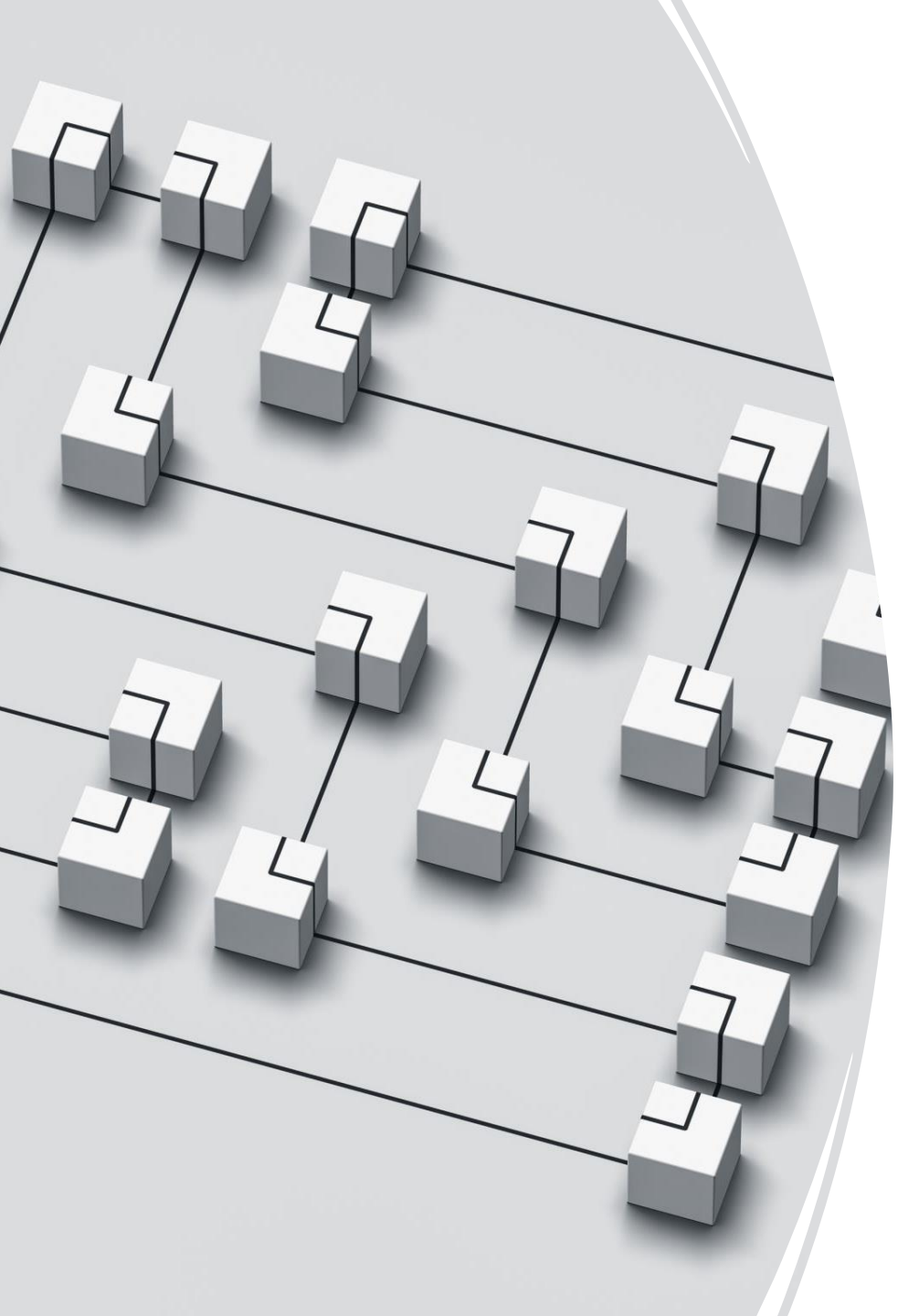
Técnicas de Programação

Usando estruturas de dados de modo eficaz

Métodos para melhorar o desempenho

- Escolha do algoritmo
  - Evite iterações desnecessárias
- Escolha da estrutura de dados
  - É mais rápido procurar um valor em um dicionário do que em uma lista
- Use funções built-in
  - Muitas são implementadas em C
- Compilando Python
  - Cython (superconjunto do Pyton), Numba (subconjunto do python) e PyPy (reimplementação do Python)
  - Código Assíncrono
  - Realiza uma tarefa enquanto aguarda outra
- Computação paralela e computação distribuída
  - MapReduce
    - Modelo de programação para BigData



Estruturas de dados

- Usar uma variedade de estruturas de dados
- Estruturas de dados mais comuns:
  - Listas
  - Tuplas
  - dicionários
  - Arrays (NumPy) e
  - dataframes (pandas)

Estruturas de dados

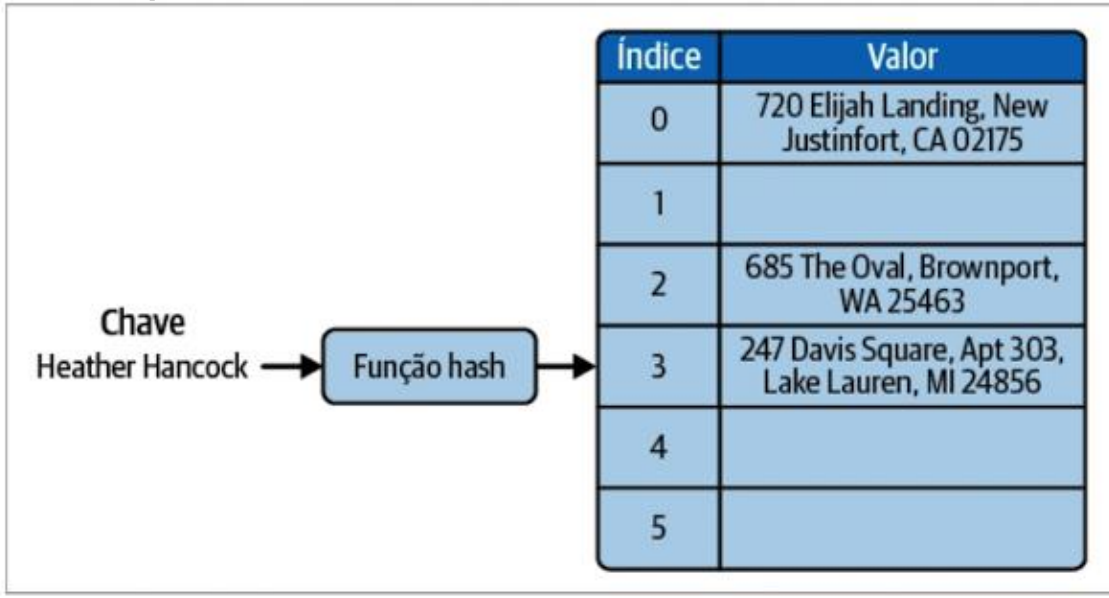
Estrutura	Características principais	Símbolo usado	Exemplo
Tupla	Imutável, tamanho fixo, acesso por índice (O(1))	()	(3.5, 7.2, 0.0)
Lista	Mutável, ordenada, permite vários tipos de dados	[]	['maçã', 'banana', 'laranja']
Array	Estrutura homogênea, mais eficiente para cálculo numérico	array.array() ou np.array()	np.array([1, 2, 3]) (com NumPy)
Dicionário	Chave-valor, acesso rápido por chave (O(1))	{}	{'nome': 'Ana', 'idade': 25}
Set	Conjunto de valores únicos, não ordenado, busca rápida (O(1))	{}	{'maçã', 'banana', 'laranja'}
DataFrame	Tabela (linhas e colunas), ideal para manipulação de dados tabulares	pd.DataFrame()	DataFrame com colunas Nome, Email, CPF

Dicionários

- Baseados em pares chave-valor
- Existem pares de elementos de dados que tem algum vínculo
- Dicionários são melhores para dados que não tem ordenação inerente

Dicionários

- Dicionários em Python são implementados por tabelas hash
- Usa função hash para transformar uma chave no índice de uma lista
- A busca em tuplas e dicionários é O(1) porque são otimizadas para acesso direto via índice (tuplas) ou chave (dicionário com tabela hash)



Criando dicionário

```python
from faker import Faker

fake = Faker('pt_BR')

# Criar um dicionário com dados de 5 clientes
clientes = {}

for _ in range(5):
    nome = fake.name()
    dados = {
        'CPF': fake.cpf(),
        'Email': fake.email(),
        'Endereço': fake.address(),
        'Data de Nascimento': fake.date_of_birth(minimum_age=18, maximum_age=65)
    }
    clientes[nome] = dados

# Exibir os dados
for nome, info in clientes.items():
    print(f"Cliente: {nome}")
    for chave, valor in info.items():
        print(f"  {chave}: {valor}")
    print("-" * 40)
```

Tempo em dicionário

```
! pip install faker
from faker import Faker
import timeit
import random

fake = Faker()

# Criar dicionário com 10 pares
dict_10 = {fake.name(): fake.email() for _ in range(10)}
chave_10 = list(dict_10.keys())[-1]  # última chave (pior caso hipotético)

# Criar dicionário com 10.000 pares
dict_10000 = {fake.name(): fake.email() for _ in range(10000)}
chave_10000 = list(dict_10000.keys())[-1]

# Teste de tempo para 10 buscas
tempo_10 = timeit.timeit(stmt="chave_10 in dict_10", globals=globals(), number=10000)
tempo_10000 = timeit.timeit(stmt="chave_10000 in dict_10000", globals=globals(), number=10000)

print("🔍 Tempo total para 10.000 buscas:")
print(f"Dicionário com     10 elementos: {tempo_10:.6f} segundos")
print(f"Dicionário com 10.000 elementos: {tempo_10000:.6f} segundos")
```

Dicionários

Limitação	Explicação
Uso de memória	Dict consome mais memória que listas por conta do hash
Colisões do hash	Muitas colisões podem degradar a performance
Acesso somente por chave	Você precisa conhecer a chave exata

Tuplas

- Array que tem tamanho específico e valores Imutáveis
- Úteis quando temos apenas alguns itens que queremos armazenar em uma estrutura de dados, e eles não serão alterados
- É armazenado no cache e não na memória
- Busca é O(1)

Tuplas

```python
# Criando uma tupla com três elementos
coordenadas = (3.5, 7.2, 0.0)
# Exibindo a tupla
print("Tupla:", coordenadas)
# Tentando alterar um valor (isso vai gerar erro!)
try:
    coordenadas[0] = 10
except TypeError as e:
    print("Erro ao tentar alterar a tupla:", e)
# Acessando um valor (busca O(1))
print("Segundo valor (índice 1):", coordenadas[1])
# Verificando se um valor existe (ainda é O(n), mas acesso direto por índice é O(1))
print("Existe o valor 7.2?", 7.2 in coordenadas)
# Tuplas são úteis quando os dados são fixos, ex:
PI = (3.14159,)  # vírgula é obrigatória se for uma tupla com 1 item
RGB_BRANCO = (255, 255, 255)
dias_da_semana = ('segunda', 'terça', 'quarta', 'quinta', 'sexta', 'sábado', 'domingo')
# Tuplas podem ser usadas como chaves em dicionários (por serem imutáveis)
cidades = {
    ('São Paulo', 'SP'): 12_000_000,
    ('Rio de Janeiro', 'RJ'): 6_700_000
}
print("População de São Paulo:", cidades[('São Paulo', 'SP')])
```

Arrays do numpy

- Biblioteca para análise de arrays
- Principal estrutura de dados: ndarray ou array n-dimensional
- Em ciência de dados, frequentemente, os dados estão em uma matriz
- Arrays permitem dados de apenas um único tipo
  - Vantagem: desempenho
  - Isso aumenta a eficiência e permite uso de operações vetorizadas em C

Criando arrays

```
import numpy as np
```

```
numeros = np.array([2,3,5,7,11])
```

```
print(type(numeros))
print(numeros)
```

Função ARRAY recebe uma coleção de elementos e retorna um novo array contendo os elementos passados
O type de dado é nparray mas a saída é array

Argumentos multidimensionais

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

atributos

- Exibe informações sobre a estrutura e conteúdo

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

```
inteiros = np.array([[1,2,3],[4,5,6]])
print(inteiros)
reais = np.array([0.0,0.1,0.3,0.4])
print(reais)
```

```
inteiros.dtype # tipo de dado dos elementos
```

```
reais.dtype
```

```
print(inteiros.ndim) #dimensão
print(reais.ndim)
print(inteiros.shape)
print(reais.shape)
```

```
print(inteiros.size) # tamanho
print(inteiros.itemsize)
print(reais.size)
print(reais.itemsize)
```

```
for row in inteiros:  #iteração pela array
  for column in row:
    print(column, end= " ")
  print()
```

Preenchendo arrays

- Funções que preenchem arrays com zeros, uns ou um determinado valor

```
np.zeros(5)
```

```
np.ones((2,4), dtype=int)
```

```
np.full((3,5), 13)
```

Preenchendo a partir de faixa de valores

- Arange()
- Linspace – números reais
  - Inicio e fim e se o valor é incluso
- Reshape
  - Transforma um array unidimensional em multidimensional

```
np.linspace(0.0,1.0,num=5)
```

```
np.arange(1,21).reshape(4,5)
```

```
np.arange(1,100001).reshape(4,25000)
```

```
np.arange(1,100001).reshape(100,1000)
```

Numpy não exibe acima de 1000 itens em um array

Performance de lista x array

- A maioria das operações com arrays executam de forma significantemente mais rápidas que listas

```python
import random
import time

start = time.time()
rolls_list = [random.randint(1, 6) for _ in range(6_000_000)]
end = time.time()
print(f"Tempo: {end - start:.4f} segundos")
```

```python
import numpy as np
import time

start = time.time()
rolls_array = np.random.randint(1, 7, size=6_000_000)
end = time.time()

print(f"Tempo: {end - start:.4f} segundos")
```

Operadores

- Operadores aritméticos com arrays e valores individuais
- Broadcasting
- Operadores entre arrays

```
numeros = np.arange(1,6)
print(numeros)
print(numeros *2)
print(numeros **3)
print(numeros)
```

```
print(numeros *[2,2,2,2,2])
```

```
numeros2 = np.linspace(1.1,5.5,5)
print(numeros2)
print(numeros * numeros2)
```

Comparação

```
numeros = np.array([11,12,13,14,15])
print(numeros >=13)
numeros2 = np.array([1.1,2.2,3.3,4.4,5.5])
print(numeros2 < numeros)
print(numeros2 == numeros)
print(numeros2 != numeros)
```

Algoritmos de cálculo Numpy

```
notas = np.array([[87,96,70], [100,87,90],[94,77,90],[100,81,82]])
print(notas.sum())
print(notas.min())
print(notas.max())
print(notas.mean())
print(notas.std())
print(notas.var())
```

```
print(notas.mean(axis=0))
print(notas.mean(axis=1))
```

Funções universais (uFuncs)

- O NumPy oferece dezenas de ufuncs independentes que realizam diversas operações elemento a elemento
- Broadcast com multiply

```
numeros = np.array([1,4,9,16,25,36])
print(np.sqrt(numeros))
numeros2= np.arange(1,7)*10
print(numero3)
```

```
np.add(numeros,numeros2) # numeros +numeros2
```

```
print(np.multiply(numeros2,5))
```

```
numeros3 = numeros2.reshape((2,3))
print(numeros3)
numeros4 = np.array([2,4,6])
print(np.multiply(numeros3,numeros4))
```

https://numpy.org/doc/stable/reference/ufuncs.html

Indexar e fatiar

- Acessar elementos individuais
- Acessar subconjuntos

```
notas = np.array([[87,96,70], [100,87,90],[94,77,90],[100,81,82]])
print(notas)
print(notas[0,1])
```

```
print(notas[1])
print(notas[0:2,1])
print(notas[:,1:3])
print(notas[::2])
```

Visões e shallow copy

- Retorna um novo objeto array com a visão original dos dados da array
  - Diferentes objets (id)
  - Visões fatiadas
  - Tente acessar um índice fora da faixa (numeros2[3])

```
numeros = np.arange(1,6)
print(numeros)
numeros2 =  numeros.view()
print(numeros2)
```

```
print(id(numeros))
print(id(numeros2))
```

```
numeros[1] *=10
print(numeros2)
```

```
numeros2[1]/=10
print(numeros)
```

```
numeros2 = numeros[0:3]
print(numeros2)
```

```
print(id(numeros))
print(id(numeros2))
```

Cópia profunda

```
numeros = np.arange(1,6)
print(numeros)
numeros2 = numeros.copy()
print(numeros2)
```

```
numeros[1]*=10
print(numeros)
print(numeros2)
```

Reshape

- Produz uma matriz a partir de um vetor

```
notas = np.array([[87,96,70],[100,87,90]])
print(notas)
notas.reshape(1,6)
print(notas)
```

```
notas.resize(1,6)
print(notas)
```

Flatten x Ravel

- Tanto flatten() quanto ravel() transformam arrays multidimensionais em **uma única dimensão**, mas com uma diferença importante:
  - Flatten faz cópia profunda e não modifica o original
  - Ravel() faz uma view(shallow copy) e pode modificar o original

```
notas = np.array([[87,96,70],[100,87,90]])
print(notas)
flattened = notas.flatten()
print(flattened)
flattened[0] =100
print(flattened)
print(notas)
```

```
reveled = notas.ravel()
print(reveled)
reveled[0] = 100
print(reveled)
print(notas)
```

transposição

- Transposição de linhas e colunas
- Retorna uma shallow copy da array

```
notas.T
```

```
notas
```

Empilhamento

No **NumPy**, empilhar arrays significa **combiná-los** em uma nova estrutura:

Tipo	Função NumPy	Efeito
Empilhamento horizontal	`np.hstack()`	Junta **lado a lado** (colunas)
Empilhamento vertical	`np.vstack()`	Junta **de cima para baixo** (linhas)

```
notas2 = np.array([[94,77,90],[100,81,82]])
print(np.hstack((notas,notas2)))
print(np.vstack((notas,notas2)))
```