NumPy - Do Básico ao Intermediário

Sumário

1. Introdução ao NumPy
2. Arrays NumPy: Criação e Propriedades
3. Indexação e Fatiamento
4. Operações Matemáticas
5. Broadcasting
6. Funções Universais (ufuncs)
7. Manipulação de Forma e Dimensões
8. Álgebra Linear
9. Estatística com NumPy
10. Entrada e Saída de Dados
11. Arrays Estruturados
12. Aplicações Práticas
13. Armadilhas Comuns (Common Pitfalls)
14. Boas Práticas e Otimização
15. Tabela de Referência Rápida
16. Recursos Adicionais

---

1. Introdução ao NumPy

NumPy (Numerical Python) é uma biblioteca muito utilizada em computação científica com Python. Ela trabalha com arrays e matrizes multidimensionais e possui funções matemáticas voltadas para esse tipo de estrutura.

Por que usar NumPy?

- **Desempenho**: operações numéricas são feitas sem laços explícitos em Python, o que torna os cálculos mais rápidos
- **Uso de memória**: arrays usam menos memória do que listas comuns quando contém muitos elementos
- **Recursos**: funções matemáticas e estatísticas prontas para uso
- **Integração**: base de bibliotecas como Pandas, SciPy e scikit-learn

Instalação e Importação

```
# Instalação (se necessário)
# pip install numpy

# Importação padrão
import numpy as np

# Verificar versão
print(np.__version__)
```

---

2. Arrays NumPy: Criação e Propriedades

Um array NumPy é uma coleção de valores organizados em uma ou mais dimensões, em que todos os elementos têm o mesmo tipo.

Métodos de Criação

`np.array(object, dtype=None)`

Cria um array a partir de uma lista ou tupla Python.

```
# Array 1D
arr1d = np.array([1, 2, 3, 4, 5])
print(arr1d)  # [1 2 3 4 5]

# Array 2D
arr2d = np.array([[1, 2, 3], [4, 5, 6]])
print(arr2d)
# [[1 2 3]
#  [4 5 6]]

# Especificando tipo de dado
arr_float = np.array([1, 2, 3], dtype=float)
print(arr_float)  # [1. 2. 3.]
```

`np.zeros(shape, dtype=float)`

Cria um array preenchido com zeros.

```
zeros_1d = np.zeros(5)
print(zeros_1d)  # [0. 0. 0. 0. 0.]

zeros_2d = np.zeros((3, 4))
print(zeros_2d)
# [[0. 0. 0. 0.]
#  [0. 0. 0. 0.]
#  [0. 0. 0. 0.]]
```

`np.ones(shape, dtype=float)`

Cria um array preenchido com uns.

```
ones_2d = np.ones((2, 3))
print(ones_2d)
# [[1. 1. 1.]
#  [1. 1. 1.]]
```

`np.full(shape, fill_value, dtype=None)`

Cria um array preenchido com um valor específico.

```
full_arr = np.full((2, 2), 7)
print(full_arr)
# [[7 7]
#  [7 7]]
```

`np.arange(start, stop, step)`

Cria um array com valores espaçados uniformemente dentro de um intervalo.

```
arr_range = np.arange(0, 10, 2)
print(arr_range)  # [0 2 4 6 8]
```

`np.linspace(start, stop, num)`

Cria um array com `num` valores igualmente espaçados entre `start` e `stop`.

```
lin_arr = np.linspace(0, 1, 5)
print(lin_arr)  # [0.   0.25 0.5  0.75 1.  ]
```

`np.eye(N, M=None, k=0)`

Cria uma matriz identidade.

```
identity = np.eye(3)
print(identity)
# [[1. 0. 0.]
#  [0. 1. 0.]
#  [0. 0. 1.]]
```

`np.random.random(size)`

Cria um array com valores aleatórios entre 0 e 1.

```
random_arr = np.random.random((2, 3))
print(random_arr)
# [[0.5488135  0.71518937 0.60276338]
#  [0.54488318 0.4236548  0.64589411]]
```

Propriedades de Arrays

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

# ndim: número de dimensões
print(arr.ndim)  # 2

# shape: formato do array (linhas, colunas, ...)
print(arr.shape)  # (2, 3)

# size: número total de elementos
print(arr.size)  # 6

# dtype: tipo de dados dos elementos
print(arr.dtype)  # int64

# itemsize: tamanho em bytes de cada elemento
print(arr.itemsize)  # 8

# nbytes: total de bytes consumidos
print(arr.nbytes)  # 48
```

Exercícios

1. Crie um array 1D com os números de 10 a 20.
2. Crie uma matriz 4x4 preenchida com o valor 5.
3. Crie um array com 10 valores igualmente espaçados entre 0 e 100.
4. Crie uma matriz identidade 5x5.
5. Crie um array 3D com forma (2, 3, 4) preenchido com zeros.

Desafios

1. Crie uma matriz de tabuleiro de xadrez 8x8 (alternando 0 e 1).
2. Gere uma matriz 5x5 de números aleatórios e normalize-a (valores entre 0 e 1).

---

3. Indexação e Fatiamento

A indexação serve para acessar valores específicos do array. O fatiamento permite selecionar partes maiores, como intervalos ou blocos.

Indexação Básica

```
# Array 1D
arr = np.array([10, 20, 30, 40, 50])

# Acessar primeiro elemento (índice 0)
print(arr[0])  # 10

# Acessar último elemento
print(arr[-1])  # 50

# Array 2D
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Acessar elemento na linha 1, coluna 2
print(arr2d[1, 2])  # 6

# Acessar linha completa
print(arr2d[0])  # [1 2 3]
```

Fatiamento (Slicing)

Sintaxe: `arr[start:stop:step]`

```
arr = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

# Elementos do índice 2 ao 5 (exclusivo)
print(arr[2:5])  # [2 3 4]

# Elementos do início até o índice 5
print(arr[:5])  # [0 1 2 3 4]

# Elementos do índice 5 até o fim
print(arr[5:])  # [5 6 7 8 9]

# Todos os elementos com passo 2
print(arr[::2])  # [0 2 4 6 8]

# Inverter array
print(arr[::-1])  # [9 8 7 6 5 4 3 2 1 0]
```

Fatiamento Multidimensional

```
arr2d = np.array([[1, 2, 3, 4],
                  [5, 6, 7, 8],
                  [9, 10, 11, 12]])

# Primeiras duas linhas, primeiras três colunas
print(arr2d[:2, :3])
# [[1 2 3]
#  [5 6 7]]

# Todas as linhas, coluna 2
print(arr2d[:, 2])  # [ 3  7 11]

# Linhas 1 e 2, colunas 1 e 2
print(arr2d[1:3, 1:3])
# [[ 6  7]
#  [10 11]]
```

Indexação Booleana

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

# Criar máscara booleana
mask = arr > 3
print(mask)  # [False False False  True  True  True]

# Aplicar máscara
print(arr[mask])  # [4 5 6]

# Forma compacta
print(arr[arr > 3])  # [4 5 6]

# Múltiplas condições
print(arr[(arr > 2) & (arr < 5)])  # [3 4]
```

Indexação Fancy

```
arr = np.array([10, 20, 30, 40, 50])

# Selecionar elementos por índices
indices = [0, 2, 4]
print(arr[indices])  # [10 30 50]

# Array 2D
arr2d = np.array([[1, 2], [3, 4], [5, 6]])

# Selecionar linhas específicas
print(arr2d[[0, 2]])
# [[1 2]
#  [5 6]]
```

Modificação com Indexação

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

# Modificar elemento único
arr[0] = 10
print(arr)  # [10  2  3  4  5]

# Modificar múltiplos elementos
arr[1:4] = [20, 30, 40]
print(arr)  # [10 20 30 40  5]

# Modificar com condição
arr[arr > 30] = 0
print(arr)  # [10 20 30  0  0]
```

Exercícios

1. Dado o array `arr = np.arange(20)`, extraia os elementos de índice par.
2. Crie uma matriz 5x5 e extraia a submatriz central 3x3.
3. Dado um array, substitua todos os valores negativos por zero.
4. Extraia a diagonal principal de uma matriz 4x4.
5. Selecione todos os elementos maiores que a média de um array.

Desafios

1. Dada uma matriz, extraia todas as bordas (primeira e última linha/coluna).
2. Crie um array de 100 elementos aleatórios e encontre os índices dos 10 maiores valores.

---

4. Operações Matemáticas

No NumPy, operações matemáticas são aplicadas diretamente aos valores do array, elemento por elemento.

Operações Aritméticas Básicas

```
a = np.array([1, 2, 3, 4])
b = np.array([10, 20, 30, 40])

# Adição
print(a + b)  # [11 22 33 44]

# Subtração
print(b - a)  # [ 9 18 27 36]

# Multiplicação
print(a * b)  # [ 10  40  90 160]

# Divisão
print(b / a)  # [10. 10. 10. 10.]

# Potenciação
print(a ** 2)  # [ 1  4  9 16]

# Módulo
print(b % 3)  # [1 2 0 1]
```

Operações com Escalares

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

# Adicionar escalar
print(arr + 10)  # [11 12 13 14 15]

# Multiplicar por escalar
print(arr * 2)  # [ 2  4  6  8 10]

# Dividir por escalar
print(arr / 2)  # [0.5 1.  1.5 2.  2.5]
```

Funções Matemáticas

`np.sqrt(x)`

Calcula a raiz quadrada de cada elemento.

```
arr = np.array([1, 4, 9, 16, 25])
print(np.sqrt(arr))  # [1. 2. 3. 4. 5.]
```

`np.exp(x)`

Calcula e^x para cada elemento.

```
arr = np.array([0, 1, 2])
print(np.exp(arr))  # [1.         2.71828183 7.3890561 ]
```

`np.log(x)`, `np.log10(x)`, `np.log2(x)`

Calcula logaritmo natural, base 10 e base 2.

```
arr = np.array([1, 10, 100])
print(np.log10(arr))  # [0. 1. 2.]
```

Funções Trigonométricas

```
angles = np.array([0, np.pi/2, np.pi])

print(np.sin(angles))
# [ 0.00000000e+00  1.00000000e+00  1.22464680e-16]
# Nota: O valor 1.22464680e-16 é essencialmente zero (erro de ponto flutuante)

print(np.cos(angles))
# [ 1.00000000e+00  6.12323400e-17 -1.00000000e+00]
# Nota: O valor 6.12323400e-17 é essencialmente zero (erro de ponto flutuante)

# Para resultados mais limpos, podemos arredondar:
print(np.round(np.sin(angles), 10))  # [0. 1. 0.]
print(np.round(np.cos(angles), 10))  # [ 1.  0. -1.]
```

`np.abs(x)` **ou** `np.absolute(x)`

Calcula o valor absoluto.

```
arr = np.array([-1, -2, 3, -4])
print(np.abs(arr))  # [1 2 3 4]
```

`np.round(x, decimals=0)`

Arredonda para o número de decimais especificado.

```
arr = np.array([1.234, 2.567, 3.891])
print(np.round(arr, 2))  # [1.23 2.57 3.89]
```

`np.floor(x)`, `np.ceil(x)`

Arredonda para baixo e para cima, respectivamente.

```
arr = np.array([1.2, 2.5, 3.9])
print(np.floor(arr))  # [1. 2. 3.]
print(np.ceil(arr))   # [2. 3. 4.]
```

Funções de Agregação

`np.sum(arr, axis=None)`

Calcula a soma dos elementos.

```
# Entendendo o parâmetro axis:
arr = np.array([[1, 2, 3], [4, 5, 6]])
print("Array original (2, 3):")
print(arr)

# axis=None (padrão): opera em todo o array
print("\nSoma total (axis=None):", np.sum(arr))  # 21

# axis=0: opera AO LONGO das linhas (colapsa linhas, mantém colunas)
print("Soma por coluna (axis=0):", np.sum(arr, axis=0))  # [5 7 9]
# Resultado tem forma (3,) - uma dimensão a menos

# axis=1: opera AO LONGO das colunas (colapsa colunas, mantém linhas)
print("Soma por linha (axis=1):", np.sum(arr, axis=1))  # [6 15]
# Resultado tem forma (2,) - uma dimensão a menos

# Regra mnemônica: axis=0 para operações "verticais", axis=1 para "horizontais"
```

`np.mean(arr, axis=None)`

Calcula a média aritmética.

```
arr = np.array([1, 2, 3, 4, 5])
print(np.mean(arr))  # 3.0
```

`np.std(arr, axis=None)`, `np.var(arr, axis=None)`

Calcula o desvio padrão e a variância.

```
arr = np.array([1, 2, 3, 4, 5])
print(np.std(arr))  # 1.4142135623730951
print(np.var(arr))  # 2.0
```

`np.min(arr)`, `np.max(arr)`

Encontra o valor mínimo e máximo.

```
arr = np.array([3, 1, 4, 1, 5, 9, 2, 6])
print(np.min(arr))  # 1
print(np.max(arr))  # 9
```

`np.argmin(arr)`, `np.argmax(arr)`

Retorna o índice do valor mínimo e máximo.

```
arr = np.array([3, 1, 4, 1, 5, 9, 2, 6])
print(np.argmin(arr))  # 1
print(np.argmax(arr))  # 5
```

Exercícios

1. Crie um array de 10 números aleatórios e calcule sua média e desvio padrão.
2. Dado um array de ângulos em graus, converta-os para radianos e calcule seus senos.
3. Crie uma matriz 3x3 e calcule a soma de cada linha e cada coluna.
4. Normalize um array (subtraia a média e divida pelo desvio padrão).
5. Encontre os índices dos 3 maiores valores em um array.

Desafios

1. Calcule a distância euclidiana entre dois pontos representados como arrays.

2. Implemente a função softmax: $\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$

---

5. Broadcasting

Broadcasting é o mecanismo que permite combinar arrays de formatos diferentes em uma mesma operação, desde que eles sejam compatíveis.

Para que o broadcasting funcione, o NumPy segue algumas regras simples de compatibilidade entre as dimensões.

Regras de Broadcasting

1. Se os arrays não têm o mesmo número de dimensões, a forma do array com menos dimensões é preenchida com 1s **à esquerda** até que ambos tenham o mesmo número de dimensões.
2. Arrays são compatíveis em uma dimensão se tiverem o mesmo tamanho **ou** se um deles tiver tamanho 1 naquela dimensão.
3. Após a verificação de compatibilidade, arrays são "expandidos" (conceitualmente) ao longo das dimensões onde tinham tamanho 1 para corresponder ao tamanho da outra dimensão.
4. Se em qualquer dimensão os tamanhos não forem compatíveis (diferentes e nenhum deles igual a 1), o NumPy retorna um erro.

Exemplos de Broadcasting

```
# Escalar com array
arr = np.array([1, 2, 3])
print(arr + 10)  # [11 12 13]

# Array 1D com array 2D
arr1d = np.array([1, 2, 3])
arr2d = np.array([[10], [20], [30]])

result = arr2d + arr1d
print(result)
# [[11 12 13]
#  [21 22 23]
#  [31 32 33]]

# Explicação do broadcasting:
# arr2d tem forma: (3, 1)
# arr1d tem forma: (3,) -> é expandido para (1, 3)
# Resultado final após broadcast: (3, 3)
```

Exemplos Práticos

```
# Normalizar colunas de uma matriz
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Média por coluna
col_means = np.mean(matrix, axis=0)  # [4. 5. 6.]

# Subtrair média de cada coluna
normalized = matrix - col_means
print(normalized)
# [[-3. -3. -3.]
#  [ 0.  0.  0.]
#  [ 3.  3.  3.]]

# Criar uma grade de coordenadas
x = np.array([0, 1, 2])
y = np.array([0, 10, 20, 30])

# Broadcasting para criar grade
X = x.reshape(1, -1)  # (1, 3)
Y = y.reshape(-1, 1)  # (4, 1)

Z = X + Y
print(Z)
# [[ 0  1  2]
#  [10 11 12]
#  [20 21 22]
#  [30 31 32]]
```

Expandindo Dimensões

`np.newaxis`

Adiciona uma nova dimensão ao array.

```
arr = np.array([1, 2, 3])
print(arr.shape)  # (3,)

# Adicionar dimensão como linha
arr_row = arr[np.newaxis, :]
print(arr_row.shape)  # (1, 3)

# Adicionar dimensão como coluna
arr_col = arr[:, np.newaxis]
print(arr_col.shape)  # (3, 1)
```

`np.expand_dims(arr, axis)`

Expande a forma do array inserindo uma nova dimensão.

```
arr = np.array([1, 2, 3])
expanded = np.expand_dims(arr, axis=0)
print(expanded.shape)  # (1, 3)
```

Exercícios

1. Normalize cada linha de uma matriz 4x5 (subtraia a média da linha).
2. Crie uma matriz de distâncias entre todos os pares de pontos dados dois arrays de coordenadas.
3. Multiplique cada coluna de uma matriz por um vetor de pesos.
4. Adicione um vetor de bias a cada linha de uma matriz.
5. Crie uma tabela de multiplicação 10x10 usando broadcasting.

Desafios

1. Implemente a função de distância euclidiana entre todas as combinações de dois conjuntos de pontos usando broadcasting.
2. Crie uma matriz RGB (100x100x3) onde cada pixel tem cor baseada em sua posição (x, y).

---

6. Funções Universais (ufuncs)

Funções universais (ufuncs) aplicam a mesma operação a cada elemento do array e seguem as regras de broadcasting do NumPy.

Características das ufuncs

- Operação vetorizada, geralmente mais eficiente que loops Python em arrays de tamanho moderado ou grande
- Suportam broadcasting automático
- Podem aceitar argumentos de saída opcionais
- Possuem métodos especiais (reduce, accumulate, outer, etc.)

ufuncs Matemáticas

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

# Unárias
print(np.sqrt(arr))      # Raiz quadrada
print(np.square(arr))    # Quadrado
print(np.exp(arr))       # Exponencial
print(np.log(arr))       # Logaritmo natural
print(np.sign(arr))      # Sinal (-1, 0, 1)

# Binárias
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

print(np.add(a, b))        # Adição
print(np.subtract(a, b))   # Subtração
print(np.multiply(a, b))   # Multiplicação
print(np.divide(a, b))     # Divisão
print(np.power(a, b))      # Potência
print(np.maximum(a, b))    # Máximo elemento a elemento
print(np.minimum(a, b))    # Mínimo elemento a elemento
```

Métodos de ufuncs

`reduce()`

Aplica a operação cumulativamente aos elementos.

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

# Soma acumulada
print(np.add.reduce(arr))  # 15 (1+2+3+4+5)

# Produto acumulado
print(np.multiply.reduce(arr))  # 120 (1*2*3*4*5)
```

`accumulate()`

Armazena resultados intermediários.

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

# Soma acumulativa
print(np.add.accumulate(arr))  # [ 1  3  6 10 15]

# Produto acumulativo
print(np.multiply.accumulate(arr))  # [  1   2   6  24 120]
```

`outer()`

Aplica a operação a todos os pares de elementos.

```
a = np.array([1, 2, 3])
b = np.array([10, 20, 30])

# Produto externo
print(np.multiply.outer(a, b))
# [[10 20 30]
#  [20 40 60]
#  [30 60 90]]
```

`at()`

Aplica a operação em índices específicos.

```
arr = np.array([1, 2, 3, 4, 5])
indices = [0, 2, 4]

np.add.at(arr, indices, 10)
print(arr)  # [11  2 13  4 15]
```

ufuncs de Comparação

```
a = np.array([1, 2, 3, 4, 5])
b = np.array([5, 4, 3, 2, 1])

print(np.equal(a, b))         # [False False  True False False]
print(np.not_equal(a, b))     # [ True  True False  True  True]
print(np.less(a, b))          # [ True  True False False False]
print(np.less_equal(a, b))    # [ True  True  True False False]
print(np.greater(a, b))       # [False False False  True  True]
print(np.greater_equal(a, b)) # [False False  True  True  True]
```

ufuncs Lógicas

```
a = np.array([True, True, False, False])
b = np.array([True, False, True, False])

print(np.logical_and(a, b))   # [ True False False False]
print(np.logical_or(a, b))    # [ True  True  True False]
print(np.logical_not(a))      # [False False  True  True]
print(np.logical_xor(a, b))   # [False  True  True False]
```

Exercícios

1. Use `np.add.reduce()` para somar todos os elementos de uma matriz 2D.
2. Calcule a soma acumulativa de um array usando `accumulate()`.
3. Crie uma tabela de verdade para operações lógicas AND, OR e XOR.
4. Use `outer()` para criar uma matriz de distâncias Manhattan entre dois conjuntos de pontos 1D.
5. Incremente elementos específicos de um array usando `at()`.

Desafios

1. Implemente uma versão vetorizada da função sigmoid usando ufuncs: $ \sigma(x) = \frac{1}{1 + e^{-x}}$
2. Calcule o produto interno de dois vetores usando apenas ufuncs (sem `np.dot`).

---

7. Manipulação de Forma e Dimensões

O NumPy permite reorganizar a forma de um array sem alterar os valores armazenados.

`reshape()`

Retorna um array com nova forma sem alterar os dados.

```
arr = np.arange(12)
print(arr)  # [ 0  1  2  3  4  5  6  7  8  9 10 11]

# Reshape para 2D
reshaped = arr.reshape(3, 4)
print(reshaped)
# [[ 0  1  2  3]
#  [ 4  5  6  7]
#  [ 8  9 10 11]]

# Reshape para 3D
reshaped_3d = arr.reshape(2, 2, 3)
print(reshaped_3d.shape)  # (2, 2, 3)

# Usar -1 para inferir dimensão
reshaped_auto = arr.reshape(3, -1)
print(reshaped_auto)
# [[ 0  1  2  3]
#  [ 4  5  6  7]
#  [ 8  9 10 11]]
```

`ravel()` e `flatten()`

Retornam um array 1D.

```
arr2d = np.array([[1, 2, 3], [4, 5, 6]])

# ravel() retorna uma view (quando possível) - mais eficiente
raveled = arr2d.ravel()
print(raveled)  # [1 2 3 4 5 6]

# CUIDADO: Modificar raveled pode afetar arr2d
raveled[0] = 999
print(arr2d)  # [[999 2 3] [4 5 6]] - foi modificado!

# flatten() sempre retorna uma cópia - mais seguro
arr2d = np.array([[1, 2, 3], [4, 5, 6]])
flattened = arr2d.flatten()
flattened[0] = 999
print(arr2d)  # [[1 2 3] [4 5 6]] - não foi modificado
```

`transpose()` e `T`

Transpõe as dimensões do array.

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr.shape)  # (2, 3)

# Transposição simples
transposed = arr.T
print(transposed.shape)  # (3, 2)
print(transposed)
# [[1 4]
#  [2 5]
#  [3 6]]

# Para arrays multidimensionais
arr3d = np.arange(24).reshape(2, 3, 4)
# Especificar ordem de eixos
transposed_3d = arr3d.transpose(2, 0, 1)
print(transposed_3d.shape)  # (4, 2, 3)
```

`swapaxes()`

Troca dois eixos de um array.

```
arr = np.arange(24).reshape(2, 3, 4)
print(arr.shape)  # (2, 3, 4)

swapped = arr.swapaxes(0, 2)
print(swapped.shape)  # (4, 3, 2)
```

`squeeze()` e `expand_dims()`

`squeeze()`

Remove dimensões de tamanho 1.

```
arr = np.array([[[1], [2], [3]]])
print(arr.shape)  # (1, 3, 1)

squeezed = arr.squeeze()
print(squeezed.shape)  # (3,)
print(squeezed)  # [1 2 3]
```

`expand_dims()`

Adiciona uma nova dimensão.

```
arr = np.array([1, 2, 3])
print(arr.shape)  # (3,)

expanded = np.expand_dims(arr, axis=0)
print(expanded.shape)  # (1, 3)

expanded2 = np.expand_dims(arr, axis=1)
print(expanded2.shape)  # (3, 1)
```

Concatenação

`np.concatenate(arrays, axis=0)`

Junta arrays ao longo de um eixo existente.

```
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6], [7, 8]])

# Concatenar verticalmente (axis=0)
result = np.concatenate([a, b], axis=0)
print(result)
# [[1 2]
#  [3 4]
#  [5 6]
#  [7 8]]

# Concatenar horizontalmente (axis=1)
result = np.concatenate([a, b], axis=1)
print(result)
# [[1 2 5 6]
#  [3 4 7 8]]
```

`np.vstack()` e `np.hstack()`

Empilham arrays vertical e horizontalmente.

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

# Empilhamento vertical
vstacked = np.vstack([a, b])
print(vstacked)
# [[1 2 3]
#  [4 5 6]]

# Empilhamento horizontal
hstacked = np.hstack([a, b])
print(hstacked)  # [1 2 3 4 5 6]
```

`np.stack()`

Junta arrays ao longo de uma nova dimensão.

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

# Stack ao longo do eixo 0
stacked = np.stack([a, b], axis=0)
print(stacked)
# [[1 2 3]
#  [4 5 6]]

# Stack ao longo do eixo 1
stacked = np.stack([a, b], axis=1)
print(stacked)
# [[1 4]
#  [2 5]
#  [3 6]]
```

Divisão

`np.split(arr, indices_or_sections, axis=0)`

Divide um array em múltiplas submatrizes.

```
arr = np.arange(9)

# Dividir em 3 partes iguais
parts = np.split(arr, 3)
print(parts)  # [array([0, 1, 2]), array([3, 4, 5]), array([6, 7, 8])]

# Dividir em índices específicos
parts = np.split(arr, [2, 5])
print(parts)  # [array([0, 1]), array([2, 3, 4]), array([5, 6, 7, 8])]
```

`np.vsplit()` e `np.hsplit()`

Divisão vertical e horizontal.

```
arr = np.arange(12).reshape(4, 3)

# Divisão vertical
vsplit_result = np.vsplit(arr, 2)
print(vsplit_result[0])
# [[0 1 2]
#  [3 4 5]]

# Divisão horizontal
hsplit_result = np.hsplit(arr, 3)
print(hsplit_result[0])
# [[ 0]
#  [ 3]
#  [ 6]
#  [ 9]]
```

Repetição

`np.repeat(arr, repeats, axis=None)`

Repete elementos de um array.

```
arr = np.array([1, 2, 3])

# Repetir cada elemento 3 vezes
repeated = np.repeat(arr, 3)
print(repeated)  # [1 1 1 2 2 2 3 3 3]

# Repetir ao longo de um eixo
arr2d = np.array([[1, 2], [3, 4]])
repeated = np.repeat(arr2d, 2, axis=0)
print(repeated)
# [[1 2]
#  [1 2]
#  [3 4]
#  [3 4]]
```

`np.tile(arr, reps)`

Constrói um array repetindo o array inteiro.

```
arr = np.array([1, 2, 3])

# Repetir o array inteiro 3 vezes
tiled = np.tile(arr, 3)
print(tiled)  # [1 2 3 1 2 3 1 2 3]

# Repetir em 2D
arr2d = np.array([[1, 2], [3, 4]])
tiled = np.tile(arr2d, (2, 3))
print(tiled)
# [[1 2 1 2 1 2]
#  [3 4 3 4 3 4]
#  [1 2 1 2 1 2]
#  [3 4 3 4 3 4]]
```

Exercícios

1. Reshape um array 1D de 20 elementos em uma matriz 4x5.
2. Transpor uma matriz 3x4 e verificar suas dimensões.
3. Concatenar três arrays 1D em um único array 2D (cada array como uma linha).
4. Dividir uma matriz 6x6 em quatro submatrizes 3x3.
5. Criar um padrão de tabuleiro 8x8 usando `tile()`.

Desafios

1. Dado um array 3D de forma (2, 3, 4), reorganize-o para forma (4, 2, 3) usando apenas `reshape` e `transpose`.
2. Implemente uma função que rotaciona uma matriz 90 graus no sentido horário usando apenas operações de forma.

---

8. Álgebra Linear

As operações de álgebra linear no NumPy ficam concentradas no módulo `np.linalg`.

Produto de Matrizes

`np.dot(a, b)` ou `@`

Calcula o produto interno de vetores ou produto matricial de matrizes.

```
# produto interno de vetores
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
print(np.dot(a, b))  # 32 (1*4 + 2*5 + 3*6)

# Produto matricial
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
print(np.dot(A, B))
# [[19 22]
#  [43 50]]

# Operador @
print(A @ B)
# [[19 22]
#  [43 50]]
```

`np.matmul(a, b)` ou `@`

Produto matricial (não funciona com escalares).

```
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
print(np.matmul(A, B))
# [[19 22]
#  [43 50]]
```

Determinante

`np.linalg.det(a)`

Calcula o determinante de uma matriz.

```
A = np.array([[1, 2], [3, 4]])
det = np.linalg.det(A)
print(det)  # -2.0
```

Inversa

`np.linalg.inv(a)`

Calcula a inversa de uma matriz.

```
A = np.array([[1, 2], [3, 4]])
A_inv = np.linalg.inv(A)
print(A_inv)
# [[-2.   1. ]
#  [ 1.5 -0.5]]

# Verificação: A @ A_inv = I
print(A @ A_inv)
# [[1.00000000e+00 0.00000000e+00]
#  [8.88178420e-16 1.00000000e+00]]
```

Autovalores e Autovetores

`np.linalg.eig(a)`

Calcula autovalores e autovetores.

```
A = np.array([[1, 2], [2, 1]])
eigenvalues, eigenvectors = np.linalg.eig(A)

print("Autovalores:", eigenvalues)     # [ 3. -1.]
print("Autovetores:\n", eigenvectors)
# [[ 0.70710678 -0.70710678]
#  [ 0.70710678  0.70710678]]
```

Solução de Sistemas Lineares

`np.linalg.solve(a, b)`

Resolve o sistema linear Ax = b.

```
# Sistema: 2x + y = 5
#          3x - 2y = 4
A = np.array([[2, 1], [3, -2]])
b = np.array([5, 4])

x = np.linalg.solve(A, b)
print(x)  # [2. 1.]

# Verificação
print(A @ x)  # [5. 4.]
```

Norma

`np.linalg.norm(x, ord=None)`

Calcula a norma de um vetor ou matriz.

```
v = np.array([3, 4])

# Norma L2 (euclidiana)
print(np.linalg.norm(v))  # 5.0

# Norma L1
print(np.linalg.norm(v, ord=1))  # 7.0

# Norma infinita
print(np.linalg.norm(v, ord=np.inf))  # 4.0
```

Traço

`np.trace(a)`

Calcula o traço de uma matriz (soma da diagonal principal).

```
A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(np.trace(A))  # 15 (1 + 5 + 9)
```

Posto (Rank)

`np.linalg.matrix_rank(a)`

Calcula o posto de uma matriz.

```
A = np.array([[1, 2], [2, 4]])
print(np.linalg.matrix_rank(A))  # 1
```

Decomposições

SVD (Decomposição em Valores Singulares)

```
A = np.array([[1, 2], [3, 4], [5, 6]])
U, s, Vt = np.linalg.svd(A)

print("U shape:", U.shape)      # (3, 3)
print("s shape:", s.shape)      # (2,)
print("Vt shape:", Vt.shape)    # (2, 2)
```

QR

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

print("Q:\n", Q)
print("R:\n", R)
```

Produto Externo

`np.outer(a, b)`

Calcula o produto externo de dois vetores.

```
a = np.array([1, 2, 3])
b = np.array([4, 5])

result = np.outer(a, b)
print(result)
# [[ 4  5]
#  [ 8 10]
#  [12 15]]
```

Produto Interno

`np.inner(a, b)`

Calcula o produto interno de vetores.

```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
print(np.inner(a, b))  # 32
```

Exercícios

1. Calcule o determinante e a inversa de uma matriz 3x3.
2. Resolva o sistema linear: 3x + 2y - z = 1, 2x - 2y + 4z = -2, -x + 0.5y - z = 0.
3. Encontre os autovalores e autovetores de uma matriz simétrica 3x3.
4. Calcule a norma L2 de um vetor 5D.
5. Verifique se duas matrizes são ortogonais multiplicando-as por suas transpostas.

Desafios

1. Implemente o método dos mínimos quadrados para ajuste linear usando operações matriciais.
2. Use SVD para comprimir uma imagem (representada como matriz).

---

9. Estatística com NumPy

O NumPy inclui funções básicas para calcular estatísticas diretamente sobre arrays.

Medidas de Tendência Central

`np.mean(a, axis=None)`

Calcula a média aritmética.

```
arr = np.array([1, 2, 3, 4, 5])
print(np.mean(arr))  # 3.0

# Média por eixo
arr2d = np.array([[1, 2, 3], [4, 5, 6]])
print(np.mean(arr2d, axis=0))  # [2.5 3.5 4.5]
print(np.mean(arr2d, axis=1))  # [2. 5.]
```

`np.median(a, axis=None)`

Calcula a mediana.

```
arr = np.array([1, 2, 3, 4, 5, 100])
print(np.median(arr))  # 3.5
```

Medidas de Dispersão

`np.std(a, axis=None, ddof=0)`

Calcula o desvio padrão.

```
arr = np.array([1, 2, 3, 4, 5])
# desvio padrão populacional (ddof=0)
print(np.std(arr))  # 1.4142135623730951

# Com correção de Bessel (ddof=1)
print(np.std(arr, ddof=1))  # 1.5811388300841898
```

`np.var(a, axis=None, ddof=0)`

Calcula a variância.

```
arr = np.array([1, 2, 3, 4, 5])
print(np.var(arr))  # 2.0
```

Valores Extremos

`np.min(a)`, `np.max(a)`

Valor mínimo e máximo.

```
arr = np.array([3, 1, 4, 1, 5, 9, 2, 6])
print(np.min(arr))  # 1
print(np.max(arr))  # 9
```

`np.ptp(a)`

Calcula o range (diferença entre o valor máximo e mínimo).

```
arr = np.array([1, 2, 3, 4, 5])
print(np.ptp(arr))  # 4 (5 - 1)
```

Percentis e Quantis

`np.percentile(a, q, axis=None)`

Calcula o percentil especificado.

```
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
print(np.percentile(arr, 50))   # 5.5 (mediana)
print(np.percentile(arr, 25))   # 3.25 (primeiro quartil)
print(np.percentile(arr, 75))   # 7.75 (terceiro quartil)
```

`np.quantile(a, q, axis=None)`

Similar ao percentile, mas q está em [0, 1].

```
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
print(np.quantile(arr, 0.5))    # 5.5
print(np.quantile(arr, 0.25))   # 3.25
```

Correlação e Covariância

`np.corrcoef(x, y=None)`

Calcula a matriz de correlação de Pearson.

```
x = np.array([1, 2, 3, 4, 5])
y = np.array([2, 4, 6, 8, 10])

corr_matrix = np.corrcoef(x, y)
print(corr_matrix)
# [[1. 1.]
#  [1. 1.]]
```

`np.cov(x, y=None)`

Calcula a matriz de covariância.

```
x = np.array([1, 2, 3, 4, 5])
y = np.array([2, 4, 5, 4, 5])

cov_matrix = np.cov(x, y)
print(cov_matrix)
# [[2.5  1.75]
#  [1.75 1.7 ]]
```

Histograma

`np.histogram(a, bins=10, range=None)`

Calcula o histograma de um conjunto de dados.

```
data = np.random.randn(1000)
hist, bin_edges = np.histogram(data, bins=10)

print("Contagens:", hist)
print("Bordas dos bins:", bin_edges)
```

Dados Binários

`np.bincount(x)`

Conta o número de ocorrências de cada valor.

```
arr = np.array([0, 1, 1, 2, 2, 2, 3])
print(np.bincount(arr))  # [1 2 3 1]
```

Ordenação

`np.sort(a, axis=-1)`

Retorna uma cópia ordenada do array.

```
arr = np.array([3, 1, 4, 1, 5, 9, 2, 6])
print(np.sort(arr))  # [1 1 2 3 4 5 6 9]

# Ordenar por eixo
arr2d = np.array([[3, 1, 4], [2, 5, 1]])
print(np.sort(arr2d, axis=1))
# [[1 3 4]
#  [1 2 5]]
```

`np.argsort(a, axis=-1)`

Retorna os índices que ordenariam o array.

```
arr = np.array([3, 1, 4, 1, 5, 9, 2, 6])
indices = np.argsort(arr)
print(indices)  # [1 3 6 0 2 4 7 5]
print(arr[indices])  # [1 1 2 3 4 5 6 9]
```

Valores Únicos

`np.unique(ar, return_counts=False)`

Encontra valores únicos em um array.

```
arr = np.array([1, 2, 2, 3, 3, 3, 4])
unique_vals = np.unique(arr)
print(unique_vals)  # [1 2 3 4]

# Com contagens
unique_vals, counts = np.unique(arr, return_counts=True)
print(unique_vals)  # [1 2 3 4]
print(counts)       # [1 2 3 1]
```

Exercícios

1. Calcule média, mediana, desvio padrão e variância de um array de 100 números aleatórios.
2. Encontre o primeiro e terceiro quartis de um conjunto de dados.
3. Calcule a correlação entre duas variáveis simuladas.
4. Crie um histograma de dados normalmente distribuídos com 5 bins.
5. Encontre os 5 maiores valores em um array usando `argsort`.

Desafios

1. Implemente uma função para detectar outliers usando o método IQR (valores fora de Q1 - 1.5*IQR e Q3 + 1.5*IQR).
2. Calcule a correlação de Spearman (baseada em ranks) usando apenas funções NumPy.

10. Entrada e Saída de Dados

É possível salvar arrays em arquivos e carregá-los depois usando funções do NumPy.

Formato Binário (.npy e .npz)

`np.save(file, arr)`

Salva um array em formato binário .npy.

```
arr = np.array([1, 2, 3, 4, 5])
np.save('meu_array.npy', arr)
```

`np.load(file)`

Carrega um array de um arquivo .npy.

```
loaded_arr = np.load('meu_array.npy')
print(loaded_arr)  # [1 2 3 4 5]
```

`np.savez(file, *args, **kwds)`

Salva múltiplos arrays em um arquivo .npz comprimido.

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([[4, 5], [6, 7]])

np.savez('multiplos_arrays.npz', primeiro=arr1, segundo=arr2)
```

Carregando arquivo .npz

```
data = np.load('multiplos_arrays.npz')
print(data['primeiro'])  # [1 2 3]
print(data['segundo'])
# [[4 5]
#  [6 7]]
```

Formato Texto

`np.savetxt(fname, X, delimiter=' ')`

Salva um array em arquivo de texto.

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
np.savetxt('dados.txt', arr, delimiter=',')
```

`np.loadtxt(fname, delimiter=' ')`

Carrega dados de um arquivo de texto.

```
loaded = np.loadtxt('dados.txt', delimiter=',')
print(loaded)
# [[1. 2. 3.]
#  [4. 5. 6.]]
```

`np.genfromtxt(fname, delimiter=None)`

Lê dados de arquivos de texto e trata casos como valores ausentes.

```
# Arquivo CSV com cabeçalho
data = np.genfromtxt('dados.csv', delimiter=',', skip_header=1)
```

Exercícios

1. Crie uma matriz 5x5 de números aleatórios, salve em formato .npy e carregue novamente.
2. Salve três arrays diferentes em um único arquivo .npz.
3. Crie uma matriz e salve em formato CSV usando `savetxt`.
4. Carregue um arquivo de texto com dados separados por tabulação.
5. Salve e carregue uma matriz complexa (com números complexos).

Desafios

1. Implemente uma função que salva um array com metadados (como descrição, data de criação) em um formato que possa ser recuperado posteriormente.
2. Carregue um arquivo CSV grande de forma eficiente, processando apenas as colunas necessárias.

---

11. Arrays Estruturados

Arrays estruturados permitem guardar, em um mesmo array, dados de tipos diferentes, organizados por campos.

Criando Arrays Estruturados

```
# Definir dtype estruturado
dt = np.dtype([('nome', 'U10'), ('idade', 'i4'), ('peso', 'f8')])

# Criar array estruturado
dados = np.array([('Alice', 25, 55.5),
                  ('Bob', 30, 75.2),
                  ('Carlos', 35, 80.0)], dtype=dt)

print(dados)
# [('Alice', 25, 55.5) ('Bob', 30, 75.2) ('Carlos', 35, 80. )]
```

Acessando Campos

```
# Acessar campo específico
print(dados['nome'])   # ['Alice' 'Bob' 'Carlos']
print(dados['idade'])  # [25 30 35]

# Acessar elemento específico
print(dados[0])        # ('Alice', 25, 55.5)
print(dados[0]['nome'])  # Alice

# Modificar valores
dados['idade'][0] = 26
print(dados['idade'])  # [26 30 35]
```

Operações com Campos

```
# Calcular média de um campo
print(np.mean(dados['peso']))  # 70.23333333333333

# Filtrar baseado em condição
adultos = dados[dados['idade'] >= 30]
print(adultos)
# [('Bob', 30, 75.2) ('Carlos', 35, 80. )]
```

Tipos de Dados Compostos

```
# dtype com campos nomeados
dt = np.dtype({'names': ('x', 'y', 'cor'),
               'formats': ('f4', 'f4', 'U10')})

pontos = np.array([(0.0, 0.0, 'vermelho'),
                   (1.0, 1.0, 'azul'),
                   (2.0, 0.5, 'verde')], dtype=dt)

print(pontos['x'])    # [0. 1. 2.]
print(pontos['cor'])  # ['vermelho' 'azul' 'verde']
```

Arrays de Registros

```
# np.rec.array cria um array de registros
dados = np.rec.array([('Alice', 25, 55.5),
                      ('Bob', 30, 75.2)],
                     dtype=[('nome', 'U10'), ('idade', 'i4'), ('peso', 'f8')])

# Acesso por atributo
print(dados.nome)   # ['Alice' 'Bob']
print(dados.idade)  # [25 30]
```

Exercícios

1. Crie um array estruturado para armazenar informações de estudantes (nome, matrícula, nota).
2. Calcule a média das notas de todos os estudantes.
3. Filtre estudantes com nota acima de 7.0.
4. Ordene o array por nota de forma decrescente.
5. Adicione um novo campo (boolean) indicando se o estudante foi aprovado (nota >= 6.0).

Desafios

1. Crie um array estruturado para representar coordenadas 3D com timestamp e converta-o para um formato que possa ser salvo em CSV.
2. Implemente um sistema simples de banco de dados em memória usando arrays estruturados com operações de inserção, busca e atualização.

Aplicações Práticas

Aplicação 1: Processamento de Imagens

```
# Criar uma imagem simples (escala de cinza)
altura, largura = 100, 100
imagem = np.random.randint(0, 256, (altura, largura), dtype=np.uint8)

# Inverter cores
imagem_invertida = 255 - imagem

# Aplicar threshold
threshold = 128
imagem_binaria = (imagem > threshold).astype(np.uint8) * 255

# Adicionar borda
borda_espessura = 5
imagem_com_borda = imagem.copy()
imagem_com_borda[:borda_espessura, :] = 255
imagem_com_borda[-borda_espessura:, :] = 255
imagem_com_borda[:, :borda_espessura] = 255
imagem_com_borda[:, -borda_espessura:] = 255
```

Aplicação 2: Simulação de Monte Carlo

```
# Estimar π usando Monte Carlo
n_pontos = 1000000

# Gerar pontos aleatórios no quadrado [0,1] x [0,1]
x = np.random.random(n_pontos)
y = np.random.random(n_pontos)

# Calcular distância à origem
distancias = np.sqrt(x**2 + y**2)

# Contar pontos dentro do círculo de raio 1
dentro_circulo = np.sum(distancias <= 1)

# Estimar π
pi_estimado = 4 * dentro_circulo / n_pontos
print(f"Estimativa de π: {pi_estimado}")
print(f"Erro: {abs(pi_estimado - np.pi)}")
```

Aplicação 3: Análise de Séries Temporais

```
# Gerar série temporal com tendência e sazonalidade
t = np.linspace(0, 4*np.pi, 1000)
tendencia = 0.5 * t
sazonalidade = 10 * np.sin(t)
ruido = np.random.normal(0, 1, len(t))
serie = tendencia + sazonalidade + ruido

# Calcular média móvel
janela = 50
media_movel = np.convolve(serie, np.ones(janela)/janela, mode='valid')

# Calcular diferenças para remover tendência
diferencas = np.diff(serie)

# Estatísticas da série
print(f"Média: {np.mean(serie):.2f}")
print(f"Desvio padrão: {np.std(serie):.2f}")
print(f"Mínimo: {np.min(serie):.2f}")
print(f"Máximo: {np.max(serie):.2f}")
```

Aplicação 4: Regressão Linear

```
# Dados de exemplo
n = 100
x = np.linspace(0, 10, n)
y_real = 2 * x + 1
y = y_real + np.random.normal(0, 2, n)  # Com ruído

# Ajuste linear usando mínimos quadrados
# y = ax + b
# Solução: (a, b) = (X^T X)^(-1) X^T y
X = np.vstack([x, np.ones(n)]).T
coeficientes = np.linalg.lstsq(X, y, rcond=None)[0]
a, b = coeficientes

print(f"Coeficientes: a={a:.2f}, b={b:.2f}")

# Predições
y_pred = a * x + b

# Erro quadrático médio
mse = np.mean((y - y_pred)**2)
print(f"MSE: {mse:.2f}")

# R²
ss_res = np.sum((y - y_pred)**2)
ss_tot = np.sum((y - np.mean(y))**2)
r2 = 1 - (ss_res / ss_tot)
print(f"R²: {r2:.4f}")
```

Aplicação 5: Transformada de Fourier

```
# Criar sinal composto
t = np.linspace(0, 1, 1000)
freq1, freq2 = 5, 20
sinal = np.sin(2*np.pi*freq1*t) + 0.5*np.sin(2*np.pi*freq2*t)

# Aplicar FFT
fft_result = np.fft.fft(sinal)
frequencias = np.fft.fftfreq(len(t), t[1] - t[0])

# Magnitude do espectro
magnitude = np.abs(fft_result)

# Encontrar frequências dominantes
idx_positivos = frequencias > 0
freq_dominantes = frequencias[idx_positivos][np.argsort(magnitude[idx_positivos])[-2:]]
print(f"Frequências dominantes: {sorted(freq_dominantes)}")
```

Aplicação 6: K-Means Clustering (Simplificado)

```
# Gerar dados de exemplo
np.random.seed(42)
n_pontos = 300

# Cluster 1
cluster1 = np.random.randn(n_pontos//3, 2) + np.array([0, 0])
# Cluster 2
cluster2 = np.random.randn(n_pontos//3, 2) + np.array([5, 5])
# Cluster 3
cluster3 = np.random.randn(n_pontos//3, 2) + np.array([0, 5])

dados = np.vstack([cluster1, cluster2, cluster3])

# Algoritmo K-Means simplificado
k = 3
max_iter = 100

# Inicializar centroides aleatoriamente
indices_aleatorios = np.random.choice(len(dados), k, replace=False)
centroides = dados[indices_aleatorios]

for iteracao in range(max_iter):
    # Atribuir cada ponto ao centroide mais próximo
    distancias = np.sqrt(((dados[:, np.newaxis] - centroides)**2).sum(axis=2))
    labels = np.argmin(distancias, axis=1)

    # Atualizar centroides
    novos_centroides = np.array([dados[labels == i].mean(axis=0) for i in range(k)])

    # Verificar convergência
    if np.allclose(centroides, novos_centroides):
        print(f"Convergiu na iteração {iteracao}")
        break

    centroides = novos_centroides

print("Centroides finais:")
print(centroides)
```

Armadilhas Comuns (Common Pitfalls)

1. Alterações não intencionais ao usar views

```
# Problema
arr = np.array([1, 2, 3, 4, 5])
subset = arr[1:4]  # É uma view!
subset[0] = 999
print(arr)  # [1 999 3 4 5] - arr foi modificado!

# Solução: criar cópia explícita
subset = arr[1:4].copy()
subset[0] = 999
print(arr)  # [1 2 3 4 5] - arr não foi modificado
```

2. Comparação de Arrays com `==`

```
# Problema
arr1 = np.array([1, 2, 3])
arr2 = np.array([1, 2, 3])

if arr1 == arr2:  # ValueError: ambiguous!
    print("Iguais")

# Solução 1: usar np.array_equal()
if np.array_equal(arr1, arr2):
    print("Iguais")

# Solução 2: usar .all()
if (arr1 == arr2).all():
    print("Iguais")
```

3. Divisão inteira inesperada

```
# Problema
arr = np.array([1, 2, 3])
result = arr / 2  # Pode não ser o que você espera com arrays inteiros

# Solução: forçar float
result = arr / 2.0
# ou
result = arr.astype(float) / 2
```

4. Broadcasting Acidental

```
# Problema
matriz = np.random.rand(5, 3)
vetor = np.random.rand(5)

# Isto pode não fazer o que você espera:
resultado = matriz + vetor  # Adiciona vetor a cada COLUNA

# Solução: seja explícito
resultado = matriz + vetor[:, np.newaxis]  # Adiciona a cada linha
```

5. Perda de Precisão com float32

```
# Problema
arr = np.array([1e10, 1, -1e10], dtype=np.float32)
print(np.sum(arr))  # Pode dar 0.0 em vez de 1.0!

# Solução: use float64 quando precisão é crítica
arr = np.array([1e10, 1, -1e10], dtype=np.float64)
print(np.sum(arr))  # 1.0
```

Boas Práticas e Otimização

1. Vetorização vs Loops

```
# MAU: Usando loops
def soma_quadrados_loop(arr):
    resultado = 0
    for x in arr:
        resultado += x ** 2
    return resultado

# BOM: Vetorizado
def soma_quadrados_vet(arr):
    return np.sum(arr ** 2)

# Comparação de desempenho
arr = np.random.rand(1000000)
%timeit soma_quadrados_loop(arr)  # ~100 ms
%timeit soma_quadrados_vet(arr)   # ~1 ms
```

2. Uso Eficiente de Memória

```
# Evitar cópias desnecessárias
arr = np.arange(1000000)

# Cria cópia
arr_copia = arr.reshape(1000, 1000)  # Cópia

# Cria view (mais eficiente)
arr_view = arr.reshape(1000, 1000, order='C')  # Pode ser view

# Verificar se é view ou cópia
print(arr_view.base is arr)  # True = view, False = cópia
```

3. Broadcasting Inteligente

```
# Expandir dimensões de forma explícita para clareza
matriz = np.random.rand(100, 50)
vetor = np.random.rand(50)

# Menos claro
resultado = matriz + vetor

# Mais claro
resultado = matriz + vetor[np.newaxis, :]
```

4. Pré-alocação de Arrays

```
# RUIM: Crescer array dinamicamente
resultado = np.array([])
for i in range(1000):
    resultado = np.append(resultado, i)

# BOM: Pré-alocar
resultado = np.zeros(1000)
for i in range(1000):
    resultado[i] = i

# MELHOR: Vetorizado
resultado = np.arange(1000)
```

5. Escolha do Tipo de Dados

```
# Economizar memória escolhendo dtype apropriado
arr_float64 = np.random.rand(1000000)  # 8 MB
arr_float32 = np.random.rand(1000000).astype(np.float32)  # 4 MB
arr_int16 = np.random.randint(0, 100, 1000000, dtype=np.int16)  # 2 MB
```

Tabela de Referência Rápida

Criação de Arrays

Função	Descrição	Exemplo
`np.array()`	Cria array de lista	`np.array([1,2,3])`
`np.zeros()`	Array de zeros	`np.zeros((3,4))`
`np.ones()`	Array de uns	`np.ones(5)`
`np.arange()`	Range de valores	`np.arange(0,10,2)`
`np.linspace()`	Valores espaçados	`np.linspace(0,1,5)`
`np.eye()`	Matriz identidade	`np.eye(3)`
`np.random.random()`	Aleatórios [0,1)	`np.random.random((2,3))`

Manipulação de Forma

Função	Descrição	Modifica Original?
`reshape()`	Muda forma	Não (view se possível)
`ravel()`	Achata	Não (view se possível)
`flatten()`	Achata	Não (sempre cópia)
`transpose()`	Transpõe	Não (view)
`squeeze()`	Remove dims 1	Não (view)

Indexação

Tipo	Sintaxe	Exemplo
Básica	`arr[i]`	`arr[0]`
Slice	`arr[start:stop:step]`	`arr[1:5:2]`
Booleana	`arr[mask]`	`arr[arr > 5]`
Fancy	`arr[indices]`	`arr[[0,2,4]]`

Agregação (com axis)

Função	axis=None	axis=0	axis=1
`sum()`	Soma total	Soma colunas	Soma linhas
`mean()`	Média total	Média colunas	Média linhas
`std()`	DP total	DP colunas	DP linhas
`min()` / `max()`	Min/Max total	Min/Max colunas	Min/Max linhas

Recursos Adicionais

Documentação Oficial

- NumPy Documentation: https://numpy.org/doc/
- NumPy User Guide: https://numpy.org/doc/stable/user/
- NumPy API Reference: https://numpy.org/doc/stable/reference/

Conceitos Importantes para Estudo Avançado

- Strides e layouts de memória
- NumPy C API
- Numba para compilação JIT
- Dask para computação paralela
- CuPy para computação em GPU

Próximos Passos

- Pandas: Análise de dados estruturados
- SciPy: Computação científica avançada
- Matplotlib: Visualização de dados
- Scikit-learn: Machine learning