

Estruturas de dados em Python (Python não tão básico)

Disciplina: Programação aplicada à engenharia cartográfica

Maurício C. M. de Paulo - D.Sc.

25 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 com Matrizes

Criação rápida

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

Operações comuns

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

Percorrendo a matriz com for

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

Principais Funções de numpy.linalg

```
import numpy as np

# Vetor e matriz de exemplo
v = np.array([3, 4])
A = np.array([[3, 1], [1, 2]])
b = np.array([9, 8])

np.linalg.norm(v)           # Norma Euclidiana (magnitude do vetor)

x = np.linalg.solve(A, b)   # Resolver sistema linear  $Ax = b$ 

np.linalg.det(A)            # Determinante

np.linalg.inv(A)            # Inversa (evitar se possível; prefira solve)

vals, vecs = np.linalg.eig(A) # Autovalores e autovetores
```

Observações importantes:

- norm calcula magnitude (L2 por padrão; pode usar `ord=1`, `ord=np.inf`, etc.).
- solve é numericamente mais estável que usar `inv(A) @ b`.
- Evite calcular a inversa explicitamente quando for resolver sistemas

Mínimos Quadrados (Least Squares)

Problema:

Resolver um sistema
sobredeterminado

$$Ax \approx b$$

minimizando o erro:

$$\min_x \|Ax - b\|_2$$

Saídas da função:

- coef → solução ótima
- residuals → soma dos erros quadráticos
- rank → posto de A
- s → valores singulares

```
import numpy as np
# Ajuste linear  $y = ax + b$ 
x_data = np.array([1, 2, 3, 4])
y_data = np.array([1.2, 1.9, 3.0, 3.9])

# Matriz do modelo linear
A = np.vstack([x_data, np.ones(len(x_data))]).T

# Resolver mínimos quadrados
coef, residuals, rank, s = np.linalg.lstsq(A, y_data, rcond=None)

a, b = coef
print("Coeficientes:", a, b)
```


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
```

Operações comuns

```
chaves = aluno.keys()
```

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:

```
def calcula_media(notas):  
    soma = 0  
    for n in notas:  
        soma += n  
    return soma / len(notas)  
  
def resultado(media):  
    if media >= 7:  
        return "Aprovado"  
    else:  
        return "Reprovado"  
  
notas = [7.0, 8.5, 6.0, 9.0]  
media = calcula_media(notas)  
print(resultado(media))
```

Boas Práticas ao Escrever Funções

1. Funções devem ter responsabilidade única

- Faça uma coisa — e faça bem.

2. Use nomes claros e descritivos

- Evite: `def calc(x):`
- Prefira: `def calcular_area_poligono(vertices):`

3. Tipagem explícita (quando possível)

```
def area_triangulo(base: float, altura: float) -> float:  
    return (base * altura) / 2
```

4. Evite efeitos colaterais

- Não modifique variáveis globais
- Prefira retorno explícito

5. Documente comportamento e contratos

```
def normalizar(v: np.ndarray) -> np.ndarray:  
    """Retorna vetor normalizado (norma L2 = 1)."""  
    return v / np.linalg.norm(v)
```

Exemplo: Gráfico de Barras a partir de um Dicionário

```
import matplotlib.pyplot as plt

def plotar_grafico_barras(dados, titulo="Gráfico de Barras",
    xlabel="Categorias", ylabel="Valores",
    rotacao_labels=0, salvar_em="plot.png"):

    # Separando chaves e valores
    labels = list(dados.keys())
    valores = list(dados.values())

    # Plot
    plt.figure() # Inicializa área de desenho.
    plt.bar(labels, valores) # Gráfico de barras.
    plt.xlabel(xlabel) # Descrição do eixo x.
    plt.ylabel(ylabel) # Descrição do eixo y.
    plt.title(titulo) # Título do plot.
    plt.xticks(rotation=rotacao_labels) # Textos do eixo X.
    plt.tight_layout() # Reduz espaços
    plt.savefig(salvar_em) # Salva o gráfico em um arquivo.
    plt.show() # Abre a janela interativa.
```


Exemplo: Gráfico de Barras a partir de um Dicionário

```
dados = {  
    "Aprendizado profundo": 10,  
    "GNSS": 9,  
    "Sensoriamento remoto": 6,  
    "Fotogrametria": 5,  
    "SIG": 7,  
    "Programação desktop": 7,  
    "Programação web backend": 8,  
    "Programação web frontend": 5,  
    "Produção cartográfica": 6,  
    "Batimetria": 2  
}  
  
plotar_grafico_barras(dados, titulo="Habilidades do Maurício",  
    xlabel="Habilidades", ylabel="Pontuação",  
    rotacao_labels=90, salvar_em="plot.png")
```

Exemplo: Gráfico de Barras a partir de um Dicionário

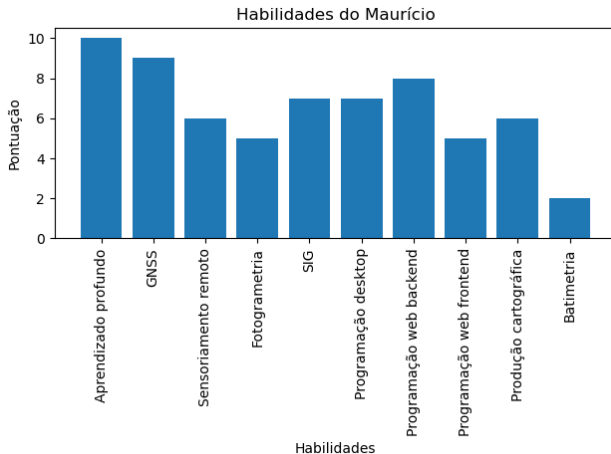


Figura: Gráfico das habilidades

Rasterio é uma biblioteca Python para leitura e escrita de dados raster geoespaciais.

Baseada em:

- GDAL

Principais capacidades:

- Leitura e escrita de GeoTIFF
- Acesso a metadados (CRS, transform)
- Operações matriciais com NumPy
- Recorte (mask) e reprojeção

Rasterio: Leitura e Escrita de Raster

Leitura:

```
import rasterio

with rasterio.open("imagem.tif") as src:
    banda = src.read(1)
    print(src.crs)
    print(src.transform)
```

Escrita:

```
with rasterio.open(
    "saida.tif", "w", driver="GTiff",
    height=banda.shape[0], width=banda.shape[1],
    count=1,
    dtype=banda.dtype,
    crs=src.crs,
    transform=src.transform,
) as dst:
    dst.write(banda, 1)
```

Rasterio: Metadados e Georreferenciamento

Elementos fundamentais:

- CRS (Coordinate Reference System)
- Transform (Affine/Afim)
- Bounds
- Resolução espacial

A transformação afim define:

$$(\text{linha}, \text{coluna}) \rightarrow (x, y)$$

Rasterio integra diretamente com:

- NumPy
- Shapely (via mask)

Rasterio: Operações Comuns

Recorte por polígono:

```
from rasterio.mask import mask

out_image, out_transform = mask(
    src,
    shapes=[geom],
    crop=True
)
```

Reprojeção:

```
from rasterio.warp import calculate_default_transform, reproject

reproject(...)
```

Leitura por janela (window):

```
window = rasterio.windows.Window(0, 0, 512, 512)
data = src.read(1, window=window)
```

Rasterio: Leitura em Janela (Windowed Reading)

Objetivo: Ler apenas parte de um GeoTIFF (melhora performance).

```
import rasterio
from rasterio.windows import Window
with rasterio.open("imagem.tif") as src:

    # define janela: coluna inicial, linha inicial, largura, altura
    window = Window(col_off=1000, row_off=2000, width=512, height=512)

    # lê somente a banda 1 nessa janela
    data = src.read(1, window=window)

    print("Shape:", data.shape)
    print("CRS:", src.crs)

    # transforma coordenadas da janela
    transform = src.window_transform(window)
    print("Transform da janela:", transform)
```

Aplicação:

- Processamento em blocos
- Trabalhar com rasters grandes
- Paralelização

Rasterio: Leitura Automática por Blocos (Block Windows)

Objetivo: Processar raster grande sem carregar tudo na memória.

```
import rasterio
with rasterio.open("imagem.tif") as src:
    # Itera automaticamente pelos blocos internos do arquivo
    for idx, window in src.block_windows(1):
        # lê somente o bloco atual (banda 1)
        data = src.read(1, window=window)

        # exemplo: calcular média do bloco
        media = data.mean()

        print(f"Bloco {idx} - média: {media}")

    # obter transform específico do bloco
    transform = src.window_transform(window)
```

Vantagens:

- Uso eficiente de memória
- Respeita estrutura interna do GeoTIFF (tiling)
- Ideal para processamento paralelo

Rasterio: Leitura de Cloud Optimized Geotiffs

```
import boto3
import rasterio
import matplotlib.pyplot as plt

from rasterio.plot import show

cog_url = 's3://sentinel-cogs/sentinel-s2-l2a-cogs/12/R/UU/2024/4/S2A_12RUU_20240420_000000_000000.tif'

# Create an AWS session with unsigned requests enabled!
aws_session = rasterio.session.AWSSession(boto3.Session(), aws_unsigned=True)

with rasterio.Env(aws_session):
    with rasterio.open(cog_url) as dataset:
        overviews = dataset.overviews(1)
        print(overviews)
        print(f"Original shape:")
        print(f"{dataset.width} x {dataset.height}")
        top_overview = len(overviews)-1
    with rasterio.open(cog_url, OVERVIEW_LEVEL=top_overview) as dataset:
        print(f"Overview {top_overview} shape:")
        print(f"{dataset.width} x {dataset.height}")
        overview_data = dataset.read(1, out_shape=(512, 512))
        show(overview_data)

plt.show()
```

Rasterio — Reprojeção

```
import rasterio
from rasterio.warp import calculate_default_transform, reproject, Resampling

src_path = "entrada.tif"
dst_path = "saida_3857.tif"

with rasterio.open(src_path) as src:

    # Define novo CRS (Web Mercator)
    dst_crs = "EPSG:3857"

    # Calcula nova transformação e dimensões
    transform, width, height = calculate_default_transform(
        src.crs, dst_crs, src.width, src.height, *src.bounds )

    profile = src.profile
    profile.update({
        "crs": dst_crs,
        "transform": transform,
        "width": width,
        "height": height
    })

    with rasterio.open(dst_path, "w", **profile) as dst:
```