Aplicações e Manipulação de Listas em Python

Explore técnicas avançadas de programação em Python, desde operações básicas com listas até estruturas de dados complexas como dicionários e tuplas.

Começar

Ver Exemplos

Seleção e Cópia de Elementos

Uma operação comum em programação é selecionar elementos de uma lista e copiá-los para outras listas baseado em critérios específicos. Por exemplo, podemos separar números pares e ímpares em listas diferentes.

O programa utiliza um loop for para percorrer cada elemento, verificando se é par ou ímpar usando o operador módulo (%). Elementos pares são adicionados à lista P, enquanto ímpares vão para a lista I.

```
V=[9,8,7,12,0,13,21]
P=[]
I=[]
for e in V:
    if e % 2 == 0:
        P.append(e)
    else:
        I.append(e)
print("Pares: ", P)
print("Impares: ", I)
```

Sistema de Controle de Cinema

Vejamos um programa prático que controla a utilização das salas de um cinema. A lista `Salas = [10,2,1,3,0]` contém o número de lugares vagos nas salas 1, 2, 3, 4 e 5, respectivamente.

01

Entrada de Dados

O programa solicita o número da sala e a quantidade de lugares desejados

02

Validação

Verifica se a sala existe e se há lugares disponíveis

03

Processamento

Atualiza o número de lugares livres após a venda

04

Relatório

Exibe a utilização atual de todas as salas

Listas com Strings

Conceito Fundamental

Strings em Python podem ser indexadas letra por letra, assim como listas. Cada elemento de uma lista pode ser uma string completa, permitindo operações sofisticadas de manipulação de texto.

Por exemplo, `S=["maçãs", "peras", "kiwis"]` cria uma lista com três strings, onde cada uma pode ser acessada individualmente.

```
S=["maçãs", "peras", "kiwis"]
print(len(S))  # 3
print(S[0])    # maçãs
print(S[1])    # peras
print(S[2])    # kiwis
```

Lista de Compras Interativa

```
compras=[]
while True:
 produto=input("Produto:")
 if produto == "fim":
 break
 compras.append(produto)
for p in compras:
 print(p)
```

Acessando Letras Individuais

Um recurso poderoso é acessar strings dentro de listas letra por letra usando um segundo índice. Isso permite manipulações complexas de texto.

Primeiro Índice

S[0] acessa o primeiro elemento da lista: "maçãs"

Segundo Índice

S[0][0] acessa a primeira letra do primeiro elemento: "m"

Combinações

S[1][1] retorna "e" de "peras", S[2][2] retorna "w" de "kiwis"

```
L=["maçãs", "peras", "kiwis"]
for s in L:
    for letra in s:
        print(letra)
```

Listas Dentro de Listas

Python permite criar listas que contêm outras listas como elementos. Além disso, os elementos de uma lista não precisam ser do mesmo tipo, oferecendo grande flexibilidade.

Produto 1

```
produto1 = [
  "maçã",
  10,
  0.30
]
```

String, inteiro, float

Produto 2

```
produto2 = [
  "pera",
  5,
  0.75
]
```

Tipos misturados

Produto 3

```
produto3 = [
  "kiwi",
  4,
  0.98
]
```

Estrutura flexível

Lista de Listas: Podemos criar uma lista chamada `compras` que contém produto1, produto2 e produto3 como elementos, criando uma estrutura bidimensional.

Sistema Completo de Lista de Compras

Um programa completo capaz de perguntar nome do produto, quantidade e preço, e no final imprimir uma lista de compras com o total.

```
compras = []
while True:
    produto = input("Produto: ")
    if produto == "fim":
        break
    quantidade = int(input("Quantidade: "))
    preço = float(input("Preço: "))
    compras.append([produto, quantidade, preço])

soma = 0.0
for e in compras:
    print("%20s x%5d %5.2f %6.2f" %
          (e[0], e[1], e[2], e[1] * e[2]))
    soma += e[1] * e[2]
print("Total: %7.2f" % soma)
```

O programa utiliza formatação de strings para criar uma saída organizada, calculando o subtotal de cada item e o total geral da compra.

Ordenação: Método Bubble Sort

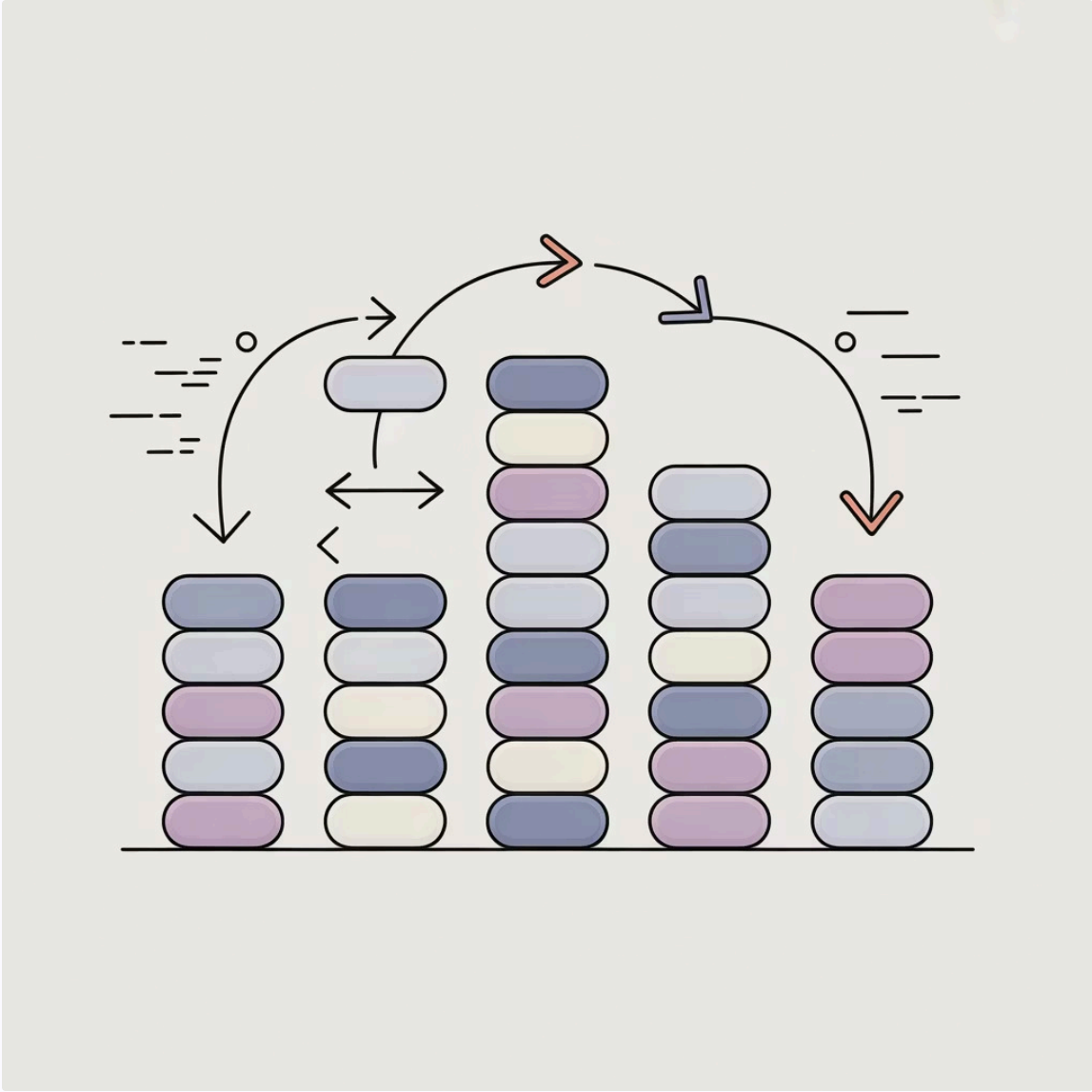
Conceito

O Bubble Sort é um algoritmo simples de ordenação que compara dois elementos por vez. Se o primeiro for maior que o segundo, eles trocam de posição.

Embora seja fácil de entender, é lento e não deve ser usado com listas grandes. Sua principal vantagem é didática.

Propriedades

- Posiciona o maior elemento na última posição a cada passagem
- Usa um marcador para detectar quando a lista está ordenada
- Elimina um elemento do fim a cada passagem completa



Implementação do Bubble Sort

```
L=[7,4,3,12,8]
fim=5
while fim > 1:
    trocou=False
    x=0
    while x<(fim-1):
        if L[x] > L[x+1]:
            trocou=True
            temp=L[x]
            L[x]=L[x+1]
            L[x+1]=temp
        x+=1
    if not trocou:
        break
    fim-=1
for e in L:
    print(e)
```

1

Inicialização

Define `fim` como tamanho da lista

2

Comparação

Compara elementos adjacentes

3

Troca

Inverte posições se necessário

4

Verificação

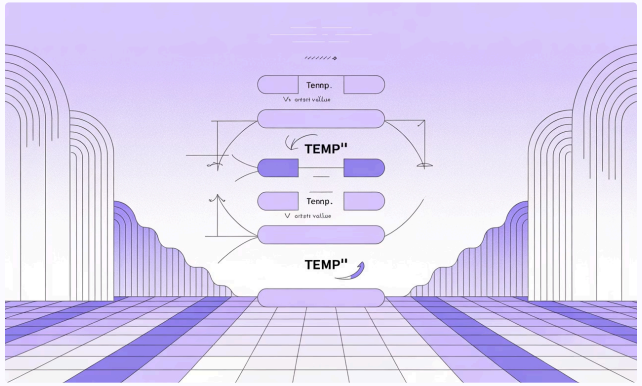
Para se não houver trocas

Processo de Troca de Valores

A troca de valores entre duas variáveis requer uma terceira variável temporária, pois cada variável só pode guardar um valor por vez.

Etapa 1: Salvar Primeiro Valor

`temp = L[x]` - Armazena o valor atual na variável temporária

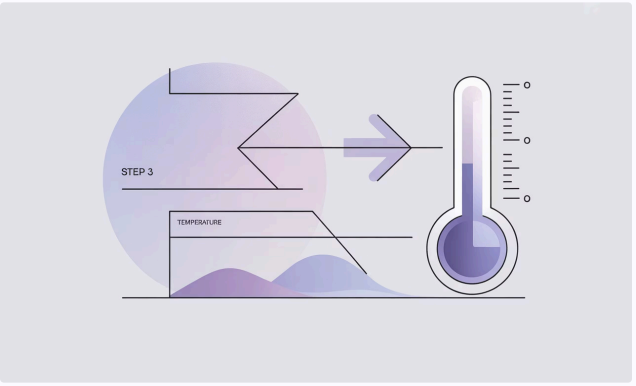


Etapa 2: Copiar Segundo Valor

`L[x] = L[x+1]` - Move o próximo valor para a posição atual

Etapa 3: Completar Troca

`L[x+1] = temp` - Coloca o valor salvo na segunda posição



Analogia das Xícaras: Imagine trocar o conteúdo de duas xícaras (uma com café, outra com leite). Você precisaria de uma terceira xícara temporária para facilitar a operação!

Dicionários em Python

Dicionários são estruturas de dados similares às listas, mas com propriedades de acesso diferentes. Um dicionário relaciona chaves a valores específicos, permitindo acesso rápido e eficiente.

Criação de Dicionários

Utilizamos chaves `{}` para criar dicionários. Cada elemento é uma combinação de chave e valor.

```
tabela = {
    "Alface": 0.45,
    "Batata": 1.20,
    "Tomate": 2.30,
    "Feijão": 1.50
}
```

Acesso por Chaves

Diferente de listas que usam índices numéricos, dicionários usam suas chaves como índice.

```
print(tabela["Alface"])
# Retorna: 0.45

tabela["Tomate"] = 2.50
# Atualiza o valor

tabela["Cebola"] = 1.20
# Adiciona nova chave
```

Operações com Dicionários

1

Verificar Existência

Use o operador `in` para verificar se uma chave existe

```
print("Manga" in tabela)  # False
print("Batata" in tabela) # True
```

2

Obter Chaves

O método `keys()` retorna todas as chaves do dicionário

```
print(tabela.keys())
# dict_keys(['Batata', 'Alface'...])
```

3

Obter Valores

O método `values()` retorna todos os valores associados

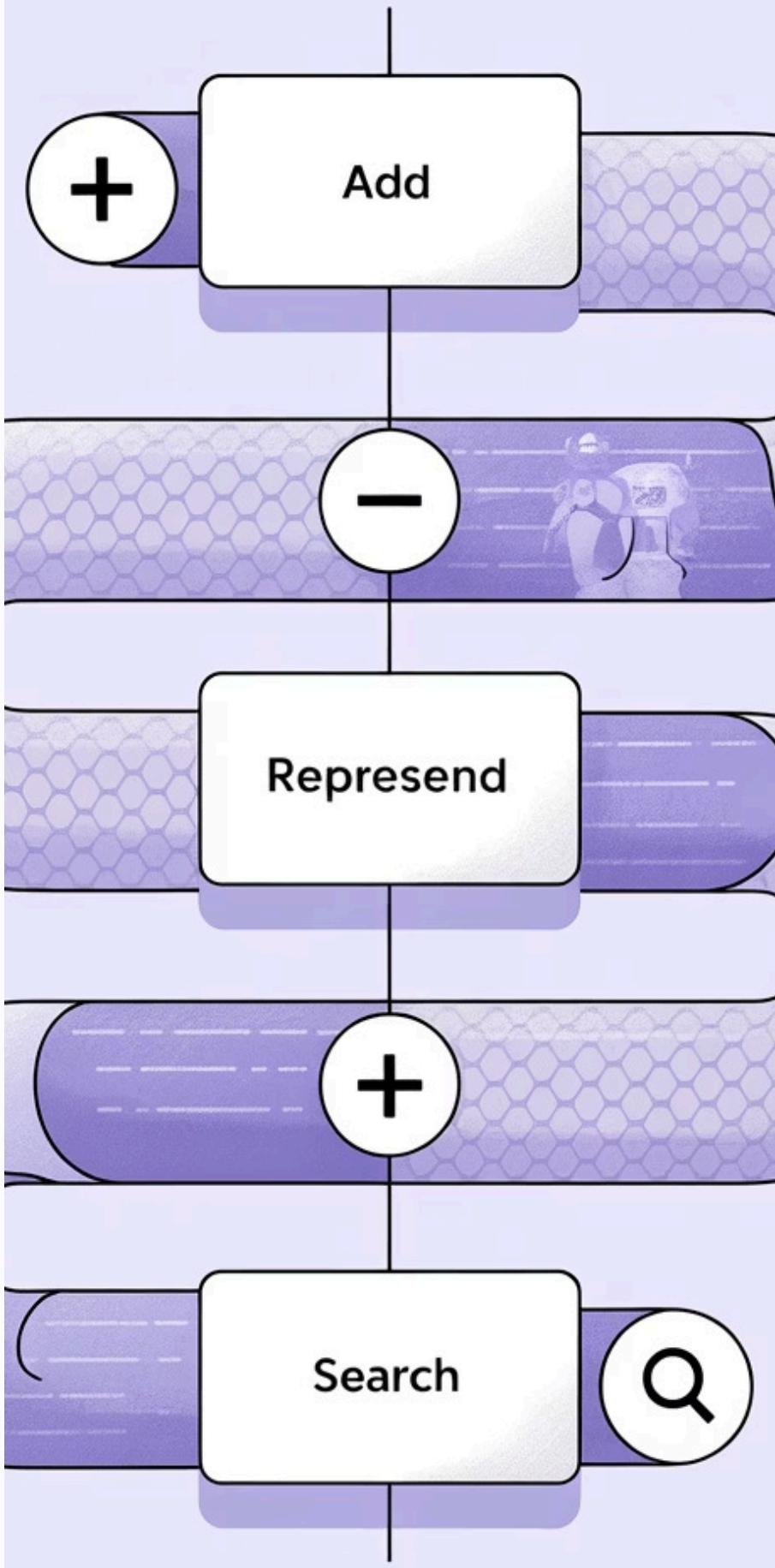
```
print(tabela.values())
# dict_values([1.2, 0.45...])
```

4

Excluir Elementos

Use a instrução `del` para remover uma associação

```
del tabela["Tomate"]
# Remove a chave e seu valor
```



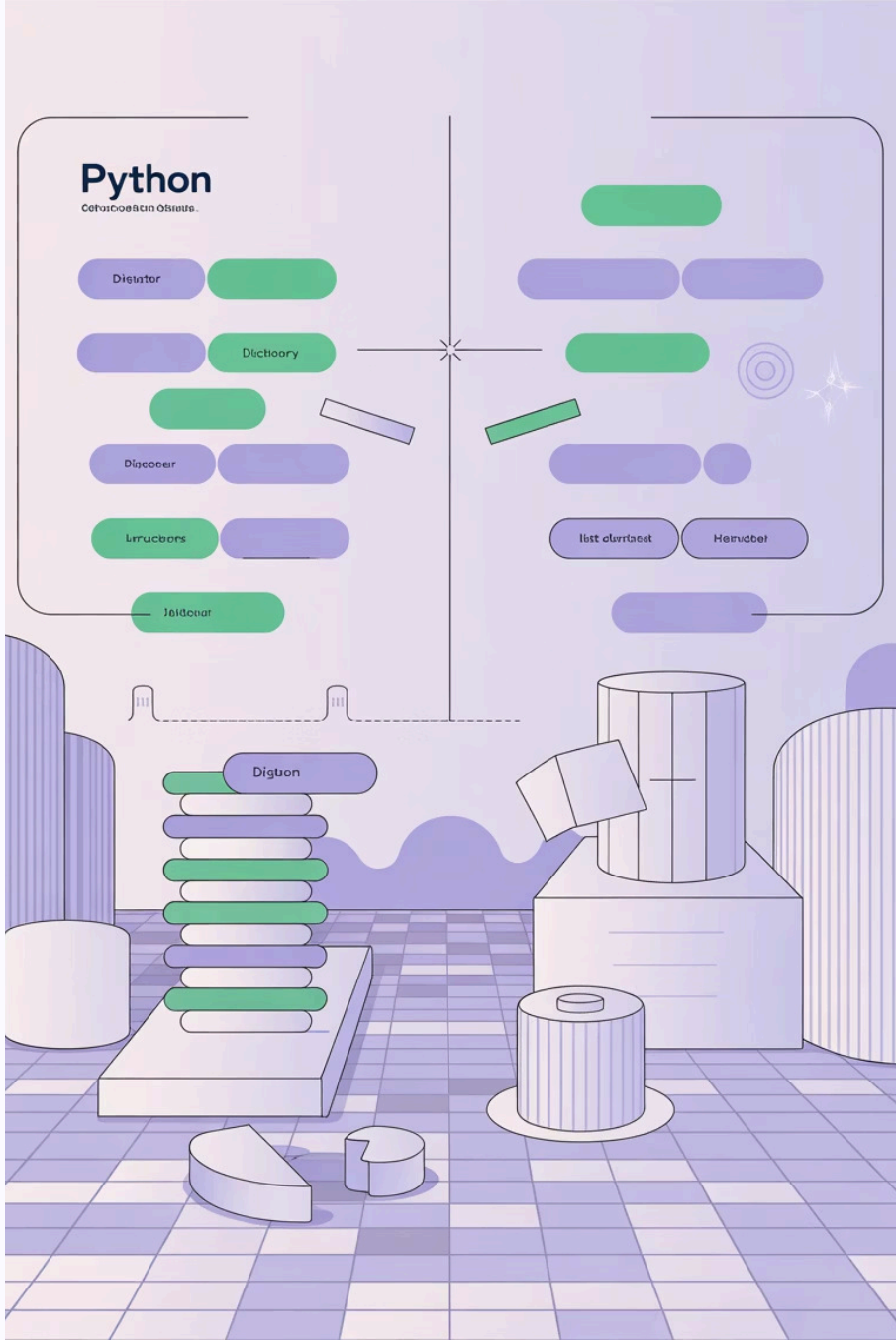
Quando Usar Dicionários vs Listas

Dicionários

- Dados acessados por chaves específicas
- Acesso rápido sem necessidade de pesquisa
- Boa velocidade com muitas chaves
- Ordem de inserção não é importante
- Exemplo: tabela de preços, cadastros

Listas

- Dados precisam preservar ordem
- Acesso sequencial é comum
- Estruturas como filas ou pilhas
- Índices numéricos fazem sentido
- Exemplo: histórico, sequências



Dicionários com Listas

Em Python, podemos ter dicionários onde as chaves são associadas a listas ou mesmo a outros dicionários, criando estruturas de dados complexas e poderosas.

```
estoque = {
    "tomate": [1000, 2.30],
    "alface": [500, 0.45],
    "batata": [2001, 1.20],
    "feijão": [100, 1.50]
}
```

Neste exemplo, cada chave (nome do produto) está associada a uma lista com dois elementos: quantidade em estoque e preço unitário.

Estrutura

Chave: nome do produto
Valor: [quantidade, preço]

Acesso

`estoque["tomate"][0]` retorna quantidade
`estoque["tomate"][1]` retorna preço

Operações

Processar vendas, atualizar estoque, calcular totais

Sistema de Vendas com Estoque

Um programa completo que processa vendas, calcula o total e atualiza o estoque automaticamente.

```
estoque = {
    "tomate": [1000, 2.30],
    "alface": [500, 0.45],
    "batata": [2001, 1.20],
    "feijão": [100, 1.50]
}

venda = [
    ["tomate", 5],
    ["batata", 10],
    ["alface", 5]
]

total = 0
for operação in venda:
    produto, quantidade = operação
    preço = estoque[produto][1]
    custo = preço * quantidade
    estoque[produto][0] -= quantidade
    total += custo
```

Recursos do Sistema

- **Desempacotamento:** `produto, quantidade = operação` simplifica o acesso aos dados
- **Cálculo automático:** Multiplica preço por quantidade
- **Atualização de estoque:** Subtrai quantidade vendida
- **Total acumulado:** Soma todos os custos

O método `items()` retorna tuplas com chave e valor para iteração eficiente.

Tuplas: Listas Imutáveis

Tuplas são estruturas de dados similares às listas, mas com a importante diferença de serem imutáveis. São ideais para representar valores constantes e realizar operações de empacotamento e desempacotamento.

Imutabilidade

Uma vez criada, uma tupla não pode ter seus elementos alterados

Sintaxe

Criadas com parênteses: `tupla = ("a", "b", "c")`

Performance

Mais rápidas que listas para operações de leitura

Uso

Ideais para dados constantes e retorno de múltiplos valores

Operações com Tuplas

Operações Suportadas

```
tupla = ("a", "b", "c")

# Indexação
print(tupla[0])  # 'a'
print(tupla[2])  # 'c'

# Fatiamento
print(tupla[1:])  # ('b', 'c')

# Repetição
print(tupla * 2)
# ('a','b','c','a','b','c')

# Comprimento
print(len(tupla))  # 3
```

Tentativa de Alteração

```
tupla[0] = "A"
# TypeError: 'tuple' object
# does not support item
# assignment
```

Iteração

```
for elemento in tupla:
    print(elemento)
# a
# b
# c
```

Tuplas suportam a maioria das operações de leitura, mas não de escrita

Empacotamento e Desempacotamento

Python permite criar tuplas usando valores separados por vírgula, independente de usar parênteses. Essa operação é chamada de empacotamento.

1 Empacotamento

```
tupla = 100, 200, 300
# (100, 200, 300)
```

Valores separados por vírgula são automaticamente convertidos em tupla

2 Desempacotamento

```
a, b = 10, 20
# a = 10
# b = 20
```

Distribui valores de uma tupla em várias variáveis

3 Troca Rápida

```
a, b = b, a
# Troca valores sem
# variável temporária
```

Atribuições simultâneas permitem trocas elegantes

Casos Especiais de Tuplas

1

Tupla com Um Elemento

Requer vírgula para indicar que é uma tupla

```
t1 = (1)    # Não é tupla: 1
t2 = (1,)   # Tupla: (1,)
t3 = 1,     # Tupla: (1,)
```

2

Tupla Vazia

Criada apenas com parênteses

```
t4 = ()
print(t4)      # ()
print(len(t4)) # 0
```

3

Conversão de Lista

Use a função `tuple()`

```
L = [1, 2, 3]
T = tuple(L)
print(T)  # (1, 2, 3)
```

4

Concatenação

Gera novas tuplas sem alterar as originais

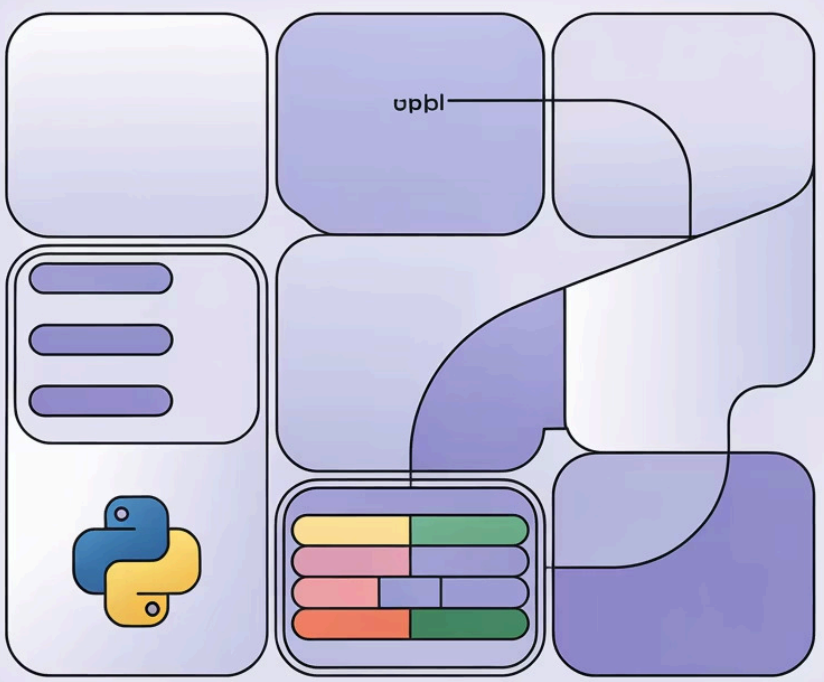
```
t1 = (1, 2, 3)
t2 = (4, 5, 6)
print(t1 + t2)
# (1, 2, 3, 4, 5, 6)
```

Tuplas com Objetos Mutáveis

Embora não possamos alterar uma tupla depois de sua criação, se ela contiver uma lista ou outro objeto mutável, este continuará funcionando normalmente.

Exemplo Prático

```
tupla = ("a", ["b", "c", "d"])

print(tupla)
# ('a', ['b', 'c', 'd'])

print(len(tupla))
# 2

print(tupla[1])
# ['b', 'c', 'd']

tupla[1].append("e")

print(tupla)
# ('a', ['b', 'c', 'd', 'e'])
```



Conceito Importante

A tupla em si não foi alterada - ela ainda contém os mesmos dois elementos: uma string e uma lista. O que mudou foi o **conteúdo** da lista que é o segundo elemento da tupla.

Imutabilidade da Tupla: A estrutura da tupla permanece constante, mas objetos mutáveis dentro dela podem ser modificados.