Estruturas de Dados em Python

Sumário

1. Listas (Lists)

Definição

Listas são estruturas de dados ordenadas e mutáveis que podem conter elementos de qualquer tipo. São representadas por colchetes `[]` e os elementos são separados por vírgulas.

Elas são usadas quando os dados precisam ser alterados durante a execução do programa.

Sintaxe

```
# Criação de listas
numeros = [1, 2, 3, 4, 5]
misturada = [1, "texto", 3.14, True]
vazia = []
aninhada = [[1, 2], [3, 4], [5, 6]]
```

Métodos mais usados

append(elemento) - Adiciona um elemento ao final da lista

```
frutas = ["maçã", "banana"]
frutas.append("laranja")
# Resultado: ["maçã", "banana", "laranja"]
```

insert(indice, elemento) - Insere um elemento em uma posição específica

```
numeros = [1, 2, 4]
numeros.insert(2, 3)
# Resultado: [1, 2, 3, 4]
```

remove(elemento) - Remove a primeira ocorrência do elemento

```
letras = ["a", "b", "c", "b"]
letras.remove("b")
# Resultado: ["a", "c", "b"]
```

pop(indice) - Remove e retorna o elemento no índice especificado (último por padrão)

```
numeros = [1, 2, 3, 4]
ultimo = numeros.pop()  # ultimo = 4
meio = numeros.pop(1)   # meio = 2
# Resultado: [1, 3]
```

extend(iteravel) - Adiciona todos os elementos de um iterável ao final da lista

```
lista1 = [1, 2, 3]
lista2 = [4, 5, 6]
lista1.extend(lista2)
# Resultado: [1, 2, 3, 4, 5, 6]
```

index(elemento) - Retorna o índice da primeira ocorrência do elemento

```
letras = ["a", "b", "c", "b"]
indice = letras.index("b")  # indice = 1
```

count(elemento) - Conta quantas vezes o elemento aparece na lista

```
numeros = [1, 2, 2, 3, 2, 4]
quantidade = numeros.count(2)  # quantidade = 3
```

sort() - Ordena a lista in-place

```
numeros = [3, 1, 4, 1, 5]
numeros.sort()
# Resultado: [1, 1, 3, 4, 5]

# Ordem decrescente
numeros.sort(reverse=True)
# Resultado: [5, 4, 3, 1, 1]
```

reverse() - Inverte a ordem dos elementos

```
numeros = [1, 2, 3, 4]
numeros.reverse()
# Resultado: [4, 3, 2, 1]
```

clear() - Remove todos os elementos da lista

```
numeros = [1, 2, 3]
numeros.clear()
# Resultado: []
```

Exemplos Práticos

```
# Gerenciamento de tarefas
tarefas = ["estudar Python", "fazer exercícios"]
tarefas.append("revisar conteúdo")
tarefas.insert(1, "assistir aula")
concluida = tarefas.pop(0)

# Processamento de dados
temperaturas = [23.5, 25.1, 22.8, 26.3, 24.0]
media = sum(temperaturas) / len(temperaturas)
temperaturas.sort()
mediana = temperaturas[len(temperaturas) // 2]
```

Exercícios

1. Crie uma lista com os números de 1 a 10 e remova todos os números pares.
2. Dada uma lista de palavras, crie uma nova lista contendo apenas as palavras com mais de 5 caracteres.
3. Implemente uma função que recebe uma lista e retorna uma nova lista sem elementos duplicados, mantendo a ordem original.
4. Crie uma função que encontra o segundo maior elemento em uma lista de números.

Desafios

1. Implemente uma função que rotaciona uma lista k posições para a direita.
2. Crie uma função que mescla duas listas ordenadas em uma única lista ordenada sem usar sort().

3. Implemente o algoritmo de ordenação Bubble Sort usando apenas operações de lista.

Aplicações

- Armazenamento de histórico de navegação
- Implementação de pilhas e filas
- Processamento de sequências de dados
- Gerenciamento de inventários
- Representação de matrizes e grafos

---

2. Tuplas (Tuples)

Definição

Tuplas são estruturas de dados ordenadas e imutáveis, representadas por parênteses `()`. Uma vez criadas, seus elementos não podem ser modificados.

Tuplas são usadas quando queremos garantir que os dados não sejam alterados acidentalmente. São mais eficientes em memória que listas e podem ser usadas como chaves de dicionários.

Sintaxe Básica

```
# Criação de tuplas
coordenadas = (10, 20)
pessoa = ("João", 25, "Engenheiro")
vazia = ()
um_elemento = (42,)  # Vírgula necessária
sem_parenteses = 1, 2, 3  # Também é uma tupla
```

Principais Operações

Acesso a elementos

```
dados = ("Python", 3.11, True)
linguagem = dados[0]  # "Python"
versao = dados[1]     # 3.11
```

count(elemento) - Conta ocorrências de um elemento

```
numeros = (1, 2, 2, 3, 2, 4)
quantidade = numeros.count(2)  # quantidade = 3
```

index(elemento) - Retorna o índice da primeira ocorrência

```
letras = ("a", "b", "c", "b")
indice = letras.index("b")  # indice = 1
```

Desempacotamento

```
coordenadas = (10, 20, 30)
x, y, z = coordenadas

# Desempacotamento com asterisco
primeiro, *resto, ultimo = (1, 2, 3, 4, 5)
# primeiro = 1, resto = [2, 3, 4], ultimo = 5
```

Exemplos Práticos

```
# Retorno múltiplo de funções
def calcular_estatisticas(numeros):
    return min(numeros), max(numeros), sum(numeros) / len(numeros)

minimo, maximo, media = calcular_estatisticas([1, 2, 3, 4, 5])

# Coordenadas geográficas
localizacao = (-23.5505, -46.6333)  # São Paulo
latitude, longitude = localizacao

# Configurações imutáveis
CONFIGURACAO = ("localhost", 8080, "admin")
```

Exercícios

1. Crie uma função que recebe uma lista e retorna uma tupla com o menor, o maior e a média dos valores.
2. Dada uma tupla de tuplas representando coordenadas, calcule a distância euclidiana entre cada par de pontos.
3. Implemente uma função que troca os valores de duas variáveis usando tuplas.
4. Crie uma tupla com informações de um livro (título, autor, ano) e desempacote seus valores.

Desafios

1. Implemente uma função que recebe uma tupla e retorna uma nova tupla com os elementos em ordem reversa sem converter para lista.
2. Crie uma função que conta quantas tuplas em uma lista de tuplas contêm um elemento específico.

Aplicações

- Retorno de múltiplos valores de funções
- Chaves de dicionários quando precisam ser compostas

- Coordenadas e pontos no espaço
- Registros de banco de dados
- Configurações que não devem ser alteradas

---

3. Conjuntos (Sets)

Definição

Conjuntos são coleções não ordenadas de elementos únicos, representados por chaves `{}` ou pela função `set()`. Não permitem elementos duplicados e não mantêm ordem.

Sets são ideais para eliminar duplicatas, testar pertinência de elementos e realizar operações matemáticas de conjuntos como união, interseção e diferença de forma eficiente.

Sintaxe Básica

```
# Criação de sets
numeros = {1, 2, 3, 4, 5}
letras = set("abracadabra")  # {'a', 'b', 'r', 'c', 'd'}
vazio = set()  # {} cria um dicionário vazio, não um set
```

Principais Métodos

add(elemento) - Adiciona um elemento ao conjunto

```
frutas = {"maçã", "banana"}
frutas.add("laranja")
# Resultado: {"maçã", "banana", "laranja"}
```

remove(elemento) - Remove um elemento (gera erro se não existir)

```
numeros = {1, 2, 3, 4}
numeros.remove(3)
# Resultado: {1, 2, 4}
```

discard(elemento) - Remove um elemento (não gera erro se não existir)

```
numeros = {1, 2, 3, 4}
numeros.discard(5)  # Não gera erro
```

pop() - Remove e retorna um elemento arbitrário

```
cores = {"vermelho", "verde", "azul"}
cor_removida = cores.pop()
```

clear() - Remove todos os elementos

```
numeros = {1, 2, 3}
numeros.clear()
# Resultado: set()
```

Operações de Conjunto

union() ou | - União de conjuntos

```
a = {1, 2, 3}
b = {3, 4, 5}
uniao = a.union(b)  # ou a | b
# Resultado: {1, 2, 3, 4, 5}
```

intersection() ou & - Interseção de conjuntos

```
a = {1, 2, 3, 4}
b = {3, 4, 5, 6}
intersecao = a.intersection(b)  # ou a & b
# Resultado: {3, 4}
```

difference() ou - - Diferença entre conjuntos

```
a = {1, 2, 3, 4}
b = {3, 4, 5, 6}
diferenca = a.difference(b)  # ou a - b
# Resultado: {1, 2}
```

symmetric_difference() ou ^ - Diferença simétrica

```
a = {1, 2, 3, 4}
b = {3, 4, 5, 6}
simetrica = a.symmetric_difference(b)  # ou a ^ b
# Resultado: {1, 2, 5, 6}
```

issubset() ou <= - Verifica se é subconjunto

```
a = {1, 2}
b = {1, 2, 3, 4}
resultado = a.issubset(b)  # ou a <= b
# Resultado: True
```

issuperset() ou >= - Verifica se é superconjunto

```
a = {1, 2, 3, 4}
b = {1, 2}
resultado = a.issuperset(b)  # ou a >= b
# Resultado: True
```

isdisjoint() - Verifica se os conjuntos são disjuntos

```
a = {1, 2, 3}
b = {4, 5, 6}
resultado = a.isdisjoint(b)
# Resultado: True
```

Exemplos Práticos

```
# Remover duplicatas
numeros_duplicados = [1, 2, 2, 3, 3, 3, 4, 5, 5]
numeros_unicos = list(set(numeros_duplicados))

# Encontrar elementos comuns
curso_python = {"Ana", "Bruno", "Carlos", "Diana"}
curso_java = {"Bruno", "Diana", "Eduardo", "Fernanda"}
ambos_cursos = curso_python & curso_java

# Verificar pertinência eficientemente
vogais = set("aeiou")
if "a" in vogais:
    print("É uma vogal")

# Análise de dados
vendas_janeiro = {"produto_a", "produto_b", "produto_c"}
vendas_fevereiro = {"produto_b", "produto_c", "produto_d"}
novos_produtos = vendas_fevereiro - vendas_janeiro
```

Exercícios

1. Dadas duas listas de números, encontre os elementos que aparecem em ambas.
2. Crie uma função que recebe uma string e retorna o número de caracteres únicos.
3. Implemente uma função que verifica se duas listas têm elementos em comum.
4. Dada uma lista de palavras, encontre todas as letras que aparecem em todas as palavras.

Desafios

1. Implemente uma função que encontra todos os subconjuntos possíveis de um conjunto (power set).
2. Crie um sistema de recomendação simples que sugere itens baseado em interesses comuns entre usuários.
3. Desenvolva uma função que identifica palíndromos ignorando caracteres repetidos.

Aplicações

- Remoção de duplicatas em grandes volumes de dados
- Análise de interseções em bancos de dados
- Implementação de filtros de Bloom
- Verificação de permissões e roles

---

4. Dicionários (Dictionaries)

Definição

Dicionários são estruturas de dados que armazenam pares chave-valor. São representados por chaves `{}` e permitem acesso rápido aos valores através de suas chaves únicas.

Dicionários são fundamentais quando precisamos associar informações e acessá-las de forma eficiente por meio de identificadores únicos, sem depender de índices numéricos.

Sintaxe Básica

```
# Criação de dicionários
pessoa = {"nome": "João", "idade": 30, "cidade": "São Paulo"}
vazio = {}
usando_dict = dict(nome="Maria", idade=25)
a_partir_de_tuplas = dict([("a", 1), ("b", 2)])
```

Principais Métodos

get(chave, padrao) - Retorna o valor da chave (ou valor padrão se não existir)

```
pessoa = {"nome": "João", "idade": 30}
nome = pessoa.get("nome")  # "João"
profissao = pessoa.get("profissao", "Não informado")  # "Não informado"
```

keys() - Retorna todas as chaves

```
dados = {"a": 1, "b": 2, "c": 3}
chaves = dados.keys()  # dict_keys(['a', 'b', 'c'])
lista_chaves = list(dados.keys())
```

values() - Retorna todos os valores

```
dados = {"a": 1, "b": 2, "c": 3}
valores = dados.values()  # dict_values([1, 2, 3])
```

items() - Retorna pares chave-valor como tuplas

```
dados = {"a": 1, "b": 2, "c": 3}
itens = dados.items()  # dict_items([('a', 1), ('b', 2), ('c', 3)])
```

update(outro_dict) - Atualiza o dicionário com pares de outro dicionário

```
config = {"tema": "escuro", "idioma": "pt"}
novos_valores = {"idioma": "en", "fonte": "12px"}
config.update(novos_valores)
# Resultado: {"tema": "escuro", "idioma": "en", "fonte": "12px"}
```

pop(chave, padrao) - Remove e retorna o valor da chave

```
dados = {"a": 1, "b": 2, "c": 3}
valor = dados.pop("b")  # valor = 2
# Resultado: {"a": 1, "c": 3}
```

popitem() - Remove e retorna o último par chave-valor

```
dados = {"a": 1, "b": 2, "c": 3}
item = dados.popitem()  # item = ('c', 3)
```

setdefault(chave, padrao) - Retorna o valor da chave ou define um valor padrão

```
contador = {}
contador.setdefault("a", 0)
contador["a"] += 1
```

clear() - Remove todos os itens

```
dados = {"a": 1, "b": 2}
dados.clear()
# Resultado: {}
```

Operações Avançadas

Acesso e modificação

```
aluno = {"nome": "Ana", "nota": 8.5}

# Acesso
nome = aluno["nome"]
nota = aluno.get("nota")

# Modificação
aluno["nota"] = 9.0

# Inserção
aluno["curso"] = "Python"

# Remoção
del aluno["curso"]
```

Iteração

```
notas = {"Ana": 9.0, "Bruno": 8.5, "Carlos": 7.5}

# Iterar sobre chaves
for nome in notas:
    print(nome)

# Iterar sobre valores
for nota in notas.values():
    print(nota)

# Iterar sobre pares chave-valor
for nome, nota in notas.items():
    print(f"{nome}: {nota}")
```

Exemplos Práticos

```
# Contador de palavras
texto = "python é uma linguagem python"
contador = {}
for palavra in texto.split():
    contador[palavra] = contador.get(palavra, 0) + 1

# Agrupamento de dados
alunos = [
    {"nome": "Ana", "turma": "A"},
    {"nome": "Bruno", "turma": "B"},
    {"nome": "Carlos", "turma": "A"}
]

por_turma = {}
for aluno in alunos:
    turma = aluno["turma"]
    if turma not in por_turma:
        por_turma[turma] = []
    por_turma[turma].append(aluno["nome"])

# Cache de resultados
cache_fibonacci = {0: 0, 1: 1}

def fibonacci(n):
    if n not in cache_fibonacci:
        cache_fibonacci[n] = fibonacci(n-1) + fibonacci(n-2)
    return cache_fibonacci[n]

# Configuração de aplicação
config = {
    "banco": {
        "host": "localhost",
        "porta": 5432,
        "usuario": "admin"
    },
    "cache": {
        "habilitado": True,
        "ttl": 3600
```

```
    }
}
```

Exercícios

1. Crie um dicionário que conte a frequência de cada letra em uma string.
2. Implemente uma função que inverte um dicionário (chaves viram valores e vice-versa).
3. Dado um dicionário de produtos e preços, crie uma função que aplica um desconto de 10%.
4. Escreva uma função que mescla dois dicionários, somando os valores de chaves comuns.

Desafios

1. Implemente uma estrutura de dados para um sistema de biblioteca que rastreie livros, autores e disponibilidade.
2. Crie um programa que receba nomes de alunos e suas notas e armazene tudo em um dicionário.
3. Você deve criar um sistema simples de inventário usando dicionários. Funcionalidades obrigatórias: adicionar produto, remover produto pelo ID, atualizar quantidade, calcular o valor total do estoque, retornar o produto mais caro.

---

5. Indexação e Fatiamento (Indexing and Slicing)

Definição

Indexação é o acesso a elementos individuais de uma sequência usando sua posição. Fatiamento é a extração de subsequências usando intervalos de índices.

Indexação e fatiamento permitem manipular partes específicas de sequências de forma simples e eficiente, facilitando operações comuns em processamento de dados.

Indexação

Índices positivos - Começam em 0 (primeiro elemento)

```
texto = "Python"
primeira = texto[0]   # 'P'
terceira = texto[2]   # 't'
```

Índices negativos - Começam em -1 (último elemento)

```
texto = "Python"
ultima = texto[-1]    # 'n'
penultima = texto[-2] # 'o'
```

Indexação em estruturas aninhadas

```
matriz = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
elemento = matriz[1][2]  # 6

dicionario = {"dados": [10, 20, 30]}
valor = dicionario["dados"][1]  # 20
```

Fatiamento (Slicing)

Sintaxe básica: `sequencia[inicio:fim:passo]`

- `inicio`: índice inicial (inclusivo)
- `fim`: índice final (exclusivo)
- `passo`: incremento entre elementos

```
numeros = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# Fatiamento básico
primeiros_cinco = numeros[0:5]   # [0, 1, 2, 3, 4]
meio = numeros[3:7]              # [3, 4, 5, 6]
ultimos_tres = numeros[-3:]      # [7, 8, 9]

# Omitindo parâmetros
do_inicio = numeros[:5]          # [0, 1, 2, 3, 4]
ate_fim = numeros[5:]            # [5, 6, 7, 8, 9]
copia = numeros[:]               # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# Usando passo
pares = numeros[::2]             # [0, 2, 4, 6, 8]
impares = numeros[1::2]          # [1, 3, 5, 7, 9]
reverso = numeros[::-1]          # [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Exemplos Práticos

```
# Extração de partes de strings
email = "usuario@exemplo.com"
usuario = email[:email.index("@")]
dominio = email[email.index("@")+1:]

# Manipulação de listas
dados = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
primeira_metade = dados[:len(dados)//2]
segunda_metade = dados[len(dados)//2:]

# Processamento de texto
texto = "   Python é incrível   "
sem_espacos = texto.strip()
primeiras_palavras = texto.split()[:3]

# Reversão de sequências
palavra = "Python"
invertida = palavra[::-1]  # "nohtyP"

numeros = [1, 2, 3, 4, 5]
invertidos = numeros[::-1]  # [5, 4, 3, 2, 1]

# Substituição usando fatiamento
lista = [1, 2, 3, 4, 5]
lista[1:4] = [20, 30, 40]  # [1, 20, 30, 40, 5]
```

Casos Especiais

```
# Índices fora do intervalo em fatiamento não geram erro
numeros = [1, 2, 3]
fatia = numeros[1:100]  # [2, 3]

# Passo negativo com início e fim
numeros = [0, 1, 2, 3, 4, 5]
reverso_parcial = numeros[5:2:-1]  # [5, 4, 3]

# Remoção usando fatiamento
lista = [1, 2, 3, 4, 5]
del lista[1:3]  # [1, 4, 5]
```

Exercícios

1. Dada uma string, extraia os três primeiros e três últimos caracteres.
2. Crie uma função que retorna uma lista sem o primeiro e o último elemento.
3. Implemente uma função que divide uma lista em duas metades.
4. Escreva uma função que extrai todos os elementos em posições pares de uma lista.

Desafios

1. Implemente uma função que rotaciona uma string k posições usando apenas fatiamento.
2. Crie uma função que verifica se uma string é palíndromo usando fatiamento.
3. Desenvolva uma função que intercala duas listas usando fatiamento e concatenação.

---

6. Mutabilidade vs Imutabilidade

Definição

Mutabilidade refere-se à capacidade de um objeto ser modificado após sua criação. **Imutabilidade** significa que o objeto não pode ser alterado depois de criado.

Tipos Mutáveis e Imutáveis

Imutáveis:
- int, float, complex
- str
- tuple
- frozenset
- bool

Mutáveis:
- list
- dict
- set
- bytearray

Comportamento de Tipos Imutáveis

```
# Strings são imutáveis
texto = "Python"
# texto[0] = "J"  # Erro! TypeError

# Operações criam novos objetos
texto = texto + " 3"  # Novo objeto criado
texto = texto.upper() # Novo objeto criado

# Tuplas são imutáveis
coordenadas = (10, 20)
# coordenadas[0] = 15  # Erro! TypeError

# Números são imutáveis
x = 5
y = x
x = x + 1  # x aponta para novo objeto, y permanece 5
```

Comportamento de Tipos Mutáveis

```
# Listas são mutáveis
numeros = [1, 2, 3]
numeros[0] = 10  # Modifica o objeto existente
numeros.append(4)  # Modifica o objeto existente

# Dicionários são mutáveis
pessoa = {"nome": "João"}
pessoa["idade"] = 30  # Modifica o objeto existente

# Sets são mutáveis
frutas = {"maçã", "banana"}
frutas.add("laranja")  # Modifica o objeto existente
```

Implicações de Mutabilidade

Atribuição e referências

```
# Com objetos mutáveis
lista_a = [1, 2, 3]
lista_b = lista_a  # Ambas apontam para o mesmo objeto
lista_b.append(4)
print(lista_a)  # [1, 2, 3, 4] - modificado!

# Com objetos imutáveis
x = 10
y = x  # Copia o valor
y = 20
print(x)  # 10 - não modificado

# Criando cópias independentes
lista_original = [1, 2, 3]
lista_copia = lista_original.copy()  # ou lista_original[:]
lista_copia.append(4)
print(lista_original)  # [1, 2, 3] - não modificado
```

Argumentos de função

```
# Passagem de mutáveis
def adicionar_elemento(lista, elemento):
    lista.append(elemento)  # Modifica a lista original

numeros = [1, 2, 3]
adicionar_elemento(numeros, 4)
print(numeros)  # [1, 2, 3, 4]

# Passagem de imutáveis
def incrementar(x):
    x = x + 1  # Cria novo objeto local
    return x

valor = 10
resultado = incrementar(valor)
print(valor)  # 10 - não modificado
```

Valor padrão mutável - armadilha comum

```
# ERRADO - não faça isso!
def adicionar_a_lista(elemento, lista=[]):
    lista.append(elemento)
    return lista

print(adicionar_a_lista(1))  # [1]
print(adicionar_a_lista(2))  # [1, 2] - inesperado!

# CORRETO
def adicionar_a_lista(elemento, lista=None):
    if lista is None:
        lista = []
    lista.append(elemento)
    return lista
```

Cópia Profunda vs Cópia Rasa

```
import copy

# Cópia rasa - copia o primeiro nível
lista_original = [[1, 2], [3, 4]]
lista_rasa = lista_original.copy()
lista_rasa[0][0] = 100
print(lista_original)  # [[100, 2], [3, 4]] - modificado!

# Cópia profunda - copia recursivamente
lista_original = [[1, 2], [3, 4]]
lista_profunda = copy.deepcopy(lista_original)
lista_profunda[0][0] = 100
print(lista_original)  # [[1, 2], [3, 4]] - não modificado
```

Exemplos Práticos

```
# Uso de tuplas para garantir integridade
CONFIGURACAO = ("localhost", 8080, True)
# Garante que a configuração não será modificada

# Quando usar mutáveis vs imutáveis
def processar_dados(dados_originais):
    # Cria cópia para não modificar original
    dados_processados = dados_originais.copy()
    dados_processados.sort()
    return dados_processados

# Tuplas como chaves de dicionário
cache = {}
ponto = (10, 20)
cache[ponto] = "valor calculado"  # OK - tupla é imutável

# lista = [10, 20]
# cache[lista] = "valor"  # Erro! Lista não pode ser chave
```

Exercícios

1. Explique por que o código `a = [1,2,3]; b = a; b.append(4)` modifica `a`.
2. Crie uma função que recebe uma lista e retorna uma versão modificada sem alterar a original.
3. Demonstre a diferença entre `copy()` e `deepcopy()` com listas aninhadas.
4. Identifique o problema no seguinte código e corrija-o:
   `python def adicionar_tarefa(tarefa, lista_tarefas=[]): lista_tarefas.append(tarefa) return lista_tarefas`

Desafios

1. Implemente uma classe `ImmutableList` que se comporta como uma lista mas não permite modificações.
2. Crie uma função que converte recursivamente todas as listas em tuplas em uma estrutura de dados aninhada.
3. Desenvolva um sistema que detecta quando objetos mutáveis são modificados inesperadamente.

---

7. Compreensões de Lista (List Comprehensions)

Definição

List comprehensions são construções sintáticas que permitem criar listas de forma concisa aplicando expressões a elementos de iteráveis, opcionalmente filtrando-os.

Compreensões de lista oferecem uma forma mais legível, concisa e frequentemente mais rápida de criar listas em comparação com loops tradicionais.

Sintaxe Básica

```
# Sintaxe geral
nova_lista = [expressao for item in iteravel]

# Com condição
nova_lista = [expressao for item in iteravel if condicao]

# Equivalente com loop tradicional
nova_lista = []
for item in iteravel:
    if condicao:
        nova_lista.append(expressao)
```

Exemplos Fundamentais

```
# Quadrados dos números de 0 a 9
quadrados = [x**2 for x in range(10)]
# Resultado: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

# Números pares de 0 a 20
pares = [x for x in range(21) if x % 2 == 0]
# Resultado: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

# Converter strings para maiúsculas
palavras = ["python", "java", "javascript"]
maiusculas = [palavra.upper() for palavra in palavras]
# Resultado: ['PYTHON', 'JAVA', 'JAVASCRIPT']

# Extrair comprimentos de strings
nomes = ["Ana", "Bruno", "Carlos"]
tamanhos = [len(nome) for nome in nomes]
# Resultado: [3, 5, 6]
```

Compreensões Aninhadas

```
# Criar matriz 3x3
matriz = [[i*3 + j for j in range(3)] for i in range(3)]
# Resultado: [[0, 1, 2], [3, 4, 5], [6, 7, 8]]

# Achatar lista de listas
matriz = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
achatada = [elemento for linha in matriz for elemento in linha]
# Resultado: [1, 2, 3, 4, 5, 6, 7, 8, 9]

# Combinações de duas listas
cores = ["vermelho", "verde", "azul"]
tamanhos = ["P", "M", "G"]
combinacoes = [f"{cor}-{tamanho}" for cor in cores for tamanho in tamanhos]
# Resultado: ['vermelho-P', 'vermelho-M', 'vermelho-G', 'verde-P', ...]
```

Compreensões com Múltiplas Condições

```
# Números divisíveis por 2 e 3
numeros = [x for x in range(50) if x % 2 == 0 if x % 3 == 0]
# Resultado: [0, 6, 12, 18, 24, 30, 36, 42, 48]

# Expressão condicional (if-else)
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
classificacao = ["par" if x % 2 == 0 else "ímpar" for x in numeros]
# Resultado: ['ímpar', 'par', 'ímpar', 'par', ...]
```

Exemplos Práticos

```
# Filtrar e transformar dados
precos = [10.50, 25.00, 15.75, 30.00, 5.25]
precos_com_desconto = [preco * 0.9 for preco in precos if preco > 10]
# Resultado: [9.45, 22.5, 14.175, 27.0]

# Processar strings
texto = "Python é uma linguagem poderosa"
vogais = [char for char in texto.lower() if char in "aeiou"]
# Resultado: ['o', 'e', 'u', 'a', 'i', 'u', 'a', 'e', 'o', 'e', 'o', 'a']

# Extrair dados de dicionários
alunos = [
    {"nome": "Ana", "nota": 8.5},
    {"nome": "Bruno", "nota": 6.0},
    {"nome": "Carlos", "nota": 9.0}
]
aprovados = [aluno["nome"] for aluno in alunos if aluno["nota"] >= 7.0]
# Resultado: ['Ana', 'Carlos']

# Criar pares (índice, valor)
frutas = ["maçã", "banana", "laranja"]
indexadas = [(i, fruta) for i, fruta in enumerate(frutas)]
# Resultado: [(0, 'maçã'), (1, 'banana'), (2, 'laranja')]

# Transpor matriz
matriz = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
transposta = [[linha[i] for linha in matriz] for i in range(len(matriz[0]))]
# Resultado: [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

Comparação com Loops Tradicionais

```
# Usando loop tradicional
quadrados = []
for x in range(10):
    if x % 2 == 0:
        quadrados.append(x**2)

# Usando list comprehension (mais conciso)
quadrados = [x**2 for x in range(10) if x % 2 == 0]

# Mais exemplos de conversão
# Loop tradicional
palavras_longas = []
for palavra in ["a", "bb", "ccc", "dddd"]:
    if len(palavra) > 2:
        palavras_longas.append(palavra.upper())

# List comprehension
palavras_longas = [p.upper() for p in ["a", "bb", "ccc", "dddd"] if len(p) > 2]
```

Exercícios

1. Crie uma list comprehension que gera os cubos dos números ímpares de 1 a 20.
2. Dada uma lista de strings, crie uma nova lista com apenas as strings que contêm a letra 'a'.
3. Crie uma list comprehension que gera todos os números de 1 a 100 que são quadrados perfeitos.
4. Dada uma lista de números, crie uma lista com "positivo", "negativo" ou "zero" para cada número.

Desafios

1. Implemente o crivo de Eratóstenes usando list comprehension para encontrar números primos até 100.
2. Crie uma list comprehension que gera todas as permutações de duas listas.

3. Desenvolva uma solução para o problema FizzBuzz usando apenas list comprehension.

Aplicações

- Processamento e limpeza de dados
- Transformações em pipelines de dados
- Filtragem de resultados de consultas
- Operações matemáticas em vetores

---

8. Compreensões de Dicionário (Dict Comprehensions)

Definição

Dict comprehensions permitem criar dicionários de forma concisa usando uma sintaxe similar às list comprehensions, gerando pares chave-valor.

Sintaxe Básica

```
# Sintaxe geral
novo_dict = {chave: valor for item in iteravel}

# Com condição
novo_dict = {chave: valor for item in iteravel if condicao}
```

Exemplos Fundamentais

```
# Quadrados como dicionário
quadrados = {x: x**2 for x in range(6)}
# Resultado: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

# Inverter chaves e valores
original = {"a": 1, "b": 2, "c": 3}
invertido = {valor: chave for chave, valor in original.items()}
# Resultado: {1: 'a', 2: 'b', 3: 'c'}

# Criar dicionário a partir de listas
nomes = ["Ana", "Bruno", "Carlos"]
idades = [25, 30, 28]
pessoas = {nome: idade for nome, idade in zip(nomes, idades)}
# Resultado: {'Ana': 25, 'Bruno': 30, 'Carlos': 28}

# Filtrar dicionário
precos = {"maçã": 3.50, "banana": 2.00, "laranja": 4.00, "uva": 8.00}
caros = {fruta: preco for fruta, preco in precos.items() if preco > 3.00}
# Resultado: {'maçã': 3.5, 'laranja': 4.0, 'uva': 8.0}
```

Transformações de Dicionários

```
# Aplicar desconto
precos = {"item1": 100, "item2": 200, "item3": 150}
com_desconto = {item: preco * 0.9 for item, preco in precos.items()}
# Resultado: {'item1': 90.0, 'item2': 180.0, 'item3': 135.0}

# Converter valores
temperaturas_c = {"manhã": 20, "tarde": 28, "noite": 22}
temperaturas_f = {momento: (temp * 9/5) + 32 for momento, temp in temperaturas_c.items()}
# Resultado: {'manhã': 68.0, 'tarde': 82.4, 'noite': 71.6}

# Transformar chaves
dados = {"nome_completo": "João Silva", "idade_anos": 30}
formatado = {chave.replace("_", " ").title(): valor for chave, valor in dados.items()}
# Resultado: {'Nome Completo': 'João Silva', 'Idade Anos': 30}
```

Exemplos Práticos

```
# Contar frequência de caracteres
texto = "hello world"
frequencia = {char: texto.count(char) for char in set(texto) if char != " "}
# Resultado: {'h': 1, 'e': 1, 'l': 3, 'o': 2, 'w': 1, 'r': 1, 'd': 1}

# Agrupar por critério
palavras = ["python", "java", "ruby", "rust", "go"]
por_tamanho = {palavra: len(palavra) for palavra in palavras}
# Resultado: {'python': 6, 'java': 4, 'ruby': 4, 'rust': 4, 'go': 2}

# Criar índice
alunos = ["Ana", "Bruno", "Carlos", "Diana"]
indice = {aluno: i for i, aluno in enumerate(alunos)}
# Resultado: {'Ana': 0, 'Bruno': 1, 'Carlos': 2, 'Diana': 3}

# Extrair campos específicos
dados = [
    {"id": 1, "nome": "Ana", "ativo": True},
    {"id": 2, "nome": "Bruno", "ativo": False},
    {"id": 3, "nome": "Carlos", "ativo": True}
]
ativos = {item["id"]: item["nome"] for item in dados if item["ativo"]}
# Resultado: {1: 'Ana', 3: 'Carlos'}

# Normalizar dados
notas = {"Ana": "8,5", "Bruno": "7,0", "Carlos": "9,5"}
notas_float = {aluno: float(nota.replace(",", ".")) for aluno, nota in notas.items()}
# Resultado: {'Ana': 8.5, 'Bruno': 7.0, 'Carlos': 9.5}
```

Compreensões Condicionais

```
# Classificação condicional
numeros = {"a": 10, "b": 15, "c": 8, "d": 20}
classificacao = {
    chave: ("alto" if valor > 12 else "baixo")
    for chave, valor in numeros.items()
}
# Resultado: {'a': 'baixo', 'b': 'alto', 'c': 'baixo', 'd': 'alto'}

# Filtro com múltiplas condições
produtos = {
    "produto1": {"preco": 50, "estoque": 10},
    "produto2": {"preco": 150, "estoque": 5},
    "produto3": {"preco": 80, "estoque": 0}
}
disponiveis = {
    nome: dados["preco"]
    for nome, dados in produtos.items()
    if dados["estoque"] > 0 and dados["preco"] < 100
}
# Resultado: {'produto1': 50}
```

Exercícios

1. Crie um dicionário que mapeia números de 1 a 10 para seus valores ao cubo.
2. Dado um dicionário de produtos e preços, crie um novo dicionário apenas com produtos acima de R$ 50.
3. Inverta um dicionário que mapeia nomes para departamentos.
4. Crie um dicionário que conta quantas vogais cada palavra de uma lista contém.

Desafios

1. Dada uma lista de palavras, crie um dicionário que mapeie cada palavra para o número de vezes que ela aparece na lista, usando apenas dict comprehension.

2. Dado um dicionário de alunos e notas, crie um novo dicionário apenas com os alunos aprovados (nota ≥ 7) e suas respectivas notas, usando dict comprehension.
3. Dado um dicionário onde as chaves são strings e os valores são números, crie um novo dicionário invertendo chaves e valores apenas para os itens cujo valor seja par, usando dict comprehension.
4. Desenvolva uma solução que mescla múltiplos dicionários somando valores de chaves duplicadas.

Aplicações

- Transformação de dados JSON
- Criação de índices e mapas reversos
- Filtragem e limpeza de configurações
- Agregação de dados por categorias
- Normalização de datasets

---

9. Compreensões de Conjunto (Set Comprehensions)

Definição

Set comprehensions permitem criar conjuntos de forma concisa, automaticamente eliminando duplicatas e sem manter ordem.

Sintaxe Básica

```
# Sintaxe geral
novo_set = {expressao for item in iteravel}

# Com condição
novo_set = {expressao for item in iteravel if condicao}
```

Exemplos Fundamentais

```
# Quadrados únicos
quadrados = {x**2 for x in range(-5, 6)}
# Resultado: {0, 1, 4, 9, 16, 25}

# Primeiras letras únicas
palavras = ["python", "java", "javascript", "perl", "php"]
iniciais = {palavra[0] for palavra in palavras}
# Resultado: {'p', 'j'}

# Comprimentos únicos
nomes = ["Ana", "Bruno", "Carlos", "Diana", "Eva"]
tamanhos = {len(nome) for nome in nomes}
# Resultado: {3, 5, 6}

# Vogais em texto
texto = "Python é uma linguagem de programação"
vogais = {char.lower() for char in texto if char.lower() in "aeiou"}
# Resultado: {'a', 'e', 'i', 'o', 'u'}
```

Operações com Set Comprehensions

```
# Eliminar duplicatas com transformação
numeros = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
dobros_unicos = {x * 2 for x in numeros}
# Resultado: {2, 4, 6, 8}

# Caracteres únicos em múltiplas strings
palavras = ["hello", "world", "python"]
todos_caracteres = {char for palavra in palavras for char in palavra}
# Resultado: {'h', 'e', 'l', 'o', 'w', 'r', 'd', 'p', 'y', 't', 'n'}

# Filtrar e transformar
valores = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
quadrados_pares = {x**2 for x in valores if x % 2 == 0}
# Resultado: {4, 16, 36, 64, 100}
```

Exemplos Práticos

```
# Extrair domínios únicos de emails
emails = [
    "user1@gmail.com",
    "user2@yahoo.com",
    "user3@gmail.com",
    "user4@hotmail.com"
]
dominios = {email.split("@")[1] for email in emails}
# Resultado: {'gmail.com', 'yahoo.com', 'hotmail.com'}

# Identificar extensões de arquivos
arquivos = ["foto1.jpg", "doc.pdf", "foto2.jpg", "planilha.xlsx", "doc2.pdf"]
extensoes = {arquivo.split(".")[-1] for arquivo in arquivos}
# Resultado: {'jpg', 'pdf', 'xlsx'}

# Encontrar dígitos únicos em números
numeros = [123, 456, 789, 321, 654]
digitos = {digito for numero in numeros for digito in str(numero)}
# Resultado: {'1', '2', '3', '4', '5', '6', '7', '8', '9'}

# Tags únicas em postagens
postagens = [
    {"titulo": "Post 1", "tags": ["python", "data"]},
    {"titulo": "Post 2", "tags": ["python", "web"]},
    {"titulo": "Post 3", "tags": ["javascript", "web"]}
]
todas_tags = {tag for post in postagens for tag in post["tags"]}
# Resultado: {'python', 'data', 'web', 'javascript'}

# Identificar anos únicos
datas = ["2020-01-15", "2021-03-20", "2020-07-10", "2022-11-05"]
anos = {data.split("-")[0] for data in datas}
# Resultado: {'2020', '2021', '2022'}
```

Diferença entre List e Set Comprehension

```
# List comprehension - mantém duplicatas e ordem
lista = [x % 3 for x in range(10)]
# Resultado: [0, 1, 2, 0, 1, 2, 0, 1, 2, 0]

# Set comprehension - remove duplicatas, sem ordem garantida
conjunto = {x % 3 for x in range(10)}
# Resultado: {0, 1, 2}

# List comprehension com strings
palavras = ["python", "java", "python", "javascript"]
lista_inicial = [p[0] for p in palavras]
# Resultado: ['p', 'j', 'p', 'j']

# Set comprehension com strings
conjunto_inicial = {p[0] for p in palavras}
# Resultado: {'p', 'j'}
```

Exercícios

1. Crie um set comprehension que extrai todos os números únicos de uma lista de strings.
2. Dado um texto, crie um conjunto com todas as palavras únicas em minúsculas.
3. Extraia todos os códigos de área únicos de uma lista de números de telefone.
4. Crie um conjunto com os restos da divisão por 7 dos números de 1 a 100.

Desafios

1. Dada uma lista de números inteiros, crie um conjunto contendo apenas os valores únicos que são múltiplos de 3, usando set comprehension.
2. Dada uma frase, crie um conjunto com todas as letras distintas usadas, ignorando espaços e diferenças entre maiúsculas e minúsculas, usando set comprehension.

3. Dadas duas listas de números, crie um conjunto com os elementos que aparecem em ambas as listas e cujo valor ao quadrado seja menor que 100, usando set comprehension.

Aplicações

- Remoção de duplicatas em processamento de dados
- Identificação de elementos únicos em logs
- Análise de vocabulário em textos
- Detecção de categorias únicas em datasets
- Validação de unicidade em sistemas

---

10. Funções Built-in

Definição

Funções built-in são funções pré-definidas disponíveis nativamente em Python, otimizadas e prontas para uso sem necessidade de importação.

len()

Descrição: Retorna o número de elementos em uma sequência ou coleção.

```
# Com strings
texto = "Python"
tamanho = len(texto)  # 6

# Com listas
numeros = [1, 2, 3, 4, 5]
quantidade = len(numeros)  # 5

# Com dicionários
pessoa = {"nome": "João", "idade": 30}
campos = len(pessoa)  # 2

# Com sets e tuplas
conjunto = {1, 2, 3}
tupla = (1, 2, 3, 4)
len(conjunto)  # 3
len(tupla)     # 4

# Aplicações práticas
def validar_senha(senha):
    return len(senha) >= 8

def esta_vazio(lista):
    return len(lista) == 0  # Ou simplesmente: not lista
```

sum()

Descrição: Retorna a soma de todos os elementos de um iterável.

```
# Soma básica
numeros = [1, 2, 3, 4, 5]
total = sum(numeros)  # 15

# Com valor inicial
total_com_bonus = sum(numeros, 10)  # 25

# Soma de valores em dicionário
vendas = {"jan": 1000, "fev": 1500, "mar": 1200}
total_vendas = sum(vendas.values())  # 3700

# Soma condicional com compreensão
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
soma_pares = sum(x for x in numeros if x % 2 == 0)  # 30

# Aplicações práticas
def calcular_media(valores):
    return sum(valores) / len(valores) if valores else 0

def total_carrinho(produtos):
    return sum(produto["preco"] * produto["quantidade"]
               for produto in produtos)

# Soma de listas aninhadas
matriz = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
total_matriz = sum(sum(linha) for linha in matriz)  # 45
```

min() e max()

Descrição: Retornam o menor e o maior elemento de um iterável ou entre argumentos.

```
# Uso básico
numeros = [3, 1, 4, 1, 5, 9, 2, 6]
menor = min(numeros)  # 1
maior = max(numeros)  # 9

# Múltiplos argumentos
menor = min(5, 2, 8, 1, 9)  # 1
maior = max(5, 2, 8, 1, 9)  # 9

# Com strings (ordem lexicográfica)
palavras = ["python", "java", "rust", "go"]
primeira = min(palavras)  # 'go'
ultima = max(palavras)    # 'rust'

# Com parâmetro key
alunos = [
    {"nome": "Ana", "nota": 8.5},
    {"nome": "Bruno", "nota": 7.0},
    {"nome": "Carlos", "nota": 9.0}
]
melhor_aluno = max(alunos, key=lambda a: a["nota"])
# Resultado: {'nome': 'Carlos', 'nota': 9.0}

pior_aluno = min(alunos, key=lambda a: a["nota"])
# Resultado: {'nome': 'Bruno', 'nota': 7.0}

# Encontrar string mais longa
palavras = ["a", "bb", "ccc", "dddd"]
mais_longa = max(palavras, key=len)  # 'dddd'

# Aplicações práticas
def amplitude(valores):
    return max(valores) - min(valores)

def normalizar(valores):
    minimo, maximo = min(valores), max(valores)
    return [(v - minimo) / (maximo - minimo) for v in valores]

# Validação de intervalos
```

```
def esta_no_intervalo(valor, valores):
    return min(valores) <= valor <= max(valores)
```

sorted()

Descrição: Retorna uma nova lista ordenada a partir de qualquer iterável.

```
# Ordenação básica
numeros = [3, 1, 4, 1, 5, 9, 2, 6]
ordenados = sorted(numeros)  # [1, 1, 2, 3, 4, 5, 6, 9]

# Ordem decrescente
decrescente = sorted(numeros, reverse=True)  # [9, 6, 5, 4, 3, 2, 1, 1]

# Ordenar strings
palavras = ["python", "java", "rust", "go"]
alfabetica = sorted(palavras)  # ['go', 'java', 'python', 'rust']

# Ordenar por comprimento
por_tamanho = sorted(palavras, key=len)  # ['go', 'rust', 'java', 'python']

# Ordenar tuplas
pessoas = [("Ana", 25), ("Bruno", 30), ("Carlos", 22)]
por_idade = sorted(pessoas, key=lambda p: p[1])
# Resultado: [('Carlos', 22), ('Ana', 25), ('Bruno', 30)]

# Ordenar dicionários por valor
scores = {"Ana": 85, "Bruno": 92, "Carlos": 78}
ranking = sorted(scores.items(), key=lambda x: x[1], reverse=True)
# Resultado: [('Bruno', 92), ('Ana', 85), ('Carlos', 78)]

# Ordenação complexa
produtos = [
    {"nome": "Produto A", "preco": 50, "estoque": 10},
    {"nome": "Produto B", "preco": 30, "estoque": 5},
    {"nome": "Produto C", "preco": 50, "estoque": 20}
]

# Ordenar por preço, depois por estoque
ordenados = sorted(produtos, key=lambda p: (p["preco"], -p["estoque"]))

# Ordenação case-insensitive
nomes = ["ana", "Bruno", "carlos", "Diana"]
ordenados = sorted(nomes, key=str.lower)  # ['ana', 'Bruno', 'carlos', 'Diana']

# Aplicações práticas
```

```
def top_n(valores, n=5):
    return sorted(valores, reverse=True)[:n]

def mediana(valores):
    ordenados = sorted(valores)
    meio = len(ordenados) // 2
    if len(ordenados) % 2 == 0:
        return (ordenados[meio-1] + ordenados[meio]) / 2
    return ordenados[meio]
```

enumerate()

Descrição: Retorna um iterador de tuplas contendo índices e valores de uma sequência.

```
# Uso básico
frutas = ["maçã", "banana", "laranja"]
for indice, fruta in enumerate(frutas):
    print(f"{indice}: {fruta}")
# 0: maçã
# 1: banana
# 2: laranja

# Começar de índice diferente
for indice, fruta in enumerate(frutas, start=1):
    print(f"{indice}: {fruta}")
# 1: maçã
# 2: banana
# 3: laranja

# Converter para lista
indexada = list(enumerate(frutas))
# Resultado: [(0, 'maçã'), (1, 'banana'), (2, 'laranja')]

# Criar dicionário com índices
frutas_dict = dict(enumerate(frutas))
# Resultado: {0: 'maçã', 1: 'banana', 2: 'laranja'}

# Encontrar índice de elementos que atendem condição
numeros = [10, 25, 30, 45, 50]
indices_maiores = [i for i, v in enumerate(numeros) if v > 30]
# Resultado: [3, 4]

# Modificar lista com base no índice
valores = [1, 2, 3, 4, 5]
modificados = [v * i for i, v in enumerate(valores)]
# Resultado: [0, 2, 6, 12, 20]

# Aplicações práticas
def encontrar_indices(lista, elemento):
    return [i for i, v in enumerate(lista) if v == elemento]

def processar_com_contexto(dados):
    resultado = []
```

```
    for i, item in enumerate(dados):
        anterior = dados[i-1] if i > 0 else None
        proximo = dados[i+1] if i < len(dados)-1 else None
        resultado.append(processar(item, anterior, proximo))
    return resultado

# Numerar linhas de arquivo
linhas = ["primeira linha", "segunda linha", "terceira linha"]
numeradas = [f"Linha {i+1}: {linha}" for i, linha in enumerate(linhas)]
```

zip()

Descrição: Combina múltiplos iteráveis em um iterador de tuplas, emparelhando elementos correspondentes.

```
# Uso básico
nomes = ["Ana", "Bruno", "Carlos"]
idades = [25, 30, 28]
pessoas = list(zip(nomes, idades))
# Resultado: [('Ana', 25), ('Bruno', 30), ('Carlos', 28)]

# Três ou mais iteráveis
nomes = ["Ana", "Bruno", "Carlos"]
idades = [25, 30, 28]
cidades = ["SP", "RJ", "BH"]
dados = list(zip(nomes, idades, cidades))
# Resultado: [('Ana', 25, 'SP'), ('Bruno', 30, 'RJ'), ('Carlos', 28, 'BH')]

# Iteráveis de tamanhos diferentes (para no menor)
lista1 = [1, 2, 3, 4, 5]
lista2 = ['a', 'b', 'c']
resultado = list(zip(lista1, lista2))
# Resultado: [(1, 'a'), (2, 'b'), (3, 'c')]

# Criar dicionário
chaves = ["nome", "idade", "cidade"]
valores = ["João", 30, "São Paulo"]
dicionario = dict(zip(chaves, valores))
# Resultado: {'nome': 'João', 'idade': 30, 'cidade': 'São Paulo'}

# Descompactar (unzip)
pares = [(1, 'a'), (2, 'b'), (3, 'c')]
numeros, letras = zip(*pares)
# numeros: (1, 2, 3)
# letras: ('a', 'b', 'c')

# Transpor matriz
matriz = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
transposta = list(zip(*matriz))
# Resultado: [(1, 4, 7), (2, 5, 8), (3, 6, 9)]

# Combinar listas em pares
lista = [1, 2, 3, 4, 5]
pares = list(zip(lista, lista[1:]))
```

```
# Resultado: [(1, 2), (2, 3), (3, 4), (4, 5)]

# Aplicações práticas
def calcular_diferencas(valores):
    return [b - a for a, b in zip(valores, valores[1:])]

def mesclar_dicionarios(chaves, dicts):
    return {k: [d[k] for d in dicts] for k in chaves}

# Iterar sobre múltiplas listas simultaneamente
precos_antes = [100, 200, 150]
precos_depois = [90, 210, 145]
for antes, depois in zip(precos_antes, precos_depois):
    variacao = ((depois - antes) / antes) * 100
    print(f"Variação: {variacao:.2f}%")

# Combinar dados de múltiplas fontes
ids = [1, 2, 3]
nomes = ["Produto A", "Produto B", "Produto C"]
precos = [10.50, 25.00, 15.75]
produtos = [
    {"id": id, "nome": nome, "preco": preco}
    for id, nome, preco in zip(ids, nomes, precos)
]
```

Exemplos

```
# Combinar múltiplas funções
dados = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]

# Estatísticas básicas
total = sum(dados)
quantidade = len(dados)
media = total / quantidade
minimo = min(dados)
maximo = max(dados)
ordenados = sorted(dados)

# Análise de frequência
from collections import Counter
frequencia = Counter(dados)
mais_comum = max(frequencia.items(), key=lambda x: x[1])

# Processamento de múltiplas listas
nomes = ["Ana", "Bruno", "Carlos", "Diana"]
notas1 = [8.5, 7.0, 9.0, 8.0]
notas2 = [7.5, 8.0, 9.5, 7.0]

# Calcular médias
medias = [
    (nome, (n1 + n2) / 2)
    for nome, n1, n2 in zip(nomes, notas1, notas2)
]

# Encontrar melhor e pior
melhor = max(medias, key=lambda x: x[1])
pior = min(medias, key=lambda x: x[1])

# Ranking ordenado
ranking = sorted(medias, key=lambda x: x[1], reverse=True)
ranking_formatado = [
    f"{i+1}. {nome}: {media:.2f}"
    for i, (nome, media) in enumerate(ranking)
]
```

Exercícios

1. Crie uma função que recebe duas listas e retorna um dicionário com elementos da primeira como chaves e da segunda como valores.
2. Implemente uma função que calcula a média móvel de uma lista de números usando zip.
3. Escreva uma função que encontra o k-ésimo maior elemento de uma lista sem usar sorted.
4. Crie uma função que normaliza uma lista de números para o intervalo [0, 1] usando min e max.
5. Implemente uma função que retorna os índices dos n maiores elementos de uma lista usando enumerate.

Desafios

1. Desenvolva uma função que implementa merge sort usando apenas sorted, len e zip.
2. Crie um sistema de ranking que considera múltiplos critérios com pesos diferentes.
3. Implemente uma função que calcula a correlação entre duas listas de números.
4. Desenvolva uma solução para o problema de agendamento de tarefas usando sorted com key customizado.

Aplicações

- Análise estatística de dados
- Ordenação e ranking de resultados
- Processamento paralelo de múltiplas sequências
- Cálculo de métricas e agregações
- Normalização e transformação de datasets
- Emparelhamento de dados relacionados

Exercícios

Nível Básico

1. Crie um programa que lê uma lista de números e retorna um dicionário com estatísticas: mínimo, máximo, média, mediana e quantidade de elementos.
2. Implemente uma função que recebe uma lista de palavras e retorna um dicionário onde as chaves são as palavras e os valores são listas de índices onde cada palavra aparece.
3. Escreva uma função que remove duplicatas de uma lista mantendo a ordem original dos elementos.

Nível Intermediário

1. Crie uma função que recebe uma lista de dicionários representando produtos (com campos nome, preço, categoria) e retorna um dicionário agrupando produtos por categoria.
2. Implemente um sistema de gerenciamento de estoque que permite adicionar produtos, remover produtos, atualizar quantidades e listar produtos por categoria usando dicionários e listas.
3. Desenvolva uma função que encontra todos os pares de números em uma lista cuja soma é igual a um valor alvo.

Nível Avançado

1. Implemente uma função que recebe uma lista de transações (cada uma com data, valor e categoria) e retorna um relatório mensal agregado por categoria.
2. Crie um sistema de cache que armazena resultados de funções custosas usando dicionários com tuplas como chaves.
3. Desenvolva uma função que implementa o algoritmo de busca binária recursiva em uma lista ordenada.
4. Implemente uma estrutura de dados de grafo usando dicionários e desenvolva um algoritmo para encontrar o caminho mais curto entre dois nós.

Desafios

Desafio 1: Sistema de Biblioteca

Implemente um sistema completo de gerenciamento de biblioteca com as seguintes funcionalidades:

- Cadastro de livros (título, autor, ISBN, ano, disponível)
- Cadastro de usuários
- Empréstimo e devolução de livros
- Busca de livros por autor, título ou ISBN
- Relatório de livros mais emprestados
- Lista de usuários com empréstimos atrasados

Use dicionários, listas, sets e tuplas apropriadamente.

Desafio 2: Análise de Texto

Crie um analisador de texto que:

- Conta frequência de palavras
- Identifica as 10 palavras mais comuns
- Calcula o comprimento médio das palavras
- Encontra palavras únicas (que aparecem apenas uma vez)
- Identifica bigramas (pares de palavras consecutivas) mais frequentes
- Gera estatísticas de pontuação

Desafio 3: Sistema de Recomendação

Desenvolva um sistema simples de recomendação baseado em:

- Usuários e seus itens favoritos (usando sets)
- Cálculo de similaridade entre usuários
- Recomendação de itens com base em usuários similares
- Ranking de recomendações por relevância

Conclusão

As estruturas de dados fundamentais em Python (listas, tuplas, conjuntos e dicionários) formam a base para praticamente qualquer programa Python. Entender como essas estruturas funcionam é fundamental para escrever código de qualidade.

Importante

1. **Listas** são mutáveis e ordenadas - use quando precisar modificar coleções
2. **Tuplas** são imutáveis e ordenadas - use para dados que não devem mudar
3. **Sets** são mutáveis e não ordenados - use para eliminar duplicatas e operações de conjunto
4. **Dicionários** mapeiam chaves a valores - use para acesso rápido por identificadores

Quando Usar Cada Estrutura

- Use **listas** quando a coleção muda ao longo da execução
- Use **tuplas** para dados constantes, retornos múltiplos para funções, chaves de dicionário
- Use **sets** para testes de pertinência, operações matemáticas de conjunto, remoção de duplicatas
- Use **dicionários** para mapeamentos chave-valor, caches, configurações

Boas Práticas

1. Escolha a estrutura com base no comportamento necessário, não por conveniência
2. Use compreensões quando elas realmente simplificarem a leitura
3. Prefira funções nativas (built-in) sempre que resolverem o problema diretamente
4. Tenha clareza sobre mutabilidade para evitar efeitos colaterais difíceis de rastrear
5. Dê preferência a objetos imutáveis quando o estado do objeto não deve mudar