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)

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

2. Método `str.format()`

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

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

```
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

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

Acesso aos elementos (indexação começa em 0)

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

Operações comuns

```
numeros.append(5)      # adiciona no final
numeros.remove(2)      # remove o valor 2
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

```
import numpy as np

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

Dimensões e acesso

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

Operações comuns

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

Operações com Matrizes

Percorrendo a matriz com for

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

Criação rápida

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

Dicionários em Python

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

Criação de dicionários

```
aluno = {
    "nome": "Ana",
    "idade": 20,
    "curso": "Eng. Cartográfica"
}
```

Acesso aos valores

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

Inserção e modificação

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

Exercícios

- 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:

```
notas = [7.0, 8.5, 6.0, 9.0]

soma = 0
    for n in notas:
        soma += n

media = soma / len(notas)

if media >= 7:
    print("Aprovado")
else:
    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))
```