Fundamentos de Programação

Estruturas de DADOS

Outras Estruturas

Aula 14

Post It da Aula Passada

Listas (Conceito)

- Coleção ordenada de dados
- Muito presente na vida cotidiana
- Uma única lista é composta por zero ou múltiplos elementos

Listas (Prática)

- Notação: []
- Listas são tipicamente atribuídas a variáveis:
  - var_lista = []
- Funções (notação ponto)

Matrizes

Nativamente, não existem matrizes em Python. Entretanto, podemos facilmente interpretar uma **lista de listas como uma matriz**

Ou seja, uma matriz nada mais é que uma lista contendo outras listas como elementos. Cada uma das listas internas são vistas como uma linha da matriz

Os elementos das listas internas são os valores de cada posição linha x coluna da matriz

Matrizes

mat_3x3 = [[1,2,3], [4,5,6], [7,8,9]]
mat_4x4 = [[1,2,3,4], [5,6,7,8], [9,10,11,12], [13,14,15,16]]
mat_2x4 = [[1,2,3,4], [5,6,7,8]]
mat_4x2 = [[1,2], [3,4], [5,6], [7,8]]

Usando essa estratégia de definição de matrizes, podemos fazer percorrer os elementos da forma clássica: i x j

Matrizes

mat_3x3 = [[1,2,3], [4,5,6], [7,8,9]]

0	1	2	
1	2	3	0
4	5	6	1
7	8	9	2

mat_4x2 = [[1,2], [3,4], [5,6], [7,8]]

0	1	
1	2	0
3	4	1
5	6	2
7	8	3

Matrizes

Para percorrermos os elementos de uma matriz podemos utilizar laços aninhados. Em particular, a estrutura de repetição *for* é muito útil nesses casos (**a própria matriz é um espaço de iteração**)

Considere a definição da seguinte matriz exemplo:

- mat_3x3 = [[1.2, 1.5, 2.8], [3.1, 2.7, 5.1], [9.2, 3.3, 5.2]]

Matrizes

Podemos percorrer cada elemento disponível, linha a linha:

```
for linha in mat_3x3:
    for elemento in mat_3x3:
        print(elemento)
```

Nesse caso não estamos manipulando os índices i (linha) e j (coluna) como geralmente fazemos na matemática

Matrizes

Podemos percorrer cada elemento disponível, linha a linha, utilizando índices i e j:

```
for i in range(len(mat_3x3)):
    for j in range(len(mat_3x3)):
        print(mat_3x3[i][j])
```

O resultado será o mesmo do exemplo anterior, porém ao invés de usarmos a própria matriz como espaço de iteração, usamos um espaço de iteração numérico gerado considerando os índices das listas

Matrizes

Podemos percorrer cada elemento disponível, coluna a coluna. Para isso, invertemos os índices i e j:

```
for j in range(len(mat_3x3)):
    for i in range(len(mat_3x3)):
        print(mat_3x3[i][j])
```

Nesse cenário, nós variamos a linha a cada iteração do laço interno, enquanto variamos a coluna em cada iteração do laço externo. Ou seja, acessamos todas as linhas de uma dada coluna antes de avançar para a próxima coluna

Matrizes

Naturalmente, visto que estamos trabalhando com listas aninhadas, todas as **funções e operadores de listas continuam sendo utilizáveis** em matrizes

count index sort append insert

pop remove del

Matrizes

Importante!

Muitas vezes quando estamos utilizando matrizes o posicionamento dos valores tem um significado intrínseco. Sendo assim, nem sempre faz sentido executar algumas funções de lista nessa estrutura de dados

Matrizes

Considere o seguinte sistema:

$$2*x + 2*y = 7$$
$$5*x - 3*y = 12$$

Organize o sistema em formato matricial utilizando uma variável matriz para os coeficientes e uma variável vetor para as constantes. **Calcule e exiba x e y utilizando a regra de Cramer**

Tuplas

Tuplas apresentam algumas características muito parecidas com listas: **são conjuntos de dados ordenados!**

Do ponto de vista de abstração, podemos entender as tuplas da mesma forma que compreendemos uma lista: uma estrutura de dados que indexa elementos pela sua posição

Tuplas

Diferente da notação de listas, tuplas utilizam parênteses para serem definidas: **()**
Como listas, tuplas podem ter de zero a múltiplos elementos, sendo estes elementos de diferentes tipos de dados: **(True, 1.0, “String”)**

E, também como listas, tuplas são naturalmente atribuíveis em variáveis: **var_tupla = (True, 1.0, “String”)**

Tuplas

Mas se temos todas essas características semelhantes, quando usaremos tuplas ao invés de listas?

Quando o conjunto de dados deve ser constante!

Diferente de listas, **tuplas não podem ter elementos removidos, modificados ou adicionados**: podemos apenas consultar os elementos

Tuplas

Se compararmos as operações que fazemos em listas com as operações possíveis em tuplas, teremos:

Possível

- consulta por índices
- manipulação head/tail
- len
- in
- index, count

Não possível

- append, insert
- pop, remove, del
- atribuição em uma posição
- sort

Tuplas

Porém, vale ressaltar que o tipo de dados ***tupla é compatível com o tipo de dados lista***. Logo, se precisarmos de uma estrutura de dados que nos permite modificar, adicionar e remover elementos, podemos realizar um *casting*:

```
var_lista = list(var_tupla)
```

O mesmo vale para o contrário:

```
var_tupla = tuple(var_lista)
```

Tuplas

Considere a seguinte **lista que contém tuplas**:
var_lista = [(40,551,2,1289), (2,68,332,698), (4,1236,65,889), (12,24,1544, 332)]

Faça um programa que verifique se, para os elementos ordenados por valor nessas tuplas, teríamos satisfeitas as seguintes condições:

- Primeira posição < 10
- Segunda posição >= 10 e < 100
- Terceira posição >= 100 e < 1000
- Quarta posição >= 1000

Remova as tuplas que não satisfazem as condições da lista var_lista e a exiba

Dicionários

Quando pensamos em um dicionário clássico, temos a imagem de uma lista de palavras onde cada uma dessas palavras conta com definições, sinônimos e aplicações

Os dicionários em Python extrapolam e generalizam essa característica de um dicionário clássico, criando uma estrutura que **associa chaves quaisquer a valores genéricos!**

Dicionários

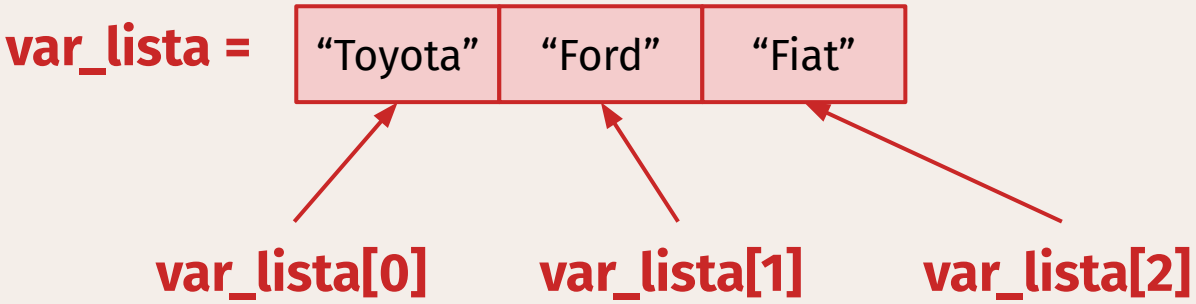
Como assim associar chaves quaisquer a valores genéricos?

Lembram dos índices das listas? Sim, aqueles que apontavam para uma posição específica das mesmas

Eles eram sempre numéricos! Ou seja, podemos **encontrar um dado específico em uma lista através da sua chave numérica**, representante da posição onde o mesmo está alocado

Dicionários

Vamos relembrar o acesso de elementos de uma LISTA via índices:



Dicionários

Diferente de listas, dicionários não utilizam o posicionamento de um elemento como a sua chave única

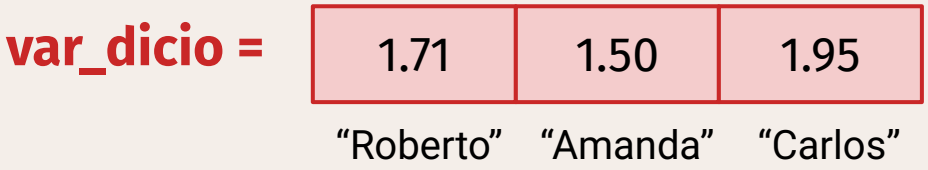
Aqui, **outros tipos de dados podem ser utilizados como chave**:

- Inteiros e Reais
- Strings
- Lógicos

Dicionários

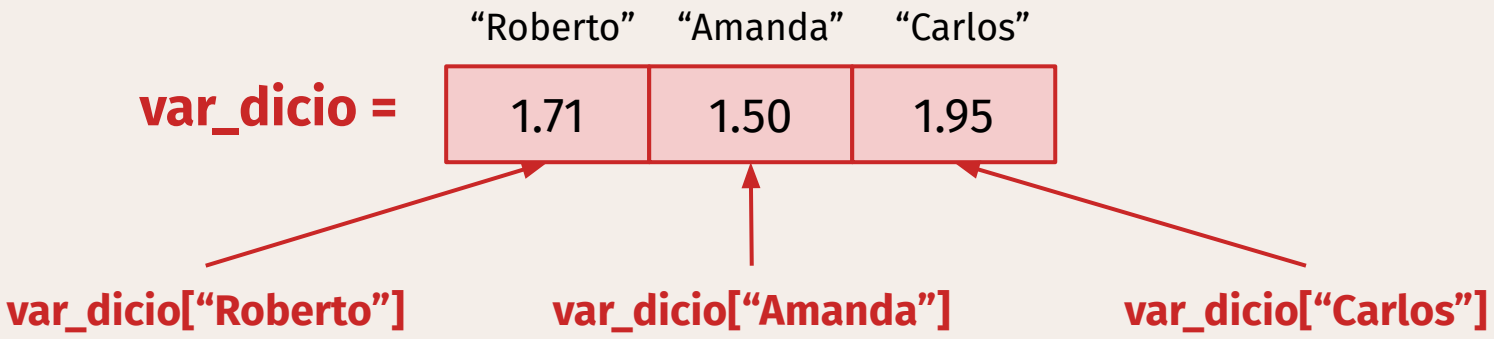
Ou seja, se estabelece uma relação conveniente entre uma chave e seu valor

Por exemplo, podemos criar um dicionário que relaciona o nome de pessoas com as suas respectivas alturas:



Dicionários

Estabelecida a relação, os índices para acesso aos valores se tornam as próprias chaves definidas



Dicionários

E como utilizar dicionários em Python?

1. A notação para a definição de uma lista em Python é dada por duas chaves: **{}**
2. Para uma predefinição de chave e valor, utilizamos dois pontos (:), a esquerda existe a chave e a direita existe o valor correspondente a ela: **{“Amanda”:1,50, “Carlos”:1.95}**

Dicionários

E como utilizar dicionários em Python?

3. Chaves, em geral, se restringem a tipos primitivos de dados, já valores podem ser absolutamente qualquer tipo de dado
4. Dicionários podem ser vazios (**{}**)
5. Dicionários são normalmente atribuídos a variáveis utilizando o operador de atribuição (**var_dicio = {“Amanda”:1,50, “Carlos”:1.95}**)

Dicionários

Vamos testar!

Crie os seguintes dicionários:
dicio_pessoas = {“Roberto”:1.71, “Amanda”:1.50, “Carlos”:1.95}
dicio_alturas = {1.50:[“Amanda”], 1.60:[], 1.70:[“Roberto”], 1.80:[], 1.90:[“Carlos”]}

Exiba os dicionários criados na tela:

```
print(dicio_pessoas)
print(dicio_alturas)
```

Dicionários

Dicionários funcionam nativamente como **espaços de iteração** (para a criação de laços *for*), porém a iteração se dá através de suas chaves, não pelos seus valores

```
for k in dicio_pessoas:
    print("Chave:", k)
    print("Valor:", dicio_pessoas[k])
```

```
for k in dicio_alturas:
    print("Chave:", k)
    print("Valor:", dicio_alturas[k])
```

Dicionários

O **operador *in*** também funciona no contexto de dicionários. Neste caso, ele verifica se a CHAVE provida à esquerda existe no conjunto de chaves do dicionário provido à direita

```
if “Carlos” in dicio_pessoas:
    print(dicio_pessoas[“Carlos”])
```

```
print(1.40 in dicio_alturas)
```

Dicionários

Ainda, podemos utilizar a **função *len*** de maneira similar ao que utilizamos para listas. Nesse caso, a função retorna a quantidade elementos (chave:valor) existentes no dicionário

```
len(dicio_pessoas)
```

```
print(len(dicio_alturas))
```

Dicionários

Para adicionar ou substituir valores em um dicionário, podemos utilizar o operador de atribuição. Para isso, indicamos a chave respectiva ao valor como o índice do dicionário utilizando colchetes

dicio_pessoas[“Carlos”] = 1.92

dicio_alturas[1.40] = []

Dicionários

A remoção de valores do dicionário é feita através da função ***pop* (notação ponto)**, recebendo como argumento a chave do elemento que deve ser removido. A função retorna o valor correspondente à chave removida

```
alt = dicio_pessoas.pop(“Roberto”)
alt = math.floor(alt * 10) / 10
dicio_alturas[alt].remove(“Roberto”)
```

```
dicio_alturas.pop(1.40)
```

Dicionários

Outra forma de remover um elemento é através do operador unário *del*. Basta utilizar o mesmo como fazemos para as listas: **à direita do operador inserir o elemento a ser removido**

del dicio_pessoas[“Amanda”]

del dicio_pessoas[1.50]

Dicionários

Eventualmente, podemos precisar das chaves existentes em um dicionário. Existe uma função em notação ponto que retorna um iterador com tais chaves: ***keys***. Como não estudaremos iteradores, podemos *castar* o mesmo para uma lista

```
lista_k = dicio_pessoas.keys()
lista_k = list(lista_k)
print(lista_k)
```

```
print(dicio_alturas.keys())
```

Dicionários

O mesmo que falamos para as chaves no *slide* anterior pode ser realizado para os valores relacionados às mesmas. Nesse caso, a função em notação ponto necessária é a ***values***

```
lista_k = dicio_pessoas.values()
lista_k = list(lista_k)
print(lista_k)
```

```
print(dicio_alturas.values())
```

Dicionários

Por fim, existe uma função em notação ponto para recuperar tuplas no formato (chave, valor) do dicionário por inteiro: ***items***. Mais uma vez o retorno é um iterador, o qual pode ser transformado em lista via *casting*

```
lista_k = dicio_pessoas.items()
lista_k = list(lista_k)
print(lista_k)
```

```
print(dicio_alturas.items())
```

Dicionários

Faça um programa para cadastro e consulta de valores de produtos

O programa deve fornecer duas opções ao usuário: cadastrar e consultar

O cadastro recebe da entrada padrão o nome do produto e seu valor em reais, **armazenando essa relação em um dicionário**
A consulta recebe o nome de um produto e **verifica se o mesmo está no dicionário, se sim, exibe o valor; se não, exibe um erro**

Exercício #14

Faça um programa que **leia um sistema 2x2 em formato matricial** da entrada padrão e o resolva utilizando a regra de Cramer

Cada coluna da matriz representa os coeficientes de uma incógnita. Devemos também **ler um vetor contendo a parte constante** das equações

O programa deve exibir como saída os valores das duas incógnitas, solução para o sistema fornecido

Fundamentos de Programação

Aula 14

Obrigado e ATÉ A PRÓXIMA!