere. Então podemos alterar um caractere de uma string? 🤔

```
1. quem = "filho"
2. quem[-1] = "a"
```

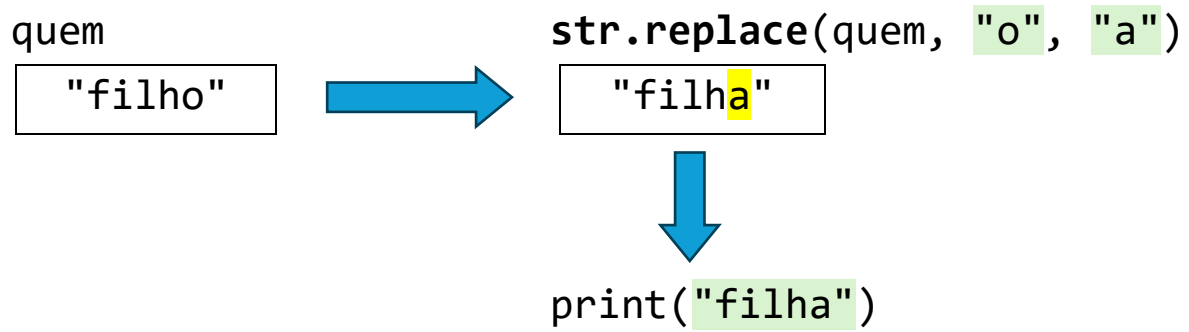
- O programa acima, gera esse erro na linha 2:

TypeError: 'str' object does not support item assignment

- Justamente porque *strings não podem ser modificadas*: são **imutáveis**!
- O que acontece, quando usamos fatiamento ou funções **str.**, é que *uma nova string é criada* em outro local da memória do computador.

```
1. quem = "filho"
2. print(str.replace(quem, "o", "a"))
```

- O que está acontecendo nesse programa:



- Veremos que diversas funções de lista *alteram a própria lista*.

5. Algumas funções de listas

- Assim como strings, fica difícil criar alguma coisa sem funções para manipular listas.
- A maioria das funções começa com o prefixo **list**.

Anexando elementos

- **list.append(lista, elemento)** anexa um elemento ao final de uma lista.
- Certamente, é a função mais utilizada de listas!

```
1. primos = [2, 3, 5, 7, 11]
2. list.append(primos, 13)
3. list.append(primos, 17)
```

primos

2	3	5	7	11
0	1	2	3	4



primos

2	3	5	7	11	13
0	1	2	3	4	5



primos

2	3	5	7	11	13	17
0	1	2	3	4	5	6

Exercício 2. Adicione tarefas.

Use **list.append()** para adicionar mais duas tarefas à sua lista de tarefas.

Inserindo elementos

- Nem sempre queremos colocar um item no fim da lista, então usamos **list.insert(lista, elemento, posicao)**
- Os elementos a partir de posicao serão deslocados para frente.

```
4. list.insert(primos, 3, 1234)
```

primos

1	2	3	1234	5	7	11	13	15
0	1	2	3	4	5	6	7	8

Procurando coisas

- Para procurar uma string em outra, temos **str.find()**
- Uma função parecida é **list.index(lista, elemento)**
- A diferença é que o elemento precisa necessariamente estar na lista, senão o Python interrompe o programa para mostrar esse erro:

ValueError: XXX is not in list

- Para evitar isso, teste antes com o operador **in**, para saber se o elemento se encontra na lista.

```
1. mercado = [  
2.     "banana",  
3.     "laranja",  
4.     "maçã",
```

```
5.     "pêra"
6. ]
7. i = list.index(mercado, "maçã")
8. print("índice da maçã:", i)
9.
10. if "banana" in mercado:
11.     j = list.index(mercado, "banana")
12.     print("índice da banana:", j)
13.
14. k = list.index(mercado, "abacate")
```

- A instrução da linha 14 vai gerar um erro (por quê?).


Retirando elementos

- Como remover um item da lista? Tem 3 opções:
 - `ultimo_item = list.pop(lista)`
retira o último item da lista e retorna, então você pode guardá-lo numa variável.
 - `list.remove(lista, item)`
simplesmente remove o item da lista, *mas não retorna*, então não podemos mais acessá-lo.
 - `del lista[indice]`
apaga o item com determinado índice. **del** vem de *delete* (apagar).

```

1. mercado = ["banana", "laranja",
2.             "maçã", "pêra"]
3.
4. ultimo = list.pop(mercado)
5. print(f"O último item era {ultimo}.")
6. print(f"O penúltimo item era {list.pop(mercado)}.")
7. print("Falta comprar:", mercado)
8.
9. print("Não tem laranjas, apagando...")
10. list.remove(mercado, "laranja")
11. print("Temos o primeiro em casa!")
12. del mercado[0]
13.
14. print(mercado)

```



Concatenação e multiplicação

- São operações que já vimos pra strings, lembra?

```

1. # Com Strings
2. print("Hello, " * 2)           # multiplicação
3. print("Hello, " + "World!")   # concatenação
4.
5. # Com Listas
6. print([1, 10] * 2)            # multiplicação
7. print([1, 10] + [100, 1000]) # concatenação

```

Ordenando listas

- **list.sort(lista)** ordena a própria lista na memória, então a ordem original é perdida.

1.	<code>nomes = ["Luke", "Lea", "Han"]</code>
2.	<code>print(nomes) # ['Luke', 'Lea', 'Han']</code>
3.	<code>list.sort(nomes)</code>
4.	<code>print(nomes) # ['Han', 'Lea', 'Luke']</code>

- Se não quisermos perder a lista original, usamos **sorted(lista)**:

1.	<code>numeros = [5, 4, 3, 2, 1]</code>
2.	<code>nova_lista = sorted(numeros)</code>
3.	<code>print(nova_lista) # [1, 2, 3, 4, 5]</code>
4.	<code>print(numeros) # [5, 4, 3, 2, 1]</code>

Máximos e mínimos

- Nos exercícios, vimos que **min** e **max** retornavam o mínimo e o máximo de uma sequência de números.
- Também são usadas com listas! 😎

1.	<code>notas = [6.5, 9.8, 7.7, 8.5]</code>
2.	<code>maior = max(notas)</code>
3.	<code>menor = min(notas)</code>

- Mais interessante, é usar essas funções e depois **list.index()** para saber a posição do elemento na lista. Continuando o exemplo:



4.	<code>i = list.index(notas, maior)</code>
5.	<code>j = list.index(notas, menor)</code>
6.	<code>print(f"Minha melhor nota é a {i+1}ª")</code>
7.	<code>print(f"Minha pior nota é a {j+1}ª")</code>

Exercício 3. Melhor tempo de volta.

Um certo corredor de kart deu 6 voltas numa corrida. Os tempos de volta foram guardados numa lista tempos. Encontre os tempos da melhor e da pior volta, imprima os tempos com as respectivas voltas.

6. Definição de tuplas

- Em certos algoritmos, é necessário proteger a sequência de dados. Por exemplo, as informações de um cliente. Para isso as **tuplas** foram criadas.
- Uma tupla é como uma lista, tem indexação, fatiamento, concatenação *etc.* Só não podemos mudar seus elementos! Tuplas são **imutáveis**!
- Em Python, o tipo de variável é **tuple** e fica entre parênteses ().

```
1. tupla_vazia = ()
2. tupla_um = (123,)
3. tupla = (1, 1, 2, 3, 5)
4. print("comprimento:", len(tupla))
```

Desempacotamento

- Geralmente, acessamos os elementos indexando-os.
- Programadores profissionais usam um truque chamado **desempacotamento**, onde você cria uma variável para cada elemento da tupla.
- Por exemplo, após a instrução:
`signo, ascendente = ("Touro", "Leão")`
teremos duas variáveis:
 - `signo` com o valor `"Touro"` e
 - `ascendente` com o valor `"Leão"`.
- Veja mais um exemplo:

1.	<code>dados = ("Roberto", 11122233300, "casado")</code>
2.	<code># Funciona mas toma espaço:</code>
3.	<code>nome = dados[0]</code>
4.	<code>cpf = dados[1]</code>
5.	<code>ecivil = dados[2]</code>
6.	
7.	<code># Agora com desempacotamento:</code>
8.	<code>nome, cpf, ecivil = dados</code>

7. Exercícios

Exercício 4. Resultado de programas.

Ao indexar a lista você obtém um elemento com seu próprio tipo de variável. Mas, ao fatiar uma lista, você obtém outra lista! Pense em qual será o resultado de cada um dos programas abaixo:

a)

1.	<code>x = [1, 2, 3]</code>
2.	<code>y = [4, 5, 6]</code>
3.	<code>z = x[:2] + y[1:]</code>
4.	<code>print(z)</code>

b)

1.	<code>m = ["a", "b", "c"] + "d"</code>
2.	<code>print(m)</code>

a)

1.	<code>periodos = ["Cretáceo", "Paleoceno",</code>
2.	<code> "Mesoceno", "Antropoceno"]</code>
3.	<code>x = periodos[1][:4]</code>
4.	<code>print(x)</code>

Exercício 5. Somando dois vetores.

Cálculo com vetores é extremamente importante. Por exemplo, aqui temos dois vetores a e b:

1.	<code>a = (10, 15)</code>
2.	<code>b = (-5, -20)</code>
3.	<code>x1 = a[0]</code>
4.	<code>y1 = a[1]</code>
5.	<code>x2 = b[0]</code>
6.	<code>y2 = b[1]</code>
7.	<code>n = (x1+x2, y1+y2)</code>
8.	<code>print(n)</code>

Há um jeito mais prática de obter as coordenadas dos vetores? E se fossem 3 componentes, x, y e z?