Guia Python

String

Principais metodos de string

Em Python, as strings possuem diversos métodos embutidos que permitem manipulação, busca, formatação, e outras operações úteis. Abaixo, listo alguns dos principais métodos para strings:

1. Manipulação de Strings

- `upper()`: Converte todos os caracteres para maiúsculas.
- `lower()`: Converte todos os caracteres para minúsculas.
- `capitalize()`: Converte o primeiro caractere da string para maiúscula.
- `title()`: Converte o primeiro caractere de cada palavra para maiúscula.
- `strip()`: Remove espaços em branco do início e do fim da string.
- `lstrip()`: Remove espaços em branco do início da string.
- `rstrip()`: Remove espaços em branco do final da string.
- `replace(old, new)`: Substitui todas as ocorrências de uma substring por outra.
- `split(separator)`: Divide a string em uma lista, usando o separador especificado.
- `join(iterable)`: Junta uma lista de strings em uma única string, usando a string como separador.

2. Busca e Verificação

- `find(substring)`: Retorna o índice da primeira ocorrência da substring, ou -1 se não for encontrada.
- `index(substring)`: Semelhante a `find()`, mas gera uma exceção se a substring não for encontrada.
- `startswith(prefix)`: Verifica se a string começa com o prefixo especificado.
- `endswith(suffix)`: Verifica se a string termina com o sufixo especificado.
- `count(substring)`: Retorna o número de vezes que a substring aparece na string.

3. Formatação

- `format()`: Formata a string, substituindo placeholders `{}` por valores especificados.
- `zfill(width)`: Preenche a string com zeros à esquerda até atingir o comprimento especificado.
- `center(width)`: Centraliza a string em um espaço de largura especificada.
- `ljust(width)`: Alinha a string à esquerda em um espaço de largura especificada.
- `rjust(width)`: Alinha a string à direita em um espaço de largura especificada.

4. Outras Operações

- `len(string)`: Retorna o comprimento da string.
- `isalnum()`: Verifica se a string é alfanumérica.

- `isalpha()`: Verifica se a string contém apenas letras.
- `isdigit()`: Verifica se a string contém apenas dígitos.
- `isspace()`: Verifica se a string contém apenas espaços em branco.

Esses métodos são os mais comumente utilizados e cobrem uma ampla gama de necessidades ao trabalhar com strings em Python.

Maneiras de Reverter uma String

Em Python, strings são imutáveis, o que significa que não existe um método direto como `reverse()` para reverter uma string. No entanto, existem várias maneiras de reverter uma string de maneira eficiente.

1. **Usando Slicing** A maneira mais comum e simples de reverter uma string é utilizando slicing.

   ```
   texto = "Python"
   texto_revertido = texto[::-1]
   print(texto_revertido)
   ```

   Saída:

   ```
   nohtyP
   ```

   - O slicing `[::-1]` significa "pegue todos os caracteres da string, mas na ordem inversa".

2. **Usando a Função** `reversed()` Outra maneira é usar a função `reversed()`, que retorna um iterador, e então, unir os caracteres de volta em uma string.

   ```
   texto = "Python"
   texto_revertido = ''.join(reversed(texto))
   print(texto_revertido)
   ```

   Saída:

   ```
   nohtyP
   ```

   - Aqui, `reversed(texto)` retorna um iterador que percorre a string de trás para frente.
   - `''.join()` é utilizado para unir os caracteres do iterador em uma nova string.

3. **Usando um Loop Manual** Embora não seja a maneira mais eficiente, você também pode reverter uma string manualmente com um loop.

```
texto = "Python"
texto_revertido = ""
for letra in texto:
    texto_revertido = letra + texto_revertido
print(texto_revertido)
```

Saída:

```
nohtyP
```

Conclusão

Embora não exista um método `reverse()` específico para strings em Python, você pode facilmente reverter uma string usando slicing, a função `reversed()`, ou até mesmo um loop. O método mais comum e eficiente é o slicing `[::-1]`.

O que são Tuplas em Python?

Tuplas são um tipo de estrutura de dados em Python, semelhantes às listas, mas com uma diferença crucial: **tuplas são imutáveis**. Isso significa que, uma vez criada, você não pode modificar, adicionar ou remover elementos de uma tupla. As tuplas são usadas quando você deseja criar uma coleção de elementos que não será alterada ao longo do tempo.

Características das Tuplas:

1. **Imutáveis**: Não podem ser alteradas após a criação.
2. **Ordenadas**: Mantêm a ordem dos elementos.
3. **Indexáveis**: Você pode acessar os elementos de uma tupla usando índices.
4. **Podem conter diferentes tipos de dados**: Inteiros, strings, floats, outras tuplas, etc.
5. **Podem ser aninhadas**: Uma tupla pode conter outras tuplas como elementos.

Criando Tuplas:

Aqui estão alguns exemplos de como criar e usar tuplas:

1. **Tupla Simples**:

```
tupla = (1, 2, 3)
print(tupla)  # Output: (1, 2, 3)
```

2. **Tupla Vazia**:

```
tupla_vazia = ()
print(tupla_vazia)  # Output: ()
```

3. **Tupla com diferentes tipos de dados**:

```
tupla_mista = (1, "Python", 3.14)
print(tupla_mista)  # Output: (1, 'Python', 3.14)
```

4. **Tupla aninhada**:

```
tupla_aninhada = (1, (2, 3), (4, 5, 6))
print(tupla_aninhada)  # Output: (1, (2, 3), (4, 5, 6))
```

5. **Acessando elementos de uma tupla**:

```
tupla = (10, 20, 30, 40)
print(tupla[0])  # Output: 10
print(tupla[2])  # Output: 30
```

6. **Tentando modificar uma tupla (causará erro)**:

```
tupla = (1, 2, 3)
tupla[0] = 10  # Isso causará um erro porque tuplas são imutáveis
```

Utilidade das Tuplas:

- **Armazenamento de dados constantes**: Como as tuplas são imutáveis, elas são ideais para armazenar dados que não devem ser alterados.
- **Chaves em dicionários**: Tuplas podem ser usadas como chaves em dicionários, enquanto listas não podem.
- **Retorno de múltiplos valores**: Funções podem retornar múltiplos valores como uma tupla.

Exemplo de Retorno de Múltiplos Valores:

```
def coordenadas():
    return (10, 20)

x, y = coordenadas()
print(x, y)  # Output: 10 20
```

1. Tuplas dentro de Listas

Você pode ter uma lista que contém tuplas como seus elementos.

Exemplo:

```
# Lista contendo tuplas
lista_de_tuplas = [(1, 2), (3, 4), (5, 6)]

# Acessando elementos das tuplas dentro da lista
for tupla in lista_de_tuplas:
    print(f"Primeiro elemento: {tupla[0]}, Segundo elemento: {tupla[1]}")

# Output:
# Primeiro elemento: 1, Segundo elemento: 2
# Primeiro elemento: 3, Segundo elemento: 4
# Primeiro elemento: 5, Segundo elemento: 6
```

2. Listas dentro de Tuplas

Tuplas podem conter listas como seus elementos. Isso é útil quando você deseja manter a imutabilidade da estrutura externa (tupla), mas ainda precisa modificar os elementos internos (listas).

Exemplo:

```
# Tupla contendo listas
tupla_de_listas = ([1, 2, 3], [4, 5, 6])

# Modificando a lista dentro da tupla
tupla_de_listas[0].append(4)

print(tupla_de_listas)  # Output: ([1, 2, 3, 4], [4, 5, 6])
```

3. Tuplas como Chaves em Dicionários

Dicionários exigem que as chaves sejam imutáveis, e como tuplas são imutáveis, elas podem ser usadas como chaves em dicionários.

Exemplo:

```
# Dicionário usando tuplas como chaves
distancias = {
    ("São Paulo", "Rio de Janeiro"): 429,
    ("São Paulo", "Belo Horizonte"): 586,
}

# Acessando valores no dicionário
print(distancias[("São Paulo", "Rio de Janeiro")])  # Output: 429
```

4. Desempacotamento de Tuplas em Estruturas de Dados

Você pode desempacotar tuplas diretamente em listas ou outras tuplas.

Exemplo:

```
# Lista de tuplas
pontos = [(1, 2), (3, 4), (5, 6)]

# Desempacotamento das tuplas
for x, y in pontos:
    print(f"x: {x}, y: {y}")

# Output:
# x: 1, y: 2
# x: 3, y: 4
# x: 5, y: 6
```

Aplicações Comuns:

- **Coordenadas e Posições**: Usar tuplas para representar coordenadas ou posições em gráficos ou jogos.
- **Agrupamento de Dados**: Retornar múltiplos valores relacionados em uma única estrutura.
- **Estruturas Imutáveis**: Quando é necessário garantir que uma coleção de dados não será alterada.

Vamos explorar como utilizar tuplas em funções Python, tanto como argumentos quanto como valores de retorno.

1. Tuplas como Argumentos de Funções

a) Função que Recebe uma Tupla como Argumento:

Você pode passar uma tupla como argumento para uma função. Isso é útil quando você quer garantir que o conjunto de valores passados seja imutável.

```
def imprime_coordenadas(coordenadas):
    x, y = coordenadas
    print(f"X: {x}, Y: {y}")

# Chamando a função com uma tupla
imprime_coordenadas((10, 20))

# Output:
# X: 10, Y: 20
```

b) Função com Vários Argumentos Agrupados em uma Tupla:

Você pode usar o operador `*` para desempacotar uma tupla diretamente nos parâmetros da função.

```
def soma(a, b, c):
    return a + b + c

# Desempacotando uma tupla como argumentos
valores = (1, 2, 3)
resultado = soma(*valores)

print(resultado)  # Output: 6
```

2. Tuplas como Retorno de Funções

Tuplas são uma maneira conveniente de retornar múltiplos valores de uma função.

a) Função que Retorna Múltiplos Valores em uma Tupla:

```
def dividir_e_resto(dividendo, divisor):
    quociente = dividendo // divisor
    resto = dividendo % divisor
    return (quociente, resto)

# Recebendo a tupla de retorno
resultado = dividir_e_resto(10, 3)

print(f"Quociente: {resultado[0]}, Resto: {resultado[1]}")

# Output:
# Quociente: 3, Resto: 1
```

b) Desempacotamento Direto do Retorno da Função:

Você pode desempacotar a tupla retornada diretamente nas variáveis.

```
def dividir_e_resto(dividendo, divisor):
    quociente = dividendo // divisor
    resto = dividendo % divisor
    return quociente, resto  # Retorno sem parênteses também funciona

# Desempacotando diretamente no momento da chamada
quociente, resto = dividir_e_resto(10, 3)

print(f"Quociente: {quociente}, Resto: {resto}")

# Output:
# Quociente: 3, Resto: 1
```

3. Passagem de Tuplas como Argumentos para Várias Funções

Tuplas também podem ser usadas para passar um conjunto fixo de dados para várias funções diferentes.

```
def calcular_area(base, altura):
    return 0.5 * base * altura

def calcular_perimetro(base, altura):
    return 2 * (base + altura)

# Usando uma tupla de argumentos para ambas as funções
dimensoes = (10, 20)

area = calcular_area(*dimensoes)
perimetro = calcular_perimetro(*dimensoes)

print(f"Área: {area}, Perímetro: {perimetro}")

# Output:
# Área: 100.0, Perímetro: 60
```

Resumo

- **Tuplas como Argumentos**: Garanta a imutabilidade dos dados passados para funções.
- **Tuplas como Retorno**: Retorne múltiplos valores de forma simples e elegante.
- **Desempacotamento**: Facilita a passagem e manipulação de múltiplos valores em funções.

O que são Listas em Python?

Listas são uma das estruturas de dados mais usadas em Python. Elas são coleções ordenadas e mutáveis de elementos que podem conter diferentes tipos de dados, como inteiros, strings, floats, outras listas, etc. Ao contrário das tuplas, as listas são mutáveis, o que significa que você pode alterar seus elementos após a criação.

Características das Listas:

1. **Mutáveis**: Você pode modificar, adicionar ou remover elementos.
2. **Ordenadas**: Mantêm a ordem de inserção dos elementos.
3. **Indexáveis**: Você pode acessar os elementos de uma lista usando índices.
4. **Podem conter diferentes tipos de dados**: Listas podem conter qualquer tipo de dado, incluindo outras listas.

Criando Listas:

Aqui estão alguns exemplos de como criar e manipular listas em Python:

1. Lista Simples:

```
lista = [1, 2, 3, 4, 5]
print(lista)  # Output: [1, 2, 3, 4, 5]
```

2. Lista Vazia:

```
lista_vazia = []
print(lista_vazia)  # Output: []
```

3. Lista com Diferentes Tipos de Dados:

```
lista_mista = [1, "Python", 3.14, True]
print(lista_mista)  # Output: [1, 'Python', 3.14, True]
```

4. Lista Aninhada (Lista dentro de uma lista):

```
lista_aninhada = [1, [2, 3], [4, 5, 6]]
print(lista_aninhada)  # Output: [1, [2, 3], [4, 5, 6]]
```

Acessando e Modificando Elementos:

1. Acessando Elementos:

Os elementos de uma lista são acessados por meio de índices, que começam em 0.

```
lista = [10, 20, 30, 40]
print(lista[0])  # Output: 10
print(lista[2])  # Output: 30
```

2. Modificando Elementos:

Você pode modificar elementos de uma lista atribuindo novos valores aos índices correspondentes.

```
lista = [10, 20, 30, 40]
lista[1] = 25
print(lista)  # Output: [10, 25, 30, 40]
```

Operações Comuns com Listas:

1. Adicionando Elementos:

Você pode adicionar elementos a uma lista usando `append`, `insert`, ou `extend`.

- `append`: Adiciona um elemento ao final da lista.

  ```
  lista = [1, 2, 3]
  lista.append(4)
  print(lista)  # Output: [1, 2, 3, 4]
  ```

- `insert`: Insere um elemento em uma posição específica.

```
lista = [1, 2, 3]
lista.insert(1, 10)
print(lista)  # Output: [1, 10, 2, 3]
```

- `extend`: Adiciona todos os elementos de outra lista ao final.

```
lista = [1, 2, 3]
lista.extend([4, 5])
print(lista)  # Output: [1, 2, 3, 4, 5]
```

2. Removendo Elementos:

Você pode remover elementos de uma lista usando `remove`, `pop`, ou `del`.

- `remove`: Remove a primeira ocorrência de um valor específico.

```
lista = [1, 2, 3, 2]
lista.remove(2)
print(lista)  # Output: [1, 3, 2]
```

- `pop`: Remove e retorna o elemento em uma posição específica (por padrão, o último elemento).

```
lista = [1, 2, 3]
elemento = lista.pop()
print(elemento)  # Output: 3
print(lista)     # Output: [1, 2]
```

- `del`: Remove um elemento em uma posição específica ou remove uma fatia da lista.

```
lista = [1, 2, 3, 4]
del lista[1]
print(lista)  # Output: [1, 3, 4]
```

3. Verificando a Presença de um Elemento:

Você pode verificar se um elemento está presente na lista usando `in`.

```
lista = [1, 2, 3]
print(2 in lista)  # Output: True
print(5 in lista)  # Output: False
```

Funções e Métodos Úteis para Listas:

- `len(lista)`: Retorna o número de elementos na lista.

```
lista = [1, 2, 3, 4]
print(len(lista))  # Output: 4
```

- `sorted(lista)`: Retorna uma nova lista ordenada.

```
lista = [3, 1, 4, 2]
print(sorted(lista))  # Output: [1, 2, 3, 4]
```

- `sum(lista)`: Retorna a soma de todos os elementos da lista (se forem numéricos).

```
lista = [1, 2, 3]
print(sum(lista))  # Output: 6
```

- `list.reverse()`: Inverte a ordem dos elementos na lista.

```
lista = [1, 2, 3]
lista.reverse()
print(lista)  # Output: [3, 2, 1]
```

Exemplo Completo:

```
# Criando uma lista de números
numeros = [1, 2, 3, 4, 5]

# Adicionando um novo número à lista
numeros.append(6)

# Inserindo um número em uma posição específica
numeros.insert(0, 0)

# Removendo um número específico
numeros.remove(3)

# Acessando elementos
primeiro = numeros[0]
ultimo = numeros[-1]

# Imprimindo a lista e elementos específicos
print(f"Lista: {numeros}")
print(f"Primeiro elemento: {primeiro}, Último elemento: {ultimo}")

# Output:
# Lista: [0, 1, 2, 4, 5, 6]
# Primeiro elemento: 0, Último elemento: 6
```

Conclusão:

As listas são extremamente versáteis e são a estrutura de dados mais comum para armazenar coleções de itens em Python. Elas são amplamente utilizadas devido à sua flexibilidade e facilidade de uso.

Vamos explorar como usar listas em Python em operações de filtragem e mapeamento utilizando as funções `filter()` e `map()`.

1. Usando `filter()` para Filtragem

A função `filter()` é usada para criar uma nova lista que contém apenas os elementos de uma lista original que satisfazem uma determinada condição. A função `filter()` recebe dois argumentos:

- Uma função que retorna `True` ou `False`.
- Uma lista ou outro iterável.

Exemplo 1: Filtrando Números Pares de uma Lista

```
# Definindo a lista de números
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Usando filter para manter apenas os números pares
pares = list(filter(lambda x: x % 2 == 0, numeros))

print(pares)  # Output: [2, 4, 6, 8, 10]
```

2. Usando `map()` para Mapeamento

A função `map()` aplica uma função a todos os itens de uma lista (ou outro iterável) e retorna um novo iterável com os resultados. Ela recebe dois argumentos:

- Uma função que transforma os elementos.
- Uma lista ou outro iterável.

Exemplo 2: Quadrado dos Números em uma Lista

```
# Definindo a lista de números
numeros = [1, 2, 3, 4, 5]

# Usando map para calcular o quadrado de cada número
quadrados = list(map(lambda x: x ** 2, numeros))

print(quadrados)  # Output: [1, 4, 9, 16, 25]
```

3. Combinando `filter()` e `map()`

Você pode combinar `filter()` e `map()` para primeiro filtrar os elementos de uma lista e, em seguida, aplicar uma transformação nos elementos filtrados.

Exemplo 3: Quadrado dos Números Pares de uma Lista

```
# Definindo a lista de números
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Filtrando os números pares e depois calculando o quadrado de cada um
quadrados_pares = list(map(lambda x: x ** 2, filter(lambda x: x % 2 == 0, numeros)))

print(quadrados_pares)  # Output: [4, 16, 36, 64, 100]
```

4. Usando `filter()` e `map()` com Funções Definidas pelo Usuário

Você também pode usar funções definidas pelo usuário em vez de lambdas para tornar o código mais legível.

Exemplo 4: Usando Funções Separadas

```
# Definindo a lista de números
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Função para verificar se um número é par
def eh_par(x):
    return x % 2 == 0

# Função para calcular o quadrado de um número
def quadrado(x):
    return x ** 2

# Filtrando números pares e depois aplicando a função de quadrado
quadrados_pares = list(map(quadrado, filter(eh_par, numeros)))

print(quadrados_pares)  # Output: [4, 16, 36, 64, 100]
```

Conclusão:

- `filter()`: Ideal para criar sublistas com base em uma condição.
- `map()`: Útil para aplicar uma transformação a cada elemento de uma lista.
- **Combinação**: `filter()` seguido de `map()` permite criar pipelines de dados onde você primeiro seleciona elementos e depois os transforma.