Trabalhando com Listas

Introdução a Listas

Como ja estamos cansados de ver em **estrutura de dados** que vemos na faculdade, **Listas** refere-se ao armazenamento de dados em formato de lista em uma variável, e claro mantem os dados armazenados em uma sequência!

E no **Python** não é diferente, e para criar listas utilizamos **[]**, como já estamos acostumados em várias linguagens de programação.

04 - Estrutura de dados > 01 - Trabalhando com listas > codigo-fonte.py > ...

```python
# Criando uma varial do tipo List
lista = [1, 10, 30]
print(type(lista))
```

```
<class 'list'>
```

Acessando elementos da lista

Para acessar os elementos dentro de uma lista não temos nada demais, basta definir qual índice da lista que queremos acessar **[{índice}]**.

```python
# Acessando elementos da lista
lista2 = [1, 10, 30]
print(lista2[0])
```

```
1
```

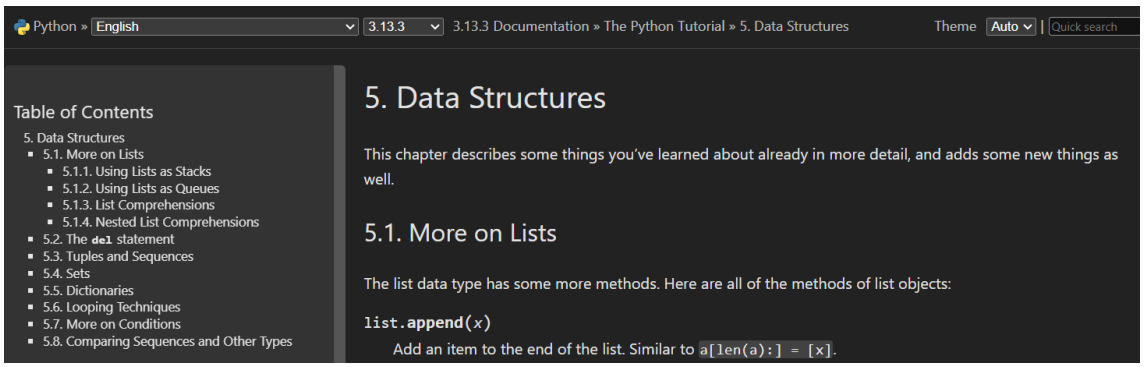
Vale destacar que no **Python** podemos acessar elementos colocando **índices negativos**, com isso a ordem da sequência dos elementos será reversa!

```python
lista2 = [1, 10, 30]
print(lista2[-1])
```

```
30
```

Funções de uma lista

Para verificarmos todas as funções de um **List** no **Python** podemos verificar diretamente pela sua **documentação**!



https://docs.python.org/3/tutorial/datastructures.html

Duplicando ou Concatenando listas com o operador (*)

Um recurso muto legal no **Python** é a forma como podemos **duplicar** ou até mesmo **concatenar** listas.

Aí você pode se perguntar... **“Ue, mas qual a diferença se em outras linguagens também podemos realizar isso!?🤔”**

E a resposta é que no **Python** podemos fazer isso de uma forma bem simples, usando o operador ***.**

Isso mesmo, no Python o operador * não serve apenas para realizar operações matemáticas de multiplicação, e sim podemos realizar outros tipos de operação, como:

- **Multiplicação de números.**

```python
3 * 4  # Resultado: 12
```

- **Repetição de sequências (listas, strings, tuplas)**

```python
[1, 2] * 3      # Resultado: [1, 2, 1, 2, 1, 2]
"ab" * 2        # Resultado: 'abab'
(5,) * 4        # Resultado: (5, 5, 5, 5)
```

- **Destructuring de argumentos em chamadas de function**

```python
def somar(a, b, c):
    return a + b + c

valores = [1, 2, 3]
somar(*valores)  # Resultado: 6
```

- **Passagem de parâmetros args ou kwargs**

```python
def imprimir(*args):
    for item in args:
        print(item)

imprimir("Oi", "tudo", "bem?")
# Saída:
# Oi
# tudo
# bem?
```

- **Destructuring em atribuição**

```
a, *meio, b = [1, 2, 3, 4, 5]
# a = 1, meio = [2, 3, 4], b = 5
```

- **Concatenação de listas**

```
a = [1, 2]
b = [3, 4]
nova_lista = [*a, *b]  # Resultado: [1, 2, 3, 4]
```

Resumindo... O operador ***** em **Python** não serve apenas para **multiplicar números**. Ele tem um comportamento mais amplo, podendo ser entendido como um operador de **"multiplicação ou expansão"**, dependendo do contexto.

Em alguns casos, ele realiza **multiplicação matemática** (quando usado entre números), mas em outros, ele **repete elementos**, **desempacota valores** ou **permite o envio de múltiplos argumentos em funções, etc**. Por isso, pode parecer que ele está "duplicando registros", quando na verdade está espalhando ou multiplicando elementos de outras formas!

Extraindo variáveis em lista

Já vimos brevemente como extrair variáveis em listas na sessão **Duplicando ou Concatenando listas com o operador (*)**, e talvez nem tenha percebido!!

Mas para extrair variáveis em listas, basta definir as variáveis na ordem certa referente a **sequência da lista**, como podemos ver:

```
# Extraindo variáveis em lista
a, b, c = [10, 20, 30]

print(a)
print(b)
print(c)
```

Porém nesse cenário somos obrigados a passar todas as variáveis de acordo com a sequência da lista. Tanto que se passarmos menos variáveis referente as sequencias da lista, teremos o seguinte erro:

```
Traceback (most recent call last):
  File "c:\Projetos\pocs\python-study\04 - Estrutura de dados\01
line 36, in <module>
    a, b = [10, 20, 30]
    ^^^^
ValueError: too many values to unpack (expected 2)
```

E para resolver isso, usaremos o operador **(*)**, até porque como já sabemos, o operador **(*)** não serve apenas para realizar operação matemática de multiplicação, mas serve também para **extrair valores de lista**, ou até mesmo realizar **destructuring(explosão)** dos dados.

```
primeiro, segundo, *outros = [10, 20, 30, 40, 50, 60]

print(primeiro)
print(segundo)
print(*outros)
```

Inclusive podemos extrair essa **"explosão"** dos valores da lista em uma outra **variável** não apenas no final das declarações, mas podemos também definir em **qualquer sequência** que quisermos.

```
firt, *meio, last = [10, 20, 30, 40, 50, 60]

print(firt)
print(meio)
print(last)
```

```
10
[20, 30, 40, 50]
60
```

Validando itens em uma lista

Um recurso muito importante em programação no geral, é a validação se um elemento existe ou contém em uma lista! E no **Python** essa verificação é muito simples, basta utilizarmos o operador **in** ao realizar a validação.

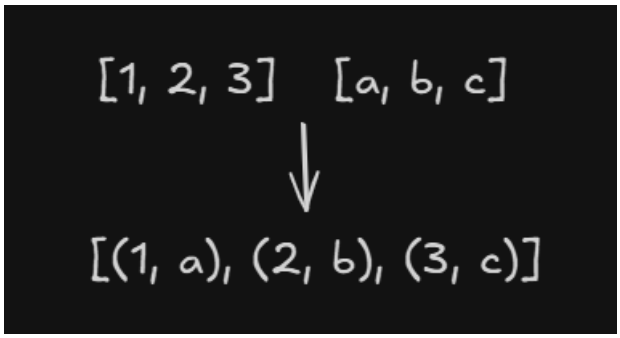
```
# Validando itens em uma lista
itens_valid = [1, 2, 3]
item_busca = 1

if item_busca in itens_valid:
    print('item encontrado')
else:
    print('item nao encontrado')
```

```
item encontrado
```

Juntando 2 listas com o ZIP

Outro recurso que temos também é **"juntar"** o conteúdo de **N** listas, fazendo com que cada elemento das listas fique armazenado em um elemento único como um **Tuple**.



Dito isso, vamos criar **2 listas independentes**, e após criado das listas vamos **"juntá-las"** criando o objeto **zip()**.

```
# Mergeando 2 listas com zip
lista1 = [1, 2, 3]
lista2 = ['a', 'b', 'c']

lista_zip = zip(lista1, lista2)
print(lista_zip)
```

```
<zip object at 0x00000294FFD77D00>
```

Como podemos ver, algo está estranho no resultado, pois mesmo não gerando erro, a saída do conteúdo não está como uma lista final, como imaginávamos.

Isso porque, de forma bem obvia ao criar um objeto **zip**, teremos um objeto **Zip** e não **List**!! Porém para resolver isso, podemos criar um **List** passando um objeto **Zip** em seu **construtor**, logo o código ficará assim.

```
lista_zip = list(zip(lista1, lista2))
print(lista_zip)
```

```
[(1, 'a'), (2, 'b'), (3, 'c')]
```

Com isso, teremos um **List** onde cada elemento se trata de um **Tuple**!!

Vale destacar também que caso alguma das listas tiver mais elementos que a outras, os **elementos** que tiverem a **MAIS** não será trazido na **lista final** juntada pelo **zip**! Como podemos ver a seguir:

```
# Mergeando 2 listas com zip
lista1 = [1, 2, 3]
lista2 = ['a', 'b', 'c', 'd', 'e']

lista_zip = list(zip(lista1, lista2))
print(lista_zip)
```

```
[(1, 'a'), (2, 'b'), (3, 'c')]
```