ESTRUTURA DE DADOS EM PYTHON

INTRODUÇÃO

Estrutura de Dados é o modo como os dados são organizados, armazenados e manipulados na memória, essas são as estruturas básicas que vamos ver, aprender e manipular ao decorrer da leitura.

Estrutura	Características	Exemplos
list	Sequência ordenada e mutável	[1, 2, 3]
tuple	Sequência ordenada e imutável	(1, 2, 3)
dict	Mapeamento chave-valor (hash table)	{'a': 1, 'b': 2}
set	Coleção de itens únicos e não ordenados	{1, 2, 3}
str	Sequência imutável de caracteres	"hello"

DETALHES TÉCNICOS: LEMBRE - TUDO EM PYTHON É UM OBJETO

- **list** - Implementada como vetor dinâmico (array dinâmico em C).
- **dict** - Implementado como hash table de alta performance.
- **set** - Internamente é um dicionário sem valores.

Classificação das Estruturas de Dados em Python:

- Sequenciais: indexados - possuem index, aquela organização sequencial de números inteiros que começa do 0 até o infinito positivo onde cada número revela uma posição do elemento da instância/objeto/elemento que está dentro da nossa estrutura de dados. Podemos colocar uma indexação negativa, mas neste caso estamos selecionando o inverso/oposto da lista.

Exemplos de estruturas indexadas: list, tuple,str.

index 0 1 2 3 4
Exemplo de indexação: lista_str = [‘a’,’b’,’c’,’d’,’e’]

puxando o elemento de index 4

print(lista_str[4]) #saída e

puxando um elemento com Slicing utilizando um index negativo

print(lista_str[-2]) # saída d, pegamos o penúltimo

- **Não-sequências:** não- indexados, esses a indexação é o próprio elemento, você pode posteriormente através de parâmetros colocar index, mas por padrão essa estrutura de dados não vem com index.

  Exemplos de estruturas de dados não-sequências: set,dict

  Exemplo com dicionários:

  discio = {“matéria”:[‘math’,’química’]}

  print(discio[“matéria”][:]) # saída [‘math’,’química’]

- **Mutáveis:** Objetos mutáveis em Python permitem que os programadores tenham objetos que podem alterar seus valores. Eles geralmente são utilizados para armazenar uma coleção de dados. Pode ser considerado como algo que sofreu mutação e o estado interno aplicável dentro de um objeto mudou, em um objeto mutável, o valor do objeto muda ao longo de um período de tempo.

  Exemplos de estruturas mutáveis(objetos): list, dict, set

  Exemplo de código:

```
numeros = [10, 20, 30]
numeros[0] = 99    # possível
```

R: veja que o elementos e indicie 0 da lista foi modificado para armazenar o número 99 na memória ram

- **Não-mutáveis:** Objetos imutáveis em Python podem ser definidos como objetos que não mudam seus valores e atributos ao longo do tempo, esses objetos tornam-se permanentes uma vez criados e inicializados, e formam uma parte crítica das estruturas de dados usadas em Python.

Exemplos de estruturas imutáveis(objetos): tuple, str, frozenset

Exemplo de código:

```
coordenadas = (10, 20)
# coordenadas[0] = 99  -> ERRO! (não pode alterar)
```

R: veja que não é possível remover, modificar, alterar os dados uma vez que foi inserido na tupla

TABELA DE COMPARAÇÃO GERAL DAS ESTRUTURAS DE DADOS BÁSICAS EM PYTHON

ESTRUTURA	ORDENADA	MUTÁVEIS	PERMITE DUPLICADOS?	DESCRIÇÃO
list (lista)	sim	sim	sim	Permite adicionar elementos repetidos. A ordem é preservada.
tuple (tupla)	sim	não	sim	Permite elementos repetidos, mas a tupla em si não pode ser alterada depois de criada.
str (string)	sim	não	sim	Caracteres podem se repetir em uma sequência de texto.
set (conjunto)	não	sim	não	Elimina automaticamente elementos repetidos. Ordem não garantida.
frozenset (conjunto congelado)	não	não	não	Igual ao set, mas é imutável. Também elimina elementos repetidos
dict (dicionário) - chaves	não	sim	não (nas chaves)	Cada chave precisa ser única; duplicações sobrescrevem valores antigos.

dict (dicionário) - valores	não	sim	sim (nos valores)	Os valores podem ser iguais entre diferentes chaves.

Comentário técnico: Quando falamos que uma estrutura "permite duplicados", significa que ela pode armazenar vários elementos iguais, ou seja, valores repetidos.

Duplicados = valores repetidos dentro da estrutura.

- Em sets e frozensets, duplicados são removidos automaticamente sem erro.
- Em dicionários, se você repete uma chave, o valor mais recente sobrescreve o anterior — não gera erro, apenas substitui silenciosamente.
- Listas, tuplas e strings aceitam duplicados porque podem ser necessários para representar sequências reais (ex.: nomes repetidos, letras repetidas).

Funções integradas do Python para melhorar a manipulação de dados em Estrutura de dados

As **funções integradas** do Python (*built-in functions*) são funções que já vêm prontas na linguagem — ou seja, você pode usá-las diretamente, sem precisar importar nenhum módulo ou biblioteca.

Elas foram criadas para realizar **tarefas comuns e frequentes** como:

- **conversão de tipos** (como `int()`, `str()`)
- **iteração** (como `enumerate()`, `zip()`)
- **cálculos matemáticos simples** (como `sum()`, `min()`, `max()`)
- **manipulação de coleções** (como `sorted()`, `len()`)
- **verificação de tipos** (`isinstance()`)

Funções Integradas Básicas - Essas funções são extremamente úteis na hora de manipular coleções como listas,tuplas, dicionários, sets, arrays e etc…

Função len(obj) - Retorna o número de elementos de um objeto sequencial ou coleção (listas, tuplas, dicionários, strings etc.).

Assinatura: def len(obj) -> int

Como funciona - Internamente, o Python chama o método especial `__len__()` do objeto. Se esse método não existir, lança `TypeError`.

Exemplo:

```
lista = [10, 20, 30]

print(len(lista))      # → 3

print(len("Python"))   # → 6
```

Dicas Avançadas:

Usar **len()** em dicionários retorna a quantidade de chaves. Evite chamar **len()** repetidamente em loops; armazene o resultado se for constante.

Função enumerate (iterable, start=0) - Produz um iterador de pares `(índice, valor)` a partir de qualquer iterável.

Assinatura : def enumerate(iterable, start: int = 0) -> Iterator[tuple[int, Any]]

Como funciona: A cada iteração, incrementa internamente um contador desde `start` e “zipa” com o valor extraído.

Exemplo:

```
for i, v in enumerate(["a", "b", "c"], start=1):
    print(i, v)
    # → 1 a
    #   2 b
    #   3 c
```

Função zip(*iterables) - Agrupa elementos de dois ou mais iteráveis, retornando um iterador de tuplas.

Assinatura: def zip(*iterables) -> Iterator[tuple]

Detalhe interno: Para de iterar quando o mais curto dos iteráveis se esgota.

Exemplo:

```
nomes = ["Ana", "Bia", "Carlos"]

idades = [28, 34, 19]

  for nome, idade in zip(nomes, idades):

      print(f"{nome} tem {idade} anos")
```

Uso Avançado: Para “descompactar” uma lista de tuplas em duas listas:

```
pares = [(1, 'a'), (2, 'b')]

nums, letras = zip(*pares)
```

Função reversed(seq) - Retorna um iterador que percorre a sequência em ordem inversa.

Assinatura: def reversed(seq) -> Iterator[Any]

Comportamento: Chama internamente `seq.__reversed__()` ou, na falta, percorre por índices de trás pra frente.

Exemplo:

```
for x in reversed([1, 2, 3]):

   print(x)   # → 3, 2, 1
```

Função sorted(iterable, *, key=None, reverse=False) - Retorna uma nova lista ordenada a partir de qualquer iterável.

Assinatura: def sorted(iterable, key: Callable = None, reverse: bool = False) -> list

Detalhes de performance: Usa o algoritmo Timsort (O(n log n) no pior caso), estável.

Exemplo:

```
frutas = ["uva", "abacaxi", "pera"]

print(sorted(frutas))  # - ['abacaxi', 'pera', 'uva']

print(sorted(frutas, key=len, reverse=True)) -  ['abacaxi', 'uva', 'pera']
```

Dica: Para ordenar dicionários por valor

```
d = {'a': 3, 'b': 1, 'c': 2}

ordenado = sorted(d.items(), key=lambda x: x[1])
```

Função sum(iterable, start=0) - Soma todos os elementos numéricos de um iterável, iniciando em `start`.

Assinatura: def sum(iterable, start: int | float = 0) -> int | float

Exemplo:

```
print(sum([1, 2, 3]))       # → 6

print(sum(range(5), 10))    # → 20  (10 + 0 + 1 + 2 + 3 + 4)
```

Função `min(*args)` / `max(*args)` - Retornam, respectivamente, o menor e o maior valor de um iterável ou de argumentos separados.

Assinatura: **def min(iterable, *,[key]) -> Any**

def max(iterable, *,[key]) -> Any

Exemplo:

```
print(min(5, 3, 8))                # → 3
print(max([10, 2, 7], key=lambda x: -x))  # → 2 (menor absoluto)
```

Função abs(x) - Retorna o valor absoluto de um número inteiro, float ou objeto que implemente `__abs__()`.

Exemplo:

```
print(abs(-7.5))  # → 7.5
```

Função round(number, ndigits=None) - Arredonda `number` para `ndigits` casas decimais (padrão inteiro mais próximo).

Detalhe: Usa “round half to even” (arredondamento bancário).

Exemplo:

```
print(round(2.675, 2))  # → 2.67
```

Função pow(x, y, mod=None) - Calcula `x**y`; se `mod` é fornecido, retorna `(x**y) % mod` de forma eficiente (útil em criptografia).

Exemplo:

```
print(pow(2, 10))       # → 1024
print(pow(2, 10, 1000))  # → 24
```

Função int(x, base=10) / float(x) / str(x)- Convertem `x` para inteiro, float ou string, respectivamente.

Exemplo: print(int("FF", base=16)) # → 255,

print(float("3.14")) # → 3.14,

print(str(100)) # → '100'

Função list(iterable) / tuple(iterable) / set(iterable)- Transformam iteráveis em lista, tupla ou conjunto (removendo duplicatas no caso de set).

Exemplo:

```
print(list("abc"))    # → ['a', 'b', 'c']

print(tuple([1,2]))   # → (1, 2)

print(set([1,2,2,3])) # → {1, 2, 3}
```

Função input(prompt=None) / print(*objects, sep=' ', end='\n') -

`input` - lê texto do usuário (devolve string).
`print` - exibe no console.

Exemplo:

```
nome = input("Seu nome: ")

print("Olá,", nome)
```

Função `id(obj)` / `dir(obj)` / `help(obj)` -

`id` retorna o identificador único (endereço na memória) de **`obj`**.

`dir` lista atributos e métodos disponíveis.
`help` exibe a docstring/documentação interativa

APROFUNDAMENTO ESPECÍFICO EM LISTAS

- Uma lista (list) é uma coleção ordenada e mutável de elementos. Pode conter tipos variados: números, strings, outras listas, objetos, etc. Logo são heterogêneos
- Usa colchetes [] para criação.

Exemplo de lista:

```
lista = [1, "texto", 3.14, True, [1,2,3]]
```

Características Principais:

- **Ordenada:** mantém a ordem de inserção
- **Mutável:** pode mudar com o tempo depois de criada
- **Permite duplicados:** permite elementos iguais armazenadas em sua estrutura
- **Tipos mistos:** Permite misturar vários tipos de dados mas cuidado para manter consistência

Principais Operações:

Criação - Exemplo de criação da lista:

```
vazia = [] # criando uma lista vazia
preenchida = [1, 2, 3] # criando uma lista preenchida com int
```

Acesso por índice - Exemplo de acesso por índice:

```
preenchida = [1, 2, 3] # criando uma lista preenchida com int
elemento = preenchida[0]  # Acessa o primeiro elemento
último = preenchida[-1]   # Acessa o último
```

Fatiamento (Slicing) - Exemplo de Slicing:

```
    preenchida = [1, 2, 3] # criando uma lista preenchida com int
    sublista = preenchida[1:3]  # Do índice 1 até o 2
```

Alterar elementos Exemplo de Alteração direta por índice:

```
preenchida[0] = 100  # Altera o primeiro elemento
```

Principais Funções/Métodos nativas para manipulação de listas

Método	Descrição	Exemplo
lista.append(x)	Adiciona x no final	lista.append(4)
lista.insert(i, x)	Insere x na posição i	lista.insert(2, "aqui")
lista.extend(iterável)	Junta outra lista ou iterável	lista.extend([7,8])
lista.remove(x)	Remove o primeiro x encontrado	lista.remove(3)
lista.pop([i])	Remove e retorna o elemento em i (último se não passar índice)	lista.pop(1)
lista.index(x)	Retorna o índice da primeira ocorrência de x	lista.index(100)
lista.count(x)	Conta quantas vezes x aparece	lista.count(1)
lista.sort()	Ordena a lista (modifica)	lista.sort()
lista.reverse()	Inverte a ordem (modifica)	lista.reverse()
lista.copy()	Retorna uma cópia da lista	nova_lista = lista.copy()
lista.clear()	Remove todos os elementos	lista.clear()

Métodos/Funções: são funções internas que pertencem a um objeto (no caso, o objeto é a lista). Estes métodos servem para manipular o estado da lista: ou seja, adicionar, remover, modificar, buscar ou reorganizar os elementos.

Definição profissional: Métodos de lista são operações encapsuladas que modificam o conteúdo ou a estrutura da lista, ou extraem informações dela, utilizando a sintaxe de chamada de método (lista.metodo())

- Eles não são funções isoladas (como print() ou len()), são ligados ao tipo de dado.
- A lista sabe como se comportar internamente quando você chama esses métodos.
- Eles usam a notação de ponto (.) porque estão associados a um objeto específico

MANIPULAÇÃO DE LISTAS COM PYTHON

Percorrendo Listas (Estruturas de Repetição):

```
nomes = ["Ana", "Carlos", "Bianca"]

for nome in nomes:
   print(nome)
```

Explicação: O for pega cada elemento da lista e executa o bloco de código. Pode ser combinado com if (tomada de decisão).

```
        nomes = ["Ana", "Carlos", "Bianca"]

for nome in nomes:
   print(nome)
        if nome == "Ana":
                print(“Anna é a melhor”)
```

Tomadas de Decisão em Listas (if dentro do for)

```
numeros = [1, 2, 3, 4, 5, 6]

for num in numeros:
   if num % 2 == 0:
      print(f"{num} é par")
   else:
      print(f"{num} é ímpar")
```

indexação e Slicing

```
# Acessando elementos
print(numeros[0])    # 1
print(numeros[-1])   # 3 (índice negativo: conta de trás para frente)

# Fatiamento
print(numeros[0:2])  # [1, 2]  (do índice 0 até 1)
print(numeros[::2])  # [1, 3]  (pegando de 2 em 2)
```

Dica técnica:

- lista[::n] → avança de n em n.
- lista[::-1] → inverte a lista rapidamente.

Copy vs Referência - CUIDADO ao copiar listas!

Exemplo:

```
# Copiando errado (aponta para mesma lista)
a = [1, 2, 3]
b = a
b.append(4)
print(a)  # [1, 2, 3, 4]  -> mudou 'a' também

# Copiando corretamente
a = [1, 2, 3]
b = a.copy()   # OU b = a[:]
b.append(4)
print(a)  # [1, 2, 3]
```

Operações comuns

```
# Soma dos elementos
print(sum([1, 2, 3]))  # 6

# Maior / Menor
print(max([1, 2, 3]))  # 3
print(min([1, 2, 3]))  # 1

# Verificar elemento
print(2 in [1, 2, 3])  # True
```

Técnicas Avançadas

List Comprehension (muito usado na prática): É uma forma compacta e rápida de criar novas listas a partir de uma existente, aplicando transformação ou filtragem diretamente.

Exemplo 1 — Transformação rápida:

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

```
# Elevar todos ao quadrado
quadrados = [x**2 for x in numeros]
print(quadrados)  # [1, 4, 9, 16, 25]
```

Explicação passo a passo:

- Para cada x dentro da lista numeros, eleva x ao quadrado (x**2).
- A lista final (quadrados) é criada automaticamente.
- formato geral: [nova_expressão for elemento in coleção]

Exemplo 2 — Filtragem rápida:

```
numeros = [1, 2, 3, 4, 5, 6]

# Pegar apenas números pares
pares = [x for x in numeros if x % 2 == 0]
print(pares)  # [2, 4, 6]
```

Explicação passo a passo:

- Percorre a lista numeros.
- Somente inclui x na nova lista se x % 2 == 0 (ou seja, número par).
- formato geral: [nova_expressão for elemento in coleção if condição]

Iterar com Índice e Valor (enumerate) - Exemplo:

```
numeros = [10, 20, 30]
for indice, valor in enumerate(numeros):
    print(f"Índice {indice}: Valor {valor}")
```

Explicação passo a passo:

- enumerate(lista) gera pares (índice, valor).
- Você pode acessar onde o elemento está (posição) e o próprio valor ao mesmo tempo.

Saída:

```
Índice 0: Valor 10
Índice 1: Valor 20   # esses são o resultado do enumerete
Índice 2: Valor 30
```

Juntar Duas Listas (zip) - Exemplo:

```
a = [1, 2, 3]
b = ['um', 'dois', 'três']

for numero, palavra in zip(a, b):
    print(numero, palavra)
```

Explicação passo a passo:

- zip(a, b) combina os elementos das duas listas em pares.
- O for percorre simultaneamente: primeiro item de a com primeiro item de b, e assim por diante.

Criar uma Matriz - Exemplo:

```
matriz = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
```

Explicação passo a passo:

- Uma lista que contém outras listas.
- Cada linha é uma lista: [1, 2, 3], [4, 5, 6], etc.
- Ideal para representar tabelas, grades, jogos, imagens, etc.
- Para acessar os elementos da matriz criada basta: print(matriz[0][1]) # Elemento linha 0, coluna 1 => 2

Iterar sobre a Matriz - Exemplo:

```
for linha in matriz:
    for valor in linha:
        print(valor, end=' ')
```

Explicação passo a passo:

- Primeiro for linha in matriz: — percorre cada linha (ou seja, uma lista dentro da matriz).

- Depois for valor in linha: — percorre cada valor dentro daquela linha.

- print(valor, end=' ') — imprime os valores na mesma linha separados por espaços (em vez de pular linha a cada print).

Realizando cálculos entre listas:

Para operações element-wise (isto é, realizando o cálculo entre o elemento da posição i de uma lista e o elemento da mesma posição em outra lista), você pode usar a função zip() junto com list comprehensions.

Exemplo:

```
lista1 = [1, 2, 3, 4, 5]
lista2 = [10, 20, 30, 40, 50]

soma = [a + b for a, b in zip(lista1, lista2)]
print("Soma:", soma)  # Output: [11, 22, 33, 44, 55]
```

Esse modelo não serve para realizar cálculos com listas de tamanhos diferentes, pois iremos iterar/percorrer até o fim da lista de menor tamanho. Ou seja, funciona bem quando ambas as listas têm o mesmo tamanho ou quando você quer iterar sobre o menor tamanho, descartando os demais.

O zip é uma função interna do Python que combina iterável (como listas, tuplas, etc.) de forma paralela, produzindo um iterador de tuplas. Cada tupla contém um elemento de cada iterável, na mesma posição. Essa função é extremamente útil quando você precisa iterar sobre múltiplas sequências ao mesmo tempo de forma sincronizada.

Serve também Para outros cálculos, basta trocar a operação matemática:

- **Produto:** [a * b for a, b in zip(lista1, lista2)]

- **Diferença:** [a - b for a, b in zip(lista1, lista2)]

- **Divisão:** [a / b for a, b in zip(lista1, lista2) if b != 0]

Usando Map e Lambda para realizar operações matemáticas entre duas listas
Outra abordagem funcional é utilizar a função map(). Essa técnica pode ser mais concisa em alguns casos

Exemplo:

```
lista1 = [1, 2, 3, 4, 5]
lista2 = [10, 20, 30, 40, 50]

soma = list(map(lambda x, y: x + y, lista1, lista2))
print("Soma usando map:", soma)
```

O map é uma função embutida do Python que aplica uma função a cada item de um iterável (ou de vários iteráveis) e retorna um iterador com os resultados. Ele é muito útil quando você quer transformar todos os elementos de uma coleção sem precisar escrever explicitamente um loop. Sintaxe básica: map(função, iterável1, iterável2, ...)

função: uma função que será aplicada a cada item do(s) iterável(is). Essa função deve aceitar o mesmo número de argumentos dos iteráveis passados.

iterável(s): o(s) iterável(is) (como listas, tuplas, etc.) cujos elementos serão passados como argumentos para a função.

O map retorna um objeto iterável (um map object) que pode ser convertido em uma lista, tupla ou qualquer outra estrutura, conforme a necessidade.

Função Lambda, uma função nativa do python. Funções lambda (ou funções anônimas) são funções pequenas, definidas de forma concisa, sem a necessidade de criar um identificador formal (nome) para elas.

Elas são muito úteis quando você precisa de uma função simples para usar rapidamente, por exemplo, em operações de ordenação, filtragem e mapeamento.

Sintaxe básica: lambda argumentos : expressão

lambda: é a palavra-chave para definir a função.

argumentos: são os parâmetros que a função recebe, separados por vírgulas.

expressão: é a única instrução que a função executa e cujo resultado será retornado automaticamente.

Utilizando o NumPy para Operações Vetorizadas - Listas são estruturas dinâmicas, flexíveis e heterogêneas que se comportam de forma similar a vetores dinâmicos, mas não são estritamente vetores no sentido de serem homogêneos e otimizados para cálculos matemáticos. Ou seja, listas são estruturas que possuem Heterogeneidade, ou seja, podem conter diferentes tipos de dados(elementos) armazenados na memória ram e isso tudo é de forma dinâmica.

Já Vetores não possuem essa característica, eles aceitam somente um tipo de dado e possuem tamanho fixo na memória, exceto se eles foram dinâmicos

Arrays/Vetores NumPy: São a escolha recomendada quando você precisa de vetores (ou matrizes) para cálculos numéricos e manipulação de dados com alta performance. Para cálculos numéricos eficientes com NumPy, é importante que os arrays sejam "retangulares" e tenham dimensões compatíveis ou aptas para o broadcasting.

Se estiver lidando com listas de tamanhos diferentes, você precisará ajustar ou transformar esses dados para que possam ser processados adequadamente. Ou seja, as listas devem ter tamanhos iguais ou compatíveis.

Exemplo:

```
import numpy as np

a = np.array([1, 2, 3])
b = np.array([10, 20, 30])
print(a + b)  # Resultado: [11, 22, 33]
```

Usando itertools.zip_longest (operações matemáticas com listas de tamanho diferentes)

Com zip_longest do módulo itertools você pode iterar sobre as listas até o final da maior delas, preenchendo os "vazios" com um valor padrão (por exemplo, 0). Assim, garante que cada iteração tenha um par de elementos para o cálculo. Para realizar cálculos element-wise (ou seja, operações que envolvem os elementos correspondentes) entre listas de tamanhos diferentes, você precisa decidir como tratar os "buracos" (valores ausentes) da lista menor.

Exemplo:

```
from itertools import zip_longest
```

```
lista1 = [1, 2, 3, 4]
lista2 = [10, 20, 30]

# Usando fillvalue=0 para preencher a lista menor

resultado = [a + b for a, b in zip_longest(lista1, lista2, fillvalue=0)]
print(resultado)  # Saída: [11, 22, 33, 4]
```

Se preferir trabalhar com NumPy, você pode converter suas listas para arrays, mas é preciso que eles tenham o mesmo tamanho para realizar operações vetorizadas. Uma abordagem é "preencher" a lista menor com um valor (como 0) até que ela tenha o mesmo tamanho da maior.

Exemplo:

```
import numpy as np

lista1 = [1, 2, 3, 4]
lista2 = [10, 20, 30]

# Determinar o tamanho máximo
max_len = max(len(lista1), len(lista2))

# Preencher as listas com 0 até terem o mesmo tamanho
pad_lista1 = lista1 + [0] * (max_len - len(lista1))
pad_lista2 = lista2 + [0] * (max_len - len(lista2))

# Converter para arrays NumPy
array1 = np.array(pad_lista1)
array2 = np.array(pad_lista2)

soma = array1 + array2
print(soma)  # Saída: [11 22 33 4]
total = sum(soma)
print(total) # 70
```

Listas Aninhadas - Basicamente Listas Aninhadas são listas dentro de outras listas, um exemplo clássico de listas aninhadas são as matrizes que criamos com listas.

APROFUNDAMENTO ESPECÍFICO EM TUPLAS

Uma tupla é uma coleção imutável de elementos ordenados e heterogêneos (podem ter tipos diferentes) de dados na memória ram. Uma vez criada, não pode ser alterada (não dá para adicionar, remover ou alterar elementos).

Criando um tupla:

```
# Com parênteses
tupla1 = (1, 2, 3)

# Sem parênteses (não é recomendado em produção por clareza)
tupla2 = 1, 2, 3

# Tupla com um único elemento: obrigatório usar vírgula
tupla3 = (1,)

# Tupla vazia
tupla4 = ()

# Convertendo uma lista em tupla
lista = [1, 2, 3]
tupla5 = tuple(lista)
```

Características Principais:

- **Ordenada:** Mantém a ordem de inserção dos elementos
- **Imutáveis:** Não permite alteração após a criação
- **Heterogêneas:** Pode conter vários tipos diferentes de dados
- **Indexável e fatiável:** Pode acessar via índices e fazer slicing
- **Iterável:** Pode ser percorrida em loops (For, While)

Operações Básicas de Tuplas:

```
# Criando uma tuple
tupla = (10, 20, 30, 40)

# Acessar elemento
print(tupla[0])  # 10

# Fatiar (slicing)
print(tupla[1:3])  # (20, 30)
```

```
# Concatenar tuplas
nova_tupla = tupla + (50, 60)
print(nova_tupla)  # (10, 20, 30, 40, 50, 60)

# Repetir tupla
repetida = tupla * 2
print(repetida)  # (10, 20, 30, 40, 10, 20, 30, 40)

# Verificar elemento
print(20 in tupla)  # True
```

Métodos de Tuplas

Apesar de ser imutável, tupla possui métodos para consulta:

Método	Descrição
.count(x)	Retorna quantas vezes x aparece
.index(x)	Retorna o índice da primeira ocorrência de x

APROFUNDAMENTO ESPECÍFICO EM DICIONÁRIOS

Os **dicionários** (`dict`) são uma das estruturas de dados mais versáteis e poderosas em Python. Permitem o armazenamento de pares *chave* → *valor*, com acesso, inserção e remoção em tempo médio constante (O(1)).

Características Principais

1. **Mapeamento chave→valor**: permite associar uma chave única a um valor (qualquer objeto).
2. **Hash table**: usa hashing para mapear chaves a índices internos.
3. **Complexidade de acesso**:
    - **Média**: O(1) para d[key], in, inserção e remoção.
    - **Pior caso**: O(n) se muitas colisões ocorreram (raro, graças a redimensionamento dinâmico).

4. **Ordem de inserção**: a partir do Python 3.7, dict preserva a ordem em que os itens foram inseridos

Operações Básicas

Acesso a valor:

```
valor = d['chave']       # KeyError se não existir
valor = d.get('chave')   # None se não existir
valor = d.get('chave', default)
```

Inserção e atualização:

```
d['nova_chave'] = valor
```

Remoção:

```
d.pop('chave')          # retorna valor, KeyError se não existir
d.pop('chave', default)  # retorna default se não existir
del d['chave']          # KeyError se não existir
d.clear()               # remove todos os itens
```

Verificações:

```
'chave' in d          # True/False
len(d)                # número de pares
```

Formas de Criar um Dicionário:

Literal - Usando chaves **{ chave:valor, chave:valor, … , chave:valor },** Chaves devem ser imutáveis: `int`, `str`, `tuple` (desde que contenha apenas imutáveis). Listas e outros dicts não são permitidos.

Exemplo:

```
usuario = {'nome': 'Ana', 'idade': 25, 'cidade': 'SP'}
```

Usando o construtor/função integrada dict(Chave = Valor) - Recebe pares nome=valor ou um iterável de tuplas:

Exemplo:

```
params = dict(altura=1.75, peso=68)
coords = dict([('x', 10), ('y', 20)])
```

A partir de sequência de pares

Exemplo:

```
pares = [('x', 10), ('y', 20)]
coordenadas = dict(pares)
```

Construção via zip e Outras Formas - A partir de duas listas:

Exemplo:

```
chaves = ['a','b','c']
valores = [1,2,3]

d = dict(zip(chaves, valores))
```

Dicionário Vazio - Simplesmente criá-lo vazio para alimentar depois

Exemplo:

```
d = {}
d = dict()
```

Principais Métodos Para Manipular Dicionário

Método	Descrição
keys()	view de chaves (dict_keys)
values()	view de valores (dict_values)
items()	view de pares (dict_items)
get(key[, default])	acesso seguro, retorna default se não existir
pop(key[, default])	remove e retorna valor; default ou KeyError se faltar
popitem()	remove último par inserido (LIFO desde Python 3.7)
update(...)	mescla outro dict ou iterável de pares
setdefault(key, v)	insere v se key não existir e retorna d[key]
clear()	remove todos os itens
copy()	retorna uma shallow copy (mesmos objetos-valor, mas novo container)

Explicação detalhada de operações no dicionário

Acesso Direto pela Chave d[key]:

Ao usar `d['chave']`, o Python **busca** diretamente no hash table o bucket correspondente ao hash da chave, retornando o valor associado em **tempo médio O(1)**.
Se a chave não existir, é levantada uma **exceção** `KeyError`, um subtipo de `LookupError`, indicando que o mapeamento não contém a chave solicitada.

Exemplo:

```
ages = {'Jim': 30, 'Pam': 28}
print(ages['Pam'])   # → 28
print(ages['Michael'])  # KeyError: 'Michael'
```

Acesso Seguro: `dict.get()`

Com d.get('chave'), o método retorna None se a chave faltar, em vez de lançar exceção, mantendo o código mais robusto em cenários incertos.
É possível fornecer um **valor padrão**: `d.get('chave', default)`, retornando `default` quando a chave não existir

Exemplo:

```
config = {'timeout': 10}
print(config.get('timeout'))      # → 10
print(config.get('retry', 3))     # → 3
```

Inserção e Atualização

Atribuição direta - Se `nova_chave` não existir, é criada em tempo O(1); se existir, seu valor é sobrescrito.

Exemplo:

```
d['nova_chave'] = valor
```

Uso de update() para várias chaves de uma vez

Exemplo:

```
d.update({'a':1, 'b':2})
```

Aceita **outro dict** ou **iterável de pares** (tuplas), mesclando-os em d

Exemplo:

```
user = {}
user['name'] = 'Alice'      # cria
user['age'] = 25            # cria
user['age'] = 26            # atualiza
user.update({'city':'SP'})  # adiciona city
```

Remoção de Itens

pop(key[, default])

Remove a chave especificada e retorna seu valor.

Se a chave faltar e default não for fornecido, lança KeyError; caso contrário, retorna default.

Exemplo:

```
nums = {1:'one', 2:'two'}
v = nums.pop(2)          # v == 'two'
v2 = nums.pop(3, 'zz')     # v2 == 'zz', sem erro
```

Verificações e Consultas

Teste de existência: `key in d`

Retorna True se key for uma das chaves de d, caso contrário False.

Também O(1) médio, pois usa hashing para checar presença no bucket.

`Exemplo:`

```
d = {'x':1}
print('x' in d)   # → True
print('y' in d)   # → False
```

Tamanho: len(d)

Devolve o número de pares armazenados em d.

Implementado internamente como atributo de contagem, portanto O(1)

Exemplo:

```
print(len({'a':1, 'b':2, 'c':3}))  # → 3
```

Iteração e View em Hash Table (Dicionário):

Exemplo:

```
for chave in usuario:                  # itera sobre chaves
    print(chave, usuario[chave])

for valor in usuario.values():         # só valores
    print(valor)

for chave, valor in usuario.items():   # pares
    print(f"{chave} → {valor}")
```

Views dinâmicas: `.keys()`, `.values()`, `.items()` refletem alterações no dict sem criar cópias.

Operações de conjunto nos Dicionário

```
a = {'x':1, 'y':2, 'z':3}
b = {'w':0, 'x':9, 'y':5}
```

Interseção de chaves - Retorna um set de chaves que estão **em ambos** os dicionários. Útil para, por exemplo, encontrar campos compartilhados em configurações ou esquemas de dados

Exemplo:

```
comuns = a.keys() & b.keys()    # {'x', 'y'}
```

União de chaves - Produz o conjunto de **todas** as chaves presentes em ao menos um dos dicionários, servindo para agregar campos de múltiplas fontes sem duplicação

Exemplo:

```
todas = a.keys() | b.keys()     # {'w','x','y','z'}
```

Diferença de chaves - Excelente para detectar campos obsoletos ou faltantes entre estruturas relacionadas

Exemplo:

```
diff = a.keys() - b.keys()      # {'z'}
```

União de pares (mantém pares idênticos) - Combina *todos* os pares de ambos os dicionários, eliminando duplicatas exatas

Exemplo:

```
pares = a.items() | b.items()
```

Exemplo Prático:

a = {'x':1, 'y':2, 'z':3}

b = {'w':0, 'x':9, 'y':5}

Interseção de chaves

comuns = a.keys() & b.keys() # {'x', 'y'}

União de chaves

todas = a.keys() | b.keys() # {'w','x','y','z'}

Diferença de chaves

diff_ab = a.keys() - b.keys() # {'z'}

diff_ba = b.keys() - a.keys() # {'w'}

Diferença simétrica - Conjunto de chaves que estão em **apenas um** dos dicionários. Ideal para auditoria rápida de divergências

sym_diff = a.keys() ^ b.keys() # {'w','z'}

Interseção de itens

```
comuns_itens= a.items() & b.items()    # set()


# União de itens

todos_itens = a.items() | b.items()    # {('x',1),('y',2),('z',3),('w',0),('x',9),('y',5)}


# Diferença de itens

diff_itens  = a.items() - b.items()    # {('x',1),('y',2),('z',3)}
```

APROFUNDAMENTO ESPECÍFICO EM SETS

set é uma coleção **não ordenada** de elementos **únicos**, implementada internamente por uma **hash table**. Sua principal vantagem é permitir operações de teoria de conjuntos (união, interseção, diferença, diferença simétrica) de forma eficiente (tempo médio O(1) para inserção, remoção e teste de pertença). Você os usa sempre que precisar garantir unicidade de itens, remover duplicatas ou executar cálculos de conjunto em grandes coleções de dados.

Quando usar SETS?

Sets são estruturas de dados que armazenam valores sem ordem e sem repetições.

Use-os quando:

- **Garantir unicidade:** eliminar duplicatas de listas ou outras coleções.
- **Teste rápido de pertença:** checar se um elemento existe em O(1) médio.
- **Operações de teoria de conjuntos:** união, interseção, diferença, diferença simétrica.
- **Filtragem e agrupamento:** extrair valores únicos em processamento de dados

Principais Métodos Para Manipular Sets

Método	Descrição
add(elem)	Adiciona elem ao set. Ignora se já existir. **O(1)** médio
remove(elem	Remove elem. Lança KeyError se não existir. **O(1)** médio
discard(elem)	Remove elem se existir; não faz nada caso não exista (não lança erro). **O(1)** médio
pop()	Remove e retorna um elemento arbitrário. Lança KeyError se o set estiver vazio. **O(1)** médio
clear()	Remove todos os elementos, deixando o set vazio. **O(n)** para esvaziar completamente

Operações Básicas com Sets

Criação de Sets - Formal Literal, Forma com Construtor, Apartir de outras Coleções como : listas, tuplas.

Exemplo:

```
s = {1, 2, 3, 2}  # forma literal
print(s)  # → {1, 2, 3}

# forma literal, veja que é o mesmo símbolo do dicionário só que sem chave:valor

# Usando o construtor set()

s = set([1, 2, 3, 2])
print(s)       # → {1, 2, 3}
s2 = set()     # set vazio

# Ele aceita Slicing também
```

Apartir de outras coleções de dados como: lista, tuple

```
lista = [1,2,2,3,4]
s = set(lista)  # elimina duplicatas
tupla = (5,5,6)
t = set(tupla)
```

Lembrando que essa coleção não aceita duplicadas/dados duplicados

Inserção ou Remoção de elementos nos set()

Exemplo:

```
s.add(4)        # adiciona 4 (se já existir, ignora)
s.remove(2)     # remove 2, KeyError se não existir
s.discard(10)   # remove se existir, não erro se não
s.pop()         # remove e retorna um elemento arbitrário
s.clear()       # esvazia o set
```

Teste de presença

Exemplo:

```
if x in s:
    print("existe")
```

Tamanho

Exemplo:

```
print(len(s))
```

Operação de Conjunto em Set - Basicamente é a mesma coisa em dicionário

```
A | B   # união: {1,2,3,4}
A & B   # interseção: {2,3}
A - B   # diferença: {1}
A ^ B   # diferença simétrica: {1,4}
```

Principais Métodos Avançados em Set()

Método	Descrição
s.update(iterable)	adiciona todos os elementos do iterável
s.intersection_update(t)	fica somente com elementos em ambos
s.difference_update(t)	remove elementos de `s` que estão em `t`
s.symmetric_difference_update(t)	atualiza para diferença simétrica
s.isdisjoint(t)	`True` se `s` e `t` não têm elementos em comum
s.issubset(t)	`True` se todo elemento de `s` está em `t`
s.issuperset(t)	`True` se `t` inteiro está em `s`

Iteração em Sets

Exemplo:

```
for elem in s:
    print(elem)
```

APROFUNDAMENTO ESPECÍFICO EM STRING (STR)

Em Python, uma **string** é uma sequência **imutável** de caracteres Unicode. Você pode pensar nela como um array unidimensional de caracteres, porém com funcionalidades especializadas para textos.

- **Imutabilidade**: uma vez criada, você **não** pode alterar os caracteres de uma string em-lugar; qualquer “modificação” gera uma nova string.

- **Unicode**: suporta textos em praticamente qualquer escrita (latim, cirílico, chinês, emojis, etc).

Criação de strings

Forma literal

Use aspas simples '...' ou duplas "..." indistintamente.

Para strings que contêm aspas internas, alterne o tipo ou use escape (\" ou \').

Exemplo:

```
s1 = "Olá, mundo!"
s2 = 'Python é divertido'
```

Stings Multilinhas:

```
s3 = """Esta é uma
string em múltiplas
linhas."""
```

Raw Strings

```
path = r"C:\Users\Nome\Desktop"
```

Prefixo r desativa escapes, útil para expressões regulares ou caminhos no Windows.

Indexação e slicing - Strings suportam acesso por índices e fatiamento

Exemplo:

```
s = "Python"

# Indexação
print(s[0])   # 'P'
print(s[-1])  # 'n'

# Slicing (start:stop:step)
print(s[1:4])   # 'yth'
print(s[:3])    # 'Pyt'
print(s[3:])    # 'hon'
print(s[::2])   # 'Pto'
```

Índices negativos contam de trás para frente.

Slice omite `start` ou `stop` para início/fim da string.

Operações básicas em Strings

Concatenação - Une duas ou mais strings numa única. O operador `+` cria uma nova string, deixando as originais inalteradas

Exemplo:

```
s1 = "Olá, "
s2 = "mundo!"
s = s1 + s2      # → "Olá, mundo!"
```

Repetição - Repete uma string um número inteiro de vezes

Exemplo:

```
s = "Aha" * 3    # → "AhaAhaAha"
```

Fatiamento (Slicing) - Extrai substrings usando **`start:stop:step`**:

```
s = "abcdef"
print(s[1:4])    # → 'bcd'    (índices 1,2,3; 4 é exclusivo)
print(s[:3])     # → 'abc'
print(s[3:])     # → 'def'
print(s[::2])    # → 'ace'    (step = 2)
```

Omitir `start` ou `stop` faz o slice ir ao início ou fim da string, respectivamente

Comprimento - Usa a função built-in `len()` para obter o número de caracteres:

```
s = "Olá!"
print(len(s))    # → 4
```

Operador de Pertença - Verifica se uma substring está contida na string principal:

```
s = "banana"
print("nan" in s)   # → True
```

```
print("xyz" not in s) # → True
```

O teste `in` faz busca O(n) na pior hipótese (escaneia a string)

Iteração sobre Strings - Forma básica de percorrer uma String

Exemplo:

```
s = ” Seu Pai ta On”

for elemento in s:
        print(elemento, end =’ ’)
```

Principais métodos de string

Strings em Python têm muitos métodos úteis. Como são imutáveis, cada método retorna **nova** string:

Método	Descrição	Exemplo
s.upper()	Converte para maiúsculas	"abc".upper() → 'ABC'
s.lower()	Converte para minúsculas	"ABC".lower() → 'abc'
s.capitalize()	Primeira letra maiúscula, resto minúsculo	"python".capitalize() → 'Python'
s.title()	Inicia cada palavra com maiúscula	"ola mundo".title() → 'Ola Mundo'
s.strip()	Remove espaços (ou outros) nas extremidades	" oi ".strip() → 'oi'
s.split(sep=None)	Divide em lista de substrings pelo separador	"a,b,c".split(",") → ['a','b','c']
sep.join(iterable)	Junta elementos com sep entre eles	"-".join(["2025","05","02"]) → '2025-05-02'
s.replace(old, new)	Substitui todas ocorrências de old por	"spam".replace("a","o") → 'spom'

	new	
s.find(sub)	Índice da primeira ocorrência de sub ou -1	"hello".find("l") → 2
s.startswith(pref)	True se inicia com pref	"test".startswith("te") → True
s.endswith(suf)	True se termina com suf	"test".endswith("st") → True
s.isnumeric()	True se todos os caracteres são numéricos	"123".isnumeric() → True
s.isalpha()	True se todos os caracteres são letras	"abc".isalpha() → True

FORMATAÇÃO DE STRINGS

Abaixo segue um panorama aprofundado das três principais formas de formatação de strings em Python — **operador %**, **método str.format()** e **f-strings** — explicando suas sintaxes, capacidades, diferenças internas e casos de uso recomendados.

Em resumo, o operador **%** (estilo C) é antigo, simples e familiar, mas limitado e menos seguro; o método **str.format()** (PEP 3101) trouxe maior flexibilidade com placeholders posicionais e nomeados, além de especificadores de formato detalhados; por fim, as **f-strings** (PEP 498) combinam concisão, performance e poderosos recursos de interpolação em tempo de execução, sendo hoje a abordagem preferida para formatação em Python 3.6+.

OPERADOR % e CONVERSORES COMUNS

Especificador	Tipo de Dado
%s	String via str()
%d / %i	Inteiro / decimal
%f	Ponto flutuante (default 6 casas)
%x / %X	Hexadecimal (minúsculo/maiúsculo)
%%	Literal %

Exemplo:

```
print("Olá, %s! Você tem %d anos." % (name, age))
```

`%s` converte o valor passado em string

`%d` formata inteiros decimais

Método `str.format()` (PEP 3101)

Posicional: `{}` substituído em ordem
Indexado: `"{1}, {0}".format('A', 'B')` → `'B, A'`

Exemplo:

```
print("Olá, {}! Você tem {} anos.".format(name, age))
```

Possui Placeholders nomeados - Melhora legibilidade e evita erros de ordem

Exemplo:

```
print("Usuário: {nome}, Idade: {idade}".format(nome=name, idade=age))
```

Possui Especificadores de formato (`format spec`) -

Exemplo:

```
# número com 2 casas decimais
"{:.2f}".format(pi)          # → '3.14'
# preenchimento e alinhamento
"{:0>5d}".format(42)         # → '00042'
```

Vantagem e Desvantagem de usar o .format()

- **Vantagens:** placeholders nomeados, formatação rica (alinhamento, padding, sinal, separador de milhares)
- **Desvantagens:** verbosidade, execução de parsing em tempo de execução torna-o ligeiramente mais lento que `%` ou f-strings

f-Strings (PEP 498, Python 3.6+) - Avalia expressões dentro de `{}` em tempo de execução

Exemplo:

```
f"Olá, {name}! Você tem {age} anos."
```

Possui Expressões embutidas - Pode chamar funções e métodos diretamente

Exemplo:

```
f"{2+3} é cinco"
f"{user.upper()}"
```

Possui Especificadores de formato - Mesmos especificadores do `format()` (mini-linguagem)

Exemplo:

```
f"Pi aproximado: {pi:.2f}"     # → 'Pi aproximado: 3.14'
f"{value:0>8X}"                # hexadecimal maiúsculo, width=8, fill '0'
```

Métodos de análise e busca para String

Método	Descrição
s.count(sub)	Conta quantas vezes `sub` aparece
s.index(sub)	Índice de `sub` ou `ValueError` se não existir
s.rfind(sub)	Última ocorrência de `sub`
s.partition(sep)	Divide em tupla `(antes, sep, depois)`

FONTES:

- Objetos mutáveis e imutáveis em Python {EXEMPLOS}
- estruturas-de-dados.pdf
- A jornada do programa - Pense em Python