# 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.

## Criando e adicionando no final

```
nomes = ["Ana", "Bia"]
nomes.append("Caio")

print(nomes)
# ['Ana', 'Bia', 'Caio']
```

## Consultando a lista

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

print(len(nomes))        # 3
print("Bia" in nomes)    # True
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.

```python
def min_indice(lista: list[int]) -> int:
    menor = 0
    n = len(lista)
    for i in range(n):
        if lista[i] < lista[menor]:
            menor = i
    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.

```
linha = "10 20 30 40"
partes = linha.split()

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.

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

```python
numeros = [1, 2, 3, 4, 5]
quadrados = [x * x for x in numeros]

print(quadrados)
# [1, 4, 9, 16, 25]
```

```python
nums = [1, 2, 3, 4, 5, 6]
pares = [x for x in nums if x % 2 == 0]

print(pares)
# [2, 4, 6]
```

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

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

```python
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.

```
letras = ["A", "B", "C", "D", "E", "F"]

print(letras[-1]) # 'F' (última)
print(letras[-2]) # 'E' (penúltima)
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.

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

```
print(letras[::2])   # ['A', 'C', 'E']
print(letras[::-1])
# ['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.

```
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.

```python
fila = ["Ana", "Caio"]
fila.insert(1, "Bia")

print(fila)
# ['Ana', 'Bia', 'Caio']
```

```python
fila = ["Ana", "Bia", "Caio"]
del fila[0]

print(fila)
# ['Bia', 'Caio']
```

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

# Removendo

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

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

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

```
one_direction = ["Harry", "Liam", "Louis", "Niall", "Zayn"]
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.

```
coordenadas = [10, 20]
x, y = coordenadas

print(x)  # 10
print(y)  # 20
```

```
a = 3
b = 7

a, b = b, a
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, $n - 1$.

```python
def ponderada(notas: list[float]) -> float:
    n = len(notas)
    nota_total = 0.0
    peso_total = 0
    for i, nota in enumerate(notas):
        nota_total += nota * i
        peso_total += i
    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.

```python
def ponderada(notas: list[float], pesos: list[int]) -> float:
    nota_total = 0.0
    peso_total = 0
    for nota, peso in zip(notas, pesos):
        nota_total += nota * peso
        peso_total += peso
    return nota_total / peso_total
```

# Tuplas

# 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.
- 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.
- 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.
- 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.
- Qualquer expressão Python pode aparecer entre `{}`.

```python
nome = "Ana"
nota = 8.75

print(f"Aluna: {nome}")
print(f"Nota final: {nota:.1f}") # 8.7
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.
- 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.

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

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

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

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

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

```
entrada = "   PyThOn   "

limpa = entrada.strip()
print(limpa)          # 'PyThOn'
```

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

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

# 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.

```
arquivo = "relatorio_final.pdf"

print(arquivo.startswith("relatorio"))  # True
print(arquivo.endswith(".pdf"))         # True
print(arquivo.endswith(".txt"))         # False
```

# 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`.
- O acesso é feito pela chave, não por posição numérica.

```
aluno = {
  "nome": "Ana",
  "idade": 20,
  "curso": "Computação",
}

print(aluno["nome"])   # Ana
print(aluno["idade"])  # 20
```

# 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.
- Não verifica diretamente os valores.

```
ra = {"ana": 12345, "bia": 23456}

print("ana" in ra)    # True
print("carlos" in ra) # False
print(12345 in ra)     # False (12345 é valor, não chave)
```

# Tuplas como chaves

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

```python
# (cidade, mes) -> temperatura média
clima = {
  ("Campinas", 1): 28.5,
  ("Campinas", 2): 27.9,
  ("Limeira", 1): 26.1,
}

print(clima[("Campinas", 2)])  # 27.9
```

# Conjuntos

# Conjuntos: quando vale usar?

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

```python
nomes = ["Ana", "Bia", "Ana", "Carlos", "Bia"]
unicos = set(nomes)

print(unicos)  # {'Ana', 'Bia', 'Carlos'}
```

# Motivo 1: remover duplicatas com facilidade

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

```
valores = [10, 20, 10, 30, 20, 40]
sem_repeticao = list(set(valores))

print(sem_repeticao)
```

# 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.

```
inscritos = {12345, 23456, 34567}

ra = 23456
if ra in inscritos:
  print("Aluno inscrito")
else:
  print("Aluno não inscrito")
```

# Motivo 3: operações de conjuntos

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

```python
python = {"Ana", "Bia", "Caio"}
java = {"Bia", "Davi"}

print(python.union(java))         # todos
print(python.intersection(java))  # em comum
print(python.difference(java))    # só python
```

# Dúvidas?