Recapitulando conceitos de Python

🧱 1. Tipos e Operações Básicas

✅ **Descrição**: Manipulação de números, textos e lógica básica.

🔤 **Sintaxe**:

```
a = 5           # int
b = 2.5         # float
s = "texto"     # string
c = True        # bool
```

🧪 **Exemplo**:

```
a = 5
b = 3
print(a + b)       # Soma
print(a % b)       # Módulo
print(a > b)       # Operador lógico
```

🔁 2. Condicionais e Laços

✅ **Descrição**: Controle de fluxo com `if` e repetição com `for` e `while`.

🔤 **Sintaxe**:

```
if cond:
    ...
elif outra_cond:
    ...
else:
    ...

for i in range(n):
    ...

while cond:
    ...
```

🧪 **Exemplo**:

```
for i in range(1, 6):
    if i % 2 == 0:
```

```
        print(f"{i} é par")
```

📦 3. Listas, Dicionários e Conjuntos

✅ **Descrição**: Armazenamento e manipulação de coleções.

🔤 **Sintaxe**:

```
lista = [1, 2, 3]
dic = {"a": 1, "b": 2}
conj = set([1, 2, 3])
```

🧪 **Exemplo**:

```
lista.append(4)
print(lista[1:3])  # slicing

dic["c"] = 3
print(dic["a"])

conj.add(4)
print(2 in conj)
```

🔁 4. Funções e Recursão

✅ **Descrição**: Blocos reutilizáveis de código. Recursão resolve problemas em etapas menores.

🔤 **Sintaxe**:

```
def soma(a, b):
    return a + b

def fatorial(n):
    if n == 0: return 1
    return n * fatorial(n - 1)
```

🧪 **Exemplo**:

```
print(soma(2, 3))        # 5
print(fatorial(5))       # 120
```

🔤 5. Manipulação de Strings

✅ **Descrição**: Extração e transformação de texto.

🔤 **Sintaxe**:

```
texto = "exemplo"
texto.upper()
texto[0:3]
texto.split()
```

🧪 **Exemplo**:

```
frase = "ola mundo"
palavras = frase.split()
print(palavras)          # ['ola', 'mundo']
```

📚 6. Bibliotecas Úteis

✅ **Descrição**: Pacotes com estruturas e algoritmos prontos.

🧪 **Exemplos**:

```
import math
print(math.gcd(20, 8))    # 4

from collections import Counter
c = Counter("banana")
print(c)                  # {'a': 3, 'b': 1, 'n': 2}

import heapq
lista = [5, 3, 8]
heapq.heapify(lista)
print(heapq.heappop(lista))  # 3
```

📊 7. Ordenação e Busca Binária

✅ **Descrição**: Classificação e localização eficiente de dados ordenados.

🧪 **Exemplo**:

```
a = [5, 1, 4, 2]
a.sort()
print(a)                          # [1, 2, 4, 5]

import bisect
idx = bisect.bisect_left(a, 4)
print(idx)                        # 2
```

💻 8. Entrada e Saída Rápida

✅ **Descrição**: Leitura de dados em massa (problemas com grande input).

🧪 **Exemplo**:

```
import sys
entrada = sys.stdin.readline
n = int(entrada())
```

💡 9. Compreensão de Listas

✅ **Descrição**: Forma compacta de criar listas.

🧪 **Exemplo**:

```
pares = [x for x in range(10) if x % 2 == 0]
print(pares)        # [0, 2, 4, 6, 8]
```

🧠 10. Algoritmo Recursivo (Backtracking simples)

✅ **Descrição**: Técnica para explorar todos os caminhos (útil em jogos, caminhos, etc).

🧪 **Exemplo**:

```
def busca(caminho):
    if caminho == "fim":
        print("Chegou")
        return
    busca("fim")

busca("inicio")
```

Otimização de código em Python:

⚡ 1. Evite loops desnecessários e cálculos repetidos

✅ **Descrição**: Calcule uma vez, reuse.

🔤 **Sintaxe**: Use variáveis para armazenar resultados.

🧪 **Exemplo ruim**:

```
for i in range(n):
    x = len(arr)   # len é recalculado toda vez
    ...
```

🧪 **Exemplo otimizado**:

```
tam = len(arr)     # calcula uma vez só
for i in range(tam):
    ...
```

⚡ 2. Use estruturas eficientes (`set`, `dict`, `heapq`)

✅ **Descrição**: Operações de busca são mais rápidas com `set` ou `dict` (`O(1)`).

🧪 **Exemplo**:

```
# Verificar duplicatas
nums = [1, 2, 3, 2]
if len(set(nums)) < len(nums):
    print("Tem duplicatas")  # ✅ muito mais rápido que nested loops
```

⚡ 3. Use `list comprehension` em vez de `for` quando possível

✅ **Descrição**: Mais rápido e conciso.

🧪 **Exemplo**:

```
# Em vez disso:
pares = []
for i in range(1000):
    if i % 2 == 0:
        pares.append(i)

# Faça assim:
pares = [i for i in range(1000) if i % 2 == 0]
```

⚡ 4. Use `map()` e `join()` para leitura rápida

✅ **Descrição**: Evita múltiplas chamadas de `input()` e `print()`.

🧪 **Exemplo**:

```
# Entrada de múltiplos inteiros
a, b, c = map(int, input().split())

# Saída rápida de lista
```

```
print(" ".join(map(str, [1, 2, 3, 4])))
```

⚡ 5. Use `sys.stdin.readline` para entrada em massa

✅ **Descrição**: Muito mais rápido que `input()` para grandes volumes.

🧪 **Exemplo**:

```
import sys
entrada = sys.stdin.readline
n = int(entrada())
```

⚡ 6. Evite recursão profunda em Python

✅ **Descrição**: O limite padrão de recursão é 1000. Pode dar erro de stack.

🧪 **Alternativa**:

```
import sys
sys.setrecursionlimit(10**6)  # ⚠️ use com cuidado!

# Ou reescreva usando pilha (DFS iterativo)
```

⚡ 7. Use `enumerate` para índice + valor em laços

✅ **Descrição**: Mais eficiente e legível que `range(len(...))`.

🧪 **Exemplo**:

```
for i, val in enumerate(lista):
    print(i, val)
```

⚡ 8. Prefira `defaultdict` para contagem ou agrupamento

✅ **Descrição**: Evita verificações com `if chave not in d`.

🧪 **Exemplo**:

```
from collections import defaultdict
contagem = defaultdict(int)
```

```
for letra in "banana":
    contagem[letra] += 1
```

⚡ 9. Evite cópia desnecessária de listas

🧪 **Ruim**:

```
nova = lista[:]   # desnecessária se você não vai modificar
```

🧪 **Bom**:
Use referências se não precisa alterar.

⚡ 10. Use `heapq` para pegar maiores/menores com eficiência

🧪 **Exemplo**:

```
import heapq
valores = [10, 3, 8, 2]
heapq.heapify(valores)      # transforma em min-heap
menor = heapq.heappop(valores)
```

Tipos de leitura de valores em python

🔹 1. Ler um valor simples

```
x = input()         # lê como string
n = int(input())    # lê como inteiro
f = float(input())  # lê como float
```

🔹 2. Ler múltiplos valores na mesma linha

✅ Com `split()` e `map()`

```
a, b = map(int, input().split())  # dois inteiros
x, y, z = map(float, input().split())  # três floats
```

🧠 Explicação:

- `input()` lê a linha como string.
- `split()` separa por espaços.
- `map(tipo, ...)` converte para o tipo desejado.

🔹 3. Ler uma lista de valores

```
lista = list(map(int, input().split()))
```

Exemplo:

```
Entrada: 1 2 3 4 5
Resultado: [1, 2, 3, 4, 5]
```

🔹 4. Ler várias linhas com número conhecido

```
n = int(input())
for _ in range(n):
    linha = input()
    print(linha)
```

Exemplo:

```
Entrada:
3
João
Maria
Carlos

Saída:
João
Maria
Carlos
```

🔹 5. Ler várias linhas com listas

```
n = int(input())
matriz = []
for _ in range(n):
    linha = list(map(int, input().split()))
    matriz.append(linha)
```

Entrada:

```
3
1 2 3
4 5 6
7 8 9
```

Resultado:

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

🔹 6. Ler até EOF (fim de arquivo)

Usado em juízes online como Beecrowd (URI).

```
while True:
    try:
        linha = input()
        print(linha)
    except EOFError:
        break
```

🔹 7. Ler múltiplos tipos misturados

```
nome, idade, nota = input().split()
idade = int(idade)
nota = float(nota)
```

Entrada:

```
Ana 17 8.5
```

🔹 8. Usar `sys.stdin` para grandes entradas (rápido)

```
import sys
entrada = sys.stdin.readline
a, b = map(int, entrada().split())
```

🔹 9. Entrada de matriz usando list comprehension

```
n = int(input())
matriz = [list(map(int, input().split())) for _ in range(n)]
```

🔹 10. Leitura direta com unpacking

```
a, b, *resto = map(int, input().split())
```

Entrada:

```
10 20 30 40 50
```

Resultado:

```
a = 10
b = 20
resto = [30, 40, 50]
```

Ordenação de valores

🧩 1. Ordenar uma lista de números

📘 Ascendente (padrão):

```
valores = [4, 2, 9, 1]
valores.sort()
print(valores)  # [1, 2, 4, 9]
```

📘 Descendente:

```
valores = [4, 2, 9, 1]
valores.sort(reverse=True)
print(valores)  # [9, 4, 2, 1]
```

🧩 2. Usar `sorted()` (cria nova lista)

```
lista = [5, 3, 8]
nova = sorted(lista)
print(nova)       # [3, 5, 8]
print(lista)      # [5, 3, 8]  ← original não muda
```

🧩 3. Ordenar lista de strings

```
nomes = ['Carlos', 'Ana', 'Bruno']
nomes.sort()
print(nomes)  # ['Ana', 'Bruno', 'Carlos']
```

📘 Descendente:

```
nomes.sort(reverse=True)
```

🧩 4. Ordenar por tamanho das palavras (usando `key`)

```
palavras = ['sol', 'girassol', 'lua']
palavras.sort(key=len)
print(palavras)  # ['sol', 'lua', 'girassol']
```

🧩 5. Ordenar tuplas por um dos elementos

```
pontos = [(3, 2), (1, 5), (2, 4)]
pontos.sort(key=lambda x: x[1])  # ordena pelo segundo valor
print(pontos)  # [(3, 2), (2, 4), (1, 5)]
```

🧩 6. Ordenar dicionário por valor

```
d = {'a': 3, 'b': 1, 'c': 2}
ordenado = sorted(d.items(), key=lambda item: item[1])
print(ordenado)  # [('b', 1), ('c', 2), ('a', 3)]
```

🧩 7. Ordenar lista com números misturados como strings

```
dados = input().split()  # Ex: 10 2 1
ordenado = sorted(map(int, dados))  # converte para int e ordena
print(ordenado)
```

🧩 8. Ordenar com critérios múltiplos

Exemplo: ordenar por idade e depois por nome.

```
pessoas = [('Ana', 20), ('Carlos', 18), ('Ana', 18)]
pessoas.sort(key=lambda x: (x[1], x[0]))
print(pessoas)
# [('Ana', 18), ('Carlos', 18), ('Ana', 20)]
```

Formatação de saida

1. F-strings (Interpolação de strings)

O que faz:
Permite incluir expressões dentro de uma string de maneira simples e eficiente. A variável ou expressão é colocada entre chaves `{}` dentro de uma string precedida pela letra `f`.

Sintaxe:

```
f"Texto {variável}"
```

Exemplo:

```
nome = "João"
idade = 25
print(f"Olá, {nome}! Você tem {idade} anos.")
```

Saída:

```
Olá, João! Você tem 25 anos.
```

2. Método `str.format()`

O que faz:
Permite inserir variáveis em uma string. É uma abordagem mais antiga que as f-strings, mas ainda muito utilizada.

Sintaxe:

```
"Texto {}".format(variável)
```

Exemplo:

```
nome = "João"
idade = 25
print("Olá, {}! Você tem {} anos.".format(nome, idade))
```

Saída:

```
Olá, João! Você tem 25 anos.
```

Usando índices ou nomes:

```
print("Nome: {0}, Idade: {1}".format(nome, idade))  # Usando índices
# Ou com nomes:
print("Nome: {nome}, Idade: {idade}".format(nome="João", idade=25))
```

Saída:

```
Nome: João, Idade: 25
```

3. Operadores de Formatação (Estilo Antigo)

O que faz:
Usa o operador `%` para inserir variáveis em strings. É um estilo mais antigo e foi substituído pelo `.format()` e f-strings, mas ainda é funcional.

Sintaxe:

```
"Texto % tipo_de_dado" % (variável)
```

Exemplo:

```
nome = "João"
idade = 25
print("Olá, %s! Você tem %d anos." % (nome, idade))
```

Saída:

```
Olá, João! Você tem 25 anos.
```

Formatação de números:

```
preco = 99.99
print("O preço é R$%.2f" % preco)
```

Saída:

```
O preço é R$99.99
```

4. Controlando Separadores e Terminadores no `print()`

O que faz:
Controla o que aparece entre os itens impressos e o que aparece no final da linha. Você pode usar `sep` para o separador e `end` para o terminador.

Sintaxe:

```
print(valores, sep="separador", end="terminador")
```

Exemplo com `sep`:

```
print("A", "B", "C", sep="-")
```

Saída:

```
A-B-C
```

Exemplo com `end`:

```
print("Olá", end=" ")
print("mundo!")
```

Saída:

```
Olá mundo!
```

5. Alinhamento e Tamanho (f-strings e `format()`)

O que faz:
Permite especificar o alinhamento de texto e a largura de uma string ou número.

Sintaxe f-strings:

```
f"{variável:<largura}"  # Alinhamento à esquerda
f"{variável:>largura}"  # Alinhamento à direita
f"{variável:^largura}"  # Alinhamento centralizado
```

Exemplo f-strings:

```
nome = "João"
print(f"{nome:<10}")  # Alinha à esquerda com largura 10
print(f"{nome:>10}")  # Alinha à direita com largura 10
print(f"{nome:^10}")  # Centraliza com largura 10
```

Saída:

```
João
      João
   João
```

Sintaxe `format()`:

```
"{:<largura}".format(variável)  # Alinhamento à esquerda
```

```
"{:>largura}".format(variável)  # Alinhamento à direita
"{:^largura}".format(variável)  # Alinhamento centralizado
```

Exemplo `format()`:

```
print("{:<10}".format("João"))
print("{:>10}".format("João"))
print("{:^10}".format("João"))
```

Saída:

```
João
      João
   João
```

6. Formatação de Números com Precisão e Largura

O que faz:
Permite controlar a precisão de números flutuantes (quantas casas decimais) e a largura total de uma string ou número.

Sintaxe f-strings:

```
f"{variável:.2f}"  # 2 casas decimais
f"{variável:10.2f}"  # Largura 10, 2 casas decimais
```

Exemplo f-strings:

```
numero = 3.14159
print(f"{numero:.2f}")  # 2 casas decimais
print(f"{numero:10.2f}")  # Largura 10, 2 casas decimais
```

Saída:

```
3.14
      3.14
```

Sintaxe `format()`:

```
"{:.2f}".format(variável)  # 2 casas decimais
"{:10.2f}".format(variável)  # Largura 10, 2 casas decimais
```

Exemplo `format()`:

```
print("{:.2f}".format(3.14159))
print("{:10.2f}".format(3.14159))
```

Saída:

```
3.14
      3.14
```

7. Formatação com Variáveis Múltiplas

O que faz:
Permite imprimir múltiplas variáveis de forma organizada e com formatação personalizada.

Sintaxe f-strings:

```
f"Texto {variável1} e {variável2}"
```

Exemplo f-strings:

```
nome = "João"
idade = 25
print(f"Nome: {nome}, Idade: {idade}")
```

Saída:

```
Nome: João, Idade: 25
```

Sintaxe `format()`:

```
"Texto {}, {}".format(variável1, variável2)
```

Exemplo `format()`:

```
nome = "João"
idade = 25
print("Nome: {}, Idade: {}".format(nome, idade))
```

Saída:

```
Nome: João, Idade: 25
```

Tipos de Sort e seus usos

🔹 1. `list.sort()` e `sorted()` (nativos e otimizados)

✔️ **Descrição:**

- Ordenação eficiente com **Timsort** (baseado em merge + insertion)
- `list.sort()` altera a lista original
- `sorted(lista)` retorna uma nova lista ordenada

🧪 **Exemplo:**

```
lista = [5, 3, 2, 8, 1]
lista.sort()
print(lista)  # [1, 2, 3, 5, 8]

lista2 = [7, 4, 6]
ordenada = sorted(lista2, reverse=True)
print(ordenada)  # [7, 6, 4]
```

🔹 2. Bubble Sort

✔️ **Descrição:**

Comparações de pares adjacentes, trocando se estiverem fora de ordem.

🧪 **Código:**

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]

lista = [4, 2, 7, 1]
bubble_sort(lista)
print(lista)  # [1, 2, 4, 7]
```

🔹 3. Selection Sort

✔️ **Descrição:**

Seleciona o menor elemento e o coloca no início repetidamente.

🧪 **Código:**

```
def selection_sort(arr):
    for i in range(len(arr)):
        min_idx = i
        for j in range(i+1, len(arr)):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]

lista = [3, 1, 4, 2]
selection_sort(lista)
print(lista)  # [1, 2, 3, 4]
```

🔹 4. Insertion Sort

✔️ **Descrição:**

Insere cada elemento no lugar certo da parte já ordenada.

🧪 **Código:**

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        chave = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > chave:
            arr[j+1] = arr[j]
            j -= 1
        arr[j+1] = chave

lista = [4, 3, 2, 1]
insertion_sort(lista)
print(lista)  # [1, 2, 3, 4]
```

🔹 5. Merge Sort

✔️ **Descrição:**

Divide a lista, ordena as partes e junta tudo ordenado.

🧪 **Código:**

```
def merge_sort(arr):
    if len(arr) > 1:
        meio = len(arr)//2
```

```
        esq = arr[:meio]
        dir = arr[meio:]

        merge_sort(esq)
        merge_sort(dir)

        i = j = k = 0
        while i < len(esq) and j < len(dir):
            if esq[i] < dir[j]:
                arr[k] = esq[i]
                i += 1
            else:
                arr[k] = dir[j]
                j += 1
            k += 1

        arr[k:] = esq[i:] + dir[j:]

lista = [6, 4, 7, 1, 3]
merge_sort(lista)
print(lista)  # [1, 3, 4, 6, 7]
```

🔹 6. Quick Sort

✔️ **Descrição:**

Escolhe um pivô, separa menores/maiores, e ordena recursivamente.

🧪 **Código:**

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivo = arr[0]
    menores = [x for x in arr[1:] if x <= pivo]
    maiores = [x for x in arr[1:] if x > pivo]
    return quick_sort(menores) + [pivo] + quick_sort(maiores)

lista = [4, 7, 1, 3, 9]
lista = quick_sort(lista)
print(lista)  # [1, 3, 4, 7, 9]
```

🔹 7. Counting Sort (inteiros não-negativos)

✔️ **Descrição:**

Conta quantas vezes cada número aparece e monta a lista ordenada.

🧪 **Código:**

```python
def counting_sort(arr):
    if not arr:
        return
    max_val = max(arr)
    count = [0] * (max_val + 1)

    for num in arr:
        count[num] += 1

    idx = 0
    for i in range(len(count)):
        for _ in range(count[i]):
            arr[idx] = i
            idx += 1

lista = [4, 2, 2, 8, 3, 3, 1]
counting_sort(lista)
print(lista)  # [1, 2, 2, 3, 3, 4, 8]
```