

Listas, tuplas, strings, dicionários e conjuntos

MC102

Listas

Relembrando

Listas guardam vários valores em ordem e são **mutáveis**: podemos inserir, remover e trocar elementos depois que a lista já foi criada.

Relembrando

Listas guardam vários valores em ordem e são **mutáveis**: podemos inserir, remover e trocar elementos depois que a lista já foi criada.

Criando e adicionando no final

```
1 nomes = ["Ana", "Bia"]
2 nomes.append("Caio")
3
4 print(nomes)
5 # ['Ana', 'Bia', 'Caio']
```

Relembrando

Listas guardam vários valores em ordem e são **mutáveis**: podemos inserir, remover e trocar elementos depois que a lista já foi criada.

Criando e adicionando no final

```
1 nomes = ["Ana", "Bia"]
2 nomes.append("Caio")
3
4 print(nomes)
5 # ['Ana', 'Bia', 'Caio']
```

Consultando a lista

```
1 nomes = ["Ana", "Bia", "Caio"]
2
3 print(len(nomes))      # 3
4 print("Bia" in nomes)  # True
5 print("Duda" in nomes) # False
```

Relembrando

Listas guardam vários valores em ordem e são **mutáveis**: podemos inserir, remover e trocar elementos depois que a lista já foi criada.

Criando e adicionando no final

```
1 nomes = ["Ana", "Bia"]
2 nomes.append("Caio")
3
4 print(nomes)
5 # ['Ana', 'Bia', 'Caio']
```

Consultando a lista

```
1 nomes = ["Ana", "Bia", "Caio"]
2
3 print(len(nomes))      # 3
4 print("Bia" in nomes)  # True
5 print("Duda" in nomes) # False
```

- `len(lista)` devolve a quantidade de elementos.
- O operador `in` testa pertencimento e devolve `True` ou `False`.

Exercício

</> Code

Faça uma função que recebe uma lista de inteiros e devolve o índice do maior elemento.

Exercício

</> Code

Faça uma função que recebe uma lista de inteiros e devolve o índice do maior elemento.

```
1 def min_indice(lista: list[int]) -> int:
```

Exercício

</> Code

Faça uma função que recebe uma lista de inteiros e devolve o índice do maior elemento.

```
1 def min_indice(lista: list[int]) -> int:  
2     menor = 0  
3     n = len(lista)
```

Exercício

</> Code

Faça uma função que recebe uma lista de inteiros e devolve o índice do maior elemento.

```
1 def min_indice(lista: list[int]) -> int:  
2     menor = 0  
3     n = len(lista)  
4     for i in range(n):  
5         if lista[i] < lista[menor]:  
6             menor = i
```

Exercício

</> Code

Faça uma função que recebe uma lista de inteiros e devolve o índice do maior elemento.

```
1 def min_indice(lista: list[int]) -> int:
2     menor = 0
3     n = len(lista)
4     for i in range(n):
5         if lista[i] < lista[menor]:
6             menor = i
7     return menor
```

Quebrando uma entrada em lista

Muitas entradas vêm em uma única linha. O método `.split()` quebra uma string em pedaços e devolve uma lista.

Quebrando uma entrada em lista

Muitas entradas vêm em uma única linha. O método `.split()` quebra uma string em pedaços e devolve uma lista.

Quebrando uma entrada em lista

Muitas entradas vêm em uma única linha. O método `.split()` quebra uma string em pedaços e devolve uma lista.

```
1 linha = "10 20 30 40"  
2 partes = linha.split()  
3  
4 print(partes) # ['10', '20', '30', '40']
```

Quebrando uma entrada em lista

Muitas entradas vêm em uma única linha. O método `.split()` quebra uma string em pedaços e devolve uma lista.

```
1 linha = "10 20 30 40"  
2 partes = linha.split()  
3  
4 print(partes) # ['10', '20', '30', '40']
```

Sem argumento, `.split()` separa pelos espaços. O resultado é uma lista de **strings**!

Compreensão de listas

Compreensão de listas é uma forma compacta de criar uma nova lista a partir de outra.

Compreensão de listas

Compreensão de listas é uma forma compacta de criar uma nova lista a partir de outra.

```
[Levante(aluno) para cada aluno em turma se aluno foi bem na prova]
```

Compreensão de listas

Compreensão de listas é uma forma compacta de criar uma nova lista a partir de outra.

```
[Levante(aluno) para cada aluno em turma se aluno foi bem na prova]
```

```
1 numeros = [1, 2, 3, 4, 5]
2 quadrados = [x * x for x in numeros]
3
4 print(quadrados)
5 # [1, 4, 9, 16, 25]
```

Compreensão de listas

Compreensão de listas é uma forma compacta de criar uma nova lista a partir de outra.

```
[Levante(aluno) para cada aluno em turma se aluno foi bem na prova]
```

```
1 numeros = [1, 2, 3, 4, 5]
2 quadrados = [x * x for x in numeros]
3
4 print(quadrados)
5 # [1, 4, 9, 16, 25]
```

```
1 nums = [1, 2, 3, 4, 5, 6]
2 pares = [x for x in nums if x % 2 == 0]
3
4 print(pares)
5 # [2, 4, 6]
```

Ela é ótima para transformar, filtrar ou ler dados de entrada de maneira curta e direta.

Compreensão de listas

Compreensão de listas é uma forma compacta de criar uma nova lista a partir de outra.

```
[Levante(aluno) para cada aluno em turma se aluno foi bem na prova]
```

```
1 numeros = [1, 2, 3, 4, 5]
2 quadrados = [x * x for x in numeros]
3
4 print(quadrados)
5 # [1, 4, 9, 16, 25]
```

```
1 nums = [1, 2, 3, 4, 5, 6]
2 pares = [x for x in nums if x % 2 == 0]
3
4 print(pares)
5 # [2, 4, 6]
```

Ela é ótima para transformar, filtrar ou ler dados de entrada de maneira curta e direta.

```
1 lista = input().split()
2 for i in range(len(lista)):
3     lista[i] = int(lista[i])
```

Compreensão de listas

Compreensão de listas é uma forma compacta de criar uma nova lista a partir de outra.

```
[Levante(aluno) para cada aluno em turma se aluno foi bem na prova]
```

```
1 numeros = [1, 2, 3, 4, 5]
2 quadrados = [x * x for x in numeros]
3
4 print(quadrados)
5 # [1, 4, 9, 16, 25]
```

```
1 nums = [1, 2, 3, 4, 5, 6]
2 pares = [x for x in nums if x % 2 == 0]
3
4 print(pares)
5 # [2, 4, 6]
```

Ela é ótima para transformar, filtrar ou ler dados de entrada de maneira curta e direta.

```
1 lista = input().split()
2 for i in range(len(lista)):
3     lista[i] = int(lista[i])
```

```
lista = [int(x) for x in input().split()]
```

Índices e Slices

Além disso, diferente de outras linguagens de programação, o python permite contar de trás para frente.

Índices e Slices

Além disso, diferente de outras linguagens de programação, o python permite contar de trás para frente.

```
1 letras = ["A", "B", "C", "D", "E", "F"]
2
3 print(letras[-1]) # 'F' (última)
4 print(letras[-2]) # 'E' (penúltima)
5 print(letras[-len(letras)]) # 'A' (primeira)
```

Índices e Slices

Além disso, diferente de outras linguagens de programação, o python permite contar de trás para frente.

```
1 letras = ["A", "B", "C", "D", "E", "F"]
2
3 print(letras[-1]) # 'F' (última)
4 print(letras[-2]) # 'E' (penúltima)
5 print(letras[-len(letras)]) # 'A' (primeira)
```

Podemos pegar fatias da lista usando `lista[inicio:fim:passo]`. O início é inclusivo e o fim é exclusivo.

Índices e Slices

Além disso, diferente de outras linguagens de programação, o python permite contar de trás para frente.

```
1 letras = ["A", "B", "C", "D", "E", "F"]
2
3 print(letras[-1]) # 'F' (última)
4 print(letras[-2]) # 'E' (penúltima)
5 print(letras[-len(letras)]) # 'A' (primeira)
```

Podemos pegar fatias da lista usando `lista[inicio:fim:passo]`. O início é inclusivo e o fim é exclusivo.

```
1 print(letras[1:4]) # ['B', 'C', 'D']
2 print(letras[:3]) # ['A', 'B', 'C']
3 print(letras[-2:]) # ['E', 'F']
```

```
1 print(letras[::2]) # ['A', 'C', 'E']
2 print(letras[::-1])
3 # ['F', 'E', 'D', 'C', 'B', 'A']
```

Exercício

</> Code

Faça uma função que recebe uma lista, um elemento e um índice. Esta função deve devolver uma lista com o elemento inserido na posição correspondente ao índice.

Exercício

</> Code

Faça uma função que recebe uma lista, um elemento e um índice. Esta função deve devolver uma lista com o elemento inserido na posição correspondente ao índice.

```
1 def inserir(lista: list, elemento, indice: int) -> list:  
2     ...
```

Exercício

</> Code

Faça uma função que recebe uma lista, um elemento e um índice. Esta função deve devolver uma lista com o elemento inserido na posição correspondente ao índice.

```
1 def inserir(lista: list, elemento, indice: int) -> list:  
2     parte1 = lista[:indice]  
3     parte2 = [elemento] # lista com só um elemento  
4     parte3 = lista[indice:]
```

Exercício

</> Code

Faça uma função que recebe uma lista, um elemento e um índice. Esta função deve devolver uma lista com o elemento inserido na posição correspondente ao índice.

```
1 def inserir(lista: list, elemento, indice: int) -> list:
2     parte1 = lista[:indice]
3     parte2 = [elemento] # lista com só um elemento
4     parte3 = lista[indice:]
5     return parte1 + parte2 + parte3
```

Inserindo e removendo

Além de alterar um valor existente, também podemos inserir em qualquer posição ou remover por índice.

Inserindo e removendo

Além de alterar um valor existente, também podemos inserir em qualquer posição ou remover por índice.

```
1 fila = ["Ana", "Caio"]
2 fila.insert(1, "Bia")
3
4 print(fila)
5 # ['Ana', 'Bia', 'Caio']
```

Inserindo e removendo

Além de alterar um valor existente, também podemos inserir em qualquer posição ou remover por índice.

```
1 fila = ["Ana", "Caio"]
2 fila.insert(1, "Bia")
3
4 print(fila)
5 # ['Ana', 'Bia', 'Caio']
```

```
1 fila = ["Ana", "Bia", "Caio"]
2 del fila[0]
3
4 print(fila)
5 # ['Bia', 'Caio']
```

Inserindo e removendo

Além de alterar um valor existente, também podemos inserir em qualquer posição ou remover por índice.

```
1 fila = ["Ana", "Caio"]
2 fila.insert(1, "Bia")
3
4 print(fila)
5 # ['Ana', 'Bia', 'Caio']
```

```
1 fila = ["Ana", "Bia", "Caio"]
2 del fila[0]
3
4 print(fila)
5 # ['Bia', 'Caio']
```

- `.insert(i, x)` coloca `x` **antes** do índice `i`.
- Já `del lista[i]` remove o elemento naquela posição.

Removendo

Removendo

O comando `.pop()` serve para remover elementos ao final da lista e devolvê-los.

Removendo

O comando `.pop()` serve para remover elementos ao final da lista e devolvê-los.

```
1 bts = ["RM", "Jin", "Suga", "J-Hope", "Jimin", "V", "Jungkook"]  
2 prox = bts.pop()
```

Removendo

O comando `.pop()` serve para remover elementos ao final da lista e devolvê-los.

```
1 bts = ["RM", "Jin", "Suga", "J-Hope", "Jimin", "V", "Jungkook"]  
2 prox = bts.pop()
```

Já o comando `.remove(x)` remove a primeira ocorrência de `x` na lista. Se `x` não estiver presente, ocorre um erro.

Removendo

O comando `.pop()` serve para remover elementos ao final da lista e devolvê-los.

```
1 bts = ["RM", "Jin", "Suga", "J-Hope", "Jimin", "V", "Jungkook"]
2 prox = bts.pop()
```

Já o comando `.remove(x)` remove a primeira ocorrência de `x` na lista. Se `x` não estiver presente, ocorre um erro.

```
1 one_direction = ["Harry", "Liam", "Louis", "Niall", "Zayn"]
2 one_direction.remove("Liam")
```

Desempacotamento

Podemos extrair vários valores de uma sequência de uma vez só, desde que a quantidade de variáveis combine com a quantidade de elementos.

Desempacotamento

Podemos extrair vários valores de uma sequência de uma vez só, desde que a quantidade de variáveis combine com a quantidade de elementos.

```
1 coordenadas = [10, 20]
2 x, y = coordenadas
3
4 print(x) # 10
5 print(y) # 20
```

Desempacotamento

Podemos extrair vários valores de uma sequência de uma vez só, desde que a quantidade de variáveis combine com a quantidade de elementos.

```
1 coordenadas = [10, 20]
2 x, y = coordenadas
3
4 print(x) # 10
5 print(y) # 20
```

```
1 a = 3
2 b = 7
3
4 a, b = b, a
5 print(a, b) # 7 3
```

Percorrendo melhor: `enumerate()`

</> Code

Faça uma função que recebe uma lista de n notas e devolve a média ponderada das notas. O peso de cada nota é correspondente à posição na lista, com o primeiro peso sendo 0 e o último, $n - 1$.

Percorrendo melhor: `enumerate()`

</> Code

Faça uma função que recebe uma lista de n notas e devolve a média ponderada das notas. O peso de cada nota é correspondente à posição na lista, com o primeiro peso sendo 0 e o último, $n - 1$.

```
1 def ponderada(notas: list[float]) -> float:  
2     ...
```

Percorrendo melhor: `enumerate()`

</> Code

Faça uma função que recebe uma lista de n notas e devolve a média ponderada das notas. O peso de cada nota é correspondente à posição na lista, com o primeiro peso sendo 0 e o último, $n - 1$.

```
1 def ponderada(notas: list[float]) -> float:
2     n = len(notas)
3     nota_total = 0.0
4     peso_total = 0
5     ...
```

Percorrendo melhor: `enumerate()`

</> Code

Faça uma função que recebe uma lista de n notas e devolve a média ponderada das notas. O peso de cada nota é correspondente à posição na lista, com o primeiro peso sendo 0 e o último, $n - 1$.

```
1 def ponderada(notas: list[float]) -> float:
2     n = len(notas)
3     nota_total = 0.0
4     peso_total = 0
5     for i, nota in enumerate(notas):
6         nota_total += nota * i
7         peso_total += i
```

Percorrendo melhor: `enumerate()`

</> Code

Faça uma função que recebe uma lista de n notas e devolve a média ponderada das notas. O peso de cada nota é correspondente à posição na lista, com o primeiro peso sendo 0 e o último, $n - 1$.

```
1 def ponderada(notas: list[float]) -> float:
2     n = len(notas)
3     nota_total = 0.0
4     peso_total = 0
5     for i, nota in enumerate(notas):
6         nota_total += nota * i
7         peso_total += i
8     return nota_total / peso_total
```

Percorrendo melhor: `zip()`

</> Code

Faça uma função que recebe duas listas de comprimento n : a primeira corresponde às notas e a segunda, aos pesos. Devolva a média ponderada.

Percorrendo melhor: `zip()`

</> Code

Faça uma função que recebe duas listas de comprimento n : a primeira corresponde às notas e a segunda, aos pesos. Devolva a média ponderada.

```
def ponderada(notas: list[float], pesos: list[int]) -> float:
```

Percorrendo melhor: `zip()`

</> Code

Faça uma função que recebe duas listas de comprimento n : a primeira corresponde às notas e a segunda, aos pesos. Devolva a média ponderada.

```
1 def ponderada(notas: list[float], pesos: list[int]) -> float:
2     nota_total = 0.0
3     peso_total = 0
```

Percorrendo melhor: `zip()`

</> Code

Faça uma função que recebe duas listas de comprimento n : a primeira corresponde às notas e a segunda, aos pesos. Devolva a média ponderada.

```
1 def ponderada(notas: list[float], pesos: list[int]) -> float:
2     nota_total = 0.0
3     peso_total = 0
4     for nota, peso in zip(notas, pesos):
5         nota_total += nota * peso
6         peso_total += peso
```

Percorrendo melhor: `zip()`

</> Code

Faça uma função que recebe duas listas de comprimento n : a primeira corresponde às notas e a segunda, aos pesos. Devolva a média ponderada.

```
1 def ponderada(notas: list[float], pesos: list[int]) -> float:
2     nota_total = 0.0
3     peso_total = 0
4     for nota, peso in zip(notas, pesos):
5         nota_total += nota * peso
6         peso_total += peso
7     return nota_total / peso_total
```

Tuplas

Tuplas: ideia principal

- Tuplas são sequências ordenadas, como listas, mas escritas com `()`.

Tuplas: ideia principal

- Tuplas são sequências ordenadas, como listas, mas escritas com `()`.
- Também podem ser criadas com `tuple(...)`.

Tuplas: ideia principal

- Tuplas são sequências ordenadas, como listas, mas escritas com `()`.
- Também podem ser criadas com `tuple(...)`.

```
1 aluno = ("Ana", 20, 8.7)
2 letras = tuple(["a", "b", "c"])
3
4 print(aluno)
5 print(letras)
```

Imutabilidade

- Depois que uma tupla é criada, não podemos alterar seus elementos.

Imutabilidade

- Depois que uma tupla é criada, não podemos alterar seus elementos.
- Não existem operações de inserção/remoção como nas listas.

Imutabilidade

- Depois que uma tupla é criada, não podemos alterar seus elementos.
- Não existem operações de inserção/remoção como nas listas.

```
1 e = ("Google", "Facebook", "Amazon")  
2 e[1] = "Samsung"
```

Imutabilidade

- Depois que uma tupla é criada, não podemos alterar seus elementos.
- Não existem operações de inserção/remoção como nas listas.

```
1 e = ("Google", "Facebook", "Amazon")  
2 e[1] = "Samsung"
```

```
TypeError: 'tuple' object does not  
support item assignment
```

Imutabilidade

- Depois que uma tupla é criada, não podemos alterar seus elementos.
- Não existem operações de inserção/remoção como nas listas.

```
1 e = ("Google", "Facebook", "Amazon")  
2 e[1] = "Samsung"
```

```
TypeError: 'tuple' object does not  
support item assignment
```

- Ainda podemos acessar normalmente com índice e slice:

Imutabilidade

- Depois que uma tupla é criada, não podemos alterar seus elementos.
- Não existem operações de inserção/remoção como nas listas.

```
1 e = ("Google", "Facebook", "Amazon")
2 e[1] = "Samsung"
```

```
TypeError: 'tuple' object does not
support item assignment
```

- Ainda podemos acessar normalmente com índice e slice:

```
1 print(empresas[0])
2 print(empresas[1:])
```

Por que usar tuplas?

- Quando os dados representam algo que não deve mudar no programa.

Por que usar tuplas?

- Quando os dados representam algo que não deve mudar no programa.
- A tupla comunica melhor a intenção: “estrutura fixa”.

Por que usar tuplas?

- Quando os dados representam algo que não deve mudar no programa.
- A tupla comunica melhor a intenção: “estrutura fixa”.

```
1 # Coordenada (x, y) de um ponto no plano
2 ponto = (3, 5)
3
4 x, y = ponto
5 print(x, y)
```

Strings

Strings: ideia principal

- Uma string é uma sequência de caracteres.

Strings: ideia principal

- Uma string é uma sequência de caracteres.
- Em Python, strings são imutáveis.

Strings: ideia principal

- Uma string é uma sequência de caracteres.
- Em Python, strings são imutáveis.
- Podemos acessar por índice e por slice, como em listas.

Strings: ideia principal

- Uma string é uma sequência de caracteres.
- Em Python, strings são imutáveis.
- Podemos acessar por índice e por slice, como em listas.

```
1 msg = "MC102"  
2  
3 print(msg[0])    # M  
4 print(msg[1:4]) # C10
```

Formatando texto com f-strings

- f-strings deixam mensagens com variáveis mais legíveis.

Formatando texto com f-strings

- f-strings deixam mensagens com variáveis mais legíveis.
- Qualquer expressão Python pode aparecer entre `{}`.

Formatando texto com f-strings

- f-strings deixam mensagens com variáveis mais legíveis.
- Qualquer expressão Python pode aparecer entre {}.

```
1 nome = "Ana"
2 nota = 8.75
3
4 print(f"Aluna: {nome}")
5 print(f"Nota final: {nota:.1f}") # 8.7
6 print(f"Aprovada? {nota >= 5.0}") # True
```

Converter string para lista de caracteres

- A função `list(...)` transforma uma string em uma lista com 1 caractere por elemento.

Converter string para lista de caracteres

- A função `list(...)` transforma uma string em uma lista com 1 caractere por elemento.
- Isso ajuda quando queremos manipular caracteres individualmente.

Converter string para lista de caracteres

- A função `list(...)` transforma uma string em uma lista com 1 caractere por elemento.
- Isso ajuda quando queremos manipular caracteres individualmente.

```
1 palavra = "python"
2 chars = list(palavra)
3
4 print(chars)      # ['p', 'y', 't', 'h', 'o', 'n']
5 print(chars[2])   # t
```

`.split()` e `.join()`

`.split(sep)` quebra uma string em uma lista.

`.split()` e `.join()`

`.split(sep)` quebra uma string em uma lista.

```
1 linha = "maçã,banana,uva"  
2 frutas = linha.split(",")  
3 print(frutas) # ['maçã', 'banana', 'uva']
```

`.split()` e `.join()`

`.split(sep)` quebra uma string em uma lista.

```
1 linha = "maçã,banana,uva"  
2 frutas = linha.split(",")  
3 print(frutas) # ['maçã', 'banana', 'uva']
```

- `sep.join(lista)` faz o caminho inverso: junta elementos em uma string.

`.split()` e `.join()`

`.split(sep)` quebra uma string em uma lista.

```
1 linha = "maçã,banana,uva"
2 frutas = linha.split(",")
3 print(frutas) # ['maçã', 'banana', 'uva']
```

- `sep.join(lista)` faz o caminho inverso: junta elementos em uma string.

```
1 texto = " - ".join(frutas)
2 print(texto) # maçã - banana - uva
```

Limpeza e padronização: `.strip()`, `.lower()`, `.upper()`

- `strip()` remove espaços extras nas pontas.

Limpeza e padronização: `.strip()`, `.lower()`, `.upper()`

- `strip()` remove espaços extras nas pontas.

```
1 entrada = "  PyTh0n  "  
2  
3 limpa = entrada.strip()  
4 print(limpa)          # 'PyTh0n'
```

Limpeza e padronização: `.strip()`, `.lower()`, `.upper()`

- `strip()` remove espaços extras nas pontas.

```
1 entrada = "  PyTh0n  "  
2  
3 limpa = entrada.strip()  
4 print(limpa)          # 'PyTh0n'
```

- `lower()` e `upper()` ajudam a comparar textos sem erro de caixa.

Limpeza e padronização: `.strip()`, `.lower()`, `.upper()`

- `strip()` remove espaços extras nas pontas.

```
1 entrada = "  PyTh0n  "  
2  
3 limpa = entrada.strip()  
4 print(limpa)          # 'PyTh0n'
```

- `lower()` e `upper()` ajudam a comparar textos sem erro de caixa.

```
1 print(limpa.lower()) # 'python'  
2 print(limpa.upper()) # 'PYTHON'
```

Prefixo e sufixo: `.startswith()` e `.endswith()`

- Verificam se uma string começa ou termina com um trecho.

Prefixo e sufixo: `.startswith()` e `.endswith()`

- Verificam se uma string começa ou termina com um trecho.
- Muito útil para validar nomes de arquivos, URLs e comandos.

Prefixo e sufixo: `.startswith()` e `.endswith()`

- Verificam se uma string começa ou termina com um trecho.
- Muito útil para validar nomes de arquivos, URLs e comandos.

```
1 arquivo = "relatorio_final.pdf"
2
3 print(arquivo.startswith("relatorio")) # True
4 print(arquivo.endswith(".pdf"))      # True
5 print(arquivo.endswith(".txt"))      # False
```

Substituição de trechos: `.replace()`

- `replace(antigo, novo)` cria uma nova string com substituições.

Substituição de trechos: `.replace()`

- `replace(antigo, novo)` cria uma nova string com substituições.
- Como strings são imutáveis, a original não é alterada.

Substituição de trechos: `.replace()`

- `replace(antigo, novo)` cria uma nova string com substituições.
- Como strings são imutáveis, a original não é alterada.

```
1 frase = "MC102 e legal"
2 nova = frase.replace("legal", "desafiadora")
3
4 print(frase) # MC102 e legal
5 print(nova)  # MC102 e desafiadora
```

Dicionários

Dicionários: visão geral

- Dicionários armazenam pares **chave: valor**.

Dicionários: visão geral

- Dicionários armazenam pares **chave: valor**.
- O acesso é feito pela chave, não por posição numérica.

Dicionários: visão geral

- Dicionários armazenam pares **chave: valor**.
- O acesso é feito pela chave, não por posição numérica.

```
1 aluno = {  
2     "nome": "Ana",  
3     "idade": 20,  
4     "curso": "Computação",  
5 }  
6  
7 print(aluno["nome"])    # Ana  
8 print(aluno["idade"])  # 20
```

Acesso por chaves e chave ausente

- `d[chave]` retorna o valor associado.

Acesso por chaves e chave ausente

- `d[chave]` retorna o valor associado.
- Se a chave não existir, ocorre `KeyError`.

Acesso por chaves e chave ausente

- `d[chave]` retorna o valor associado.
- Se a chave não existir, ocorre `KeyError`.
- `d.get(chave)` evita erro e retorna `None` (ou valor padrão).

Acesso por chaves e chave ausente

- `d[chave]` retorna o valor associado.
- Se a chave não existir, ocorre `KeyError`.
- `d.get(chave)` evita erro e retorna `None` (ou valor padrão).

```
1 notas = {"Ana": 8.5, "Bia": 9.0}
2
3 print(notas["Ana"])          # 8.5
4 print(notas.get("Carlos"))  # None
5 print(notas.get("Carlos", 0.0)) # 0.0
```

Operador **in** em dicionários

- O operador **in** verifica presença de chave no dicionário.

Operador `in` em dicionários

- O operador `in` verifica presença de chave no dicionário.
- Não verifica diretamente os valores.

Operador `in` em dicionários

- O operador `in` verifica presença de chave no dicionário.
- Não verifica diretamente os valores.

```
1 ra = {"ana": 12345, "bia": 23456}
2
3 print("ana" in ra)      # True
4 print("carlos" in ra)  # False
5 print(12345 in ra)     # False (12345 é valor, não chave)
```

Tuplas como chaves

- Chaves de dicionário devem ser imutáveis.

Tuplas como chaves

- Chaves de dicionário devem ser imutáveis.
- Tuplas funcionam muito bem como chaves compostas.

Tuplas como chaves

- Chaves de dicionário devem ser imutáveis.
- Tuplas funcionam muito bem como chaves compostas.

```
1 # (cidade, mes) -> temperatura média
2 clima = {
3     ("Campinas", 1): 28.5,
4     ("Campinas", 2): 27.9,
5     ("Limeira", 1): 26.1,
6 }
7
8 print(clima[("Campinas", 2)]) # 27.9
```

Conjuntos

Conjuntos: quando vale usar?

- Conjuntos guardam elementos sem repetição.

Conjuntos: quando vale usar?

- Conjuntos guardam elementos sem repetição.
- São úteis quando o objetivo é unicidade, e não ordem.

Conjuntos: quando vale usar?

- Conjuntos guardam elementos sem repetição.
- São úteis quando o objetivo é unicidade, e não ordem.

```
1 nomes = ["Ana", "Bia", "Ana", "Carlos", "Bia"]
2 unicos = set(nomes)
3
4 print(unicos) # {'Ana', 'Bia', 'Carlos'}
```

Motivo 1: remover duplicatas com facilidade

- Converter lista em conjunto remove repetições automaticamente.

Motivo 1: remover duplicatas com facilidade

- Converter lista em conjunto remove repetições automaticamente.
- Depois, podemos voltar para lista se necessário.

Motivo 1: remover duplicatas com facilidade

- Converter lista em conjunto remove repetições automaticamente.
- Depois, podemos voltar para lista se necessário.

```
1 valores = [10, 20, 10, 30, 20, 40]
2 sem_repeticao = list(set(valores))
3
4 print(sem_repeticao)
```

Motivo 2: teste de pertencimento eficiente

- Se vamos fazer muitas consultas do tipo “x está aqui?”, conjunto é uma boa escolha.

Motivo 2: teste de pertencimento eficiente

- Se vamos fazer muitas consultas do tipo “x está aqui?”, conjunto é uma boa escolha.
- Usamos o operador `in` normalmente.

Motivo 2: teste de pertencimento eficiente

- Se vamos fazer muitas consultas do tipo “x está aqui?”, conjunto é uma boa escolha.
- Usamos o operador `in` normalmente.

```
1 inscritos = {12345, 23456, 34567}
2
3 ra = 23456
4 if ra in inscritos:
5     print("Aluno inscrito")
6 else:
7     print("Aluno não inscrito")
```

Motivo 3: operações de conjuntos

- Modelam problemas de comparação entre grupos.

Motivo 3: operações de conjuntos

- Modelam problemas de comparação entre grupos.
- União, interseção e diferença deixam o código direto.

Motivo 3: operações de conjuntos

- Modelam problemas de comparação entre grupos.
- União, interseção e diferença deixam o código direto.

```
1 python = {"Ana", "Bia", "Caio"}
2 java = {"Bia", "Davi"}
3
4 print(python.union(java))      # todos
5 print(python.intersection(java)) # em comum
6 print(python.difference(java))  # só python
```

Dúvidas?