Capítulo 3

Fatiamento (slicing)

Manipular estruturas de dados de forma prática é um diferencial para qualquer profissional de linguagem de programação. Segundo Lambert (2022), a linguagem Python oferece recursos avançados para manipulação eficiente de dados, incluindo o fatiamento (slicing), uma técnica poderosa para refinar e extrair subconjuntos de elementos em listas. Essa abordagem permite maior flexibilidade e otimização no processamento de informações, tornando-se uma ferramenta essencial para programadores que buscam proporcionar eficiência e clareza ao código. A prática de programação com a aplicação adequada de fatiamento acelera o processo de desenvolvimento e gera um custo computacional menor frente a outros métodos baseados em ausência de força bruta (por exemplo, o loops).

O primeiro subcapítulo trata de fatiamento de listas no geral, apresentando a sintaxe básica e, principalmente, a mentalidade essencial para realizar o fatiamento desejado. No próximo subcapítulo, é abordada a manipulação de strings – lembrando que essa manipulação, na linguagem de programação Python, trata-se de considerar uma lista de caracteres em que as técnicas de fatiamento se tornam bem-vindas para que se realizem edições.

1 Fatiamento de listas

O fatiamento de listas é uma técnica em Python que permite acessar, modificar e manipular subconjuntos de elementos dentro de uma lista de maneira eficiente e expressiva. Utilizando uma sintaxe intuitiva, o fatiamento oferece uma maneira poderosa de interagir com estruturas de dados sequenciais, evitando a necessidade de loops explícitos e tornando o código mais legível e com melhor desempenho. A sintaxe básica para a aplicação de fatiamento é:

```
lista[início:fim:passo]
```

Em que:

- *início* é o índice do primeiro elemento a ser incluído no fatiamento (opcional; padrão é 0).
- *fim* é o índice do primeiro elemento a ser excluído do fatiamento (opcional; padrão é o tamanho da lista).
- *passo* define a variação entre os índices extraídos (opcional; padrão é 1).

Um primeiro exemplo é a extração de uma sublista a partir de uma lista de origem, conforme a seguir.

Exemplo 1

```
numeros = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
sublista = numeros[2:7]  # Do índice 2 ao 6
print(sublista)  # Saída: [2, 3, 4, 5, 6]
```

É importante observar que a regra do fatiamento consiste em que o primeiro parâmetro diz respeito ao primeiro elemento a ser incluído no fatiamento, e o segundo parâmetro seria o primeiro elemento a ser excluído do fatiamento; sempre da esquerda para a direita.

Portanto, no exemplo 1, o primeiro parâmetro indica que o índice 2 é o primeiro a ser incluído (logo, os índices 0 e 1 estão excluídos do fatiamento). Por sua vez, o índice 7 é o segundo parâmetro, sendo o primeiro a ser excluído (logo, os índices 7, 8 e 9 estão excluídos do fatiamento). O resultado do fatiamento é a sublista [2, 3, 4, 5, 6].

Outro aspecto importante é quando o parâmetro do passo é utilizado, permitindo uma seleção alternada e controlada dos elementos da lista, como mostra o exemplo 2.

Exemplo 2

```
numeros = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
pares = numeros[::2]  # Pegando elementos com passo 2
print(pares)  # Saída: [0, 2, 4, 6, 8]
```

No exemplo 2, os parâmetros de início e fim estão ajustados para seu funcionamento padrão: o primeiro elemento a ser incluído é o índice zero, e o primeiro elemento a ser excluído da lista é referente ao tamanho da lista; portanto, toda a lista está sendo contemplada. Porém, o parâmetro de passo 2 faz a coleta alternada partindo do primeiro elemento do resultado do fatiamento, gerando a saída apresentada no exemplo 2.

No fatiamento, também é possível trabalhar com o passo negativo, o que resulta na coleta inversa de elementos, partindo do primeiro elemento resultante do fatiamento, como apresentado no próximo exemplo (exemplo 3).

Exemplo 3

```
numeros = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
invertida = numeros[::-1]  # Passo negativo para inverter
print(invertida)  # Saída: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Como já abordado, os valores padrão dos dois primeiros parâmetros do fatiamento resultam em coletar toda a lista. O passo com valor negativo -1 resulta na coleta inversa dos valores (do maior índice resultante

do fatiamento para o menor), se constituindo em uma técnica simples para inverter elementos de listas.

Outra funcionalidade interessante é a modificação de partes de uma lista, substituindo elementos de forma direta, como apresentado exemplo 4.

Exemplo 4

```
numeros = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
numeros[2:5] = [20, 30, 40]  # Substituindo elementos
print(numeros)  # Saída: [0, 1, 20, 30, 40, 5, 6, 7, 8, 9]
```

A partir do fatiamento realizado (incluindo os elementos a partir do índice 2 e excluindo os elementos a partir do índice 5), podemos realocar uma outra lista com o mesmo número de elementos; neste caso, 3 elementos. Mesmo que alocados dinamicamente, a operação de substituição é realizada internamente pelo interpretador Python.

Além da substituição, o segmento de lista selecionado pode ser removido por meio do comando del, conforme o exemplo 5 a seguir.

Exemplo 5

```
numeros = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
del numeros[2:5]  # Remove os elementos do índice 2 ao 4
print(numeros)  # Saída: [0, 1, 5, 6, 7, 8, 9]
```

No exemplo 5, o segmento selecionado é extraído, gerando uma lista com o resultado da junção dos elementos não contemplados no fatiamento.

Em resumo, o uso do fatiamento traz diversos benefícios, incluindo:

- Legibilidade: a sintaxe é concisa e fácil de entender.
- Eficiência: opera diretamente na lista, evitando loops desnecessários.

- Flexibilidade: permite realizar extrações, substituições e manipulações complexas de forma simples.

IMPORTANTE

É fundamental ter em mente que, ao utilizar o fatiamento para extrair partes de uma lista em Python, o resultado será sempre uma nova lista, completamente independente da original. Esse comportamento pode ter implicações significativas no consumo de memória, especialmente ao manipular grandes volumes de dados, pois cada operação de fatiamento cria uma cópia dos elementos selecionados. Consequentemente, em cenários que envolvem listas extensas, é importante avaliar alternativas mais eficientes, como iteradores ou views, para evitar o consumo desnecessário de memória e melhorar o desempenho geral do programa.

2 Operações com strings

As strings são uma das estruturas de dados mais fundamentais em Python. Elas representam sequências de caracteres e são imutáveis; ou seja, uma vez criadas, não podem ser modificadas diretamente. No entanto, Python oferece diversas operações para manipular strings de forma eficiente. Entre essas operações, destacam-se concatenação, repetição, acesso por índice, fatiamento e diversas funções embutidas para modificação e análise de texto.

A concatenação de strings em Python é realizada por meio do operador +, que permite unir duas ou mais sequências de caracteres em uma única string. Essa operação é útil para a construção dinâmica de textos, como mostra o exemplo 6 a seguir.

Exemplo 6

```
s1 = “Olá”
s2 = “Mundo”
resultado = s1 + “ “ + s2
print(resultado)  # Saída: Olá Mundo
```

No exemplo 6, as strings “Olá” e “Mundo” foram unidas, com um espaço intermediário para melhor legibilidade.

O operador * permite a repetição de uma string por um determinado número de vezes. Essa técnica é útil para criar padrões de texto que se repetem, conforme apresenta o exemplo 7.

Exemplo 7

```
s = “Python “
print(s * 3)  # Saída: Python Python Python
```

Também existem funções embutidas que apoiam sua manipulação. Por exemplo, a função len() retorna o número de caracteres presentes em uma string, como mostra o exemplo a seguir.

Exemplo 8

```
s = “Python”
print(len(s))  # Saída: 6
```

Um caractere específico pode ser acessado por seu índice. Em Python, cada caractere de uma string pode ser acessado individualmente por meio de índices, cuja contagem começa do zero. Índices negativos permitem a contagem a partir do final da string. Dessa forma, podemos acessar tanto os primeiros caracteres quanto os últimos de forma simples.

Exemplo 9

```
s = "Python"
print(s[0]) # Saída: P (primeiro caractere)
print(s[-1]) # Saída: n (último caractere)
```

No entanto, referente ao fatiamento de strings, as regras são as mesmas aplicadas às listas, utilizando-se a sintaxe string[início:fim:passo]. O índice de início é inclusivo, enquanto o índice de fim é exclusivo. A seguir, alguns exemplos.

Exemplo 10

```
s = "Python"
print(s[0:4])   # Saída: Pyth (pega do índice 0 ao 3)
print(s[:4])    # Saída: Pyth (começa no início por padrão)
print(s[2:])    # Saída: thon (vai do índice 2 até o final)
print(s[-4:])   # Saída: thon (pega os últimos quatro caracteres)
print(s[::2])   # Saída: Pto (pula de 2 em 2)
```

Seguindo as mesmas premissas de fatiamento de strings, podemos, por exemplo, inverter a ordem de uma string (essa técnica é frequentemente utilizada para verificar palíndromos). Vejamos mais um exemplo.

Exemplo 11

```
s = "Python"
print(s[::-1])  # Saída: nohtyP
```

A linguagem Python também oferece operadores de verificação de um dado ou parte de um dado em uma string através dos operadores in (retorna valor "verdadeiro" se a substring estiver contida na string analisada) e not in (retorna valor "falso" se a substring estiver contida na string analisada). Veja o exemplo 12 a seguir.

Exemplo 12

```
s = "Python é incrível"
print("Python" in s)   # True
print("Java" not in s) # True
```

Considerações finais

Neste capítulo, exploramos a técnica de fatiamento (slicing) em Python, aplicada tanto a listas quanto a strings. Vimos como essa funcionalidade permite acessar, modificar e manipular subconjuntos de elementos de forma eficiente, reduzindo a necessidade de loops e tornando o código mais legível. No caso das listas, abordamos a sintaxe básica do fatiamento lista[início:fim:passo] e a utilização de passos positivos e negativos, além de operações como substituição e remoção de elementos. Já nas strings, destacamos a aplicação do slicing para extração de trechos, inversão de caracteres e verificação de substrings. A partir desses conceitos, o leitor agora dispõe de uma base sólida para manipular dados de maneira otimizada em Python, em que é possível aproveitar ao máximo o poder do fatiamento para tornar o código mais conciso e com melhor desempenho.

Referências

LAMBERT, Kenneth A. **Fundamentos de Python**: estruturas de dados. São Paulo: Cengage Learning Brasil, 2022. *E-book*.