

Departamento de Engenharia de Sistemas e Computação

NumPy

Vetores, operações vetorizadas e aplicações numéricas

João Araujo

Algoritmos Computacionais I

Aula 21

Roteiro

- Por que NumPy? Listas *versus* arrays (Cap. 5.14)
- Criando arrays: `array`, `arange`, `linspace`, `zeros`, `ones`
- Operações vetorizadas e funções universais
- Indexação, fatiamento e filtros booleanos
- Matrizes, eixos (`axis`) e álgebra linear
- Armadilhas: homogeneidade e visões de memória

Por que NumPy?

Listas do Python

- Flexíveis (tipos misturados)
- + concatena, * repete
- Cálculos exigem *laços* ⇒ lentos

Arrays do NumPy

- Elementos de **um só tipo**
- Operações aplicadas elemento a elemento
- Código em C por baixo: muito mais rápido

É a base da computação científica em Python: **Pandas**, **Matplotlib**, SciPy e *machine learning* são construídos sobre NumPy.

Concatenar *versus* somar

```
import numpy as np

a = [1, 2, 3]
b = [4, 5, 6]
c = a + b
print(c)                # [1, 2, 3, 4, 5, 6]   concatenação

a_num = np.array(a)
b_num = np.array(b)
c_num = a_num + b_num
print(c_num)            # [5 7 9]              soma vetorial
```

- `import numpy as np`: apelido universal
- Com arrays, +, -, *, /, ** atuam elemento a elemento

Fonte: `code/Cap5/prog.5.h_algebralinear.py`

Criando arrays

```
import numpy as np
a = np.array([1, 2, 3])                 # de uma lista
print(np.arange(0, 1, 0.25))            # [0.   0.25 0.5  0.75]
print(np.linspace(0, 1, 5))             # [0.   0.25 0.5  0.75 1.  ]
print(np.zeros(3))                      # [0. 0. 0.]
print(np.ones((2, 3)))                  # matriz 2 x 3 de uns
print(np.eye(2))                        # identidade 2 x 2
print(a.shape, a.dtype, a.size)         # (3,) int64 3
```

- `arange(ini, fim, passo)`: como `range`, mas aceita passo *real*
- `linspace(ini, fim, n)`: n pontos igualmente espaçados, incluindo o fim
- `shape` = dimensões; `dtype` = tipo dos elementos

Operações vetorizadas

```
a = np.array([1, 2, 3, 4])
print(a * 2)            # [2 4 6 8]
print(a ** 2)           # [ 1  4  9 16]
print(a + 10)           # [11 12 13 14]
print(1 / a)            # [1.   0.5  0.333...  0.25]
print(np.sqrt(a))       # [1.   1.41421356 1.73205081 2.]
x = np.linspace(0, np.pi, 5)
print(np.sin(x).round(3))    # [0.    0.707 1.    0.707 0.   ]
```

- Sem laço `for`: o código fica curto, claro e rápido
- **Funções universais** (`ufuncs`): `sin`, `cos`, `exp`, `log`, `sqrt`...
- Uma tabela de senos (Aula 12) agora são duas linhas

Velocidade: lista *versus* array

```
import numpy as np, random, time
num = 10_000_000
lista1 = random.choices(range(1, 10001), k=num)
lista2 = random.choices(range(1, 10001), k=num)

t1 = time.time()
lista3 = [lista1[i] * lista2[i] for i in range(num)]
print('Lista:', time.time() - t1)

l1, l2 = np.array(lista1), np.array(lista2)
t1 = time.time()
l3 = l1 * l2
print('NumPy:', time.time() - t1)
```

Em geral o NumPy é dezenas de vezes mais rápido (confira na sua máquina!).

Fonte: `code/Cap5/prog.5.i_compara_listas_e_arrays.py`

Estatística e agregações

```
notas = np.array([5, 9, 5.5, 10, 8])
print(notas.sum(), notas.mean())        # 37.5  7.5
print(notas.min(), notas.max())         # 5.0  10.0
print(notas.std(), np.median(notas))    # desvio-padrão, mediana
print(notas.argmax())                   # 3 (posição do maior)
print(np.sort(notas))                   # [ 5.   5.5  8.   9.  10. ]
```

Média ponderada (álgebra linear em uma linha):

```
pesos = np.array([1, 2, 4, 1, 2])
print(np.sum(notas * pesos) / np.sum(pesos))    # 7.1
print(np.average(notas, weights=pesos))         # 7.1
```

Indexação e filtros booleanos

```
import numpy as np

notas = np.array([5, 9, 5.5, 10, 8.4])
print(notas > 6)                    # [False  True False  True  True]
print('Notas > 6:', notas[notas > 6])       # [ 9.   10.   8.4]
print('Notas <= 6:', notas[notas <= 6])     # [5.  5.5]
print(((notas > 6) & (notas < 10)).sum())  # entre 6 e 10: 2
notas[notas < 5.5] = 5.5            # altera somente os que satisfazem
```

- Uma **máscara booleana** dentro dos colchetes seleciona elementos
- Combine com & (e), | (ou), ~ (não), com parênteses
- É a base de filtros em Pandas (Aula 28)

Fonte: `code/Cap5/prog.5.g_numpy_turma.boolean.py`

Matrizes: `shape`, fatias e eixos

```
import numpy as np

m = np.array([[1, 2, 3],
              [4, 5, 6]])
print(m.shape)             # (2, 3)
print(m[1, 2])             # 6         (linha 1, coluna 2)
print(m[:, 0])             # [1 4]     (coluna 0)
print(m[0, :])             # [1 2 3]   (linha 0)
print(m.sum(axis=0))       # [5 7 9]   (soma de cada COLUNA)
print(m.sum(axis=1))       # [ 6 15]   (soma de cada LINHA)
print(m.T)                 # transposta
```

- `m[i, j]` em vez de `m[i][j]`
- `axis=0` colapsa as *linhas*; `axis=1`, as *colunas*

Fonte: `code/Cap5/prog.5.f_numpy_com_axis.py`

Álgebra linear com NumPy

Resolver o sistema $\begin{cases} 2x + y = 3 \\ x + 3y = 5 \end{cases}$:

```
A = np.array([[2, 1],
              [1, 3]])
b = np.array([3, 5])
x = np.linalg.solve(A, b)
print(x)                  # [0.8 1.4]
print(A @ x)              # [3. 5.]   (produto matricial; confere!)
print(np.linalg.det(A))   # 5.0 (aprox.)
print(np.linalg.inv(A))   # matriz inversa
```

- `*` é elemento a elemento; `@` é produto **matricial**
- Aplicações: circuitos (leis de Kirchhoff), estruturas, balanços de massa

Aplicação: ajuste de reta (dados experimentais)

```
V = np.array([5.0, 10.0, 15.0, 20.0])      # tensão (V)
I = np.array([0.25, 0.52, 0.76, 1.01])     # corrente (A)

a, b = np.polyfit(V, I, 1)                 # reta I = a V + b
print(f"I = {a:.4f} V + {b:.4f}")    # I = 0.0504 V + 0.0050
R = 1 / a
print(f"Resistência estimada: {R:.2f} ohms")    # 19.84
```

- A mesma tabela de ensaios da Aula 19, agora com análise numérica de verdade
- Na Aula 27 vamos plotar os pontos e a reta com Matplotlib

Armadilhas do NumPy

1. **Homogeneidade:** todos os elementos têm um tipo.

```
print(np.array(['Python', 1, 2, 3]))   # ['Python' '1' '2' '3']  (tudo vira texto!)
```

2. **Fatias são *visões*, não cópias:**

```
import numpy as np

a = [1, 2, 3, 4, 5];            b = a[1:];  b[0] = 42   # lista: a NÃO muda
a = np.array([1, 2, 3, 4, 5]);  b = a[1:];  b[0] = 42   # array: a MUDA!
print(a)                         # [ 1 42  3  4  5]
b = a[1:].copy()                 # para copiar de verdade
```

Fonte: `code/Cap5/prog.5.43_numpy_erros.py`

Atividade

1. Crie um array com 100 pontos de 0 a 2π e calcule $\sin$ e $\cos$; verifique que $\sin^2 + \cos^2 = 1$ para todos os pontos.
2. Dado o array de temperaturas de 24 horas, calcule média, máxima, hora da máxima, e quantas horas ficaram acima da média.
3. Resolva o circuito de duas malhas (sistema 3×3) com `np.linalg.solve`.
4. Reescreva, com NumPy, o programa Monte Carlo para estimar π sem nenhum laço `for`: `x = np.random.rand(n); y = np.random.rand(n); 4 * np.mean(x**2 + y**2 < 1)`

Para praticar: exercícios do livro

Site do livro – Capítulo 5

- Seção 5.14 (NumPy) do livro; as médias ponderadas e filtros da aula podem ser comparados com os exercícios de listas **5.7 a 5.13**
- Refaça com NumPy: **5.14** (fatiamento negativo), **5.25** (divisíveis por 7 e não por 3)

Enunciados e soluções comentadas em
`https://www.araujo.eng.uerj.br/livro/exercicios/cap5/`

Exercícios adicionais (sem solução): `https://www.araujo.eng.uerj.br/livro/extras/`

Resumo da aula

- `np.array`: elementos de um só tipo, operações elemento a elemento
- `arange`, `linspace`, `zeros`, `ones`; funções universais `sin`, `exp`, `sqrt`...
- Filtros booleanos: `a[a > 6]`; `axis` define em que direção agregar
- Álgebra linear: `@`, `np.linalg.solve`, `polyfit`
- Cuidado: fatias são visões; use `.copy()`
- Próxima aula: strings