Coleções em Python: Tuplas, Listas, Sets e Dicionários

Informações do Autor

- **Nome:** ANIMA Lab
- **Data de atualização**: 07/10/2025

Introdução

Todo programa precisa lidar com dados. No Python, os tipos de dados básicos (como números, textos e valores lógicos) são usados para representar informações simples.
No entanto, em muitas situações, é necessário agrupar e organizar vários valores para manipulá-los de forma eficiente.

Para isso, o Python oferece as coleções, também chamadas de estruturas de dados compostas. Essas coleções permitem armazenar múltiplos elementos dentro de uma única variável, facilitando tarefas como:

- percorrer listas de informações;
- associar dados por nome;
- eliminar duplicatas;
- realizar buscas e filtragens.

Cada tipo de coleção possui características próprias — algumas permitem alterações, outras mantêm os dados fixos; algumas preservam a ordem dos elementos, outras não.

> Escolher o tipo correto de coleção é uma decisão fundamental na construção de qualquer programa.

Coleções em Python

As coleções são estruturas que armazenam e organizam grupos de dados de forma estruturada.

Python possui quatro tipos principais de coleções integradas:

- **Tuplas**: elementos em uma sequência com posições fixas (posições numeradas) e imutáveis (não podem ser alterados após a criação).
- **Listas**: elementos em uma sequência com posições fixas (posições numeradas) e mutáveis, podem ser alteradas durante a execução.
- **Sets (conjuntos)**: elementos sem uma ordem definida e sem posições numeradas (índices), sem duplicatas e mutáveis.

- **Dicionários**: pares de chave e valor, onde cada chave é única e serve como identificador do valor associado.

Tuplas e Lista

Tuplas e listas são tratadas juntas para destacar semelhanças e diferenças importantes antes de entrar em detalhes de uso.

O que é uma sequência

Sequência é um tipo de dado que armazena vários valores em uma ordem definida. Tuplas e listas são sequências.

- ordem significa posição fixa de cada elemento (índices 0, 1, 2, …)
- dá para acessar pelo índice e percorrer com laços

Exemplo:

```
seq = ["A", "B", "C"]   # lista
print(seq[0])           # A
print(seq[-1])          # C
```

Tupla x Lista: a diferença central

- Tupla é imutável: depois de criada, não aceita inserir, remover ou alterar elementos
- Lista é mutável: permite inserir, remover e alterar elementos

Exemplo:

```
t = (10, 20, 30)     # tupla
l = [10, 20, 30]     # lista

# l pode ser alterada
l[1] = 99            # ok

# t não pode ser alterada
# t[1] = 99
# TypeError: 'tuple' object
# does not support item assignment
```

Quando usar cada uma

Use tupla quando:

- os dados são fixos por natureza (coordenadas, datas, intervalos, parâmetros de configuração carregados e congelados)
- você quer sinalizar no código que aquela coleção não deve mudar
- precisa usar como chave de dicionário (tuplas podem ser chaves; listas não)

Use lista quando:

- o conjunto de dados muda ao longo do programa (cadastros, filas de processamento, amostras que crescem)
- é necessário inserir, remover ou reordenar elementos
- vai aplicar transformações sucessivas (filtrar, mapear, ordenar)

Exemplos

```
coordenada = (-16.68, -49.25) # tupla: posição no mapa
leituras = [72, 75, 73, 74] # lista: batimentos coletados ao longo d
```

Criando tuplas e listas

Tuplas

Uma tupla é criada usando parênteses `( )` e separando os elementos por vírgulas.
Ela pode conter números, textos ou outros tipos de dados.
Como é imutável, é ideal para representar informações fixas que não devem ser modificadas durante o programa.

Exemplo 1

```
tup1 = (60, 62, 65, 68)
print(tup1)
```

Saída:

```
(60, 62, 65, 68)
```

Cada valor da tupla ocupa uma posição específica: `60` está na posição 0, `62` na posição 1, `65` na posição 2 e `68` na posição 3.

Tuplas são usadas quando se quer garantir que os dados não sejam alterados, como:

- coordenadas geográficas
- datas de eventos
- faixas de medição que permanecem fixas

Listas

Uma lista é criada usando colchetes `[]`. Diferente das tuplas, listas podem ser modificadas: é possível adicionar, remover ou alterar elementos depois de criadas.

Exemplo 2

Criando uma lista de nomes:

```
nomes = ["Maria", "João", "Ana"]
print(nomes)
```

Saída:

```
['Maria', 'João', 'Ana']
```

Assim como nas tuplas, cada item possui um índice: `Maria` está na posição 0, `João` na posição 1 e `Ana` na posição 2.

Listas são ideais para representar conjuntos de dados que mudam ao longo do tempo, como:

- cadastros de participantes
- séries de medições contínuas
- filas de processamento

As duas estruturas compartilham o mesmo comportamento de acesso por índice e podem armazenar qualquer tipo de dado, mas diferem no propósito:

- tuplas garantem estabilidade dos dados
- listas oferecem flexibilidade para manipulação

Acesso e percurso

- índices começam em 0
- índices negativos contam a partir do fim (-1 é o último)
- for percorre na ordem de inserção

```
dados = ("Maria", 72, "Goiânia")   # poderia ser lista também
print(dados[0])                    # Maria
print(dados[-1])                   # Goiânia

for item in dados:
    print(item)
```

Cópia x referência

- Tuplas são imutáveis, então compartilhar a mesma tupla não traz efeitos colaterais de alteração.
- Listas são mutáveis; atribuir uma lista a outra variável não cria cópia, apenas outro nome para o mesmo objeto.

```
l1 = [1, 2, 3]
l2 = l1          # referência
l2[0] = 99
print(l1)        # [99, 2, 3]

l3 = l1[:]       # cópia rasa
l3[1] = 100
print(l1)        # [99, 2, 3]
print(l3)        # [99, 100, 3]
```

Desempacotamento de tuplas e listas

Tanto tuplas quanto listas podem armazenar vários valores relacionados. Em muitos casos, é necessário usar esses valores separadamente e acessar cada um por índice torna o código mais difícil de entender e de manter.

O desempacotamento é uma forma direta e legível de atribuir os valores de uma coleção a variáveis individuais, respeitando a ordem em que aparecem. Essa técnica evita o uso de índices e torna o código mais descritivo e sem ambiguidades.

Exemplo com tupla

```
coordenadas = (16.68, -49.25)
latitude, longitude = coordenadas
print("Latitude:", latitude, "| Longitude:", longitude)
```

Saída:

```
Latitude: 16.68 | Longitude: -49.25
```

O Python distribui automaticamente os valores da tupla `coordenadas` nas variáveis `latitude` e `longitude`, seguindo a ordem original.

Exemplo com lista

O mesmo princípio vale para listas. O desempacotamento também funciona quando os dados estão em uma estrutura mutável.

```
dados_paciente = ["Maria", 72, "Goiânia"]
nome, idade, cidade = dados_paciente
print(nome, "-", idade, "anos -", cidade)
```

Saída:

```
Maria - 72 anos - Goiânia
```

Nesse caso, o desempacotamento facilita o uso de dados extraídos de cadastros, tabelas ou planilhas, onde as posições têm significados conhecidos.

Ignorando valores

Em algumas situações, nem todos os elementos da coleção são relevantes para o que se deseja fazer.
Nesses casos, é possível descartar valores no momento do desempacotamento usando o sublinhado _, que representa uma variável descartável.

Essa prática é útil quando o dado existe, mas não será utilizado no programa, evitando a criação de variáveis desnecessárias e deixando o código mais claro.

Exemplo com tupla:

```
registro = ("João", 68, "Anápolis", "Ativo")
nome, idade, _, status = registro
print(nome, "-", idade, "anos -", status)
```

Saída:

```
João - 68 anos - Ativo
```

O terceiro valor ("Anápolis") foi ignorado, pois não é necessário na saída.

O mesmo comportamento se aplica a listas, já que o desempacotamento funciona com qualquer coleção ordenada:

```
registro = ["Maria", 72, "Goiânia", "Ativa"]
nome, idade, _, status = registro
print(nome, "-", idade, "anos -", status)
```

Saída:

```
Maria - 72 anos - Ativa
```

O uso de _ é uma **convenção do Python** que comunica claramente a intenção de que determinado valor não será utilizado, contribuindo para a legibilidade e manutenção do código.

Aplicação prática

O desempacotamento é uma técnica útil quando uma coleção agrupa várias informações relacionadas e o programa precisa utilizá-las de forma separada.
Em vez de acessar os elementos pelos índices (como `dados[0]`, `dados[1]`, etc.), é possível atribuir diretamente cada valor a uma variável significativa, o que torna o código mais legível e fácil de manter.

Essa abordagem é comum em cenários como:

- funções que retornam múltiplos resultados;
- registros armazenados em listas ou tuplas;
- leitura de dados estruturados vindos de arquivos, planilhas ou bancos de dados.

Exemplo aplicado:

```
dados = ("Carlos", 70, 120, 80)  # nome, idade, pressão sistólica, p
nome, idade, sistolica, diastolica = dados

print(f"Paciente: {nome} ({idade} anos)")
print(f"Pressão arterial: {sistolica}/{diastolica} mmHg")
```

Saída:

```
Paciente: Carlos (70 anos)
Pressão arterial: 120/80 mmHg
```

Nesse exemplo, cada variável recebe um valor da tupla `dados`, o que permite manipular as informações de forma semântica — ou seja, com nomes que expressam seu significado. Esse

tipo de estrutura é muito útil em aplicações de análise de dados, monitoramento de saúde, leitura de sensores ou processamento de registros de pesquisa, onde cada posição representa um campo fixo e conhecido.

Funções e operações comuns

Ao trabalhar com coleções como tuplas ou listas, é importante saber como consultar informações sobre o conteúdo armazenado.
Essas funções permitem contar elementos, localizar valores e verificar a presença de dados específicos — operações comuns em análises de registros, classificações ou medições.

Essas operações funcionam tanto em tuplas quanto em listas.
A diferença é que as tuplas são imutáveis (não podem ser alteradas), enquanto as listas permitem modificações, inserções e remoções.

Principais funções e operadores:

- `len(colecao)` — retorna o número total de elementos.
- `colecao.count(valor)` — conta quantas vezes o valor aparece.
- `colecao.index(valor)` — retorna o índice (posição) da primeira ocorrência do valor.
- `in` — verifica se um elemento está presente na coleção.

Exemplo prático com tupla:

```
faixas = ("baixo", "médio", "alto", "médio", "baixo")

print(len(faixas))            # quantidade total de elementos
print(faixas.count("médio"))  # quantas vezes "médio" aparece
print(faixas.index("alto"))   # posição de "alto"

if "baixo" in faixas:
    print("Categoria 'baixo' identificada.")
```

Saída:

```
5
2
2
Categoria 'baixo' identificada.
```

Exemplo equivalente com lista:

```
faixas = ["baixo", "médio", "alto", "médio", "baixo"]
faixas.append("muito alto")  # modificação possível em listas

print(len(faixas))
print(faixas.count("baixo"))
print("muito alto" in faixas)
```

Saída:

```
6
2
True
```

Coleções aninhadas e heterogêneas

Tanto tuplas quanto listas podem armazenar elementos de tipos diferentes, como números, textos, valores lógicos ou até outras coleções.

Essa característica é chamada de heterogeneidade e é importante quando um mesmo registro contém informações variadas, que precisam permanecer agrupadas.

As tuplas são imutáveis, o que garante que os dados permaneçam fixos. As listas, por outro lado, são mutáveis, permitindo a inclusão ou alteração dos elementos quando necessário. A escolha entre as duas depende da natureza dos dados: se devem permanecer constantes (tupla) ou se podem ser modificados (lista).

Exemplo com tupla:

```
pessoa = ("Maria", 72, ("Goiânia", "GO"))
print("Nome:", pessoa[0])
print("Cidade:", pessoa[2][0], "-", pessoa[2][1])
```

Exemplo com lista equivalente:

```
pessoa = ["Maria", 72, ["Goiânia", "GO"]]
pessoa[1] = 73  # alteração permitida apenas em listas
print("Nome:", pessoa[0])
print("Cidade:", pessoa[2][0], "-", pessoa[2][1])
```

Saída:

```
Nome: Maria
Cidade: Goiânia - GO
```

Essas estruturas são adequadas para representar:

- registros fixos (nome, idade, endereço);
- coordenadas geográficas;
- medições compostas por múltiplos parâmetros;
- informações organizadas em camadas, como alunos, cursos e notas.

O uso de tuplas ou listas depende do tipo de dado:

- tuplas, quando as informações não devem ser alteradas;
- listas, quando é preciso atualizar ou expandir os dados.

Coleções como retorno de funções

Funções podem devolver mais de um valor agrupando-os em uma tupla ou lista. Essa prática é comum em cálculos e medições, onde há mais de um resultado a ser utilizado posteriormente.

Exemplo com tupla:

```python
def calcular_medidas(valores):
    media = sum(valores) / len(valores)
    minimo = min(valores)
    maximo = max(valores)
    return media, minimo, maximo

resultados = calcular_medidas((65, 70, 72, 68))
print("Média, Mínimo, Máximo:", resultados)
```

Saída:

```
Média, Mínimo, Máximo: (68.75, 65, 72)
```

Os valores retornados podem ser desempacotados diretamente:

```python
media, minimo, maximo = calcular_medidas([65, 70, 72, 68])
print("Média:", media, "| Mínimo:", minimo, "| Máximo:", maximo)
```

Saída:

```
Média: 68.75 | Mínimo: 65 | Máximo: 72
```

Quando os valores não precisam ser alterados, o retorno pode ser uma tupla. Se houver necessidade de manipulação posterior (adição, atualização ou exclusão de dados), o retorno como lista é mais adequado.

Exercício-1

Análise de coordenadas de um centro de convivência

Um pesquisador deseja mapear os centros de convivência de idosos de uma cidade. Cada centro é representado por uma tupla no formato `(nome, latitude, longitude)`.

Crie um programa que:

1. Armazene pelo menos 4 centros de convivência em uma lista de tuplas.
2. Exiba todos os centros com suas coordenadas.
3. Peça ao usuário para digitar o nome de um centro e mostre sua localização.
4. Calcule a média das latitudes e longitudes (posição média da cidade).

Exercício-2

Monitoramento de saúde de idosos

Um pesquisador monitora os batimentos cardíacos de um grupo de idosos. Cada idoso é representado por um dicionário com nome e lista de batimentos registrados durante o dia.

Crie um programa que:

1. Armazene pelo menos 3 idosos com 5 medições cada.
2. Mostre o nome e a média dos batimentos de cada idoso.
3. Exiba o nome do idoso com a maior média de batimentos.
4. Permita adicionar um novo registro de batimentos a um idoso específico.

Sets (conjuntos)

Um conjunto é uma coleção que armazena elementos únicos, sem uma ordem específica.

Ao contrário de listas e tuplas, os conjuntos não mantêm a sequência dos itens e não permitem valores repetidos. Essa estrutura é útil quando o foco está na presença ou ausência de elementos, e não em sua posição.

Os conjuntos são amplamente utilizados para eliminar duplicidades, identificar valores distintos e realizar comparações entre grupos de dados. Em ciência de dados, por exemplo,

podem ser aplicados na filtragem de amostras, detecção de categorias únicas ou operações de cruzamento entre conjuntos de informações.

Características principais

- Não ordenado: os elementos não possuem uma sequência definida.
- Não indexado: não há acesso por índices.
- Mutável: novos elementos podem ser adicionados e removidos.
- Sem duplicatas: qualquer valor repetido é automaticamente ignorado.

Criando conjuntos

Um conjunto é criado utilizando chaves `{}` ou a função `set()`. Elementos duplicados são automaticamente removidos no momento da criação.

Exemplo:

```
idades = {60, 65, 70, 65, 75}
print(idades)
```

Saída:

```
{65, 70, 75, 60}
```

Apesar de “65” aparecer duas vezes na definição, ele é armazenado apenas uma vez. A ordem dos elementos pode variar a cada execução, já que os conjuntos não mantêm posição.

Também é possível criar um conjunto a partir de outras coleções, como listas ou tuplas:

```
nomes = ["Maria", "Carlos", "Ana", "Carlos"]
conjunto_nomes = set(nomes)
print(conjunto_nomes)
```

Saída:

```
{'Carlos', 'Ana', 'Maria'}
```

Operações com conjuntos

Os conjuntos permitem operações matemáticas úteis para análise de dados:

- União (`|` ou `union()`): junta os elementos de dois conjuntos.
- Interseção (`&` ou `intersection()`): retorna apenas os elementos comuns.
- Diferença (`-` ou `difference()`): mostra o que está em um conjunto e não no outro.
- Diferença simétrica (`^` ou `symmetric_difference()`): retorna elementos que estão em apenas um dos conjuntos.

Exemplo:

```
grupo_a = {"Carlos", "Ana", "Marcos"}
grupo_b = {"Ana", "João", "Paula"}

print("União:", grupo_a | grupo_b)
print("Interseção:", grupo_a & grupo_b)
print("Diferença (A - B):", grupo_a - grupo_b)
print("Diferença Simétrica:", grupo_a ^ grupo_b)
```

Saída:

```
União: {'Paula', 'Carlos', 'Marcos', 'Ana', 'João'}
Interseção: {'Ana'}
Diferença (A - B): {'Marcos', 'Carlos'}
Diferença Simétrica: {'João', 'Marcos', 'Paula', 'Carlos'}
```

Essas operações são úteis para comparar grupos, como participantes de diferentes turmas, categorias de produtos ou conjuntos de clientes que compraram em períodos distintos.

Criando conjuntos

Um conjunto é criado com chaves `{}` ou com a função `set()`. Ao serem definidos, os elementos duplicados são automaticamente removidos.

```
frutas = {"maçã", "banana", "laranja", "maçã"}
print(frutas)
```

Saída:

```
{'banana', 'laranja', 'maçã'}
```

No exemplo acima, o valor `"maçã"` aparece apenas uma vez, mesmo tendo sido incluído duas vezes. Isso ocorre porque os conjuntos armazenam apenas elementos únicos.

Para criar um conjunto vazio, é necessário usar a função `set()`.

```
vazio = set()
print(vazio)
```

Saída:

```
set()
```

Adicionando e removendo elementos

Os conjuntos são mutáveis, permitindo a inserção e a remoção de itens conforme necessário. Os principais métodos são:

- `add(x)` — adiciona um elemento ao conjunto.
- `remove(x)` — remove um elemento existente (gera erro se ele não estiver presente).
- `discard(x)` — remove um elemento, sem gerar erro caso ele não exista.
- `clear()` — remove todos os elementos do conjunto, deixando-o vazio.

Exemplo:

```
participantes = {"Maria", "João", "Ana"}
participantes.add("Carlos")     # adiciona um novo nome
participantes.remove("João")    # remove o nome informado
print(participantes)
```

Saída:

```
{'Maria', 'Ana', 'Carlos'}
```

Essas operações são úteis em situações como o controle de participantes de um evento, a atualização de categorias distintas ou o gerenciamento de conjuntos de dados sem repetições.

Por exemplo, é possível incluir automaticamente novos nomes em um grupo, garantir que não existam duplicatas e remover registros que não fazem mais parte do conjunto.

Operações entre conjuntos

Os conjuntos permitem realizar operações semelhantes às usadas na matemática, o que os torna muito úteis em tarefas de comparação e análise de dados. Essas operações ajudam a identificar elementos comuns, exclusivos ou compartilhados entre grupos de informações,

como alunos matriculados em diferentes turmas, categorias de clientes ou sensores ativos em diferentes dias.

```
a = {"Maria", "João", "Ana"}
b = {"João", "Carlos", "Pedro"}

print(a | b)   # união
print(a & b)   # interseção
print(a - b)   # diferença
print(a ^ b)   # diferença simétrica
```

Saída:

```
{'Maria', 'João', 'Ana', 'Carlos', 'Pedro'}
{'João'}
{'Maria', 'Ana'}
{'Maria', 'Ana', 'Carlos', 'Pedro'}
```

Explicação das operações:

- **União (|)** — combina todos os elementos de ambos os conjuntos, sem repetições.
- **Interseção (&)** — retorna apenas os elementos que aparecem nos dois conjuntos.
- **Diferença (-)** — mostra o que está em `a` e não está em `b`.
- **Diferença simétrica (^)** — retorna os elementos que pertencem a apenas um dos conjuntos.

Essas operações são utéis para identificar relações entre grupos, eliminar redundâncias e encontrar interseções de categorias.

Por exemplo, podem ser aplicadas para comparar conjuntos de clientes ativos e inativos, detectar sobreposição de participantes em eventos ou cruzar conjuntos de dados em uma análise estatística.

Iterando sobre conjuntos

Mesmo que os conjuntos não possuam uma ordem definida, ainda é possível percorrer seus elementos usando um laço `for`. Isso permite executar ações sobre cada item do conjunto, como exibir valores, realizar contagens ou aplicar verificações de pertinência.

```
frutas = {"maçã", "banana", "laranja"}
```

```
for fruta in frutas:
    print(fruta)
```

Saída (a ordem pode variar a cada execução):

```
banana
laranja
maçã
```

Cada elemento é visitado uma vez, sem repetição, o que torna essa estrutura eficiente para percorrer coleções em que a unicidade dos dados é importante.

Aplicações práticas

Os conjuntos são úteis em diversas situações em que se deseja garantir que não existam valores repetidos ou verificar a presença de um item em um grupo. Eles também são empregados em comparações entre conjuntos de dados, como cruzar cadastros, verificar interseções de informações ou remover redundâncias em listas grandes.

Principais aplicações:

- eliminar valores duplicados em uma lista;
- verificar se um item pertence a um grupo;
- comparar e cruzar informações entre diferentes coleções.

Exemplo — eliminando duplicatas em uma lista de nomes:

```
nomes = ["Maria", "João", "Ana", "Maria", "Carlos", "Ana"]
nomes_unicos = list(set(nomes))
print(nomes_unicos)
```

Saída (a ordem pode variar):

```
['Carlos', 'Ana', 'Maria', 'João']
```

O uso do `set()` converte a lista em um conjunto, removendo automaticamente as repetições. Em seguida, a conversão de volta para lista (`list()`) reorganiza os dados em uma estrutura indexável.

Essa técnica é comum em pré-processamento de dados, como ao limpar cadastros duplicados, padronizar informações de usuários ou consolidar registros antes de uma análise estatística.

Exercício-3

Cadastro de idosos por cidade

Um centro de convivência quer analisar de quais cidades vêm os idosos atendidos. Durante a coleta de dados, podem ocorrer duplicações nos nomes das cidades. Crie um programa que:

1. Leia uma lista de cidades onde os idosos residem.
2. Converta essa lista em um conjunto para remover duplicatas.
3. Mostre a lista original e o conjunto resultante.
4. Calcule quantas cidades distintas estão representadas.

Dicionários: coleções associativas de chave-valor

Um dicionário é uma estrutura de dados que armazena informações na forma de pares **chave: valor**, em vez de depender de posições numéricas. Cada valor é identificado por uma chave única, o que permite acessar e modificar dados de forma direta e descritiva.

Diferente de listas e tuplas, em que os elementos são acessados por índices, os dicionários permitem usar palavras, números ou tuplas imutáveis como chaves, tornando o código mais claro e flexível.

Os dicionários são amplamente utilizados em aplicações de ciência de dados, pois permitem representar registros completos, armazenar configurações, mapear categorias e integrar informações em formatos estruturados como JSON.

Criando dicionários

Um dicionário é definido com chaves `{}`, contendo pares `chave: valor` separados por vírgulas. Cada chave deve ser única dentro do mesmo dicionário.

```
pessoa = {"nome": "Maria", "idade": 72, "cidade": "Goiânia"}
print(pessoa)
```

Saída:

```
{'nome': 'Maria', 'idade': 72, 'cidade': 'Goiânia'}
```

Também é possível criar um dicionário vazio e adicionar os elementos de forma incremental.

Esse método é útil quando os dados são obtidos ao longo da execução do programa, como na leitura de arquivos ou preenchimento interativo.

```
idoso = {}
idoso["nome"] = "João"
idoso["idade"] = 70
idoso["cidade"] = "Anápolis"
print(idoso)
```

Saída:

```
{'nome': 'João', 'idade': 70, 'cidade': 'Anápolis'}
```

Essa estrutura é adequada para representar registros que têm campos bem definidos, como informações pessoais, dados de sensores ou parâmetros de configuração.

Cada chave funciona como uma “etiqueta” que descreve o conteúdo armazenado, facilitando a compreensão e a manipulação dos dados.

Acessando e modificando valores

Os dicionários permitem o acesso direto aos valores por meio das suas chaves.

Isso torna a recuperação de informações mais intuitiva, pois o nome da chave descreve o dado que está sendo consultado.

```
pessoa = {"nome": "Maria", "idade": 72, "cidade": "Goiânia"}
print(pessoa["nome"])
```

Saída:

```
Maria
```

Se uma chave não existir, o Python gera um erro do tipo `KeyError`. Para evitar isso, pode-se usar o método `get()`, que permite definir um valor padrão para retornar quando a chave não for encontrada.

```
print(pessoa.get("telefone", "não informado"))
```

Saída:

```
não informado
```

Além de acessar informações, os dicionários permitem modificar ou adicionar novos pares `chave: valor` durante a execução. Quando uma chave já existe, o valor associado é substituído; quando não existe, o par é adicionado ao dicionário.

```
pessoa["idade"] = 73                # atualiza o valor existente
pessoa["telefone"] = "(62) 99999-0000"  # adiciona uma nova chave
print(pessoa)
```

Saída:

```
{'nome': 'Maria', 'idade': 73, 'cidade': 'Goiânia', 'telefone': '(62
```

Removendo elementos

Os dicionários permitem remover elementos individualmente ou de forma seletiva. A remoção pode ser feita com o método `pop()`, que exclui uma chave específica e retorna o valor associado a ela. Se a chave não existir, é possível definir um valor padrão para evitar erros.

```
pessoa = {"nome": "Maria", "idade": 73, "cidade": "Goiânia"}

removido = pessoa.pop("telefone", "não cadastrado")
print("Telefone:", removido)

pessoa.pop("cidade")
print(pessoa)
```

Saída:

```
Telefone: não cadastrado
{'nome': 'Maria', 'idade': 73}
```

Além de `pop()`, há outros métodos úteis para remoção:

- `del dicionario[chave]` — apaga o item sem retornar o valor.
- `clear()` — remove todos os elementos, deixando o dicionário vazio.

Exemplo:

```
cadastro = {"nome": "João", "idade": 70, "cidade": "Anápolis"}
del cadastro["idade"]     # remove a chave "idade"
```

```
print(cadastro)

cadastro.clear()          # limpa todo o conteúdo
print(cadastro)
```

Saída:

```
{'nome': 'João', 'cidade': 'Anápolis'}
{}
```

Iterando sobre dicionários

Percorrer um dicionário é uma forma prática de acessar e exibir suas informações, seja para gerar relatórios, processar registros ou realizar verificações. O modo mais comum é iterar diretamente sobre as chaves, usando um laço `for`.

```
pessoa = {"nome": "Maria", "idade": 73, "cidade": "Goiânia"}

for chave in pessoa:
    print(chave, ":", pessoa[chave])
```

Saída:

```
nome : Maria
idade : 73
cidade : Goiânia
```

Quando é necessário acessar chaves e valores simultaneamente, o método `items()` é o mais adequado, pois retorna ambos em pares.

```
for chave, valor in pessoa.items():
    print(chave, "→", valor)
```

Saída:

```
nome → Maria
idade → 73
cidade → Goiânia
```

Em alguns casos, é interessante percorrer o dicionário em **ordem alfabética das chaves**. Isso pode ser feito convertendo o conjunto de chaves em uma lista e aplicando a função `sorted()`.

```
for chave in sorted(pessoa.keys()):
    print(chave, ":", pessoa[chave])
```

Saída:

```
cidade : Goiânia
idade : 73
nome : Maria
```

Funções e métodos úteis

Os dicionários possuem funções e métodos específicos que facilitam o gerenciamento e a consulta de seus elementos. Esses recursos são usados em diferentes situações, como na contagem de registros, atualização de informações ou extração de dados para análise.

Método/Função	Descrição
`len(d)`	Retorna a quantidade de pares chave-valor presentes no dicionário
`keys()`	Retorna uma coleção com todas as chaves do dicionário
`values()`	Retorna uma coleção com todos os valores armazenados
`items()`	Retorna uma sequência de pares (chave, valor)
`pop(chave)`	Remove e retorna o valor associado à chave especificada
`clear()`	Remove todos os itens, deixando o dicionário vazio
`update()`	Atualiza ou adiciona vários pares de chave e valor de uma só vez

Exemplo prático:

```
dados = {"nome": "João", "idade": 70, "cidade": "Anápolis"}

print(len(dados))          # quantidade de pares
print(list(dados.keys()))  # todas as chaves
print(list(dados.values()))# todos os valores
print(list(dados.items())) # pares chave-valor
```

```
dados.update({"idade": 71, "telefone": "(62) 99999-0000"})
print(dados)
```

Saída:

```
3
['nome', 'idade', 'cidade']
['João', 70, 'Anápolis']
[('nome', 'João'), ('idade', 70), ('cidade', 'Anápolis')]
{'nome': 'João', 'idade': 71, 'cidade': 'Anápolis', 'telefone': '(62
```

Dicionários aninhados

Um dicionário pode conter outros dicionários em seu interior, criando uma estrutura hierárquica de informações. Esse formato é útil quando os dados possuem múltiplos níveis de detalhe, como registros de filiais, departamentos, centros de atendimento ou sensores distribuídos.

```
centros = {
    "Centro Norte": {"idosos": 45, "coordenador": "Ana"},
    "Vila Nova": {"idosos": 60, "coordenador": "Carlos"},
    "Setor Sul": {"idosos": 38, "coordenador": "João"}
}

for nome, dados in centros.items():
    print(f"{nome}: {dados['idosos']} idosos (coord. {dados['coorden
```

Saída:

```
Centro Norte: 45 idosos (coord. Ana)
Vila Nova: 60 idosos (coord. Carlos)
Setor Sul: 38 idosos (coord. João)
```

Nesse exemplo, cada chave principal representa um centro de convivência, e o valor associado é outro dicionário contendo os dados específicos daquele local.

Esse tipo de estrutura é bastante usado em aplicações que precisam lidar com registros compostos, como bancos de dados simulados em memória ou leitura de arquivos em formato JSON.

Cópia e referência

Os dicionários são estruturas mutáveis. Quando um dicionário é atribuído a uma nova variável, ambas passam a compartilhar o mesmo objeto na memória. Isso significa que qualquer alteração feita em uma delas se reflete na outra.

```
dados = {"nome": "Maria", "idade": 72}
referencia = dados
referencia["idade"] = 80

print(dados)      # {'nome': 'Maria', 'idade': 80}
print(referencia) # {'nome': 'Maria', 'idade': 80}
```

Nesse caso, `dados` e `referencia` apontam para o mesmo dicionário. Para criar uma cópia independente, deve-se usar o método `copy()`. A cópia é um novo objeto, e mudanças feitas nela não afetam o dicionário original.

```
dados = {"nome": "Maria", "idade": 72}
copia = dados.copy()
copia["idade"] = 80

print(dados)  # {'nome': 'Maria', 'idade': 72}
print(copia)  # {'nome': 'Maria', 'idade': 80}
```

Essa distinção entre referência e cópia é importante em programas que manipulam grandes volumes de dados, evitando alterações acidentais em coleções compartilhadas. Ela também é essencial em funções que recebem dicionários como parâmetros, pois qualquer modificação interna pode afetar o objeto original se não houver cópia explícita.

Atualizando informações com `.update()`

O método `.update()` é usado para adicionar novas chaves ou alterar valores existentes em um dicionário. Ele combina o conteúdo de outro dicionário, incorporando ou substituindo informações conforme necessário. Essa operação é muito prática para atualizar registros ou mesclar dados vindos de diferentes fontes.

```
idoso = {"nome": "João", "idade": 68}
atualizacao = {"idade": 69, "cidade": "Anápolis"}

idoso.update(atualizacao)
print(idoso)
```

Saída:

```
{'nome': 'João', 'idade': 69, 'cidade': 'Anápolis'}
```

No exemplo acima, o valor da chave `"idade"` foi atualizado e a nova chave `"cidade"` foi adicionada. Esse método é frequentemente utilizado em rotinas de integração de dados, onde registros precisam ser sincronizados ou complementados com informações externas.

Verificando chaves

Antes de acessar uma chave em um dicionário, é importante verificar se ela existe para evitar erros. O operador `in` realiza essa verificação de forma simples e eficiente.

```
pessoa = {"nome": "Ana", "idade": 75}
if "idade" in pessoa:
    print("Idade registrada:", pessoa["idade"])
```

Saída:

```
Idade registrada: 75
```

Essa técnica é útil em situações em que os dados podem estar incompletos ou vir de diferentes origens, como formulários, planilhas ou arquivos JSON.

Compreensão de dicionário

A compreensão de dicionário é uma forma compacta de criar ou transformar dicionários a partir de listas, tuplas ou outros dicionários. Ela combina a lógica de um laço `for` e uma expressão de atribuição em uma única linha, tornando o código mais legível e direto.

```
nomes = ["Maria", "João", "Ana"]
idades = [72, 68, 75]
dados = {nomes[i]: idades[i] for i in range(len(nomes))}
print(dados)
```

Saída:

```
{'Maria': 72, 'João': 68, 'Ana': 75}
```

No exemplo, o dicionário `dados` é construído associando cada nome à respectiva idade. Essa técnica é útil para transformar dados tabulares em estruturas associativas que facilitam consultas e análises.

Outro exemplo: criar um dicionário que relacione cada nome ao dobro da idade.

```
dobro = {nome: idade * 2 for nome, idade in dados.items()}
print(dobro)
```

Saída:

```
{'Maria': 144, 'João': 136, 'Ana': 150}
```

Esse tipo de construção é usado em análises e pré-processamentos de dados, especialmente quando é necessário aplicar cálculos ou filtros antes de consolidar informações em um formato de dicionário.

Conversão entre dicionários e listas

Em muitas situações, é necessário converter um dicionário em listas para manipulação de dados, especialmente quando se trabalha com estruturas tabulares, planilhas ou bibliotecas de análise como o pandas. Essa conversão permite acessar separadamente as chaves e os valores, facilitando a visualização e o processamento das informações.

```
pessoa = {"nome": "Maria", "idade": 72, "cidade": "Goiânia"}
chaves = list(pessoa.keys())
valores = list(pessoa.values())

print(chaves)
print(valores)
```

Saída:

```
['nome', 'idade', 'cidade']
['Maria', 72, 'Goiânia']
```

O método `keys()` retorna todas as chaves e `values()` retorna todos os valores. Essa separação é útil, por exemplo, para transformar dados de um dicionário em colunas de uma tabela ou em linhas de um arquivo CSV.

Integração com JSON

O formato **JSON (JavaScript Object Notation)** é amplamente usado para armazenar e transferir dados entre sistemas, por ser leve e fácil de interpretar. Em Python, os dicionários

se integram naturalmente com JSON, pois ambos seguem a mesma estrutura de chave e valor.

Para converter um dicionário em texto JSON, usa-se a função `json.dumps()`. Para fazer o caminho inverso — ler o texto e reconstruir o dicionário — usa-se `json.loads()`.

```
import json

dados = {"nome": "Maria", "idade": 72, "cidade": "Goiânia"}
json_texto = json.dumps(dados, ensure_ascii=False)
print(json_texto)

# Convertendo de volta
recuperado = json.loads(json_texto)
print(recuperado["nome"])
```

Saída:

```
{"nome": "Maria", "idade": 72, "cidade": "Goiânia"}
Maria
```

Exercício-4

Sistema de registro de idosos

Um pesquisador quer criar um sistema simples de registro de idosos usando dicionários. Cada registro deve armazenar o nome, idade, cidade e número de atividades realizadas.

O programa deve:

1. Permitir cadastrar pelo menos 3 idosos em um dicionário principal. Cada idoso é identificado pela chave “nome”.
2. Mostrar todos os registros.
3. Permitir buscar um idoso pelo nome e exibir seus dados.
4. Calcular e mostrar a média das idades cadastradas.
5. Permitir atualizar a cidade de um idoso.

Fatiamento em Coleções Python

Durante o processamento de dados, é comum precisar analisar apenas uma parte de uma coleção em vez de trabalhar com todos os elementos.

Em situações como análise de séries temporais, extração de registros específicos ou comparação de subconjuntos de dados, copiar apenas um trecho da estrutura é mais eficiente e organizado do que duplicar ou percorrer tudo manualmente.

Essa técnica é chamada de fatiamento (*slicing*) e é aplicada a várias estruturas em Python — como tuplas, listas, strings e, em menor grau, a dicionários (de forma indireta, pois eles não são indexados).

O fatiamento usa a sintaxe geral:

```
colecao[início:fim:passo]
```

- `início`: posição do primeiro elemento incluído (padrão = início da coleção).
- `fim`: posição onde o corte termina (não inclusiva).
- `passo`: intervalo entre os elementos (padrão = 1).

---

1) Fatiamento em tuplas

As tuplas são imutáveis, o que significa que não podem ser alteradas depois de criadas. O fatiamento é útil para selecionar partes específicas sem modificar a original, criando uma nova tupla com os elementos desejados.

Exemplo:

```
idades = (60, 62, 65, 68, 70, 72)

# Criando fragmentos da tupla original
inicio = idades[:3]     # primeiros três valores
final = idades[3:]      # últimos três valores
intervalo = idades[::2] # um valor a cada dois
print(inicio, final, intervalo)
```

Saída:

```
(60, 62, 65) (68, 70, 72) (60, 65, 70)
```

Exemplo Prático:

Um pesquisador tem as idades de idosos cadastrados em um programa de saúde e quer analisar apenas uma faixa etária específica.

```
idades = (60, 62, 65, 68, 70, 72, 74, 75, 77)
```

```
# Selecionando subconjuntos de interesse
faixa_inicial = idades[:4]      # primeiros quatro registros
faixa_intermediaria = idades[4:7]  # valores centrais
faixa_final = idades[-3:]       # últimos registros

print("Faixa inicial:", faixa_inicial)
print("Faixa intermediária:", faixa_intermediaria)
print("Faixa final:", faixa_final)
```

Saída:

```
Faixa inicial: (60, 62, 65, 68)
Faixa intermediária: (70, 72, 74)
Faixa final: (74, 75, 77)
```

---

2) Fatiamento em listas

As listas permitem as mesmas operações de fatiamento, mas, por serem mutáveis, é possível alterar ou substituir os fragmentos diretamente. Isso é útil em processos de atualização, filtragem e reamostragem de dados.

Exemplo:

```
temperaturas = [32, 33, 31, 30, 29, 28, 27]

# Analisando subconjuntos
manhã = temperaturas[:3]
tarde = temperaturas[3:5]
noite = temperaturas[5:]

# Alterando uma fatia
temperaturas[2:5] = [35, 36, 37]

print(manhã, tarde, noite)
print("Lista modificada:", temperaturas)
```

Saída:

```
[32, 33, 31] [30, 29] [28, 27]
Lista modificada: [32, 33, 35, 36, 37, 28, 27]
```

Exemplo Prático

Um sistema de monitoramento registrou temperaturas ao longo do dia, mas algumas leituras estão incorretas e precisam ser ajustadas.

```
temperaturas = [22, 23, 25, 30, 29, 28, 26, 24]

# Corrigindo leituras incorretas da tarde (índices 3 a 5)
temperaturas[3:6] = [27, 27, 26]

# Selecionando apenas manhã (0 a 3) e noite (6 em diante)
manha = temperaturas[:3]
noite = temperaturas[6:]

print("Temperaturas ajustadas:", temperaturas)
print("Manhã:", manha)
print("Noite:", noite)
```

Saída:

```
Temperaturas ajustadas: [22, 23, 25, 27, 27, 26, 26, 24]
Manhã: [22, 23, 25]
Noite: [26, 24]
```

---

3) Fatiamento em Strings

Strings em Python funcionam como sequências de caracteres e também podem ser fatiadas. Esse recurso é útil para extrair partes de textos, códigos, datas ou identificadores, sem precisar de funções adicionais.

Exemplo Prático

Um sistema de identificação usa códigos no formato `ID2025GO001`, onde cada parte tem um significado:

- `ID` → tipo de dado
- `2025` → ano
- `GO` → estado
- `001` → número sequencial

```
codigo = "ID2025GO0001"

tipo = codigo[:2]
ano = codigo[2:6]
estado = codigo[6:8]
numero = codigo[8:]

print("Tipo:", tipo)
print("Ano:", ano)
print("Estado:", estado)
print("Número:", numero)
```

Saída:

```
Tipo: ID
Ano: 2025
Estado: GO
Número: 001
```

---

4) Fatiamento em Dicionários

Os dicionários não são indexados por posição, então o fatiamento direto (`dicionario[1:3]`) não é possível. Entretanto, pode-se converter as chaves ou valores em listas e aplicar o fatiamento sobre elas. Essa abordagem é usada para criar subconjuntos de registros ou limitar a visualização de dados.

```
dados = {"Maria": 72, "João": 68, "Ana": 75, "Carlos": 70, "Paula":

# Convertendo em listas
nomes = list(dados.keys())
idades = list(dados.values())

# Selecionando os três primeiros registros
print(nomes[:3])
print(idades[:3])
```

Saída:

```
['Maria', 'João', 'Ana']
[72, 68, 75]
```

Exemplo Prático

Um centro de convivência deseja gerar relatórios apenas dos primeiros registros de uma base de idosos.

```
idosos = {"Maria": 72, "João": 68, "Ana": 75, "Carlos": 70, "Paula":

# Convertendo em listas
nomes = list(idosos.keys())
idades = list(idosos.values())

# Selecionando os três primeiros registros
nomes_parciais = nomes[:3]
idades_parciais = idades[:3]

for i in range(len(nomes_parciais)):
    print(f"{nomes_parciais[i]}: {idades_parciais[i]} anos")
```

Saída:

```
Maria: 72 anos
João: 68 anos
Ana: 75 anos
```

Criação de coleções a partir de outras

Em muitas aplicações, é comum gerar uma nova coleção a partir de outra existente. Essa operação é importante quando se deseja filtrar, transformar ou reorganizar dados sem alterar os originais.

Python oferece maneiras diretas e eficientes de fazer isso em todas as coleções principais: tuplas, listas, conjuntos e dicionários.

---

1) Criando listas a partir de tuplas

As listas são mutáveis, enquanto as tuplas são imutáveis. Converter uma tupla em lista é útil quando se deseja modificar ou expandir os dados sem perder sua estrutura original.

Exemplo

Um sistema recebe uma tupla com idades registradas, mas o pesquisador precisa ajustar os valores para análise.

```
idades_tupla = (60, 62, 65, 68, 70)

# Conversão para lista
idades_lista = list(idades_tupla)

# Aplicando transformação
idades_lista = [idade + 1 for idade in idades_lista]  # simula novo

print("Tupla original:", idades_tupla)
print("Lista atualizada:", idades_lista)
```

Saída:

```
Tupla original: (60, 62, 65, 68, 70)
Lista atualizada: [61, 63, 66, 69, 71]
```

Esse processo é útil quando dados imutáveis precisam ser ajustados ou corrigidos sem comprometer o registro original.

---

2) Criando conjuntos a partir de listas

Os conjuntos (`set`) eliminam automaticamente valores duplicados. Converter uma lista em conjunto é uma forma prática de limpar dados repetidos ou identificar valores únicos.

Exemplo

Um levantamento de participantes foi feito em diferentes dias e contém repetições de nomes.

```
nomes_lista = ["Maria", "João", "Ana", "Maria", "Carlos", "Ana"]
nomes_unicos = set(nomes_lista)

print("Lista original:", nomes_lista)
print("Conjunto sem duplicatas:", nomes_unicos)
```

Saída:

```
Lista original: ['Maria', 'João', 'Ana', 'Maria', 'Carlos', 'Ana']
```

```
Conjunto sem duplicatas: {'Carlos', 'João', 'Ana', 'Maria'}
```

A conversão simplifica a limpeza de dados, garantindo que cada valor apareça apenas uma vez.

---

3) Criando tuplas ou listas a partir de dicionários

Os dicionários armazenam dados em pares de chave e valor. É possível extrair apenas as chaves, apenas os valores ou ambos para formar novas coleções.

Exemplo

Um centro de convivência mantém um dicionário com nomes e idades. O pesquisador deseja gerar uma lista de idades e uma tupla de nomes para análise estatística.

```
idosos = {"Maria": 72, "João": 68, "Ana": 75, "Carlos": 70}

nomes = tuple(idosos.keys())
idades = list(idosos.values())

print("Nomes:", nomes)
print("Idades:", idades)
```

Saída:

```
Nomes: ('Maria', 'João', 'Ana', 'Carlos')
Idades: [72, 68, 75, 70]
```

Essas coleções podem ser usadas em gráficos, cálculos ou visualizações sem precisar manipular diretamente o dicionário.

---

4) Criando dicionários a partir de listas

A função `zip()` permite combinar duas listas — uma de chaves e outra de valores — para formar um dicionário. Isso é útil para transformar dados de diferentes fontes em um formato associativo.

Exemplo

Um sistema coleta nomes e idades em listas separadas. Com `zip()`, é possível criar um dicionário para relacionar cada nome à respectiva idade.

```
nomes = ["Maria", "João", "Ana"]
idades = [72, 68, 75]

dados = dict(zip(nomes, idades))
print(dados)
```

Saída:

```
{'Maria': 72, 'João': 68, 'Ana': 75}
```

Essa conversão é comum ao importar dados de planilhas, em que colunas diferentes precisam ser associadas.

Gerando novas coleções com compreensão

Durante a manipulação de dados, é comum precisar criar uma nova coleção a partir de uma já existente, aplicando filtros, cálculos ou transformações sobre seus elementos.

Uma forma tradicional de fazer isso seria usar um laço `for` para percorrer os itens e construir uma nova lista manualmente. Porém, o Python oferece uma forma mais direta e legível: as compreensões (*comprehensions*).

As compreensões permitem criar coleções em uma única linha, combinando um laço `for`, uma condição opcional e uma expressão de transformação. Elas podem ser aplicadas a listas, conjuntos e dicionários.

A estrutura geral é:

```
nova_colecao = [expressao for item in colecao if condicao]
```

- **expressao**: define o valor que será incluído na nova coleção.
- **item**: representa cada elemento da coleção original.
- **condicao (opcional)**: filtra quais itens serão incluídos.

---

1) Compreensão de listas

As compreensões de lista são as mais comuns. Permitem criar listas filtradas ou transformadas com pouco código.

Exemplo

Um pesquisador tem uma lista de idades coletadas em um levantamento e deseja gerar uma nova lista apenas com os idosos acima de 70 anos.

```
idades = [68, 72, 75, 69, 80, 65]

idosos = [idade for idade in idades if idade > 70]
print(idosos)
```

Saída:

```
[72, 75, 80]
```

A lista `idosos` é construída automaticamente, sem precisar de laços ou condicionais explícitos.

Exemplo

Em vez de filtrar, também é possível transformar os valores durante a criação. O exemplo abaixo cria uma nova lista com as idades acrescidas de 5 anos, simulando uma projeção futura.

```
idades = [68, 72, 75, 69, 80, 65]
projecao = [idade + 5 for idade in idades]
print(projecao)
```

Saída:

```
[73, 77, 80, 74, 85, 70]
```

Essa técnica é usada em análises de tendências e projeções em bases de dados numéricos.

---

2) Compreensão de conjuntos

Quando o objetivo é obter apenas valores únicos, usa-se a mesma lógica entre chaves `{}`. A compreensão de conjunto remove duplicatas automaticamente.

Exemplo

```
categorias = ["A", "B", "C", "A", "B", "D"]
distintas = {c for c in categorias}
```

```
print(distintas)
```

Saída:

```
{'C', 'B', 'A', 'D'}
```

Nesse caso, o conjunto é criado com os elementos distintos da lista original, útil para identificar faixas, grupos ou classes únicas em amostras.

---

3) Compreensão de dicionário

As compreensões também podem criar dicionários, associando chaves e valores derivados de uma coleção existente. Elas são úteis para converter dados tabulares em estruturas associativas.

Exemplo

Um pesquisador possui duas listas separadas e deseja combiná-las em um dicionário.

```
nomes = ["Maria", "João", "Ana"]
idades = [72, 68, 75]
dados = {nomes[i]: idades[i] for i in range(len(nomes))}
print(dados)
```

Saída:

```
{'Maria': 72, 'João': 68, 'Ana': 75}
```

Cada nome é associado à idade correspondente. Essa técnica é amplamente usada para consolidar dados importados de planilhas ou arquivos CSV.

Exemplo

Com base no dicionário criado, é possível gerar outro que relacione cada nome ao dobro da idade:

```
dobro = {nome: idade * 2 for nome, idade in dados.items()}
print(dobro)
```

Saída:

```
{'Maria': 144, 'João': 136, 'Ana': 150}
```

Esse tipo de operação é útil em cálculos de pontuação, projeções ou ajustes de variáveis durante o pré-processamento de dados.

---

4) Compreensão aninhada

As compreensões também podem ser combinadas, permitindo criar coleções bidimensionais (como listas de listas). Isso é usado quando os dados têm estrutura em forma de grade ou tabela.

Exemplo

Um pesquisador quer simular um conjunto de medições de pressão arterial (sistólica e diastólica) em três dias.

```
pressao = [(120 + i, 80 + i) for i in range(3)]
print(pressao)
```

Saída:

```
[(120, 80), (121, 81), (122, 82)]
```

Cada tupla representa uma medição diferente. Essa abordagem simplifica a geração de dados simulados ou séries estruturadas.

Resumo - Comparativo das Coleções em Python

A tabela abaixo resume as principais características das coleções integradas do Python.

Estrutura	Mutabilidade	Ordenação	Indexação	Permite Duplicatas?
Tupla	Imutável	Ordenada	Indexada	Sim
Lista	Mutável	Ordenada	Indexada	Sim
Set	Mutável	Não ordenada	Não indexada	Não
Dicionário	Mutável	Não ordenada*	Indexado por chave	Não (chaves únicas)

* Desde o Python 3.7, os dicionários preservam a ordem de inserção, mas conceitualmente continuam sendo tratados como não ordenados.

Observações finais

- **Tuplas** são ideais para representar registros fixos e imutáveis.
- **Listas** são usadas quando há necessidade de alterar, inserir ou remover elementos.
- **Sets** são úteis para eliminar duplicatas e realizar operações de comparação entre grupos.
- **Dicionários** são a melhor opção para armazenar dados nomeados, como cadastros e configurações.