Programação para Ciência de Dados

Introdução ao NumPy - Arrays e Operações Vetorizadas

Arthur Casals

14 de Outubro de 2025

Avisos

- Primeira entrega - conteúdo (60% da nota): 17/10 (sexta-feira)
- Interação via Zoom

Agenda

- Fundamentos de Arrays NumPy
- Indexação e Slicing
- Operações Vetorizadas
- Estatísticas e Agregações

Recapitulação Aula 02

O que já sabemos:

- Estruturas aninhadas (listas de listas)
- Collections: defaultdict, Counter, deque, namedtuple
- Comprehensions avançadas e generators
- Algoritmos de busca e complexidade
- Big O notation

Hoje vamos aprender:

- NumPy: biblioteca fundamental para computação científica
- Arrays multidimensionais eficientes
- Operações vetorizadas e performance

Bloco 1

Fundamentos de Arrays NumPy

O que é NumPy?

NumPy (Numerical Python):

- Biblioteca fundamental para computação científica em Python
- Fornece arrays multidimensionais eficientes
- Implementado em C: performance próxima a código compilado
- Base para Pandas, SciPy, Scikit-learn, e muitas outras

Principais características:

- **ndarray:** Estrutura de array N-dimensional
- **Homogêneo:** Todos elementos do mesmo tipo
- **Contíguo em memória:** Acesso rápido
- **Vetorização:** Operações em arrays inteiros
- **Broadcasting:** Operações entre arrays de diferentes shapes

Por que NumPy?

Limitações das listas Python:

- **Heterogêneas:** Podem conter tipos diferentes
- **Não otimizadas:** Operações elemento-a-elemento são lentas
- **Sem vetorização:** Precisa de loops explícitos
- **Alto overhead:** Cada elemento é um objeto Python completo

Por que NumPy Revoluciona Tudo?

Python

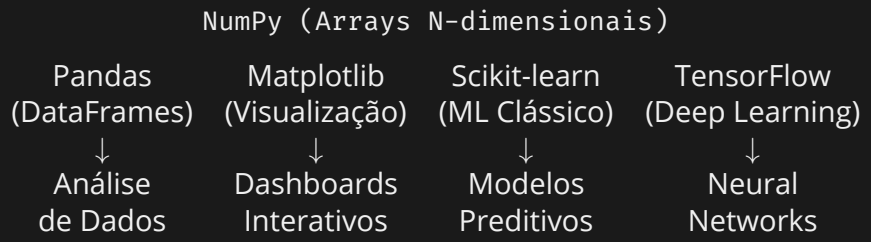
```python
# Python puro: soma de 1 milhao de numeros
import time
numeros = list(range(1000000))
start = time.time()
soma = sum(numeros)
tempo_python = time.time() - start

# NumPy: mesma operacao
import numpy as np
arr = np.arange(1000000)
start = time.time()
soma = np.sum(arr)
tempo_numpy = time.time() - start

#Resultado típico: NumPy é 50-100x mais rápido
```

NumPy como Fundação do Ecossistema Científico

Estrutura do ecossistema:



Nota Importante

Dominar NumPy = Dominar todo o ecossistema

Instalação e Importação

Instalação:

Bash

```bash
# Google Colab: ja vem instalado
# Local (via pip):
pip install numpy

# Verificar versao:
pip show numpy

```

Instalação e Importação

Importação (convenção universal):

</> Python

```python
import numpy as np

# Verificar versao
print(np.__version__)  # 1.24.3 (ou similar)

```

Nota Importante

Sempre use `import numpy as np` - é a convenção padrão

Arrays NumPy: Conceitos Fundamentais

O que é um Array NumPy:

- **Homogêneo:** Todos elementos do mesmo tipo
- **Contíguo:** Dados organizados sequencialmente na memória
- **N-dimensional:** Pode ter qualquer número de dimensões
- **Eficiente:** Implementado em C, operações vetorizadas

Diferenças das Listas Python:

</> Python

```python
# Lista Python: heterogenea, flexivel, lenta
lista = [1, 2.5, 'texto', [1,2,3]]

# Array NumPy: homogeneo, rapido, restritivo
arr = np.array([1, 2, 3, 4, 5])  # Todos int
arr_float = np.array([1.0, 2, 3, 4, 5])  # Todos float

```

Arrays NumPy *vs* Listas Python

Lista Python:

- Array de ponteiros
- Cada elemento: objeto Python completo
- Memória fragmentada
- Overhead significativo
- Tipos mistos permitidos

```
[ptr → int, ptr → float, ptr → str]
```

Array NumPy:

- Bloco contíguo de memória
- Dados brutos (sem overhead)
- Acesso direto e rápido
- Cache-friendly
- Tipo único (homogêneo)

```
[1.0, 2.0, 3.0, 4.0, ...]
```

Nota Importante

NumPy sacrifica flexibilidade (tipos mistos) por **performance massiva**

Criando Arrays NumPy

Principais métodos de criação:

Função	Descrição
`np.array()`	Cria array a partir de lista/tupla
`np.zeros()`	Array preenchido com zeros
`np.ones()`	Array preenchido com uns
`np.full()`	Array preenchido com valor específico
`np.arange()`	Sequência com passo (como range)
`np.linspace()`	N valores espaçados uniformemente
`np.eye()`	Matriz identidade
`np.random.random()`	Valores aleatórios [0, 1)

Criando Arrays: np.array()

A partir de listas:

Python

```python
# Array 1D
arr1d = np.array([1, 2, 3, 4, 5])
print(arr1d)  # [1 2 3 4 5]
print(type(arr1d))  # <class 'numpy.ndarray'>

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

```

Criando Arrays: np.array()

Especificando tipo de dados:

Python

```python
# Float por padrao se houver decimais
arr_float = np.array([1.0, 2.0, 3.0])
print(arr_float.dtype)  # float64

# Forcar tipo inteiro
arr_int = np.array([1.5, 2.7, 3.9], dtype=int)
print(arr_int)  # [1 2 3] - trunca!

# Tipos especificos
arr_int32 = np.array([1, 2, 3], dtype=np.int32)
arr_float32 = np.array([1, 2, 3], dtype=np.float32)

```

Criando Arrays: Zeros e Ones

Python

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

# Matriz de zeros
zeros_2d = np.zeros((3, 4))
print(zeros_2d.shape)  # (3, 4)

# Array de uns
ones = np.ones((2, 3))
print(ones)
# [[1. 1. 1.]
#  [1. 1. 1.]]

```

Criando Arrays: np.full()

Python

```python
# Preencher com valor especifico
arr = np.full((3, 3), 7)
print(arr)
# [[7 7 7]
#  [7 7 7]
#  [7 7 7]]

# Util para inicializacao
matriz_inicial = np.full((100, 100), -1)

```

Criando Arrays: np.arange()

Similar ao range(), mas retorna array:

</> Python

```python
# Basico: arange(stop)
arr1 = np.arange(10)
print(arr1)  # [0 1 2 3 4 5 6 7 8 9]

# Com inicio: arange(start, stop)
arr2 = np.arange(5, 10)
print(arr2)  # [5 6 7 8 9]

# Com passo: arange(start, stop, step)
arr3 = np.arange(0, 20, 2)
print(arr3)  # [ 0  2  4  6  8 10 12 14 16 18]

```

Criando Arrays: np.arange()

Funciona com floats:

Python

```python
# Passo decimal
arr_float = np.arange(0, 1, 0.1)
print(arr_float)
# [0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9]

# Cuidado: precisao de ponto flutuante!
arr = np.arange(0, 1, 0.3)
print(arr)  # [0.  0.3 0.6 0.9]
print(len(arr))  # 4 (esperado!)

```

Criando Arrays: np.arange()

⚠ Atenção

Para passos fracionários, prefira `np.linspace()`

Criando Arrays: np.linspace()

N valores igualmente espaçados:

Python

```python
# linspace(start, stop, num)
arr = np.linspace(0, 10, 5)
print(arr)  # [ 0.   2.5  5.   7.5 10. ]

# Util para graficos
x = np.linspace(0, 2*np.pi, 100)  # 100 pontos
y = np.sin(x)

# Excluir endpoint
arr = np.linspace(0, 10, 5, endpoint=False)
print(arr)  # [0. 2. 4. 6. 8.]

```

np.arange() vs np.linspace()

Python

```python
# arange: voce especifica o PASSO
arr1 = np.arange(0, 10, 2)
print(arr1)  # [0 2 4 6 8]
print(len(arr1))  # 5 - nao sabemos de antemao

# linspace: voce especifica o NUMERO de elementos
arr2 = np.linspace(0, 10, 5)
print(arr2)  # [ 0.   2.5  5.   7.5 10. ]
print(len(arr2))  # 5 - exatamente o que pedimos

```

np.arange() vs np.linspace()

np.arange() - Preferido quando:

Vantagens:

- **Performance:** Mais rápido (loop simples)
- **Inteiros:** Perfeito para índices
- **Memória:** Menor overhead
- **Previsível:** Comportamento igual ao `range()`

Casos de uso:

- Iteração com índices
- Sequências inteiras conhecidas
- Substituir `range()` com arrays

np.linspace() - Preferido quando:

Vantagens:

- **Precisão:** Evita erros de ponto flutuante
- **Garantias:** Sempre inclui endpoints
- **Controle:** Número exato de pontos
- **Matemática:** Divisão uniforme garantida

Casos de uso:

- Gráficos e visualizações
- Amostragem uniforme
- Cálculos científicos

np.arange() vs np.linspace()

np.arange()

Python

```python
# Otimo: indices, passos conhecidos
indices = np.arange(1000)
pares = np.arange(0, 100, 2)

# PROBLEMA: imprecisao com floats
arr = np.arange(0, 1, 0.1)
print(len(arr))  # 10 ou 11?
# Depende de arredondamento!

```

np.linspace()

Python

```python
# Otimo: divisao uniforme, graficos
x = np.linspace(0, 10, 100)
y = np.sin(x)  # 100 pontos exatos

# SEMPRE preciso
arr = np.linspace(0, 1, 11)
print(len(arr))  # 11 - garantido!
print(arr[-1])   # 1.0 - exato!

```

np.arange() vs np.linspace()

⚠ Atenção

Armadilha Clássica: `np.arange(0.1, 1.0, 0.1)` pode ter 9 ou 10 elementos devido a imprecisão de ponto flutuante! Use `np.linspace(0.1, 1.0, 10)` para garantir 10 elementos exatos.

Nota Importante

Regra Prática:
Inteiros e passos conhecidos: `arange` (mais rápido)
Floats, gráficos, ou número exato de pontos: `linspace` (mais seguro)

Criando Arrays: Matrizes Especiais

Python

```python
# Matriz identidade
identidade = np.eye(4)
print(identidade)
# [[1. 0. 0. 0.]
#  [0. 1. 0. 0.]
#  [0. 0. 1. 0.]
#  [0. 0. 0. 1.]]

# Matriz diagonal
diagonal = np.diag([1, 2, 3, 4])
print(diagonal)
# [[1 0 0 0]
#  [0 2 0 0]
#  [0 0 3 0]
#  [0 0 0 4]]
```

Criando Arrays: Aleatórios

Python

```python
# Valores aleatorios entre 0 e 1
arr_rand = np.random.random((3, 3))
print(arr_rand)

# Inteiros aleatorios
arr_int = np.random.randint(0, 10, size=(2, 5))
print(arr_int)  # Valores entre 0 e 9

# Distribuicao normal
arr_norm = np.random.randn(1000)  # media=0, std=1
print(arr_norm.mean())  # ~0
print(arr_norm.std())   # ~1

```

Reprodutibilidade: Seeds

Python

```python
# Sem seed: resultados diferentes a cada execucao
print(np.random.random(3))
print(np.random.random(3))  # Valores diferentes!

# Com seed: resultados reprodutíveis np.random.seed(42)
print(np.random.random(3))  # [0.37 0.95 0.73]

np.random.seed(42)
print(np.random.random(3))  # [0.37 0.95 0.73] - igual!

```

Atributos de Arrays

Python

```python
arr = np.array([[1, 2, 3, 4],
                [5, 6, 7, 8]])
# Dtype: tipo dos elementos
print(arr.dtype)  # int64

# Shape: dimensoes do array
print(arr.shape)  # (2, 4) - 2 linhas, 4 colunas

# Size: numero total de elementos
print(arr.size)  # 8

# Ndim: numero de dimensoes
print(arr.ndim)  # 2

```

Tipos de Dados (dtypes)

Principais tipos:

Tipo	Descrição	Exemplos
`int8, int16, int32, int64`	Inteiros com sinal	-128 a 127 (int8)
`uint8, uint16, ...`	Inteiros sem sinal	0 a 255 (uint8)
`float16, float32, float64`	Ponto flutuante	3.14159
`bool`	Booleano	True, False
`complex64, complex128`	Números complexos	1+2j

Por que importa?

- **Memória:** float32 usa metade da memória de float64
- **Performance:** Operações com tipos menores são mais rápidas
- **Precisão:** float64 tem mais precisão que float32

Trabalhando com dtypes

Python

```python
# Verificar dtype
arr = np.array([1, 2, 3])
print(arr.dtype)  # int64 (padrao em 64-bit)

# Especificar dtype na criacao
arr_float32 = np.array([1, 2, 3], dtype=np.float32)
print(arr_float32.dtype)  # float32

# Converter dtype
arr_int = np.array([1.5, 2.7, 3.9])
arr_converted = arr_int.astype(int)
print(arr_converted)  # [1 2 3]

```

Compreendendo Shape

Shape: tupla descrevendo dimensões

- **1D:** `(n,)` - Vetor de n elementos
- **2D:** `(n, m)` - Matriz n×m (n linhas, m colunas)
- **3D:** `(n, m, p)` - n matrizes m×p
- **nD:** `(d1, d2, ..., dn)` - n dimensões

Exemplos visuais:

- `shape=(5,)`: `[1, 2, 3, 4, 5]`
- `shape=(2, 3)`: matriz 2×3
- `shape=(2, 3, 4)`: 2 matrizes 3×4

Nota Importante

Lembre: Shape se lê da ”maior”para ”menor”dimensão

Arrays de Diferentes Dimensões

Python

```python
# 1D
arr_1d = np.array([1, 2, 3, 4])
print(f"1D shape: {arr_1d.shape}")  # (4,)

# 2D
arr_2d = np.array([[1, 2], [3, 4], [5, 6]])
print(f"2D shape: {arr_2d.shape}")  # (3, 2)

# 3D
arr_3d = np.array([[[1, 2, 3, 4], [3, 4, 5, 6], [6, 7, 8, 9]],
                   [[5, 6, 7, 8], [7, 8, 9, 0], [0, 1, 2, 3]]])
print(f"3D shape: {arr_3d.shape}")  # (2, 3, 4)

```

Mais Atributos Úteis

Python

```python
arr = np.array([[1, 2, 3], [4, 5, 6]], dtype=np.float32)

# Itemsize: bytes por elemento
print(arr.itemsize)  # 4 (float32 = 4 bytes)

# Nbytes: total de bytes ocupados
print(arr.nbytes)  # 24 (6 elementos * 4 bytes)

# Strides: bytes para andar em cada dimensao
print(arr.strides)  # (12, 4)
# 12 bytes para proxima linha
# 4 bytes para proximo elemento na linha

```

Bloco 2

Indexação e Slicing

Indexação em NumPy

Tipos de indexação:

- **Básica:** Acesso por índice `arr[i]`
- **Slicing:** Fatias `arr[start:stop:step]`
- **Fancy indexing:** Arrays de índices `arr[[0, 2, 4]]`
- **Boolean indexing:** Máscaras booleanas `arr[arr > 5]`
- **Multidimensional:** `arr[i, j]` ou `arr[i][j]`

Diferença importante:

- Python: índices negativos contam do fim
- NumPy: mesma convenção + suporte multidimensional

Indexação Básica 1D

Python

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

# Acesso por indice (comeca em 0)
print(arr[0])   # 10
print(arr[2])   # 30
print(arr[-1])  # 50 (ultimo)
print(arr[-2])  # 40 (penultimo)

# Modificar elementos
arr[0] = 100
print(arr)  # [100  20  30  40  50]

```

Indexação Multidimensional

</> Python

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

# Acesso: arr[linha, coluna]
print(matriz[0, 0])  # 1
print(matriz[1, 2])  # 6
print(matriz[-1, -1])  # 9

# Tambem funciona (menos eficiente)
print(matriz[0][0])  # 1

```

Nota Importante

Slicing Básico 1D

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

Python

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

# Fatiamento basico
print(arr[2:5])      # [2 3 4]
print(arr[:4])       # [0 1 2 3] - do inicio ate 4
print(arr[5:])       # [5 6 7 8 9] - de 5 ate o fim
print(arr[::2])      # [0 2 4 6 8] - passo 2
print(arr[::-1])     # [9 8 7 6 5 4 3 2 1 0] - reverso

```

Slicing Multidimensional

</> Python

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

# Selecionar linha
print(matriz[0, :])  # [1 2 3 4]

# Selecionar coluna
print(matriz[:, 1])  # [ 2  6 10]

# Sub-matriz
print(matriz[0:2, 1:3])
# [[ 2  3]
#  [ 6  7]]

```

Slicing: Exemplos Avançados

</> Python

```python
matriz = np.arange(20).reshape(4, 5)
print(matriz)
# [[ 0  1  2  3  4]
#  [ 5  6  7  8  9]
#  [10 11 12 13 14]
#  [15 16 17 18 19]]

# Linhas pares, colunas impares
print(matriz[::2, 1::2])
# [[ 1  3]
#  [11 13]]

```

Views vs Copies

Diferença fundamental:

- **View (visualização):** Referência ao array original
  - Não copia dados
  - Modificações afetam o original
  - Eficiente em memória
- **Copy (cópia):** Novo array independente
  - Duplica dados
  - Modificações não afetam o original
  - Usa mais memória

Quando ocorre cada um?

- Slicing básico: **view**
- Fancy/Boolean/Combined indexing: **copy**

Views vs Copies: Exemplo

Python

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

# Slicing cria VIEW
view = arr[1:4]
view[0] = 999
print(arr)  # [  1 999   3   4   5] - MODIFICOU!

# Criar copia explicita
arr = np.array([1, 2, 3, 4, 5])
copia = arr[1:4].copy()
copia[0] = 999
print(arr)  # [1 2 3 4 5] - nao modificou

```

Verificando Views vs Copies

Python

```python
arr = np.array([1, 2, 3, 4, 5])
view = arr[:]
copia = arr[:].copy()

# Verificar se compartilham memoria
print(np.shares_memory(arr, view))   # True
print(np.shares_memory(arr, copia))  # False

# Base de uma view aponta pro original
print(view.base is arr)    # True
print(copia.base is None)  # True

```

Fancy Indexing

O que é?

- Usar arrays de índices para selecionar elementos
- Permite seleção não-sequencial
- Sempre retorna **cópia** (não view)

Quando usar?

- Selecionar elementos específicos
- Reordenar elementos
- Amostragem de dados
- Embaralhar arrays

Fancy Indexing: Exemplos

Python

```python
arr = np.array([10, 20, 30, 40, 50, 60])

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

# Pode repetir indices
indices = [0, 0, 1, 1, 2]
print(arr[indices])  # [10 10 20 20 30]

# Arrays 2D
indices = np.array([[0, 1], [2, 3]])
print(arr[indices])
# [[10 20]
#  [30 40]]

```

Fancy Indexing Multidimensional

</> Python

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

# Selecionar elementos especificos
linhas = [0, 2, 2] # Última linha selecionada duas vezes
colunas = [1, 0, 3] #[1] na linha 0, [0] na linha 2, [3] na linha 2
print(matriz[linhas, colunas])  # [ 2  9 12]
# elemento [0,1]=2, [2,0]=9, [2,3]=12

```

Boolean Indexing (Máscara booleana)

Conceito:

- Usar array de booleanos como índice
- True: seleciona elemento
- False: ignora elemento
- Fundamental para filtragem de dados

Como funciona:

1. Criar condição que retorna array booleano
2. Usar array booleano como índice
3. NumPy seleciona apenas elementos True

Nota Importante

Boolean indexing é **extremamente poderoso** para análise de dados

Boolean Indexing: Básico

Python

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

# Criar mascara booleana
mascara = arr > 5
print(mascara)
# [False False False False False True True True True True]

# Usar mascara para filtrar
resultado = arr[mascara]
print(resultado)  # [ 6  7  8  9 10]

# Forma compacta (mais comum)
print(arr[arr > 5])  # [ 6  7  8  9 10]

```

Boolean Indexing: Condições Múltiplas

Python

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

# AND logico: &
print(arr[(arr > 3) & (arr < 8)])  # [4 5 6 7]

# OR logico: |
print(arr[(arr < 3) | (arr > 8)])  # [ 1  2  9 10]

# NOT logico: ~
print(arr[~(arr > 5)])  # [1 2 3 4 5]

# Multiplas condicoes
print(arr[(arr % 2 == 0) & (arr > 4)])  # [ 6  8 10]

```

Boolean Indexing: Aplicações

Python

```python
# Substituir valores que atendem condicao
arr = np.array([1, 2, 3, 4, 5, 6])
arr[arr > 3] = 0
print(arr)  # [1 2 3 0 0 0]

# Contar elementos que atendem condicao
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
count = np.sum(arr > 5)
print(count)  # 5

# Verificar se algum/todos atendem
print(np.any(arr > 8))   # True
print(np.all(arr > 0))   # True
```

Combined Indexing

O que é Indexação Combinada?

- Usar **múltiplos tipos** de indexação simultaneamente
- Cada dimensão pode usar técnica diferente
- Permite seleções complexas e precisas
- Essencial para análise de dados multidimensionais

Tipos que podem ser combinados:

1. **Básica** (`arr[2]`) + **Slicing** (`arr[:, 1:3]`)
2. **Boolean** (`arr[mask]`) + **Slicing** (`arr[mask, :3]`)
3. **Fancy** (`arr[[0,2]]`) + **Slicing** (`arr[[0,2], 1:4]`)
4. **Boolean** (`arr > 5`) + **Fancy** (`arr[mask][[0,2]]`)

Regras importantes:

- Ordem importa: `arr[mask, :]` $\neq$ `arr[:, mask]`
- Combinações complexas tendem a retornar cópias

Combined Indexing

Nota Importante

Poder máximo: Combinar indexações permite extrair dados específicos de arrays multidimensionais complexos em uma única operação!

Combined Indexing: Exemplos Práticos

Setup: Dados de vendas (produtos × meses)

</> Python

```python
# Matriz: 5 produtos x 12 meses
np.random.seed(42) #Experimento reprodutível
vendas = np.random.randint(100, 1000, (5, 12)) #Matriz 5x12: [100,1000)
produtos = ['A', 'B', 'C', 'D', 'E']
meses = ['Jan', 'Fev', 'Mar', 'Abr', 'Mai', 'Jun',
         'Jul', 'Ago', 'Set', 'Out', 'Nov', 'Dez']

```

Combined Indexing: Exemplos Práticos

Exemplo 1: Boolean + Slicing

Python

```python
# Produtos com vendas > 500 em Jan, apenas Q1
mask = vendas[:, 0] > 500  # Mascara em Janeiro
q1_vendas = vendas[mask, :3]  # Q1 desses produtos
print(f"Shape: {q1_vendas.shape}")
# Produtos que passaram + 3 meses

```

Combined Indexing: Exemplos Práticos

Exemplo 2: Fancy + Slicing

Python

```python
# Produtos A e C (indices 0, 2), meses de verao
produtos_sel = [0, 2]
meses_verao = slice(5, 8)  # Jun, Jul, Ago
verao_AC = vendas[produtos_sel, meses_verao]
print(f"Shape: {verao_AC.shape}")  # (2, 3)

```

Combined Indexing: Exemplos Práticos

Exemplo 3: Boolean em múltiplas dimensões

Python

```python
# Meses onde pelo menos um produto vendeu > 800
mask_meses = np.any(vendas > 800, axis=0)
# Produtos que venderam > 600 na media anual
mask_produtos = np.mean(vendas, axis=1) > 600
# Combinar ambas as mascaras
resultado = vendas[mask_produtos][:, mask_meses]

```

Combined Indexing: Exemplos Práticos

Exemplo 4: Combinação tripla (avançado)

Python

```python
# Passo 1: Produtos com media anual > 500
mask_prod = vendas.mean(axis=1) > 500

# Passo 2: Desses, selecionar indices especificos
indices_interesse = [0, 2]  # Primeiro e terceiro

# Passo 3: Apenas segundo semestre
vendas_filtradas = vendas[mask_prod][indices_interesse, 6:]
print(f"Shape final: {vendas_filtradas.shape}")

```

Combined Indexing: Exemplos Práticos

Exemplo 5: Modificação combinada

Python

```python
# Zerar vendas < 300 apenas em Q4 dos produtos B e D
vendas[[1, 3], 9:][(vendas[[1, 3], 9:] < 300)] = 0

```

Atenção

Combinações complexas podem ser menos legíveis. Use variáveis intermediárias para clareza!

np.where(): Seleção Condicional

Sintaxe: `np.where(condition, x, y)`

</> Python

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

# Se condicao True, usa x; senao usa y
resultado = np.where(arr > 3, 'alto', 'baixo')
print(resultado)
# ['baixo' 'baixo' 'baixo' 'alto' 'alto' 'alto']

# Substituir valores
arr_novo = np.where(arr > 3, arr * 10, arr)
print(arr_novo)  # [ 1  2  3 40 50 60]

```

np.where(): Encontrar Índices

</> Python

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

# Sem segundo argumento: retorna indices
indices = np.where(arr > 5)
print(indices)  # (array([5, 6, 7, 8, 9]),)
print(indices[0])  # [5 6 7 8 9]

# Usar indices para acessar valores
print(arr[indices])  # [ 6  7  8  9 10]

```

np.where(): Encontrar Índices

Python

```python
# Multidimensional
matriz = np.array([[1, 2], [3, 4], [5, 6]]) # [[1 2] [3 4] [5 6]]
mask = matriz > 3# Verifica elementos > 3
print(mask) # [[False False] [False True] [True True]]
# Índices em que os elementos sao maiores que 3
linhas, colunas = np.where(matriz > 3)
print(linhas, colunas)  # [1 2 2] [1 0 1]

```

Exemplo Prático: Filtragem de Dados

Python

```python
# Simular dados de sensores
temperaturas = np.random.uniform(15, 35, 20)
print(f"Media: {temperaturas.mean():.1f}°C")

# Encontrar temperaturas anomalas (> 30°C)
anomalias = temperaturas[temperaturas > 30]
print(f"Anomalias: {len(anomalias)}")
print(f"Valores: {anomalias}")

# Substituir anomalias pela media
temp_corrigidas = np.where(temperaturas > 30,
                           temperaturas.mean(),
                           temperaturas)

```

Bloco 3

Operações Vetorizadas

O que é Vetorização?

Definição:

- Aplicar operações em arrays inteiros de uma vez
- Sem loops explícitos em Python
- Operações implementadas em C otimizado

Vantagens:

- **Velocidade:** 10x-100x mais rápido que loops
- **Código limpo:** Mais legível e conciso
- **Paralelização:** NumPy pode usar múltiplos cores
- **Otimizações:** SIMD, cache-friendly

Nota Importante

Regra de ouro: Evite loops explícitos - use vetorização!

Loop vs Vetorização: Exemplo

</> Python

```python
# COM LOOP (lento)
lista = list(range(1000000))
resultado = []
for x in lista:
    resultado.append(x * 2)
# Tempo: ~100ms

# VETORIZADO (rapido)
arr = np.arange(1000000)
resultado = arr * 2
# Tempo: ~1ms - 100x mais rapido!

```

Operações Aritméticas Básicas

Elemento-a-elemento:

Python

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

# Operacoes com escalares
print(arr + 10)   # [11 12 13 14 15]
print(arr * 2)    # [ 2  4  6  8 10]
print(arr ** 2)   # [ 1  4  9 16 25]
print(arr / 2)    # [0.5 1.  1.5 2.  2.5]

# Operacoes entre arrays (mesmo tamanho)
arr2 = np.array([10, 20, 30, 40, 50])
print(arr + arr2)  # [11 22 33 44 55]

```

Operações Aritméticas: Mais Exemplos

</> Python

```python
arr1 = np.array([10, 20, 30])
arr2 = np.array([1, 2, 3])

# Divisao inteira
print(arr1 // arr2)  # [10 10 10]

# Modulo (resto)
print(arr1 % arr2)   # [0 0 0]

# Potencia
print(arr2 ** 3)     # [ 1  8 27]

# Negacao
print(-arr1)         # [-10 -20 -30]

```

Broadcasting

O que é broadcasting?

- Mecanismo para operar arrays de shapes diferentes
- NumPy ”estica”automaticamente o array menor
- Sem cópias de dados (eficiente!)

Regras de broadcasting:

1. Dimensões: da direita para a esquerda
2. Se arrays têm número diferente de dimensões, adiciona dimensões de tamanho 1 à esquerda do shape menor
3. Arrays são compatíveis em uma dimensão se:
   - Têm o mesmo tamanho, OU
   - Um deles tem tamanho 1
4. Depois do broadcasting, cada array se comporta como se tivesse o shape máximo

Broadcasting: Exemplo Simples

</> Python

```python
# Escalar com array - broadcasting automatico
arr = np.array([1, 2, 3])
print(arr + 10)  # [11 12 13]
# 10 é "esticado" para [10, 10, 10]

# Array 1D com array 2D
matriz = np.array([[1, 2, 3],
                   [4, 5, 6]])
vetor = np.array([10, 20, 30])
print(matriz + vetor)
# [[11 22 33]
#  [14 25 36]]

```

Broadcasting: Visualização

Python

```python
# Shape (3, 4) + shape (4,)
matriz = np.ones((3, 4))
vetor = np.array([1, 2, 3, 4])

resultado = matriz + vetor
print(resultado)
# [[2. 3. 4. 5.]
#  [2. 3. 4. 5.]
#  [2. 3. 4. 5.]]

# vetor (4,) foi "esticado" para (3, 4)
# conceitualmente: [[1,2,3,4], [1,2,3,4], [1,2,3,4]]

```

Broadcasting: Exemplos Avançados

</> Python

```python
# Shape (3, 1) + shape (4,) -> shape (3, 4)
col = np.array([[1], [2], [3]])  # (3, 1)
row = np.array([10, 20, 30, 40])  # (4,)

resultado = col + row
print(resultado)
# [[11 21 31 41]
#  [12 22 32 42]
#  [13 23 33 43]]

```

Broadcasting: Quando Falha

Python

```python
# Shapes incompativeis
arr1 = np.ones((3, 4))    # (3, 4)
arr2 = np.ones((3,))      # (3,)

# Isto vai FALHAR!
# resultado = arr1 + arr2
# ValueError: operands could not be broadcast together

# Solucao: reshape para compatibilidade
arr2_reshaped = arr2.reshape(3, 1)  # (3, 1)
resultado = arr1 + arr2_reshaped  # Funciona!

```

Broadcasting: Quando Falha

⚠ Atenção

Sempre verifique shapes antes de operar arrays!

Funções Universais (ufuncs)

O que são ufuncs?

- Operações matemáticas vetorizadas
- Operam elemento-a-elemento
- Implementadas em C otimizado
- Suportam broadcasting
- Muito mais rápidas que equivalentes Python

Categorias principais:

- Matemáticas: sin, cos, exp, log, sqrt
- Comparação: greater, less, equal
- Lógicas: logical_and, logical_or
- Arredondamento: round, floor, ceil

Ufuncs Matemáticas

Python

```python
arr = np.array([1, 4, 9, 16, 25])

# Raiz quadrada
print(np.sqrt(arr))  # [1. 2. 3. 4. 5.]

# Exponencial
print(np.exp([0, 1, 2]))  # [1.    2.72  7.39]

# Logaritmo
print(np.log([1, np.e, np.e**2]))  # [0. 1. 2.]

# Potencia
print(np.power(arr, 2))  # [  1  16  81 256 625]

```

Ufuncs Trigonométricas

Python

```python
# Angulos em radianos
angulos = np.array([0, np.pi/4, np.pi/2, np.pi])

print(np.sin(angulos))
# [0.000 0.707 1.000 0.000]

print(np.cos(angulos))
# [ 1.000  0.707  0.000 -1.000]

# Converter graus para radianos
graus = np.array([0, 45, 90, 180])
radianos = np.deg2rad(graus)
print(np.sin(radianos))

```

Ufuncs de Arredondamento

Python

```python
arr = np.array([1.2, 2.5, 3.7, 4.1, 5.9])

# Arredondar para inteiro mais proximo
print(np.round(arr))  # [1. 2. 4. 4. 6.]

# Arredondar para baixo (floor)
print(np.floor(arr))  # [1. 2. 3. 4. 5.]

# Arredondar para cima (ceil)
print(np.ceil(arr))   # [2. 3. 4. 5. 6.]

# Truncar (remover parte decimal)
print(np.trunc(arr))  # [1. 2. 3. 4. 5.]

```

Operações de Comparação

Python

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

# Comparacoes elemento-a-elemento
print(arr1 > arr2)
# [False False False  True  True]

print(arr1 == arr2)
# [False False  True False False]

# Funcoes equivalentes
print(np.greater(arr1, arr2))  # mesmo que >
print(np.equal(arr1, arr2))    # mesmo que ==

```

Operações Lógicas

Python

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

# Multiplas condicoes
mask1 = arr > 2
mask2 = arr < 5

# AND logico
print(np.logical_and(mask1, mask2))
# [False False  True  True False False]

# OR logico
print(np.logical_or(mask1, mask2))
# [False  True  True  True  True  True]

```

Exemplo Prático: Normalização Min-Max

Python

```python
# Dados originais
dados = np.array([10, 20, 30, 40, 50])

# Normalizacao Min-Max: (x - min) / (max - min)
min_val = dados.min()
max_val = dados.max()

normalizados = (dados - min_val) / (max_val - min_val)
print(normalizados)
# [0.   0.25 0.5  0.75 1.  ]

# Tudo vetorizado - sem loops!

```

Exemplo Prático: Cálculo de Distância

Python

```python
# Pontos 2D
ponto1 = np.array([0, 0])
pontos = np.array([[1, 1],
                   [2, 2],
                   [3, 3]])

# Distancia euclidiana vetorizada
diferencas = pontos - ponto1  # broadcasting
distancias = np.sqrt(np.sum(diferencas**2, axis=1))
print(distancias)
# [1.41421356 2.82842712 4.24264069]

```

Desempenho: Loop vs Vetorização

</> Python

```python
import time
# Criar dados
n = 1000000
lista = list(range(n))
arr = np.arange(n)

# Timing: loop Python
start = time.time()
resultado = [x**2 for x in lista]
tempo_loop = time.time() - start

# Timing: vetorizacao NumPy
start = time.time()
resultado = arr**2
tempo_numpy = time.time() - start
```

Desempenho: Resultados

Python

```python
print(f"Loop Python: {tempo_loop:.4f}s")
print(f"NumPy:       {tempo_numpy:.4f}s")
print(f"Speedup:     {tempo_loop/tempo_numpy:.1f}x")

# Resultado tipico:
# Loop Python: 0.1234s
# NumPy:       0.0012s
# Speedup:     102.8x

```

Nota Importante

NumPy é **100x+ mais rápido** que loops Python puros!

Bloco 4

Estatísticas e Agregações

Funções de Agregação

O que são agregações?

- Funções que reduzem array a valor escalar (ou menor dimensão)
- Aplicam operação em todo o array
- Fundamentais para análise estatística

Principais funções:

- **Tendência central:** mean, median, mode
- **Dispersão:** std, var, min, max
- **Soma/produto:** sum, prod, cumsum, cumprod
- **Contagem:** count_nonzero, unique

Funções Estatísticas Básicas

Python

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

# Medidas de tendencia central
print(f"Media: {arr.mean()}")       # 5.5
print(f"Mediana: {np.median(arr)}")  # 5.5

# Medidas de dispersao
print(f"Desvio padrao: {arr.std()}")  # 2.87
print(f"Variancia: {arr.var()}")      # 8.25

# Valores extremos
print(f"Minimo: {arr.min()}")  # 1
print(f"Maximo: {arr.max()}")  # 10

```

Somas e Produtos

Python

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

# Soma total
print(arr.sum())  # 15

# Produto total
print(arr.prod())  # 120

# Soma cumulativa
print(np.cumsum(arr))  # [ 1  3  6 10 15]

# Produto cumulativo
print(np.cumprod(arr))  # [  1   2   6  24 120]

```

Percentis e Quartis

Python

```python
dados = np.array([10, 20, 30, 40, 50, 60, 70, 80, 90, 100])

# Percentis especificos
print(np.percentile(dados, 25))   # 32.5 (Q1)
print(np.percentile(dados, 50))   # 55.0 (mediana)
print(np.percentile(dados, 75))   # 77.5 (Q3)

# Multiplos percentis de uma vez
percentis = np.percentile(dados, [25, 50, 75])
print(percentis)  # [32.5 55.  77.5]

```

Valores Únicos e Contagens

Python

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

# Valores unicos
unicos = np.unique(arr)
print(unicos)  # [1 2 3 4]

# Valores unicos com contagens
valores, contagens = np.unique(arr, return_counts=True)
print(valores)    # [1 2 3 4]
print(contagens)  # [1 2 3 4]

# Contar nao-zeros
print(np.count_nonzero(arr))  # 10
```

Operações de Agregação: Exemplos Condensados

Python

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

# Estatisticas globais
soma_total = np.sum(dados)        # 45
media = np.mean(dados)            # 5.0
mediana = np.median(dados)        # 5.0
desvio = np.std(dados)            # 2.58
variancia = np.var(dados)         # 6.67
minimo = np.min(dados)            # 1
maximo = np.max(dados)            # 9

# Posicoes de min/max
pos_min = np.argmin(dados)        # 0
pos_max = np.argmax(dados)        # 8
```

O Parâmetro axis

O que é axis?

- Define a direção da agregação
- `axis=None`: opera em todo o array (padrão)
- `axis=0`: opera ao longo das linhas (↓)
- `axis=1`: opera ao longo das colunas (→)

Como pensar:

- `axis=0`: ”colapsa”as linhas (resultado tem menos linhas)
- `axis=1`: ”colapsa”as colunas (resultado tem menos colunas)

Nota Importante

Dica visual: axis aponta na direção que será **removida**

axis em Arrays 2D: Exemplo Visual

</> Python

```python
matriz = np.array([[1, 2, 3],
                   [4, 5, 6]])
print(f"Shape: {matriz.shape}")  # (2, 3)

# Sem axis: soma tudo
print(matriz.sum())  # 21

# axis=0: soma ao longo das linhas
print(matriz.sum(axis=0))  # [5 7 9]
# soma cada coluna: [1+4, 2+5, 3+6]

# axis=1: soma ao longo das colunas
print(matriz.sum(axis=1))  # [ 6 15]
# soma cada linha: [1+2+3, 4+5+6]

```

axis: Mais Exemplos

Python

```python
matriz = np.array([[10, 20, 30],
                   [40, 50, 60]])

# Media por coluna
print(matriz.mean(axis=0))  # [25. 35. 45.]

# Media por linha
print(matriz.mean(axis=1))  # [20. 50.]

# Maximo por coluna
print(matriz.max(axis=0))   # [40 50 60]

# Minimo por linha
print(matriz.min(axis=1))   # [10 40]

```

axis em Arrays 3D

Python

```python
# Array 3D: (2, 3, 4)
arr_3d = np.arange(24).reshape(2, 3, 4)
print(f"Shape: {arr_3d.shape}")  # (2, 3, 4)

# axis=0: colapsa primeira dimensao
print(arr_3d.sum(axis=0).shape)  # (3, 4)

# axis=1: colapsa segunda dimensao
print(arr_3d.sum(axis=1).shape)  # (2, 4)

# axis=2: colapsa terceira dimensao
print(arr_3d.sum(axis=2).shape)  # (2, 3)

```

keepdims: Manter Dimensões

</> Python

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

# Sem keepdims: resultado e 1D
resultado = matriz.sum(axis=0)
print(resultado) # [5 7 9]
print(resultado.shape)  # (3,)

# Com keepdims: mantem dimensao
resultado = matriz.sum(axis=0, keepdims=True)
print(resultado.shape)  # (1, 3)
print(resultado)
# [[5 7 9]]

# Útil para broadcasting posterior

```

Índices de Valores Extremos

Python

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

# Indice do minimo
print(np.argmin(arr))  # 1

# Indice do maximo
print(np.argmax(arr))  # 2

# Pode usar com axis
matriz = np.array([[1, 5, 3],
                   [2, 4, 6]])
print(np.argmax(matriz, axis=0))  # [1 0 1]
print(np.argmax(matriz, axis=1))  # [1 2]

```

Ordenação

Python

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

# Ordenar (retorna copia)
print(np.sort(arr))  # [10 20 30 40 50]

# Ordenar in-place
arr.sort()
print(arr)  # [10 20 30 40 50]

# Indices que ordenariam o array (argsort)
arr = np.array([30, 10, 50, 20, 40])
indices = np.argsort(arr)
print(indices)  # [1 3 0 4 2]
print(arr[indices])  # [10 20 30 40 50]

```

Ordenação Multidimensional

</> Python

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

# Ordenar cada linha
print(np.sort(matriz, axis=1))
# [[1 3 5]
#  [2 4 6]]

# Ordenar cada coluna
print(np.sort(matriz, axis=0))
# [[3 1 2]
#  [6 4 5]]

```

Exemplo Prático: Análise de Notas

Python

```python
# Notas de 5 alunos em 3 provas
notas = np.array([[8.5, 7.0, 9.0],
                  [6.0, 8.5, 7.5],
                  [9.5, 9.0, 9.5],
                  [5.0, 6.0, 7.0],
                  [7.5, 8.0, 8.5]])

# Media por aluno (cada linha)
medias_alunos = notas.mean(axis=1)
print(f"Medias dos alunos: {medias_alunos}")

# Media por prova (cada coluna)
medias_provas = notas.mean(axis=0)
print(f"Medias das provas: {medias_provas}")

```

Exemplo Prático: Análise de Notas

Python

```python
# Melhor nota de cada aluno
print(f"Melhores notas: {notas.max(axis=1)}")

# Pior nota de cada prova
print(f"Piores por prova: {notas.min(axis=0)}")

# Desvio padrao das notas
print(f"Desvio geral: {notas.std():.2f}")

# Aluno com maior media
melhor_aluno = np.argmax(medias_alunos)
print(f"Melhor aluno: {melhor_aluno} (media {medias_alunos[melhor_aluno
    ]:.2f})")

```

Desempenho: NumPy *vs* Python Puro

</> Python

```python
# Criar dados
lista = list(range(1000000))
arr = np.arange(1000000)

# Python puro
import time
start = time.time()
media_lista = sum(lista) / len(lista)
tempo_python = time.time() - start

# NumPy
start = time.time()
media_arr = arr.mean()
tempo_numpy = time.time() - start

```

Desempenho: Resultados

Python

```python
print(f"Python: {tempo_python:.4f}s")
print(f"NumPy:  {tempo_numpy:.4f}s")
print(f"Speedup: {tempo_python/tempo_numpy:.1f}x")

# Resultado tipico:
# Python: 0.0234s
# NumPy:  0.0003s
# Speedup: 78.0x

# NumPy é muito mais rapido!
# Para operações estatísticas, NumPy é 50-100x mais rápido.

```

Aplicação: Titanic com NumPy

Python

```python
# Simular dados do Titanic (survived, age, fare)
np.random.seed(42)
n = 891  # numero real de passageiros

survived = np.random.randint(0, 2, n)
ages = np.random.uniform(1, 80, n)
fares = np.random.uniform(0, 500, n)

# Estatisticas
print(f"Taxa sobrevivencia: {survived.mean():.2%}")
print(f"Idade media: {ages.mean():.1f}")
print(f"Tarifa media: ${fares.mean():.2f}")

```

Aplicação: Titanic com NumPy (cont.)

Python

```python
# Analise condicional
sobreviventes = ages[survived == 1]
nao_sobreviventes = ages[survived == 0]

print(f"Idade media sobreviventes: {sobreviventes.mean():.1f}")
print(f"Idade media nao-sobreviventes: {nao_sobreviventes.mean():.1f}")

# Tarifa por grupo
tarifa_sobrev = fares[survived == 1]
print(f"Tarifa media sobreviventes: ${tarifa_sobrev.mean():.2f}")

```

Demonstração Prática

Live Coding

Vamos praticar juntos:

1. (notebook guiado)

Duração: 15 minutos

Revisão da Aula

Bloco 1 - Fundamentos:

- Criação de arrays (array, zeros, ones, arange, linspace)
- Tipos de dados (dtypes)
- Atributos (shape, size, ndim)

Bloco 2 - Indexação:

- Indexação básica e multidimensional
- Slicing avançado
- Fancy indexing e boolean indexing
- Views vs copies
- np.where()

Revisão da Aula

Bloco 3 - Vetorização:

- Operações vetorizadas
- Broadcasting
- Funções universais (ufuncs)
- Performance vs loops

Bloco 4 - Estatísticas:

- Funções de agregação
- Parâmetro axis
- Estatísticas descritivas
- Aplicações práticas

Conceitos-Chave para Lembrar

1. **Arrays são homogêneos** - todos elementos mesmo tipo
2. **Slicing cria views** - use .copy() para independência
3. **Broadcasting** - operações entre shapes diferentes
4. **Vetorização > Loops** - sempre prefira operações vetorizadas
5. **axis=0** $\rightarrow$ **linhas, axis=1** $\downarrow$ **colunas**
6. **Boolean indexing** - poderoso para filtragem
7. **Desempenho** - NumPy é 10-100x mais rápido

Nota Importante

Estes conceitos são **fundamentais** para o resto do curso!

Próxima Aula: NumPy Avançado

Aula 04 - O que vem:

- Arrays multidimensionais (reshape, transpose, flatten)
- Operações matriciais (dot, @, produtos)
- Álgebra linear (determinante, inversa, autovalores)
- Concatenação e stacking
- I/O de arrays

Preparação:

- Revisar conceitos de álgebra linear básica
- Praticar os exemplos desta aula
- Experimentar com dados do Titanic

Dicas de Estudo

Para dominar NumPy:

1. **Pratique, pratique, pratique**
   - Refaça os exemplos da aula
   - Experimente variações
2. **Documentação é sua amiga**
   - `https://numpy.org/doc/`
   - Exemplos excelentes
3. **Pense em vetorização**
   - ”Como fazer sem loop?”
   - Reformule problemas para operações vetorizadas
4. **Use o notebook de atividades guiadas**
   - Exercícios progressivos
   - Soluções comentadas

A Seguir: Aula Prática

✋ Exercício Prático

Tempo: 60 minutos
Entrega: via Moodle (notebook)
Tarefa: (atualizada em aula)

Obrigado

Próxima aula: NumPy Avançado
Quinta-feira, 16/10