Neste capítulo estudaremos uma estrutura de dados, ou seja, uma forma estruturada de armazenar múltiplos dados. Você provavelmente já estudou pelo menos outra estrutura de dados em Python: a **lista**.

Vamos brevemente revisar os conceitos de lista, pois eles serão úteis para compreender nossa nova estrutura, a **tupla**.

Focaremos em funcionalidades, não em funções prontas e métodos de lista.

**1. Revisão de listas**

**1.1. Criando uma lista e acessando elementos**

A lista é uma coleção de objetos em Python. Criando uma única variável para representar a lista, podemos armazenar múltiplos valores. Internamente, esses valores são representados por seus **índices**: um número inteiro, iniciando em zero e incrementando com passo unitário.

Podemos criar uma lista através da função list ou utilizando um par de colchetes:

lista1 = list() # uma lista vazia

lista2 = [] # outra lista vazia

linguagens = ['Python', 'JavaScript', 'SQL'] # uma lista com 3 elementos

dados\_variados = [3.14, 1000, True, 'abacate'] # uma lista com dados de tipos diversos

lista\_de\_listas = [ ['Curso', 'Módulo 1', 'Módulo 2'], ['Data Science', 'Lógica de Programação I', 'Lógica de Programação II'], ['Web Full Stack', 'Front End Estático', 'Front End Dinâmico']]

print(linguagens[0]) # imprime "Python"

print(linguagens[1]) # imprime "JavaScript"

print(dados\_variados[2]) # imprime True

print(lista\_de\_lista[2][0]) # imprime "Web Full Stack"

**1.2. Iterando uma lista**

Como os elementos em uma lista são representados por números inteiros, podemos facilmente percorrê-la variando seu índice de maneira automatizada:

for indice in range(4):

print(dados\_variados[indice])

Apesar de funcionar, essa forma é considerada pouco legível. Existe uma maneira mais direta de percorrer uma lista. Ao trocarmos a função *range* do nosso *for* pela própria lista, ele irá **copiar** cada elemento da lista para a variável temporária. Assim conseguimos facilmente, e de maneira bem legível, acessar todos os elementos de uma lista:

for elemento in dados\_variados:

print(elemento)

Caso tenhamos listas aninhadas, podemos percorrê-las aninhando loops. Utilizamos um loop para cada "nível" de lista. Execute o bloco abaixo e veja seu resultado na tela:

for linha in lista\_de\_listas:

for elemento in linha:

print(elemento)

**1.3. *Slicing* de listas**

Uma operação bastante comum em uma lista é extrair apenas uma parte dela. Para isso você deve informar, pelo menos, o índice inicial e o índice final, sendo que o índice final não irá entrar na conta - dizemos que ele é um valor "exclusivo", como se fosse um intervalo aberto naquele ponto.

frutas = ['abacate', 'banana', 'carambola', 'damasco', 'embaúba', 'framboesa', 'goiaba']

algumas\_frutas = frutas[2:5]

print(algumas\_frutas) # resultado: ['carambola', 'damasco', 'embaúba']

Você pode deixar um dos dois índices em branco. Na ausência do primeiro índice, a operação se iniciará com o índice 0 da lista original. Na ausência do segundo, a operação seguirá até o final da lista original.

primeiras3 = frutas[:3]

print(primeiras3)

ultimas3 = frutas[4:]

print(ultimas3)

Também podemos utilizar índices negativos. O índice -1 acessa o último elemento da lista, o -2 acessa o penúltimo, e assim sucessivamente. Podemos reescrever o exemplo anterior utilizando números negativos para facilitar ainda mais a legibilidade e tornar o exemplo mais genérico e independente do tamanho da lista:

ultimas3 = frutas[-3:]

print(ultimas3)

Você deve se recordar que atribuir uma lista para outra **não** copia a lista. Veja o exemplo abaixo:

copia\_frutas = frutas

copia\_frutas.append('heisteria')

print(frutas) # resultado na tela: ['abacate', 'banana', 'carambola', 'damasco', 'embaúba', 'framboesa', 'goiaba', 'heisteria']

Quando atribuímos uma lista existente para uma variável, a variável irá referenciar **a mesma** lista na memória. Portanto, operações realizadas na lista "nova" irão afetar a lista "antiga". Não houve a criação de uma nova lista de fato.

Uma forma fácil de **copiar** de verdade uma lista para outra lista é utilizar um slice indo do início até o final da lista original:

copia\_frutas = frutas[:]

print(copia\_frutas)

copia\_frutas e frutas não referenciam a mesma lista. Elas referenciam duas listas diferentes contendo os mesmos elementos, mas modificações feitas em uma delas não irão afetar a outra.

É possível passar mais um valor representando um passo ou salto. Por exemplo, podemos pegar os elementos das posições ímpares da lista começando na posição 1 e adotando salto igual a 2:

impares = frutas[1::2]

print(impares)

Com saltos negativos, é fácil inverter uma lista:

frutas\_inv = frutas[-1::-1]

print(frutas\_inv)

**1.4. Concatenação de listas**

Uma operação útil em listas é a concatenação. Quando "somamos" duas listas, utilizando o operador +, teremos uma nova lista com os elementos das duas listas originais em ordem de aparição:

ds = ['Python', 'SQL', 'R']

web = ['HTML', 'CSS', 'JavaScript']

linguagens = ds + web

print(linguagens)

#resultado: ['Python', 'SQL', 'R', 'HTML', 'CSS', 'JavaScript']

**2. Tuplas**

**2.1. Operações básicas**

Assim como as listas, tuplas também são coleções de objetos. Elas podem armazenar diversos objetos de diferentes tipos. Elas também possuem índice, que se comporta da mesma maneira que os índices de uma lista. Podemos criar tuplas utilizando parênteses ou a função tuple. Caso a tupla possua pelo menos 2 elementos, não precisamos dos parênteses, basta separar os valores por vírgula, apesar de ser **recomendável** utilizá-los para evitar ambiguidades:

tupla1 = tuple() # uma tupla vazia

tupla2 = () # outra tupla vazia

linguagens = ('Python', 'JavaScript', 'SQL') # uma tupla com 3 elementos

dados\_variados = 3.14, 1000, True, 'abacate' # uma tupla declarada sem parênteses

tupla\_de\_tuplas = ( ('Curso', 'Módulo 1', 'Módulo 2'), ('Data Science', 'Lógica de Programação I', 'Lógica de Programação II'), ('Web Full Stack', 'Front End Estático', 'Front End Dinâmico'))

print(linguagens[0]) # imprime "Python"

print(linguagens[1]) # imprime "JavaScript"

print(dados\_variados[2]) # imprime True

print(tupla\_de\_tuplas[2][0]) # imprime "Web Full Stack"

Todas as outras operações que revisamos hoje em listas podem ser realizadas com tuplas:

* iteração através de um loop do tipo for
* *slicing* passando índice inicial, final e salto
* concatenação

É possível também fazer conversão de lista para tupla e vice-versa:

lista\_frutas = ['abacate', 'banana', 'carambola', 'damasco', 'embaúba', 'framboesa', 'goiaba']

tupla\_frutas = tuple(lista\_frutas)

print(tupla\_frutas)

nova\_lista\_frutas = list(tupla\_frutas)

print(nova\_lista\_frutas)

**2.2. Imutabilidade**

Listas possuem uma propriedade que a tupla **não** possui: **mutabilidade**. O código abaixo irá funcionar para a operação na lista, mas irá falhar para a operação na tupla:

lista\_frutas = ['abacate', 'banana', 'carambola', 'damasco', 'embaúba', 'framboesa', 'goiaba']

tupla\_frutas = ('abacate', 'banana', 'carambola', 'damasco', 'embaúba', 'framboesa', 'goiaba')

lista\_frutas[0] = 'ananás'

print(lista\_frutas)

tupla\_frutas[0] = 'ananás' # erro

Por que alguém escolheria uma estrutura imutável? Por que usar uma lista com *menos recursos*? O principal motivo é precisamente quando não convém alterar os dados. Em Python é muito difícil "proibir" algo: nada impede outro programador de transformar a tupla em uma lista, fazer alterações na lista e salvar a nova tupla "por cima" da velha, utilizando o mesmo nome para a nova.

Porém, quando utilizamos uma tupla, estamos **sinalizando** que aqueles dados **não deveriam** ser alterados, e ninguém irá conseguir alterá-los por acidente. Para alterá-los será necessário realizar uma série de conversões de maneira intencional.

Isso aumenta um pouco a segurança e confiabilidade de nosso código em certas situações, evitando a alteração indevida de dados que podem ser críticos para o bom funcionamento do nosso programa.

Adicionalmente, em alguns contextos muito específicos - quantidades muito grandes de dados com uma grande quantidade de operações de **leitura** versus poucas ou nenhuma operação de **escrita** - a tupla pode oferecer desempenho significativamente superior à lista. São poucas as situações onde você sentirá essa diferença, seja porque para quantidades muito baixas de dados a lista possui uma série de otimizações que podem torná-la até mais veloz do que a tupla, seja porque para quantidades relativamente grandes de dados a lista ainda será razoavelmente rápida.

**2.3. Desempacotamento de tupla**

Uma operação bastante interessante que podemos realizar com tuplas é o **desempacotamento** de tuplas, que aparecerá em muitos locais com seu nome em inglês, *tuple unpacking*.

O desempacotamento de tupla é uma operação que permite facilmente atribuir o conteúdo de uma tupla a variáveis individuais, sem a necessidade de escrever múltiplas linhas de código e manipular índices. Vejamos um exemplo básico:

x, y, z = ('Lista', 'Tupla', 'Dicionário')

print(x) # Lista

print(y) # Tupla

print(z) # Dicionário

Uma limitação inicial que temos com essa técnica é que precisamos utilizar exatamente 1 variável para cada elemento da tupla, mesmo que não estejamos interessados em todos os seus elementos. Podemos contornar isso utilizando o operador **\***. Ao utilizá-lo em uma das variáveis no desempacotamento, estamos sinalizando que ele pode receber múltiplos valores, formando uma coleção com parte dos valores:

linguagens = ('Python', 'JavaScript', 'HTML', 'CSS', 'R')

primeira, \*resto = linguagens

print(primeira) # Python

print(resto) # ['JavaScript', 'HTML', 'CSS', 'R']

\*resto, ultima = linguagens

print(resto) # ['Python', 'JavaScript', 'HTML', 'CSS']

print(ultima) # R

primeira, \*meio, ultima = linguagens

print(primeira) # Python

print(meio) # ['JavaScript', 'HTML', 'CSS']

print(ultima) # R

O desempacotamento também pode ser utilizado com listas. Um dos principais motivos para ele frequentemente ser lembrado como uma operação de tupla foi que ele inicialmente só existia, de fato, para tuplas, e foi implementado para listas em versões mais recentes do Python. Outro motivo está relacionado à imutabilidade: como a tupla é imutável, temos mais garantias de que sabemos qual dado está em cada posição dela, tornando essa operação mais confiável em tuplas do que em listas.

**2.4. Operações com tuplas "implícitas"**

O Python oferece alguns truques que permitem escrever códigos mais enxutos do que em outras linguagens, e parte desses truques utiliza sintaxe de tupla. Por exemplo, para criar duas variáveis e atribuir valores simultaneamente a elas, podemos utilizar vírgulas:

x, y = 10, 20

print(x) # 10

print(y) # 20

Outro truque relacionado bastante comum é inverter o valor de duas variáveis:

y, x = x, y

print(x) # 20

print(y) # 10

Esse tipo de operação é considerado *açúcar sintático*, ou seja, não acrescenta funcionalidades novas, apenas cria formas mais simples e legíveis de realizar operações que já éramos capazes de realizar anteriormente.

Internamente, o Python está usando lógica de criar e desempacotar tuplas para realizar esse tipo de operação.

**3. Facilidades para iteração**

Sempre que possível, é preferível iterar uma coleção - seja ela uma lista ou uma tupla - de maneira direta utilizando for sem índices. Há alguns problemas onde pode ser difícil escapar do índice, pois nossa lógica irá depender de posição de alguma maneira.

Veremos duas estruturas que irão nos auxiliar a fazer iteração por índice sem precisar utilizar uma estrutura pouco legível como:

for indice in range(len(lista)):

...

lista[indice] = ...

...

**3.1. Enumerate**

Considere um problema qualquer onde o índice importa. Por exemplo, suponha que você possua uma lista de strings e gostaria de exibi-la intercalando uma em letra maiúscula e outra em letra minúscula (assim como frequentemente representamos tabelas intercalando as cores de suas linhas em editores de planilha para melhorar a legibilidade).

A lógica desse problema poderia ser resolvida usando índice:

lista\_frutas = ['abacate', 'banana', 'carambola', 'damasco', 'embaúba', 'framboesa', 'goiaba']

for indice in range(len(lista\_frutas)):

if indice % 2 == 0:

print(lista\_frutas[indice].upper())

else:

print(lista\_frutas[indice].lower())

Existe uma ferramenta em Python que pode nos ajudar a escrever de maneira mais *pythonica*, sem precisar acessar lista por índice: o enumerate. Primeiro, vamos entender o que ele faz e, em seguida, veremos como deixar o código mais limpo:

for x in enumerate(lista\_frutas):

print(x)

Saída na tela:

(0, 'abacate')

(1, 'banana')

(2, 'carambola')

(3, 'damasco')

(4, 'embaúba')

(5, 'framboesa')

(6, 'goiaba')

O enumerate montou uma estrutura onde cada elemento é uma tupla, sendo o primeiro elemento da tupla um índice da lista, e o segundo o valor associado àquele índice. Aplicando desempacotamento de tupla no for, podemos ter, simultaneamente, índice e valor em variáveis separadas, na prática percorrendo a lista tanto por índice quanto por valor. Refazendo o exemplo das maiúsculas/minúsculas:

for indice, valor in enumerate(lista\_frutas):

if indice % 2 == 0:

print(valor.upper())

else:

print(valor.lower())

**3.2. Zip**

Vamos pensar em um problema onde precisamos percorrer duas listas simultaneamente. Por exemplo, considere que temos uma lista com os nomes de todos os alunos de uma turma, e outra com as notas, na mesma ordem. Como faríamos para acessar, simultaneamente, o nome de um aluno e a sua nota?

Esse é um problema onde, a princípio, utilizaríamos índice. Se usarmos o mesmo índice nas duas listas, estamos na prática percorrendo ambas as listas simultaneamente:

alunos = ['Paul', 'John', 'George', 'Ringo']

notas = [10, 9.5, 7, 6]

for indice in range(len(alunos)):

print(f'Aluno {alunos[indice]}: {notas[indice]}')

Saída na tela:

Aluno Paul: 10

Aluno John: 9.5

Aluno George: 7

Aluno Ringo: 6

Vamos ver agora o zip em ação para compreender como ele funciona:

for x in zip(alunos, notas):

print(x)

Saída na tela:

('Paul', 10)

('John', 9.5)

('George', 7)

('Ringo', 6)

Assim como no enumerate, o zip montou tuplas. Cada tupla representa 1 posição das listas originais, e cada posição dentro da tupla representa o dado de uma das listas. Ou seja, cada elemento do zip contém 1 elemento de cada lista original, na ordem que eles apareceram nas listas originais. Logo, ele permite percorrer 2 listas simultaneamente.

Novamente podemos aplicar desempacotamento de tuplas em nosso loop e acessar os dados de cada lista individualmente de maneira legível:

alunos = ['Paul', 'John', 'George', 'Ringo']

notas = [10, 9.5, 7, 6]

for aluno, nota in zip(alunos, notas):

print(f'Aluno {aluno}: {nota}')

Saída na tela:

Aluno Paul: 10

Aluno John: 9.5

Aluno George: 7

Aluno Ringo: 6