

Programação para Ciência de Dados

Estruturas de Dados Avançadas e Algoritmos Básicos

Arthur Casals

09 de Outubro de 2025

- ▶ Suporte ao Moodle
- ▶ Interação via Zoom

Agenda

- ▶ Estruturas aninhadas e coleções
- ▶ Compreensões avançadas e geradores
- ▶ Complexidade e desempenho
- ▶ Algoritmos de busca e complexidade

Recapitulação Aula 01

O que já sabemos:

- ▶ Variáveis e tipos de dados básicos
- ▶ Estruturas condicionais (if/elif/else)
- ▶ Loops (for/while)
- ▶ Funções básicas
- ▶ Projeto Titanic com Python puro

Hoje vamos evoluir para:

- ▶ Estruturas de dados mais sofisticadas
- ▶ Manipulação eficiente de dados

Bloco 1

Estruturas aninhadas e coleções

O que vamos aprender:

- ▶ Listas aninhadas e representação de matrizes
- ▶ Operações matriciais básicas em Python puro
- ▶ Coleções: ferramentas da biblioteca padrão
- ▶ Manipulação eficiente de dados tabulares

Listas Aninhadas

Representando Estruturas Bidimensionais

O que são listas aninhadas?

- ▶ Listas que contêm outras listas como elementos
- ▶ Forma natural de representar matrizes e tabelas
- ▶ Base para manipulação de dados antes de bibliotecas especializadas

Aplicações práticas:

- ▶ Planilhas e tabelas de dados
- ▶ Matrizes matemáticas
- ▶ Tabuleiros de jogos
- ▶ Pixels de imagens
- ▶ Dados de séries temporais multivariadas

Criando Listas Aninhadas

Criação manual:

</> Python

```
1 # Matriz 3x3: lista de linhas
2 matriz = [
3     [1, 2, 3],
4     [4, 5, 6],
5     [7, 8, 9]
6 ]
7 # Tabela de dados: lista de registros
8 alunos = [
9     ['Ana', 8.5, 'SP'],
10    ['Bruno', 7.2, 'RJ'],
11    ['Carlos', 9.1, 'MG']
12 ]
```

Criando Matrizes Programaticamente

Python

```
1 # Loop aninhado: criar matriz 4x5 preenchida com zeros
2 linhas, colunas = 4, 5
3 matriz = []
4 for i in range(linhas):
5     linha = []
6     for j in range(colunas):
7         linha.append(0)
8     matriz.append(linha)
9 print(matriz)
10
11 # [[0, 0, 0, 0, 0],
12 #  [0, 0, 0, 0, 0],
13 #  [0, 0, 0, 0, 0],
14 #  [0, 0, 0, 0, 0]]
```

List Comprehension (Compreensão de Listas)

- ▶ É uma forma concisa de criar e manipular listas em Python, reduzindo a quantidade de código necessário para gerar listas a partir de operações ou filtragem de elementos.
- ▶ Permite aplicar uma expressão a cada item de uma sequência, possivelmente filtrando elementos com condições, e gera uma nova lista como resultado.
- ▶ A sintaxe básica é: (expressão) for (item) in (lista), podendo incluir if para condições.
- ▶ Expressões podem ser aninhadas

Criação com List Comprehension

</> Python

```
1 # Matriz de zeros
2 linhas, colunas = 4, 5
3 matriz = [[0 for j in range(colunas)] for i in range(linhas)]
4
5 # Matriz identidade 4x4
6 n = 4
7 identidade = [[1 if i == j else 0 for j in range(n)]
8               for i in range(n)]
9 print(identidade)
10 # [[1, 0, 0, 0],
11 #  [0, 1, 0, 0],
12 #  [0, 0, 1, 0],
13 #  [0, 0, 0, 1]]
```

Acessando Elementos

</> Python

```
1 matriz = [  
2     [1, 2, 3],  
3     [4, 5, 6],  
4     [7, 8, 9]  
5 ]  
6  
7 # Acessar linha  
8 primeira_linha = matriz[0] # [1, 2, 3]  
9  
10 # Acessar elemento especifico: linha 1, coluna 2  
11 elemento = matriz[1][2] # 6  
12
```

Acessando Elementos

</> Python

```
1 matriz = [  
2     [1, 2, 3],  
3     [4, 5, 6],  
4     [7, 8, 9]  
5 ]  
6  
7 # Modificar elemento  
8 matriz[0][1] = 99  
9 print(matriz[0])  # [1, 99, 3]  
10  
11 # Acessar ultima linha, ultimo elemento  
12 ultimo = matriz[-1][-1]  # 9  
13
```

Armadilha Comum: Referências Compartilhadas

</> Python

```
1 # ERRADO - Todas as linhas apontam para o mesmo objeto!
2 matriz_errada = [[0] * 3] * 4
3 matriz_errada[0][0] = 1
4 print(matriz_errada)
5 # [[1, 0, 0],
6 #  [1, 0, 0], # Ops! Todas as linhas mudaram!
7 #  [1, 0, 0],
8 #  [1, 0, 0]]
9
```

Armadilha Comum: Referências Compartilhadas

</> Python

```
1 # CORRETO - Cada linha é um objeto diferente
2 matriz_certa = [[0] * 3 for _ in range(4)]
3 matriz_certa[0][0] = 1
4 print(matriz_certa)
5 # [[1, 0, 0],
6 #  [0, 0, 0], # Correto! Apenas primeira linha mudou
7 #  [0, 0, 0],
8 #  [0, 0, 0]]
9
```

Iterando sobre Listas Aninhadas

Iterar por linhas:

Python

```
1 matriz = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
2
3 # Iterar linha por linha
4 for linha in matriz:
5     print(linha)
6
7 # Iterar elemento por elemento
8 for linha in matriz:
9     for elemento in linha:
10         print(elemento, end=' ')
11     print() # Nova linha apos cada linha da matriz
12
```

Iterando sobre Listas Aninhadas

Iterar por linhas:

</> Python

```
1 matriz = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
2
3 # Iterar com indices
4 for i in range(len(matriz)):
5     for j in range(len(matriz[i])):
6         print(f"matriz[{i}][{j}] = {matriz[i][j]}")
7
```

Operações Matriciais: Adição

</> Python

```
1 def somar_matrizes(A, B):
2     """Soma duas matrizes de mesma dimensao."""
3     # Verificar dimensoes
4     if len(A) != len(B) or len(A[0]) != len(B[0]):
5         raise ValueError("Matrizes devem ter mesmas dimensoes")
6
7     linhas = len(A)
8     colunas = len(A[0])
9
10    # Criar matriz resultado
11    resultado = [[0 for j in range(colunas)]
12                 for i in range(linhas)]
13
```

Operações Matriciais: Adição

</> Python

```
1 def somar_matrizes(A, B):
2     """(continuação)"""
3
4     # Somar elemento por elemento
5     for i in range(linhas):
6         for j in range(colunas):
7             resultado[i][j] = A[i][j] + B[i][j]
8
9     return resultado
10
11 # Exemplo de uso
12 A = [[1, 2], [3, 4]]
13 B = [[5, 6], [7, 8]]
14 C = somar_matrizes(A, B)
15 print(C)  # [[6, 8], [10, 12]]
```

Operações Matriciais: Transposição

Python

```
1 def transpor_matriz(matriz):
2     """Retorna a transposta da matriz."""
3     linhas = len(matriz)
4     colunas = len(matriz[0])
5     # Criar matriz transposta (colunas x linhas)
6     transposta = [[0 for i in range(linhas)]
7                   for j in range(colunas)]
8     # Transpor: linha vira coluna, coluna vira linha
9     for i in range(linhas):
10         for j in range(colunas):
11             transposta[j][i] = matriz[i][j]
12     return transposta
```

Operações Matriciais: Transposição

</> Python

```
1 # Exemplo
2 matriz = [[1, 2, 3], [4, 5, 6]]
3 print("Original:", matriz)
4 # [[1, 2, 3], [4, 5, 6]] - Matriz 2x3
5
6 print("Transposta:", transpor_matriz(matriz))
7 # [[1, 4], [2, 5], [3, 6]] - Matriz 3x2
8
```

Operações Matriciais: Multiplicação

</> Python

```
1 def multiplicar_matrizes(A, B):
2     """Multiplica duas matrizes A e B."""
3     linhas_A = len(A)
4     colunas_A = len(A[0])
5     colunas_B = len(B[0])
6
7     # Verificar se multiplicacao e possivel
8     if colunas_A != len(B):
9         raise ValueError("Numero de colunas de A deve ser igual ao numero
10        de linhas de B")
```

Operações Matriciais: Multiplicação

</> Python

```
1  """(continuação)"""
2  # Criar matriz resultado
3  resultado = [[0 for j in range(colunas_B)]
4               for i in range(linhas_A)]
5
6  # Multiplicacao matricial
7  for i in range(linhas_A):
8      for j in range(colunas_B):
9          for k in range(colunas_A):
10             resultado[i][j] += A[i][k] * B[k][j]
11  return resultado
```

Trabalhando com Dados Tabulares

Python

```
1 # Tabela: lista de listas (cada linha e um registro)
2 vendas = [
3     ['Produto', 'Quantidade', 'Preco'], # Cabecalho
4     ['Notebook', 10, 2500.00],
5     ['Mouse', 50, 25.00],
6     ['Teclado', 30, 150.00],
7     ['Monitor', 15, 800.00]
8 ]
9
10 # Acessar dados
11 cabecalho = vendas[0]
12 dados = vendas[1:] # Todos exceto cabecalho
13
14
```

Trabalhando com Dados Tabulares

</> Python

```
1 """(continuação)"""
2
3 # Processar dados
4 for registro in dados:
5     produto, qtd, preco = registro
6     total = qtd * preco
7     print(f"{produto}: {qtd} unidades = R$ {total:.2f}")
8
```

Coleções: Estruturas de Dados Especializadas

- ▶ Estruturas de dados otimizadas para casos específicos
- ▶ Parte da biblioteca padrão Python
- ▶ Mais eficientes que soluções manuais
- ▶ Código mais limpo e expressivo

Principais tipos:

- ▶ `defaultdict` - Dicionário com valores padrão
- ▶ `Counter` - Contador de elementos
- ▶ `deque` - Fila de duas pontas eficiente
- ▶ `namedtuple` - Tupla com campos nomeados

defaultdict

- ▶ `defaultdict` é uma subclasse de `dict` encontrada no módulo *collections*.
- ▶ Permite definir um **valor padrão automático** para chaves não existentes, evitando erros caso você tente acessar uma chave ainda não criada.
- ▶ Exemplo: se você criar `d = defaultdict(int)`, acessar uma chave ainda não existente retorna `0` (o padrão de `int`), em vez de gerar `KeyError`.
- ▶ Possibilita simplificar códigos que acumulam elementos em listas, contadores, etc., sem necessidade de testar previamente a existência da chave.

Usando defaultdict

</> Python

```
1 # Contar palavras - forma tradicional (verbosa)
2 texto = "python e legal python e poderoso"
3 palavras = texto.split() #Cria uma lista de substrings utilizando o espaço
   em branco como separador
4
5 contagem = {}
6 for palavra in palavras:
7     if palavra not in contagem:
8         contagem[palavra] = 0
9     contagem[palavra] += 1
10
11 print(contagem) # {'python': 2, 'e': 2, 'legal': 1, 'poderoso': 1}
12
```

Usando defaultdict

Python

```
1 from collections import defaultdict
2
3 texto = "python e legal python e poderoso"
4 palavras = texto.split() #Cria uma lista de substrings utilizando o espaço
   em branco como separador
5
6 contagem = defaultdict(int) # int() retorna 0
7 for palavra in palavras:
8     contagem[palavra] += 1 # Não precisa verificar existência!
9
10 print(dict(contagem)) # Mesmo resultado, código mais limpo
11
```

defaultdict: Diferentes Tipos de Valores Padrão

Python

```
1 from collections import defaultdict
2 # Agrupar items por categoria
3 categorias = defaultdict(list) # list() retorna []
4 produtos = [
5     ('Notebook', 'Eletronicos'),
6     ('Mouse', 'Eletronicos'),
7     ('Caderno', 'Papeleria'),
8     ('Caneta', 'Papeleria')
9 ]
10 for produto, categoria in produtos:
11     categorias[categoria].append(produto)
12 print(dict(categorias))
13 # {'Eletronicos': ['Notebook', 'Mouse'],
14 #  'Papeleria': ['Caderno', 'Caneta']}
```

defaultdict: Diferentes Tipos de Valores Padrão

</> Python

```
1 from collections import defaultdict
2 # Contar ocorrencias com valor padrao customizado
3 contador = defaultdict(lambda: 0) # Funcao anônima que retorna sempre 0
4 contador['a'] += 1 # 'a' não existia antes, então vira 0+1 = 1
5 print(contador) # defaultdict(<function <lambda> ...>, {'a': 1})
6 print(contador['b']) # 0 (valor padrao)
7
```

Bônus: funções anônimas (lambda)

Funções lambda:

- ▶ São funções anônimas, ou seja, não possuem nome.
- ▶ São definidas em uma única linha e geralmente usadas para tarefas simples e rápidas.
- ▶ Úteis quando se deseja criar uma função pequena que será usada apenas uma vez ou como argumento para funções como `map`, `filter` e `sorted`, facilitando operações rápidas e sem a necessidade de definir uma função tradicional com `def`.
- ▶ Sintaxe: `lambda argumentos: expressão`

Bônus: funções anônimas (lambda)

</> Python

```
1 soma = lambda x, y: x + y
2 print(soma(5, 3))
3 # Saída: 8
4
```

Counter: Contador Especializado

- ▶ Counter é uma estrutura de dados da biblioteca *collections* em Python, projetada para contar objetos hashable de forma eficiente.
- ▶ Implementação baseada no dict (utiliza as operações internas de dicionário).
- ▶ Armazena os elementos como *chaves* do dicionário e suas respectivas contagens como *valores*.
- ▶ Permite operações como recuperar os elementos mais comuns usando o método `.most_common()`. Exemplo: `Counter('aabbbc').most_common(2)` retorna `[('b', 3), ('a', 2)]`.

Counter: Contador Especializado

- ▶ Suporta aritmética de multisets ($\{a, a, b, b, c, c\}$), como soma, subtração, união e interseção de contadores, além de métodos como `.elements()` para listar todos os itens conforme a contagem.
- ▶ Facilita estatísticas rápidas e análise de frequência em textos, listas e outros dados, evitando código extra para inicialização ou teste de chaves ausentes.

Counter: Contador Especializado

</> Python

```
1 from collections import Counter
2 # Contar elementos em lista
3 frutas = ['maca', 'banana', 'maca', 'laranja', 'banana', 'maca']
4 contagem = Counter(frutas)
5 print(contagem) # Counter({'maca': 3, 'banana': 2, 'laranja': 1})
6 # Metodos uteis
7 print(contagem.most_common(2)) # [('maca', 3), ('banana', 2)]
8 print(contagem['maca']) # 3
9 print(contagem['uva']) # 0 (nao da erro!)
10 # Contar caracteres em string
11 texto = "mississippi"
12 letras = Counter(texto)
13 print(letras) # Counter({'i': 4, 's': 4, 'p': 2, 'm': 1})
```

Counter: Operações Matemáticas

</> Python

```
1 from collections import Counter
2 # Criar contadores
3 vendas_janeiro = Counter({'Notebook': 5, 'Mouse': 10, 'Teclado': 8})
4 vendas_fevereiro = Counter({'Notebook': 3, 'Mouse': 15, 'Monitor': 4})
5 # Somar contadores
6 total = vendas_janeiro + vendas_fevereiro
7 print(total)
8 # Counter({'Mouse': 25, 'Notebook': 8, 'Teclado': 8, 'Monitor': 4})
9 # Subtrair contadores
10 diferenca = vendas_fevereiro - vendas_janeiro
11 print(diferenca) # Counter({'Mouse': 5, 'Monitor': 4})
12 # Intersecao (minimo)
13 comum = vendas_janeiro & vendas_fevereiro
14 print(comum) # Counter({'Notebook': 3, 'Mouse': 10})
```

deque: Fila de Duas Pontas

- ▶ Uma fila de duas pontas (deque) é uma estrutura de dados que permite a inserção e remoção de elementos tanto no início quanto no final da fila.
- ▶ Em Python, o deque é implementado pela classe deque do módulo collections, e normalmente é construído sobre uma lista duplamente ligada, o que garante operações eficientes para adicionar ou remover na frente e no final.
- ▶ Operações básicas: `append()`, `appendleft()`, `pop()`, `popleft()`.
- ▶ É útil para simular filas, pilhas reversíveis, buffers circulares, e algoritmos que exigem manipulação bidirecional dos dados.

deque: Fila de Duas Pontas

Diferenças em relação a listas

- ▶ Listas (`list`) permitem inserção e remoção em qualquer posição, mas são otimizadas apenas para adicionar/remover elementos no final (operador `append` e `pop`). Inserir ou remover elementos no início de uma lista é menos eficiente ($O(n)$), pois exige deslocamento de todos os demais elementos.
- ▶ Deque foi feito precisamente para operações rápidas nas duas pontas, sendo superior para casos com necessidade de manipular dados tanto à esquerda quanto à direita.
- ▶ Ambas permitem armazenar tipos variados de dados, mas o deque oferece melhor desempenho em cenários de acesso e modificação bidirecional.

deque: Fila de Duas Pontas

</> Python

```
1 from collections import deque
2 # Criar fila
3 fila = deque([1, 2, 3])
4 # Adicionar no final (como lista)
5 fila.append(4)          # deque([1, 2, 3, 4])
6 # Adicionar no inicio (rapido!)
7 fila.appendleft(0)      # deque([0, 1, 2, 3, 4])
8 # Remover do final
9 fila.pop()              # retorna 4, deque([0, 1, 2, 3])
10 # Remover do inicio (rapido!)
11 fila.popleft()          # retorna 0, deque([1, 2, 3])
12 # Rotacionar elementos
13 fila.rotate(1)          # deque([3, 1, 2])
14 fila.rotate(-1)         # deque([1, 2, 3])
```

namedtuple: Tuplas com Campos Nomeados

Diferenças e vantagens em relação a tuplas

- ▶ Nas `namedtuples` você pode acessar cada campo pelo nome (ex: `ponto.x`), enquanto nas tuplas normais o acesso é apenas por índice (`ponto[0]`). Isso torna o código mais legível e autoexplicativo.
- ▶ O uso de nomes para os campos ajuda a entender para que serve cada valor na estrutura, tornando o código mais fácil de manter. Com tuplas comuns, é preciso lembrar o significado de cada posição.
- ▶ Ambas são imutáveis, ou seja, uma vez criada, não é possível alterar seus valores.
- ▶ Funcionalidades extra: Métodos como `_fields`, `_asdict()`, `_replace()`, que facilitam inspeção e manipulação dos dados contidos na estrutura.

namedtuple: Tuplas com Campos Nomeados

Diferenças e vantagens em relação a tuplas

- ▶ `namedtuple` é uma alternativa leve a classes simples, sem a sobrecarga de memória de instâncias comuns e mais rápidas que dicionários, pois não possuem dicionário próprio por instância.
- ▶ Para usar *namedtuple*, é preciso importar de `collections` ou `typing`, enquanto tuplas usam apenas parênteses.
- ▶ A exibição (`print`) traz os nomes dos campos e os valores na `namedtuple` (ex: `Point(x=3, y=5)`), ao contrário da tupla, que mostra só os valores (ex: `(3, 5)`)

namedtuple: Tuplas com Campos Nomeados

</> Python

```
1 from collections import namedtuple
2 # Definir estrutura
3 Aluno = namedtuple('Aluno', ['nome', 'idade', 'nota'])
4 # Criar instancias
5 aluno1 = Aluno('Ana', 20, 8.5)
6 aluno2 = Aluno(nome='Bruno', idade=22, nota=7.2)
7 print(aluno1.nome)      # Ana
8 print(aluno1.nota)      # 8.5
9 print(aluno1[0])        # Ana
10 # aluno1.nota = 9.0    # AttributeError!
11 # Converter para dicionario
12 print(aluno1._asdict())
13 # {'nome': 'Ana', 'idade': 20, 'nota': 8.5}
```

Live Coding

Vamos codificar juntos:

1. Abra o Google Colab
2. Utilize o notebook disponibilizado no Moodle
3. Siga os passos da demonstração

Bloco 2

Compreensões Avançadas e Generators

Bloco 2: Compreensões Avançadas e Generators

O que vamos aprender:

- ▶ List comprehensions aninhadas para estruturas complexas
- ▶ Dictionary e set comprehensions avançadas
- ▶ Generators: iteração eficiente com yield
- ▶ Criação de iterators customizados
- ▶ Técnicas de transformação de dados

Estrutura do bloco:

1. List comprehensions aninhadas
2. Dict/Set comprehensions
3. Generators e yield
4. Iterators customizados

List Comprehensions Aninhadas

Quando usar comprehensions aninhadas?

- ▶ Processar matrizes e estruturas bidimensionais
- ▶ Transformar dados aninhados
- ▶ Gerar combinações de elementos
- ▶ Achatar (*flatten*) estruturas aninhadas

Estrutura mental:

```
[expressao for item_interno in lista_interna] for item_externo in  
lista_externa]
```

💡 Nota Importante

Dica: Leia de fora para dentro, da esquerda para direita

Criando Matrizes com Comprehensions

</> Python

```
1 # Matriz 3x4 preenchida com zeros
2 matriz = [[0 for j in range(4)] for i in range(3)]
3 print(matriz)
4 # [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
5
6 # Matriz com valores baseados em posicao
7 matriz = [[i * 4 + j for j in range(4)] for i in range(3)]
8 print(matriz)
9 # [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]
10
```

Criando Matrizes com Comprehensions

</> Python

```
1 # Matriz identidade
2 n = 4
3 identidade = [[1 if i == j else 0 for j in range(n)]
4               for i in range(n)]
5 print(identidade)
6 # [[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]]
7
8 # Tabela de multiplicacao
9 tabuada = [[i * j for j in range(1, 11)] for i in range(1, 11)]
10
```

Processando Estruturas Aninhadas

</> Python

```
1 # Matriz original
2 matriz = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
3
4 # Dobrar todos os valores
5 dobrada = [[x * 2 for x in linha] for linha in matriz]
6 print(dobrada)
7 # [[2, 4, 6], [8, 10, 12], [14, 16, 18]]
8
9 # Filtrar valores pares de cada linha
10 pares = [[x for x in linha if x % 2 == 0] for linha in matriz]
11 print(pares)
12 # [[2], [4, 6], [8]]
13
```

Processando Estruturas Aninhadas

</> Python

```
1 # Matriz original
2 matriz = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
3
4 # Transpor matriz usando comprehension
5 transposta = [[matriz[i][j] for i in range(len(matriz))]
6               for j in range(len(matriz[0]))]
7 print(transposta)
8 # [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
9
```

Achatando Listas Aninhadas (Flatten)

</> Python

```
1 # Lista aninhada
2 matriz = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
3
4 # Achatar para lista simples
5 flat = [elemento for linha in matriz for elemento in linha]
6 print(flat)
7 # [1, 2, 3, 4, 5, 6, 7, 8, 9]
8
9 # Achatar com condicao
10 flat_pares = [x for linha in matriz for x in linha if x % 2 == 0]
11 print(flat_pares)
12 # [2, 4, 6, 8]
13
```

Achatando Listas Aninhadas (Flatten)

</> Python

```
1 # Achatar e transformar
2 flat_quadrados = [x**2 for linha in matriz for x in linha]
3 print(flat_quadrados)
4 # [1, 4, 9, 16, 25, 36, 49, 64, 81]
5
6 # Lista de listas com tamanhos diferentes
7 irregular = [[1, 2], [3, 4, 5], [6]]
8 flat = [x for sublista in irregular for x in sublista]
9 print(flat) # [1, 2, 3, 4, 5, 6]
10
```

Produto Cartesiano com Comprehensions

</> Python

```
1 # Produto cartesiano de duas listas
2 cores = ['vermelho', 'azul', 'verde']
3 tamanhos = ['P', 'M', 'G']
4
5 produtos = [(cor, tamanho) for cor in cores for tamanho in tamanhos]
6 print(produtos)
7 # [('vermelho', 'P'), ('vermelho', 'M'), ('vermelho', 'G'),
8 #  ('azul', 'P'), ('azul', 'M'), ('azul', 'G'),
9 #  ('verde', 'P'), ('verde', 'M'), ('verde', 'G')]
10
```

Produto Cartesiano com Comprehensions

</> Python

```
1 # Combinacoes com condicao
2 numeros1 = [1, 2, 3]
3 numeros2 = [3, 4, 5]
4 pares = [(x, y) for x in numeros1 for y in numeros2 if x < y]
5 print(pares)
6 # [(1, 3), (1, 4), (1, 5), (2, 3), (2, 4), (2, 5), (3, 4), (3, 5)]
7
8 # Coordenadas de um grid
9 grid = [(x, y) for x in range(3) for y in range(3)]
10 print(grid)
11 # [(0,0), (0,1), (0,2), (1,0), (1,1), (1,2), (2,0), (2,1), (2,2)]
12
```

Dictionary Comprehensions

Sintaxe básica:

`{chave: valor for item in iteravel}`

Python

```
1 # Quadrados dos numeros
2 quadrados = {x: x**2 for x in range(1, 6)}
3 print(quadrados)
4 # {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
5
6 # Inverter dicionario
7 original = {'a': 1, 'b': 2, 'c': 3}
8 invertido = {v: k for k, v in original.items()}
9 print(invertido)
10 # {1: 'a', 2: 'b', 3: 'c'}
11
```

Dictionary Comprehensions

Sintaxe básica:

```
{chave: valor for item in iteravel}
```

</> Python

```
1 # Criar dicionario a partir de listas
2 nomes = ['Ana', 'Bruno', 'Carlos']
3 idades = [25, 30, 28]
4 pessoas = {nome: idade for nome, idade in zip(nomes, idades)}
5 print(pessoas)
6 # {'Ana': 25, 'Bruno': 30, 'Carlos': 28}
7
```

Compreensões com Condições

</> Python

```
1 # Filtrar valores
2 precos = {'maca': 3.50, 'banana': 2.00, 'laranja': 4.00, 'uva': 8.00}
3 caros = {fruta: preco for fruta, preco in precos.items()
4         if preco > 3.00}
5 print(caros)
6 # {'maca': 3.50, 'laranja': 4.00, 'uva': 8.00}
7
8 # Transformar chaves e valores
9 texto = "python"
10 freq = {char.upper(): texto.count(char)
11         for char in set(texto)}
12 print(freq)
13 # {'P': 1, 'Y': 1, 'T': 1, 'H': 1, 'O': 1, 'N': 1}
14
```

Compreensões com Condições

</> Python

```
1 # Condicional no valor (if-else)
2 numeros = [1, 2, 3, 4, 5, 6]
3 paridade = {n: 'par' if n % 2 == 0 else 'impar'
4             for n in numeros}
5 print(paridade)
6 # {1: 'impar', 2: 'par', 3: 'impar', 4: 'par', 5: 'impar', 6: 'par'}
7
```

Set Comprehensions (Conjuntos sem Duplicatas)

Sintaxe:

`{expressao for item in iteravel}`

</> Python

```
1 # Criar set de quadrados
2 quadrados = {x**2 for x in range(-5, 6)}
3 print(quadrados)
4 # {0, 1, 4, 9, 16, 25} # Sem duplicatas!
5
6 # Extrair caracteres unicos
7 texto = "mississippi"
8 letras = {char for char in texto}
9 print(letras)
10 # {'m', 'i', 's', 'p'}
11
```

Set Comprehensions (Conjuntos sem Duplicatas)

Sintaxe:

`{expressao for item in iteravel}`

</> Python

```
1 # Set com condicao
2 numeros = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
3 pares_unicos = {x for x in numeros if x % 2 == 0}
4 print(pares_unicos)
5 # {2, 4}
6
7 # Comprimentos de palavras
8 palavras = ["python", "java", "c", "javascript", "ruby"]
9 comprimentos = {len(p) for p in palavras}
10 print(comprimentos)
11 # {1, 4, 6, 10}
12
```

Generators: Iteração Eficiente

Definição:

- ▶ Funções que retornam iteradores
- ▶ Produzem valores sob demanda (lazy evaluation)
- ▶ Não armazenam todos os valores na memória
- ▶ Usam a palavra-chave `yield`

Vantagens:

- ▶ **Eficiência de memória:** Não criam listas completas
- ▶ **Performance:** Processamento sob demanda
- ▶ **Infinitos:** Podem representar sequências infinitas
- ▶ **Pipeline:** Ideal para processamento em etapas

Generators: Iteração Eficiente

Quando usar:

- ▶ Datasets grandes que não cabem na memória
- ▶ Processamento de streams de dados
- ▶ Quando você não precisa de todos os valores ao mesmo tempo

Criando Generators com yield

</> Python

```
1 # Generator simples
2 def contar_ate(n):
3     """Gera numeros de 1 ate n."""
4     i = 1
5     while i <= n:
6         yield i
7         i += 1
8 # Usar generator
9 for numero in contar_ate(3):
10     print(numero)
11 # 1
12 # 2
13 # 3
14
```

Criando Generators com yield

</> Python

```
1 # Generator e um iterador - pode converter para lista
2 numeros = list(contar_ate(10))
3 print(numeros) # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
4
5 # Mas cuidado - consome o generator!
6 gen = contar_ate(3)
7 print(list(gen)) # [1, 2, 3]
8 print(list(gen)) # [] - generator esgotado!
9
```

Generator vs Lista: Comparação

</> Python

```
1 # LISTA - armazena tudo na memória
2 def quadrados_lista(n):
3     resultado = []
4     for i in range(n):
5         resultado.append(i ** 2)
6     return resultado
7
8 lista = quadrados_lista(1000000) # 1 milhão de números!
9 # Ocupa muita memória
10
```

Generator vs Lista: Comparação

</> Python

```
1 # GENERATOR - produz valores sob demanda
2 def quadrados_generator(n):
3     for i in range(n):
4         yield i ** 2
5
6 gen = quadrados_generator(1000000)
7
8 primeiros_5 = []
9 for i, valor in enumerate(gen):
10     if i >= 5:
11         break
12     primeiros_5.append(valor)
13 print(primeiros_5) # [0, 1, 4, 9, 16]
14 # Resto nunca foi calculado!
15
```

Generator Expressions

Similar a list comprehension, mas com parênteses:

</> Python

```
1 # List comprehension - cria lista completa
2 lista = [x**2 for x in range(10)]
3 print(type(lista)) # <class 'list'>
4
5 # Generator expression - cria generator
6 gen = (x**2 for x in range(10))
7 print(type(gen))   # <class 'generator'>
8
9 # Usar generator
10 for valor in gen:
11     print(valor, end=' ')
12 # 0 1 4 9 16 25 36 49 64 81
13
```

Generator Expressions

Similar a list comprehension, mas com parênteses:

</> Python

```
1 # Generator expression com condicao
2 pares = (x for x in range(20) if x % 2 == 0)
3 print(list(pares))
4 # [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
5
6 # Pode passar direto para funcoes
7 soma = sum(x**2 for x in range(100)) # Sem colchetes!
8 print(soma) # 328350
9
```

Generators: Exemplos Práticos

</> Python

```
1 # Ler arquivo linha por linha (eficiente!)
2 def ler_arquivo(nome_arquivo):
3     """Generator que le arquivo linha por linha."""
4     with open(nome_arquivo, 'r') as arquivo:
5         for linha in arquivo:
6             yield linha.strip()
7
8 # Usar apenas as primeiras N linhas
9 # for i, linha in enumerate(ler_arquivo('dados.txt')):
10 #     if i >= 10:
11 #         break
12 #     print(linha)
13
```

Generators: Exemplos Práticos

</> Python

```
1 # Fibonacci com generator (sequencia infinita!)
2 def fibonacci():
3     """Gera sequencia de Fibonacci infinitamente."""
4     a, b = 0, 1
5     while True:
6         yield a
7         a, b = b, a + b
8
9 # Pegar apenas primeiros 10 numeros
10 fib = fibonacci()
11 primeiros_10 = [next(fib) for _ in range(10)]
12 print(primeiros_10)
13 # [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
14
```

Pipeline de Generators: Processamento em Etapas

</> Python

```
1 # Pipeline: generator alimenta outro generator
2 def numeros(n):
3     """Gera numeros de 1 a n."""
4     for i in range(1, n + 1):
5         yield i
6 def quadrados(nums):
7     """Eleva cada numero ao quadrado."""
8     for n in nums:
9         yield n ** 2
10 def pares(nums):
11     """Filtra apenas numeros pares."""
12     for n in nums:
13         if n % 2 == 0:
14             yield n
```

Pipeline de Generators: Processamento em Etapas

</> Python

```
1  """(continuação)"""
2  # Criar pipeline
3  nums = numeros(20)
4  quads = quadrados(nums)
5  resultado = pares(quads)
6
7  # Processar (lazy - so calcula quando necessario)
8  print(list(resultado))
9  # [4, 16, 36, 64, 100, 144, 196, 256, 324, 400]
10
```

Iterators Customizados

O que é um iterator?

- ▶ Objeto que implementa o protocolo de iteração
- ▶ Deve ter método `__iter__()` que retorna `self`
- ▶ Deve ter método `__next__()` que retorna próximo valor
- ▶ Lança `StopIteration` quando acabam os elementos

Por que criar iterators customizados?

- ▶ Mais controle sobre o processo de iteração
- ▶ Manter estado entre iterações
- ▶ Criar iteradores mais complexos
- ▶ Entender melhor como Python funciona

💡 Nota Importante

Nota: Generators são uma forma mais simples de criar iterators!

Live Coding

Vamos codificar juntos:

1. Abra o Google Colab
2. Utilize o notebook disponibilizado no Moodle
3. Siga os passos da demonstração

Bloco 3

Algoritmos de Busca e Complexidade

Bloco 3: Algoritmos e Complexidade

O que vamos aprender:

- ▶ Notação Big O e complexidade
- ▶ Algoritmos de busca (linear e binária)
- ▶ Ordenação em Python (`sorted()`, parâmetro `key`)
- ▶ Impacto na escolha de estruturas de dados
- ▶ Medição básica de performance

💡 Nota Importante

Objetivo: Fundamentos para trabalhar eficientemente com dados

Big O Notation: Introdução

Por que estudar complexidade?

- ▶ Prever comportamento com dados grandes
- ▶ Escolher estrutura de dados apropriada
- ▶ Identificar gargalos de desempenho

O que é Big O?

- ▶ Descreve como tempo/espço cresce com tamanho da entrada (n)
- ▶ Foca no caso assintótico (n muito grande)
- ▶ Ignora constantes: $O(2n + 5) \rightarrow O(n)$
- ▶ Usa termo dominante: $O(n^2 + n) \rightarrow O(n^2)$

Big O Notation: Introdução

Exemplo motivador:

- ▶ Algoritmo $O(n)$: 1.000.000 itens $\rightarrow \sim 1.000.000$ operações
- ▶ Algoritmo $O(\log n)$: 1.000.000 itens $\rightarrow \sim 20$ operações!
- ▶ **Complexidade influencia velocidade absoluta**

Complexidades Comuns

Do mais rápido ao mais lento:

Notação	Nome	n=1000	Exemplo
$O(1)$	Constante	1	Acesso a array[i]
$O(\log n)$	Logarítmica	~ 10	Busca binária
$O(n)$	Linear	1.000	Busca linear
$O(n \log n)$	Linearítmica	~ 10.000	sorted()
$O(n^2)$	Quadrática	1.000.000	Loops duplos
$O(2^n)$	Exponencial	$\sim 10^{300}$	Força bruta

Regra prática:

- ▶ $O(1)$, $O(\log n)$, $O(n)$: **Excelente**
- ▶ $O(n \log n)$: **Bom** (maioria dos algoritmos de ordenação)
- ▶ $O(n^2)$: **Aceitável apenas para n pequeno**
- ▶ $O(2^n)$, $O(n!)$: **Evitar!**

Exemplos de Complexidade

</> Python

```
1 # O(1) - Constante
2 lista = [1, 2, 3, 4, 5]
3 elemento = lista[0]           # Acesso direto
4 if 3 in {1, 2, 3}:           # Set lookup
5     pass
6
7 # O(n) - Linear
8 total = sum(lista)           # Uma passagem
9 if 3 in lista:               # Busca linear
10     pass
11
```

Exemplos de Complexidade

</> Python

```
1 # O(n^2) - Quadratica
2 for i in range(len(lista)):      # n iteracoes
3     for j in range(len(lista)):  # n iteracoes
4         print(lista[i] + lista[j]) # Total: n * n
5
6 # O(log n) - Logaritmica
7 busca_binaria(lista_ordenada, valor) # Divide por 2
8
9 # O(n log n) - Linearitmica
10 sorted(lista)                    # Timsort
11
```

Complexidade de Estruturas de Dados

Operações comuns:

Estrutura	Acesso	Busca	Inserção
Lista	$O(1)$	$O(n)$	$O(1)^*$
Tupla	$O(1)$	$O(n)$	N/A
Set	N/A	$O(1)$	$O(1)$
Dict	$O(1)$	$O(1)$	$O(1)$

*append no final; inserção no meio é $O(n)$

Implicações práticas:

- ▶ **Busca frequente?** Use set ou dict, não lista
- ▶ **Acesso por índice?** Use lista
- ▶ **Verificar pertencimento?** Set é $O(1)$, lista é $O(n)$
- ▶ **Ordem importa?** Lista; ordem não importa? Set

Analizando Complexidade de Código

</> Python

```
1 # Exemplo 1
2 def exemplo1(lista):
3     soma = 0                # O(1)
4     for num in lista:      # O(n)
5         soma += num        # O(1)
6     return soma
7 # Total: O(n)
8
9 # Exemplo 2
10 def exemplo2(lista):
11     for i in range(len(lista)): # O(n)
12         for j in range(len(lista)): # O(n)
13             print(lista[i] + lista[j])
14 # Total: O(n²)
15
```

Analisando Complexidade de Código

</> Python

```
1 # Exemplo 3 - Cuidado com operacoes escondidas!
2 def exemplo3(lista1, lista2):
3     comuns = []
4     for x in lista1:          # O(n)
5         if x in lista2:      # O(m) - busca linear!
6             comuns.append(x)
7     return comuns
8 # Total: O(n * m) - muito lento!
9
```

Algoritmos de Busca

Dois algoritmos fundamentais:

▶ Busca Linear

- ▶ Percorre elemento por elemento
- ▶ Funciona em qualquer lista
- ▶ Simples, mas pode ser lenta

▶ Busca Binária

- ▶ Divide lista ao meio a cada passo
- ▶ **Requer lista ordenada**
- ▶ Muito mais rápida para listas grandes

Quando usar cada uma?

- ▶ **Linear:** Listas pequenas, dados não ordenados, busca única
- ▶ **Binária:** Listas grandes ordenadas, buscas frequentes

Implementação dos Algoritmos

</> Python

```
1 # Busca Linear - O(n)
2 def busca_linear(lista, valor):
3     for i in range(len(lista)):
4         if lista[i] == valor:
5             return i
6     return -1
7
```

Implementação dos Algoritmos

</> Python

```
1 # Busca Binaria -  $O(\log n)$ 
2 def busca_binaria(lista, valor):
3     esquerda, direita = 0, len(lista) - 1
4
5     while esquerda <= direita:
6         meio = (esquerda + direita) // 2
7
8         if lista[meio] == valor:
9             return meio
10        elif valor < lista[meio]:
11            direita = meio - 1 # Buscar esquerda
12        else:
13            esquerda = meio + 1 # Buscar direita
14    return -1
15
```

Implementação dos Algoritmos

</> Python

```
1 # Exemplo
2 numeros = [1, 3, 5, 7, 9, 11, 13]
3 print(busca_binaria(numeros, 7))    # 3
4 print(busca_linear([5,2,8,1,9], 8)) # 2
5
```

Busca em Python

Python

```
1 # Operador 'in' - busca linear automatica
2 if 8 in [5, 2, 8, 1, 9]:
3     print("Encontrado!")
4
5 # Metodo index() - retorna posicao
6 numeros = [5, 2, 8, 1, 9]
7 try:
8     pos = numeros.index(8)
9     print(f"Posicao: {pos}") # 2
10 except ValueError:
11     print("Nao encontrado")
12
```

Busca em Python

</> Python

```
1 # Modulo bisect - busca binaria pronta
2 import bisect
3
4 ordenados = [1, 3, 5, 7, 9, 11, 13]
5 pos = bisect.bisect_left(ordenados, 7)
6 if pos < len(ordenados) and ordenados[pos] == 7:
7     print(f"Encontrado na posicao {pos}")
8
9 # count() - contar ocorrencias
10 lista = [1, 2, 3, 2, 4, 2, 5]
11 print(lista.count(2)) # 3
12
```

Ordenação em Python

Duas funções fundamentais:

- ▶ `sorted()`
 - ▶ Ordena elementos de **qualquer coleção iterável**
 - ▶ Retorna uma **cópia** ordenada da coleção original
 - ▶ Sintaxe: `sorted(iterable, key=None, reverse=False)`
- ▶ `sort()`
 - ▶ Ordena elementos de uma **lista**
 - ▶ **Altera a lista original**
 - ▶ Sintaxe: `lista.sort(key=None, reverse=False)`

Ordenação em Python

Duas formas principais:

</> Python

```
1 # 1. sorted() - retorna NOVA lista ordenada
2 numeros = [5, 2, 8, 1, 9]
3 ordenados = sorted(numeros)
4 print(ordenados) # [1, 2, 5, 8, 9]
5 print(numeros)   # [5, 2, 8, 1, 9] - original intacto
6
7 # 2. .sort() - ordena IN-PLACE (modifica original)
8 numeros = [5, 2, 8, 1, 9]
9 numeros.sort()
10 print(numeros)   # [1, 2, 5, 8, 9] - modificado!
11
```

Ordenação em Python

</> Python

```
1 # Ordem reversa
2 numeros = [5, 2, 8, 1, 9]
3 decrescente = sorted(numeros, reverse=True)
4 print(decrescente) # [9, 8, 5, 2, 1]
5
6 # Funciona com strings
7 palavras = ['python', 'java', 'c', 'javascript']
8 print(sorted(palavras)) # ['c', 'java', 'javascript', 'python']
9
```

Ordenação em Python

Nota Importante

Python usa **Timsort**: $O(n \log n)$ - muito eficiente!

Ordenação Personalizada: Parâmetro key

O parâmetro key define o critério de ordenação

</> Python

```
1 # Ordenar por comprimento
2 palavras = ['python', 'java', 'c', 'javascript']
3 por_tamanho = sorted(palavras, key=len)
4 print(por_tamanho) # ['c', 'java', 'python', 'javascript']
5
6 # Ordenar por valor absoluto
7 numeros = [-5, 2, -8, 1, 9, -3]
8 por_abs = sorted(numeros, key=abs)
9 print(por_abs) # [1, 2, -3, -5, -8, 9]
10
```

Ordenação Personalizada: Parâmetro key

O parâmetro key define o critério de ordenação

</> Python

```
1 # Ordenar tuplas por segundo elemento
2 alunos = [('Ana', 8.5), ('Bruno', 7.2), ('Carlos', 9.1)]
3 por_nota = sorted(alunos, key=lambda x: x[1])
4 print(por_nota) # [('Bruno', 7.2), ('Ana', 8.5), ('Carlos', 9.1)]
5
6 # Ordenar dicionarios
7 produtos = [
8     {'nome': 'Mouse', 'preco': 25},
9     {'nome': 'Teclado', 'preco': 150},
10    {'nome': 'Notebook', 'preco': 2500}
11 ]
12 por_preco = sorted(produtos, key=lambda x: x['preco'])
13 print(por_preco[0]['nome']) # 'Mouse'
```

Múltiplos Critérios e Módulo operator

Python

```
1 # Ordenar por multiplos criterios usando tuplas
2 alunos = [
3     ('Ana', 'SP', 8.5),
4     ('Bruno', 'RJ', 7.2),
5     ('Carlos', 'SP', 9.1),
6     ('Diana', 'RJ', 8.5)
7 ]
8
9 # Por cidade, depois por nota (decrecente)
10 ordenados = sorted(alunos, key=lambda x: (x[1], -x[2]))
11 # [('Diana', 'RJ', 8.5), ('Bruno', 'RJ', 7.2),
12 #  ('Carlos', 'SP', 9.1), ('Ana', 'SP', 8.5)]
```

Múltiplos Critérios e Módulo operator

</> Python

```
1 # Usando operator (mais legível)
2 from operator import itemgetter
3
4 vendas = [
5     {'produto': 'A', 'valor': 100, 'qtd': 5},
6     {'produto': 'B', 'valor': 100, 'qtd': 3}
7 ]
8
9 # Ordenar por valor, depois quantidade
10 ordenados = sorted(vendas, key=itemgetter('valor', 'qtd'))
11
```

Múltiplos Critérios e Módulo operator

Nota Importante

Python garante **ordenação estável**: elementos iguais mantêm ordem original

Técnicas de Otimização

Escolher estrutura de dados adequada

</> Python

```
1 # LENTO -  $O(n * m)$ 
2 def elementos_comuns_v1(lista1, lista2):
3     comuns = []
4     for x in lista1:           #  $O(n)$ 
5         if x in lista2:       #  $O(m)$  - busca linear!
6             comuns.append(x)
7     return comuns
8
```

Técnicas de Otimização

Escolher estrutura de dados adequada

</> Python

```
1 # RAPIDO -  $O(n + m)$ 
2 def elementos_comuns_v2(lista1, lista2):
3     set2 = set(lista2)          #  $O(m)$ 
4     comuns = []
5     for x in lista1:           #  $O(n)$ 
6         if x in set2:          #  $O(1)$  - busca em set!
7             comuns.append(x)
8     return comuns
9
```

Técnicas de Otimização

Escolher estrutura de dados adequada

</> Python

```
1 # MAIS LIMPO - usando set intersection
2 def elementos_comuns_v3(lista1, lista2):
3     return list(set(lista1) & set(lista2)) # O(n + m)
4
5 # Diferença: para n=m=10000
6 # v1: ~100.000.000 operacoes
7 # v2 e v3: ~20.000 operacoes (5000x mais rapido!)
8
```

Mais Exemplos de Otimização

</> Python

```
1 # Verificar duplicatas
2 # LENTO -  $O(n^2)$ 
3 def tem_duplicatas_v1(lista):
4     for i in range(len(lista)):
5         for j in range(i + 1, len(lista)):
6             if lista[i] == lista[j]:
7                 return True
8     return False
9
10 # RAPIDO -  $O(n)$ 
11 def tem_duplicatas_v2(lista):
12     return len(lista) != len(set(lista))
13
```

Mais Exemplos de Otimização

Python

```
1 # Contar frequências
2 # LENTO -  $O(n^2)$ 
3 def contar_v1(lista):
4     contagem = {}
5     for elemento in lista:
6         contagem[elemento] = lista.count(elemento) #  $O(n)$ !
7     return contagem
8
9 # RAPIDO -  $O(n)$ 
10 def contar_v2(lista):
11     contagem = {}
12     for elemento in lista:
13         contagem[elemento] = contagem.get(elemento, 0) + 1
14     return contagem
15
```

Medindo Performance

Módulo timeit e magic commands

</> Python

```
1 import timeit
2 # Comparar duas abordagens
3 def abordagem1():
4     lista = list(range(1000))
5     return [x * 2 for x in lista]
6 def abordagem2():
7     lista = list(range(1000))
8     return list(map(lambda x: x * 2, lista))
9
10 tempo1 = timeit.timeit(abordagem1, number=1000)
11 tempo2 = timeit.timeit(abordagem2, number=1000)
12 print(f"Abordagem 1: {tempo1:.4f}s")
13 print(f"Abordagem 2: {tempo2:.4f}s")
```

Medindo Performance

Módulo timeit e magic commands

</> Python

```
1 # No Google Colab/Jupyter
2 # %timeit sum(range(1000))
3 # %timeit [x for x in range(1000)]
4
5 # Comparar busca
6 # lista = list(range(10000))
7 # %timeit 5000 in lista           # O(n)
8 # %timeit 5000 in set(lista)     # O(1)
```

Quando e Como Otimizar?

Regras de ouro:

1. Escreva código claro primeiro

- ▶ "Premature optimization is the root of all evil"
- ▶ Otimize apenas se necessário

2. Meça antes de otimizar

- ▶ Use `timeit` / `%timeit`
- ▶ Identifique o gargalo real

3. Priorize algoritmos sobre micro-otimizações

- ▶ $O(n^2) \rightarrow O(n)$ vale muito mais
- ▶ Do que pequenas melhorias em loops

4. Considere o trade-off

- ▶ Legibilidade vs. performance
- ▶ Tempo vs. espaço de memória

Quando e Como Otimizar?

Nota Importante

Foque em otimizar o código crítico (20% que usa 80% do tempo)

Resumo: Pontos-Chave

Algoritmos:

- ▶ Busca linear: $O(n)$, qualquer lista
- ▶ Busca binária: $O(\log n)$, lista ordenada
- ▶ Use `sorted()` e parâmetro `key` para ordenação customizada

Complexidade:

- ▶ Big O descreve crescimento do tempo/espço
- ▶ $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2)$
- ▶ Escolha estrutura de dados pela complexidade das operações

Otimização:

- ▶ Use `set/dict` para buscas frequentes ($O(1)$ vs $O(n)$)
- ▶ Evite loops duplos quando possível
- ▶ Meça antes de otimizar
- ▶ Legibilidade também importa!

Guia Prático de Decisões

Quando usar cada estrutura:

Situação	Use
Buscar elemento frequentemente	Set ou Dict
Manter ordem e acessar por índice	Lista
Garantir unicidade	Set
Relacionar chave-valor	Dict
Dados imutáveis	Tupla
Buscas em lista ordenada	bisect + lista

Checklist de otimização:

1. Identifiquei o gargalo com medição?
2. Posso usar estrutura de dados mais eficiente?
3. Estou fazendo trabalho desnecessário?
4. Há operações $O(n)$ dentro de loops?
5. A otimização vale a perda de legibilidade?

Próximos Passos

O que vem a seguir:

- ▶ **NumPy:** Operações vetorizadas eficientes
- ▶ Arrays multidimensionais
- ▶ Operações matemáticas otimizadas
- ▶ Processamento de grandes volumes de dados

Como o que aprendemos se conecta:

- ▶ NumPy usa operações $O(n)$ internamente otimizadas em C
- ▶ Estruturas de dados apropriadas = código mais rápido
- ▶ Conceitos de complexidade aplicam-se a qualquer linguagem

💡 Nota Importante

Dica: Pratique analisar complexidade de código que você lê/escreve!

Live Coding

Vamos codificar juntos:

1. Abra o Google Colab
2. Utilize o notebook disponibilizado no Moodle
3. Siga os passos da demonstração

Exercício Prático

Tempo: 60 minutos

Entrega: via Moodle (notebook)

Tarefa:

1. (o enunciado será atualizado durante a aula)

Obrigado