Aula 4 – Listas e Matrizes

Objetivo da Aula

Conhecer e aplicar corretamente as listas, as matrizes e outras estruturas de dados.

Apresentação

Para desenvolver um programa em uma linguagem de programação, é comum utilizar dados que precisam ficar armazenados em uma variável. O problema é que, quando se utiliza uma grande quantidade de dados, as variáveis ficam muito trabalhosas.

Em situações como essa, as linguagens de programação usam outras formas de armazenamento de dados, conhecidas como listas e matrizes, que permitem armazenar e gerenciar o acesso a um volume de dados de uma forma menos trabalhosa do que variáveis.

A linguagem Python possui algumas maneiras além das listas e matrizes, e vamos analisar aqui essas outras alternativas.

1. Listas em Python

Em Python, uma lista é uma estrutura de dados que pode armazenar vários valores em uma única variável. É uma sequência ordenada de valores separados por vírgulas e delimitada por colchetes.

A lista pode conter qualquer tipo de valor, incluindo números, strings, outras listas e objetos complexos. Além disso, os elementos de uma lista podem ser acessados, individualmente, por meio de seus índices, que começam em 0 para o primeiro elemento da lista.

Por exemplo, veja como criar uma lista simples em Python:

```
1   lista = [1, 2, 3, 4, 5]
```

Aqui estão alguns dos principais recursos das listas em Python:

- Adicionar elementos: você pode adicionar elementos a uma lista, usando o método append() ou o operador +. Por exemplo:

```
lista = [1, 2, 3]
lista.append(4)
# lista agora é [1, 2, 3, 4]

lista2 = [5, 6]
lista += lista2
# lista agora é [1, 2, 3, 4, 5, 6]
```

- Acessar elementos: você pode acessar elementos de uma lista, usando índices numéricos. Por exemplo:

```
lista = [1, 2, 3, 4, 5]
primeiro_elemento = lista[0] # retorna 1
terceiro_elemento = lista[2] # retorna 3
```

- Alterar elementos: você pode alterar elementos de uma lista, atribuindo um novo valor a um índice específico. Por exemplo:

```
lista = [1, 2, 3]
lista[1] = 4
# lista agora é [1, 4, 3]
```

- Remover elementos: você pode remover elementos de uma lista, usando o método remove() ou a palavra-chave del. Por exemplo:

```
lista = [1, 2, 3, 4, 5]
lista.remove(3)
# lista agora é [1, 2, 4, 5]

del lista[0]
# lista agora é [2, 4, 5]
```

- Comprimento da lista: você pode obter o comprimento de uma lista, usando a função len(). Por exemplo:

```
lista = [1, 2, 3, 4, 5]
tamanho_da_lista = len(lista) # retorna 5
```

Agora, vamos dar um exemplo de como usar uma lista em um loop for:

```
notas = [9.5, 7.8, 6.0, 8.5, 10.0]
soma = 0

for nota in notas:
    soma += nota

media = soma / len(notas)
print("A média das notas é:", media)
```

Nesse exemplo, criamos uma lista de notas e inicializamos a variável soma como zero. Em seguida, usamos um loop for para iterar sobre cada elemento da lista notas. Em cada iteração, adicionamos o valor da nota à variável soma. Depois que o loop termina, calculamos a média, dividindo a soma pelo comprimento da lista, e exibimos o resultado.

2. Matrizes em Python

Em Python, matrizes são uma forma de representar uma coleção bidimensional de dados. Elas são compostas de linhas e colunas e podem ser usadas para armazenar uma grande quantidade de informações em uma estrutura organizada e acessível.

Para criar uma matriz em Python, podemos utilizar a estrutura de lista aninhada, que consiste em criar uma lista de listas. Cada lista interna representa uma linha da matriz, e cada elemento dessa lista representa uma coluna.

Por exemplo, a seguinte matriz de dimensões 3x3 pode ser representada em Python como uma lista aninhada:

```
matriz = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
```

A fim de acessar um elemento específico da matriz, podemos utilizar a notação de colchetes para indicar a linha e a coluna desejada. Por exemplo, para acessar o elemento na segunda linha e terceira coluna (valor 6), podemos escrever:

```
elemento = matriz[1][2]
```

Podemos percorrer todos os elementos de uma matriz, utilizando dois loops for aninhados. O primeiro loop percorre as linhas, e o segundo loop percorre as colunas:

```
7
8   for i in range(len(matriz)):
9       for j in range(len(matriz[i])):
10          print(matriz[i][j])
```

Podemos, também, modificar os valores de uma matriz, utilizando a mesma notação de colchetes:

```
11
12  matriz[1][2] = 10
```

Com isso, o valor anterior da posição (2,3) (6) é substituído pelo valor 10.

As matrizes em Python são amplamente utilizadas em diversas áreas, como em processamento de imagens e jogos e em análise de dados.

3. Outras Coleções em Python

3.1. Conjuntos em Python

Em Python, um conjunto (set) é uma coleção não ordenada de elementos únicos e imutáveis. Em outras palavras, um conjunto não permite elementos duplicados, e não é possível alterar um elemento individualmente após sua inclusão no conjunto. Conjuntos são implementados com a classe set em Python.

Para criar um conjunto, basta usar a função set() e passar uma sequência de elementos como argumento. Por exemplo:

```
1   meu_set = set([1, 2, 3, 3, 4, 5])
2   print(meu_set)
3   A saída será: {1, 2, 3, 4, 5}
```

Observe que a duplicata do número 3 foi removida automaticamente do conjunto.

Os conjuntos em Python suportam diversas operações matemáticas, como união, interseção e diferença. Para realizar essas operações, podemos utilizar os métodos correspondentes ou os operadores matemáticos correspondentes.

Por exemplo, para realizar a união de dois conjuntos, podemos usar o método union() ou o operador | (pipe):

```
conjunto1 = {1, 2, 3}
conjunto2 = {2, 3, 4}
uniao = conjunto1.union(conjunto2)
print(uniao)
A saída será: {1, 2, 3, 4}
```

Também é possível fazer a interseção de dois conjuntos, utilizando o método intersection() ou o operador & (e comercial):

```
conjunto1 = {1, 2, 3}
conjunto2 = {2, 3, 4}
intersecao = conjunto1.intersection(conjunto2)
print(intersecao)
A saída será: {2, 3}
```

Além disso, os conjuntos em Python suportam a operação de diferença, que pode ser realizada com o método difference() ou o operador - (hífen):

```
conjunto1 = {1, 2, 3}
conjunto2 = {2, 3, 4}
diferenca = conjunto1.difference(conjunto2)
print(diferenca)
A saída será: {1}
```

Outras operações suportadas por conjuntos incluem testar se um elemento pertence a um conjunto, adicionar e remover elementos, entre outras.

```
meu_set = set([1, 2, 3, 4, 5])
print(2 in meu_set) # Verifica se o elemento 2 está presente no conjunto
meu_set.add(6) # Adiciona o elemento 6 ao conjunto
print(meu_set)
meu_set.remove(2) # Remove o elemento 2 do conjunto
print(meu_set)
A saída será:
True
{1, 2, 3, 4, 5, 6}
{1, 3, 4, 5, 6}
```

Em resumo, conjuntos em Python são úteis para armazenar coleções de elementos únicos e realizar operações matemáticas entre eles.

3.2. Tuplas em Python

As tuplas em Python são semelhantes às listas, mas com a diferença fundamental de que as tuplas são imutáveis – ou seja, uma vez criada, não é possível adicionar, remover ou modificar os elementos da tupla. A sintaxe para criar uma tupla é usar parênteses (), separando os elementos por vírgulas:

```
#Exemplo:
tupla = (1, 2, 3, 4, 5)
```

Ao contrário das listas, as tuplas não suportam operações como append(), remove(), pop() e sort(). No entanto, elas têm algumas vantagens em relação às listas, como o menor uso de memória e a capacidade de serem usadas como chaves de dicionários.

Para acessar um elemento específico, em uma tupla, podemos usar a mesma sintaxe que usamos para as listas, ou seja, o índice do elemento entre colchetes:

```
#Exemplo:
tupla = (1, 2, 3, 4, 5)
print(tupla[2]) # resultado: 3
```

Podemos, também, usar o operador de fatiamento para obter uma sub-tupla a partir da tupla original:

```
#Exemplo:
tupla = (1, 2, 3, 4, 5)
print(tupla[1:4]) # resultado: (2, 3, 4)
```

Além disso, é possível usar tuplas em atribuições múltiplas, ou seja, atribuir vários valores a várias variáveis de uma só vez:

```
#Exemplo:
tupla = (1, 2, 3)
a, b, c = tupla
print(a) # resultado: 1
print(b) # resultado: 2
print(c) # resultado: 3
```

As tuplas também podem ser usadas em loops for, permitindo percorrer todos os elementos da tupla.

```
#Exemplo:
tupla = (1, 2, 3, 4, 5)
for elemento in tupla:
    print(elemento)
```

Em resumo, as tuplas são uma estrutura de dados imutável em Python, que permite armazenar um conjunto de valores e os acessar por meio de índices ou loops for. Embora não suportem operações de modificação, as tuplas têm algumas vantagens em relação às listas, como o menor uso de memória e a capacidade de serem usadas como chaves de dicionários.

3.3. Dicionários em Python

Em Python, um dicionário é uma coleção de pares chave-valor que é mutável, indexada e não ordenada. Cada elemento no dicionário é representado por uma chave única e pelo valor associado a ela. A chave pode ser de qualquer tipo imutável, como uma string, um número ou uma tupla, enquanto o valor pode ser de qualquer tipo, como um inteiro, uma lista, uma string, um dicionário etc. O dicionário é representado por chaves {}, e os elementos são separados por vírgulas.

Para criar um dicionário em Python, pode-se utilizar a seguinte sintaxe:

```
meu_dicionario = {'chave1': valor1, 'chave2': valor2,...}
```

Aqui está um exemplo simples de como criar e acessar elementos de um dicionário em Python:

```
# Criando um dicionário
meu_dicionario = {'nome': 'João', 'idade': 25, 'cidade': 'São Paulo'}
# Acessando elementos do dicionário
print(meu_dicionario['nome'])    # Output: João
print(meu_dicionario['idade'])   # Output: 25
print(meu_dicionario['cidade'])  # Output: São Paulo
```

Para adicionar um novo elemento ao dicionário, basta atribuir um valor a uma nova chave:

```
# Adicionando um novo elemento ao dicionário
meu_dicionario['profissão'] = 'Engenheiro'
# Acessando o novo elemento adicionado
print(meu_dicionario['profissão'])  # Output: Engenheiro
```

Para atualizar um elemento existente no dicionário, basta atribuir um novo valor à chave existente:

```
11  # Atualizando um elemento existente no dicionário
12  meu_dicionario['idade'] = 30
13  # Acessando o elemento atualizado
14  print(meu_dicionario['idade'])   # Output: 30
```

Para remover um elemento do dicionário, pode-se utilizar o método pop() ou a instrução del:

```
15
16  # Removendo um elemento do dicionário com o método pop()
17  valor_removido = meu_dicionario.pop('cidade')
18  print(meu_dicionario)          # Output: {'nome': 'João', 'idade': 30, 'profissão':
    'Engenheiro'}
19  print(valor_removido)          # Output: São Paulo
20  # Removendo um elemento do dicionário com a instrução del
21  del meu_dicionario['idade']
22  print(meu_dicionario)          # Output: {'nome': 'João', 'profissão': 'Engenheiro'}
```

Os dicionários também podem ser percorridos com um loop for. É possível percorrer as chaves, os valores ou ambos:

```
23
24  # Percorrendo as chaves do dicionário
25  for chave in meu_dicionario:
26      print(chave)
27  # Percorrendo os valores do dicionário
28  for valor in meu_dicionario.values():
29      print(valor)
30  # Percorrendo as chaves e os valores do dicionário
31  for chave, valor in meu_dicionario.items():
32      print(chave, valor)
```

Em resumo, os dicionários são uma estrutura de dados poderosa e versátil em Python, permitindo armazenar informações em forma de pares chave-valor. Eles são úteis para muitas tarefas, desde o armazenamento de dados em bancos de dados até o processamento de linguagem natural.

Considerações Finais da Aula

Coleções de dados são fundamentais na programação, porque permitem organizar e manipular dados, de maneira eficiente e acessível, facilitando a implementação de códigos e funcionalidades complexas.

Em Python, existem diversas coleções de dados que podem ser usadas para armazenar e manipular valores:

- **Listas**: são uma coleção ordenada e mutável de elementos. É possível adicionar, remover e modificar elementos em uma lista. As listas são definidas utilizando colchetes [];
- **Matrizes**: são uma coleção bidimensional de elementos organizados em linhas e colunas. As matrizes oferecem uma maneira eficiente de processar grandes conjuntos de dados relacionados, já que as operações matemáticas podem ser executadas de forma vetorizada;
- **Tuplas**: são uma coleção ordenada e imutável de elementos. Ao contrário das listas, não é possível modificar os elementos que nela são criados. Elas são definidas utilizando parênteses ();
- **Conjuntos**: são uma coleção não ordenada de elementos únicos. Os conjuntos são mutáveis e são definidos utilizando chaves {};
- **Dicionários**: são uma coleção de pares chave-valor, em que cada valor é acessado por meio de sua chave. Os dicionários são mutáveis e são definidos utilizando chaves {}.

Cada tipo de coleção tem suas próprias características e métodos específicos, permitindo aos programadores escolher a melhor opção para cada situação.

Materiais Complementares

Livro:

Use a cabeça! Python
2019, Paul Barry. Alta Books.
Esse livro apresenta mais informações sobre os conceitos básicos da linguagem Python, além de sua sintaxe e seus comandos básicos.

Site

AlexandreLouzada/Pyquest: Pyquest/envExemplo/Lista03
2023, Alexandre N. Louzada. Github.
O link indicado permite o acesso a um repositório com várias listas de exemplo de programas em Python utilizando coleções.
Link para acesso: https://github.com/AlexandreLouzada/Pyquest/tree/master/envExemplo/Lista03 (acesso em 24 maio 2023.)

Referências

ALVES, William P. ***Programação Python***: aprenda de forma rápida. [s.l.]: Editora Saraiva, 2021.

PYTHON SOFTWARE FOUNDATION. ***Python Language Site***. Documentation, 2023. Página de documentação. Disponível em: https://docs.python.org/pt-br/3/tutorial/index.html. Acesso em: 8 mar. 2023.