

# Python não tão básico

Disciplina: Programação aplicada à engenharia cartográfica

Maurício C. M. de Paulo - D.Sc.

6 de fevereiro de 2026

# Objetivos

- Aprender a formatar strings em Python.
- Aprender as estruturas de dados mais usadas.
- Aprender a usar funções para organizar o código.

# Formatação de strings em Python

Existem três formas principais de formatar strings em Python:

## 1. Operador % (legado – evitar)

```
1 nome = "Ana"
2 idade = 20
3 msg = "Nome: %s, Idade: %d" % (nome, idade)
```

## 2. Método str.format()

```
1 msg = "Nome: {}, Idade: {}".format(nome, idade)
2 msg = "Nome: {n}, Idade: {i}".format(n=nome, i=idade)
```

## 3. f-strings (recomendado, Python $\geq$ 3.6)

```
1 msg = f"Nome: {nome}, Idade: {idade}"
```

### Vantagens das f-strings:

- Mais legíveis
- Mais concisas
- Permitem expressões: `f"{x + y}"`

# Listas em Python

**Listas** são estruturas que armazenam uma sequência de valores, podendo conter elementos de tipos diferentes.

## Criação de listas

```
1 numeros = [1, 2, 3, 4]
2 nomes = ["Ana", "João", "Maria"]
3 mistura = [10, "texto", 3.14]
```

## Acesso aos elementos (indexação começa em 0)

```
1 print(numeros[0])    # 1
2 print(nomes[2])      # Maria
3 print(numeros[-1])   # último elemento
```

## Operações comuns

```
1 numeros.append(5)    # adiciona no final
2 numeros.remove(2)    # remove o valor 2
3 tamanho = len(numeros) # quantidade de elementos
```

# Matrizes com NumPy

O **NumPy** fornece o tipo `ndarray`, usado para trabalhar com vetores e matrizes de forma eficiente.

## Importação e criação

```
1 import numpy as np
2
3 A = np.array([
4     [1, 2, 3],
5     [4, 5, 6]
6 ])
```

## Dimensões e acesso

```
1 print(A.shape)      # (2, 3)
2 print(A[0, 1])      # elemento da 1ª linha, 2ª coluna
```

## Operações comuns

```
1 B = A * 2           # operação elemento a elemento
2 C = A.T              # transposta
```

# Operações com Matrizes

## Percorrendo a matriz com for

```
1 for i in range(len(matriz)):  
2     for j in range(len(matriz[i])):  
3         print(matriz[i][j])
```

## Criação rápida

```
1 Z = np.zeros((3, 3))  
2 I = np.eye(3)
```

# Dicionários em Python

**Dicionários** armazenam pares **chave** → **valor**, permitindo acesso rápido aos dados por meio da chave.

## Criação de dicionários

```
1 aluno = {  
2     "nome": "Ana",  
3     "idade": 20,  
4     "curso": "Eng. Cartográfica"  
5 }
```

## Acesso aos valores

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

## Inserção e modificação

```
1 aluno["matricula"] = "20231234"  
2 aluno["idade"] = 21
```

- Criar uma lista e calcular a média
- Somar todos os elementos de uma matriz
- Criar um dicionário de que relaciona o nome do aluno à sua nota.



Utilizando funções para organizar o código.

# Código sem funções

Considere um programa que calcula a média de notas de um aluno:

```
1 notas = [7.0, 8.5, 6.0, 9.0]
2
3 soma = 0
4     for n in notas:
5         soma += n
6
7 media = soma / len(notas)
8
9 if media >= 7:
10     print("Aprovado")
11 else:
12     print("Reprovado")
```

# Problemas do código sem funções

Embora o código funcione, ele apresenta limitações:

- Tudo está misturado em um único bloco
- Difícil de reutilizar em outro programa
- Difícil de testar partes específicas
- Pouca clareza sobre **o que** cada parte faz

**Problema central:** tarefas diferentes não estão separadas.

Problemas complexos devem ser divididos em problemas menores, com soluções **reusáveis**.

Bônus: Podemos dividir tarefas e cada aluno implementar uma parte do problema?

# Código com funções

Agora o mesmo programa dividido em funções:

```
1 def calcula_media(notas):
2     soma = 0
3     for n in notas:
4         soma += n
5     return soma / len(notas)
6
7 def resultado(media):
8     if media >= 7:
9         return "Aprovado"
10    else:
11        return "Reprovado"
12
13 notas = [7.0, 8.5, 6.0, 9.0]
14 media = calcula_media(notas)
15 print(resultado(media))
```