Coleções

1. Introdução

Em Python, as coleções referem-se a estruturas de dados que agrupam vários elementos, permitindo operações como armazenamento, acesso, modificação e iteração. Embora o termo "array" seja mais comum em linguagens como C ou Java, em Python, a estrutura equivalente é a lista, uma coleção versátil e mutável. Além das listas, Python oferece tuplas, dicionários, conjuntos (sets) e, para casos específicos, o tipo array do módulo array ou a biblioteca numpy.

O que são coleções? Uma coleção é um objeto que organiza múltiplos elementos em uma estrutura coesa, permitindo manipulações eficientes. Python implementa coleções como tipos embutidos, projetados para flexibilidade e desempenho. As principais coleções incluem:

- Listas: Sequências mutáveis que armazenam elementos ordenados de qualquer tipo.
- Tuplas: Sequências imutáveis que mantêm a ordem dos elementos.
- Dicionários (dict): Estruturas de pares chave-valor, otimizadas para buscas por chave.
- Conjuntos (set e frozenset): Coleções não ordenadas de elementos únicos, ideais para operações matemáticas.
- Arrays (array.array): Estruturas convencionais do módulo array, otimizadas para eficiência de memória.

Cada coleção possui características específicas que determinam sua adequação a diferentes problemas, como ordenação, mutabilidade e eficiência computacional.

2. Propriedades das Coleções

Listas:

Mutabilidade: Mutável, permitindo adição, remoção ou alteração de elementos. Ordenação: Mantêm a ordem de inserção. Indexação: Acessadas por índices inteiros (base 0). Flexibilidade: Suportam elementos heterogêneos.

Tuplas:

Mutabilidade: Imutáveis, impossibilitando alterações após criação. Ordenação: Mantêm a ordem dos elementos. Indexação: Suporta acesso por índices, como listas. Eficiência: Mais folhas em memória que listas.

Dicionários:

Mutabilidade: Mutável, permitindo manipulação de pares chave-valor. Ordenação: Desde Python 3.7, preserva a ordem de inserção. Indexação: Acessados por chaves imutáveis (ex.: strings, números, tuplas). Eficiência: Otimizado para buscas rápidas por chave.

Conjuntos:

Mutabilidade: set é mutável; frozenset é imutável. Ordenação: Não ordenado, sem suporte a indexação. Unicidade: Garantir elementos únicos, eliminando duplicatas. Operações: Apoiar operações de conjunto (união, interseção, diferença).

Matrizes:

Mutabilidade: Mutável, mas restrita a tipos homogêneos. Ordenação: Mantêm a ordem dos elementos. Eficiência: Otimizados para dados numéricos, consumindo menos memória que listas.

3. Como Usar Coleções

As coleções são manipuladas por meio de métodos embutidos e entregas específicas. A seguir, são apresentados exemplos detalhados para cada tipo, ilustrando operações comuns e cenários práticos.

Listas Listas são criadas com colchetes ([]) ou a função list(). Eles suportam métodos como anexar, pop, inserir e remover.

Exemplo: Gerenciamento de tarefas

```
tarefas = ["Estudar Python", "Fazer compras", "Enviar e-mail"]
tarefas.append("Reunião às 14h")  # Adiciona ao final
tarefas.remove("Fazer compras")    # Remove elemento específico
tarefas[1] = "Enviar e-mail urgente"  # Altera elemento
print(tarefas)  # Saída: ['Estudar Python', 'Enviar e-mail urgente', 'Reuni

# Exemplo 2: Filtragem de números pares
numeros = [1, 2, 3, 4, 5, 6]
pares = [n for n in numeros if n % 2 == 0]
print(pares)  # Saída: [2, 4, 6]
```

Tuplas Tuplas são criadas com pares (()) ou tuple(). Sua imutabilidade as torna ideais para dados constantes.

Exemplo: Coordenadas geográficas

```
ponto = (-23.5505, -46.6333)  # Latitude e longitude de São Paulo
latitude, longitude = ponto    # Desempacotamento
print(f"Latitude: {latitude}, Longitude: {longitude}")  # Saída: Latitude:

# Exemplo 2: Configurações fixas
config = ("localhost", 8080, "admin")
print(config[1])  # Saída: 8080
# config[1] = 9090  # Erro: tuplas são imutáveis
```

Dicionários Dicionários são criados com chaves ({}) ou dict(). Métodos como get, update e pop facilitam manipulações.

Exemplo: Cadastro de produto

```
produto = {"id": 101, "nome": "Notebook", "preco": 3500.00}
produto["estoque"] = 10  # Adiciona nova chave
produto.update({"preco": 3400.00, "marca": "Tech"})  # Atualiza múltiplos v
print(produto.get("nome", "Desconhecido"))  # Saída: Notebook

# Exemplo 2: Contagem de ocorrências
frases = ["gato", "cachorro", "gato", "pássaro"]
contagem = {}
for palavra in frases:
    contagem[palavra] = contagem.get(palavra, 0) + 1
print(contagem)  # Saída: {'gato': 2, 'cachorro': 1, 'pássaro': 1}
```

Conjuntos Conjuntos são criados com chaves ({}) ou set(). Métodos como adição, união e intersecção são comuns.

Exemplo: Eliminação de duplicatas

```
emails = ["ana@ex.com", "bob@ex.com", "ana@ex.com"]
emails_unicos = set(emails)
print(emails_unicos)  # Saída: {'ana@ex.com', 'bob@ex.com'}

# Exemplo 2: Interseção de interesses
interesses_ana = {"Python", "Java", "SQL"}
interesses_bob = {"Python", "C++", "SQL"}
comuns = interesses_ana & interesses_bob
print(comuns)  # Saída: {'Python', 'SQL'}
```

Arrays O módulo array cria arrays homogêneos, especificando o tipo de dado (ex.: 'i' para inteiros). da matriz importar matriz

Exemplo: Armazenamento de temperaturas

```
temperaturas = array('f', [23.5, 24.0, 22.8])
temperaturas.append(25.1)
print(temperaturas)  # Saída: array('f', [23.5, 24.0, 22.8, 25.1])

# Exemplo 2: Cálculo de média
media = sum(temperaturas) / len(temperaturas)
print(f"Média: {media:.2f}")  # Saída: Média: 23.85
```

4. Casos de Uso

- Listas: Indicadas para listas de tarefas, históricos de transações ou dados planejados que mudam frequentemente.
- Tuplas: Usadas para configurações fixas, como conexões de banco de dados (host, porta, usuário) ou coordenadas.
- Dicionários: Ideais para cadastros (ex.: informações de clientes por ID) ou contagens de frequência.
- Conjuntos: Aplicáveis para eliminar duplicatas (ex.: lista de e-mails únicos) ou comparar grupos (ex.: interesses comuns).
- Matrizes: Recomendados para processamento numérico intensivo, como análise de dados sensoriais ou cálculos estatísticos.

5. Melhores Práticas

Seleção da Coleção Correta:

Escolha listas para sequências mutáveis, tuplas para dados imutáveis, dicionários para mapeamentos e conjuntos para unicidade. Use array ou numpy para dados numéricos homogêneos em aplicações de alto desempenho.

Nomenclatura Descritiva:

Adote nomes que reflitam o conteúdo, como clientes ou configurações, seguindo a convenção PEP 8.

Validação de Entradas:

Verifique tipos e valores antes de manipular coleções, evitando erros como índices inválidos ou chaves inexistentes.

```
def acessar_elemento(lista, indice):
    if not isinstance(lista, list) or indice >= len(lista):
        raise IndexError("Índice inválido ou entrada não é uma lista")
    return lista[indice]
```

Evitar Modificações Durante a Iteração:

Modificar coleções durante a iteração pode causar erros ou comportamentos imprevisíveis.

```
# Errado
for item in lista:
    lista.remove(item)  # Pode pular elementos
# Correto
lista[:] = [item for item in lista if not deve_remover(item)]
```

Usar aninhamento:

Aninhamento em lista, dicionário e conjunto são mais legíveis e eficientes que os loops tradicionais.

```
# Em vez de:
quadrados = []
for i in range(10):
    quadrados.append(i ** 2)
# Use:
quadrados = [i ** 2 for i in range(10)]
```

Aproveitar Métodos Embutidos:

Métodos como list.sort, dict.get e set.union são otimizados e devem ser preferidos.

```
# Em vez de:
if chave in dicionario:
    valor = dicionario[chave]
else:
    valor = padrão
# Use:
valor = dicionario.get(chave, padrão)
```

Garantir Imutabilidade Quando Necessário:

Use tuplas ou frozenset para dados que não devem ser alterados, especialmente em sistemas concorrentes.

Gerenciar Memória:

Para grandes coleções, prefira array ou numpy em vez de listas. Evite solicitações de cópias com copy.copy ou copy.deepcopy apenas quando necessário.

6. Exemplos completos

Aqui está um exemplo completo de cadastro CRUD usando apenas listas.

Aqui tem um exemplo completo de cadastro CRUD usando cojuntos, set e frozenset.