**Explicação criada por ia**

Python oferece várias estruturas de dados para armazenar e manipular coleções. Aqui estão as principais formas de coleção:

1. Listas (list)

- São ordenadas e mutáveis.

- Elementos são acessados por índice.

- Sintaxe: [elemento1, elemento2, ...]

- Exemplo: frutas = ['maçã', 'banana', 'laranja']

- Métodos: append(), insert(), remove(), sort()

2. Tuplas (tuple)

- São ordenadas e imutáveis.

- Elementos são acessados por índice.

- Sintaxe: (elemento1, elemento2, ...)

- Exemplo: cores = ('vermelho', 'azul', 'amarelo')

- Métodos: index(), count()

3. Conjuntos (set)

- São não ordenados e mutáveis.

- Elementos são únicos.

- Sintaxe: {elemento1, elemento2, ...}

- Exemplo: numeros = {1, 2, 3, 4, 4} (não permite duplicados)

- Métodos: add(), remove(), union(), intersection()

4. Dicionários (dict)

- São não ordenados e mutáveis.

- Elementos são acessados por chave.

- Sintaxe: {chave: valor, ...}

- Exemplo: pessoa = {'nome': 'João', 'idade': 30}

- Métodos: keys(), values(), items(), get()

5. Conjuntos Ordenados (frozenset)

- São não ordenados e imutáveis.

- Elementos são únicos.

- Sintaxe: frozenset({elemento1, elemento2, ...})

- Exemplo: numeros = frozenset({1, 2, 3, 4})

- Métodos: union(), intersection()

Semelhanças

- Todas as coleções podem ser iteradas usando for.

- Todas as coleções têm métodos para verificar a existência de elementos (in).

Diferenças

- Ordenação: listas e tuplas são ordenadas, enquanto conjuntos e dicionários não.

- Mutabilidade: listas, conjuntos e dicionários são mutáveis, enquanto tuplas e conjuntos ordenados são imutáveis.

- Acesso: listas e tuplas são acessadas por índice, enquanto conjuntos são acessados por valor e dicionários por chave.

Escolhendo a coleção certa

- Use listas para dados ordenados e mutáveis.

- Use tuplas para dados ordenados e imutáveis.

- Use conjuntos para dados não ordenados e únicos.

- Use dicionários para dados chave-valor.

- Use conjuntos ordenados para dados não ordenados e imutáveis.

**Documentação python**

3.1.3. Listas

Python inclui diversas estruturas de dados *compostas*, usadas para agrupar outros valores. A mais versátil é *list* (lista), que pode ser escrita como uma lista de valores (itens) separados por vírgula, entre colchetes. Os valores contidos na lista não precisam ser todos do mesmo tipo.

>>>

**>>>** quadrados = [1, 4, 9, 16, 25]

**>>>** quadrados

[1, 4, 9, 16, 25]

Como strings (e todos os tipos embutidos de [sequência](https://docs.python.org/pt-br/3.13/glossary.html#term-sequence)), listas pode ser indexados e fatiados:

>>>

**>>>** quadrados[0] *# indexação retorna o item*

1

**>>>** quadrados[-1]

25

**>>>** quadrados[-3:] *# fatiamento retorna uma nova lista*

[9, 16, 25]

As listas também suportam operações como concatenação:

>>>

**>>>** quadrados + [36, 49, 64, 81, 100]

[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

Diferentemente de strings, que são [imutáveis](https://docs.python.org/pt-br/3.13/glossary.html#term-immutable), listas são [mutáveis](https://docs.python.org/pt-br/3.13/glossary.html#term-mutable), ou seja, é possível alterar elementos individuais de uma lista:

>>>

**>>>** cubos = [1, 8, 27, 65, 125] *# algo errado aqui*

**>>>** 4 \*\* 3 *# o cubo de 4 é 64, não 65!*

64

**>>>** cubos[3] = 64 *# substitui o valor errado*

**>>>** cubos

[1, 8, 27, 64, 125]

Você também pode adicionar novos itens no final da lista, usando o *método* list.append() (estudaremos mais a respeito dos métodos posteriormente):

>>>

**>>>** cubos.append(216) *# adiciona o cubo de 6*

**>>>** cubos.append(7 \*\* 3) *# e o cubo de 7*

**>>>** cubos

[1, 8, 27, 64, 125, 216, 343]

A atribuição simples em Python nunca copia dados. Quando você atribui uma lista a uma variável, a variável se refere à *lista existente*. Quaisquer alterações que você fizer na lista por meio de uma variável serão vistas por todas as outras variáveis que se referem a ela:

>>>

**>>>** rgb = ["Vermelho", "Verde", "Azul"]

**>>>** rgba = rgb

**>>>** id(rgb) == id(rgba) *# elas referenciam o mesmo objeto*

True

**>>>** rgba.append("Alf")

**>>>** rgb

["Vermelho", "Verde", "Azul", "Alf"]

Todas as operações de fatiamento devolvem uma nova lista contendo os elementos solicitados. Isso significa que o seguinte fatiamento devolve uma [cópia rasa](https://docs.python.org/pt-br/3.13/library/copy.html#shallow-vs-deep-copy) da lista:

>>>

**>>>** rgba\_correto = rgba[:]

**>>>** rgba\_correto[-1] = "Alfa"

**>>>** rgba\_correto

["Vermelho", "Verde", "Azul", "Alfa"]

**>>>** rgba

["Vermelho", "Verde", "Azul", "Alf"]

Atribuição a fatias também é possível, e isso pode até alterar o tamanho da lista ou remover todos os itens dela:

>>>

**>>>** letras = ['a', 'b', 'c', 'd', 'e', 'f', 'g']

**>>>** letras

['a', 'b', 'c', 'd', 'e', 'f', 'g']

**>>>** *# substitui alguns valores*

**>>>** letras[2:5] = ['C', 'D', 'E']

**>>>** letras

['a', 'b', 'C', 'D', 'E', 'f', 'g']

**>>>** *# agora remove-os*

**>>>** letras[2:5] = []

**>>>** letras

['a', 'b', 'f', 'g']

**>>>** *# limpa a lista substituindo todos os elementos por uma lista vazia*

**>>>** letras[:] = []

**>>>** letras

[]

A função embutida [len()](https://docs.python.org/pt-br/3.13/library/functions.html" \l "len" \o "len) também se aplica a listas:

>>>

**>>>** letras = ['a', 'b', 'c', 'd']

**>>>** len(letras)

4

É possível aninhar listas (criar listas contendo outras listas), por exemplo:

>>>

**>>>** a = ['a', 'b', 'c']

**>>>** n = [1, 2, 3]

**>>>** x = [a, n]

**>>>** x

[['a', 'b', 'c'], [1, 2, 3]]

**>>>** x[0]

['a', 'b', 'c']

**>>>** x[0][1]

'b'

3.2. Primeiros passos para a programação

Claro, podemos usar o Python para tarefas mais complicadas do que somar 2+2. Por exemplo, podemos escrever o início da [sequência de Fibonacci](https://pt.wikipedia.org/wiki/Sequ%C3%AAncia_de_Fibonacci) assim:

>>>

**>>>** *# Sequência de Fibonacci:*

**>>>** *# a soma de dois elementos define a próxima*

**>>>** a, b = 0, 1

**>>> while** a < 10:

**...**  print(a)

**...**  a, b = b, a+b

**...**

0

1

1

2

3

5

8

Este exemplo introduz diversas características ainda não mencionadas.

* A primeira linha contém uma atribuição múltipla: as variáveis a e b recebem simultaneamente os novos valores 0 e 1. Na última linha há outro exemplo de atribuição múltipla demonstrando que expressões do lado direito são sempre avaliadas primeiro, antes da atribuição. As expressões do lado direito são avaliadas da esquerda para a direita.
* O laço de repetição [while](https://docs.python.org/pt-br/3.13/reference/compound_stmts.html" \l "while) executa enquanto a condição (aqui: a < 10) permanece verdadeira. Em Python, como em C, qualquer valor inteiro que não seja zero é considerado verdadeiro; zero é considerado falso. A condição pode também ser uma cadeia de caracteres ou uma lista, ou qualquer sequência; qualquer coisa com um tamanho maior que zero é verdadeiro, enquanto sequências vazias são falsas. O teste usado no exemplo é uma comparação simples. Os operadores padrões de comparação são os mesmos de C: < (menor que), > (maior que), == (igual), <= (menor ou igual), >= (maior ou igual) e != (diferente).
* O *corpo* do laço é *indentado*: indentação em Python é a maneira de agrupar comandos em blocos. No console interativo padrão você terá que digitar tab ou espaços para indentar cada linha. Na prática você vai preparar scripts Python mais complicados em um editor de texto; a maioria dos editores de texto tem facilidades de indentação automática. Quando um comando composto é digitado interativamente, deve ser finalizado por uma linha em branco (já que o interpretador não tem como adivinhar qual é a última linha do comando). Observe que toda linha de um mesmo bloco de comandos deve ter a mesma indentação.
* A função [print()](https://docs.python.org/pt-br/3.13/library/functions.html#print) escreve o valor dos argumentos fornecidos. É diferente de apenas escrever a expressão no interpretador (como fizemos anteriormente nos exemplos da calculadora) pela forma como lida com múltiplos argumentos, quantidades de ponto flutuante e strings. As strings são impressas sem aspas, e um espaço é inserido entre os itens, assim você pode formatar bem o resultado, dessa forma:

>>>

**>>>** i = 256\*256

**>>>** print('O valor de i é', i)

O valor de i é 65536

O argumento nomeado *end* pode ser usado para evitar uma nova linha após a saída ou finalizar a saída com uma string diferente:

>>>

**>>>** a, b = 0, 1

**>>> while** a < 1000:

**...**  print(a, end=',')

**...**  a, b = b, a+b

**...**

0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,