Programação Python

Aula 03: Listas

Prof. Eduardo Corrêa Gonçalves

23/03/2021

Sumário

Introdução (1/2)

- **Lista**
  - Coleção ordenada de *n* elementos.

  - Cada elemento está associado a um **índice**
    - Número que indica a posição do elemento na sequência.
    - O primeiro índice é *0* e o último *n-1*.
  - Características:
    - Permite elementos **duplicados**.
    - **Mutável** – pode ser alterada.
    - **Iterável** – capaz de retornar seus elementos um por vez em um laço.
    - **Sequência** – elementos possuem ordem determinada.
    - **Popular** – Estrutura de dados (ED) nativa mais usada do Python.
    - **Similar ao array** das linguagens tradicionais, porém mais flexível.

Introdução (2/2)

- **Lista**
  - Há **3 tipos** de lista:
    - **Lista simples** (elementos de um único tipo básico)

[0]	[1]	[2]	[3]	
John	Yoko	Julian	Sean	*lst_familia*

    - **Lista mista** (elementos de diferentes tipos)

[0]	[1]	[2]	[3]	
pen drive	25.90	laptop	2690	*lst_eletronicos*

    - **Lista bidimensional** ou **2d** (lista de listas)

[0]	[1]	[2]	
[5, 0, 3, 1]	[10, 20]	[0,1, 2]	*lst_numeros*

Criação de Listas

- **Criando Listas**
  - Devemos especificar uma sequência de valores entre colchetes, onde os valores devem estar separados por vírgula:
    - escritores = ['Jorge Amado', 'José Saramago', 'Aldous Huxley']
    - sequencia_fibonacci = [0,1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
    - lista_vazia = []
    - lst_mista = ['Pen Drive', 25.90, 'Laptop', 2690]
    - m = [[1, 2, 3],
      [4, 5, 6]] *#lista de listas (similar a uma matriz)*

Propriedades e Operações (1/9)

- **Operações básicas**:
  - Recuperar elemento do índice *i*
  - Modificar elemento do índice *i*
  - Verificar se elemento *x* pertence à lista
  - Iterar (percorrer todos os elementos *em ordem*)
  - Fatiar (obter uma sublista)
  - Adicionar um elemento *x* (em qualquer posição)
  - Remover um elemento
  - Contar número de ocorrências do elemento *x*
  - Obter índice do elemento *x*
  - Inverter
  - Ordenar
  - ...

Propriedades e Operações (2/9)

- **Recuperando elementos pelo índice**

```
lst = ["John", "Yoko", "Julian", "Sean"]

primeiro = lst[0]              # 'John'
ultimo = lst[3]                # 'Sean'
ultimo_tambem = lst[-1]        #também retorna 'Sean'
penultimo = lst[-2]            # 'Julian'
```

- **Modificando um elemento pelo índice**

```
lst[1] = "Yoko Ono"        # ["John", "Yoko Ono", "Julian", "Sean"]
lst[0] = "John Lennon"     # ["John Lennon", "Yoko Ono", "Julian", "Sean"]
```

Propriedades e Operações (3/9)

- **Verificando se elemento pertence à lista**

```
lst = ["John", "Yoko", "Julian", "Sean"]

"Julian" in lst        # True
"Ringo" in lst         # False
```

- **Obtendo as propriedades de uma lista**

```
# tipo do objeto lista
type(lst)              # <class 'list'>

# tamanho de lst
len(lst)               # 4
```

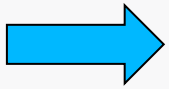
- **Comparação de Listas – é feita elemento por elemento, de forma lexicográfica**

```
[1,2,3] > [1, 5, 10]          # False
[20] > [10, 998, 800]         # True
```

Propriedades e Operações (4/9)

- **Iterando**

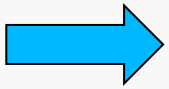
```
lst = ["John", "Yoko", "Julian", "Sean"]

for pessoa in lst:
    print(pessoa)
```



```
John
Yoko
Julian
Sean
```

- **Iterando com base nos índices**

```
for k in range(len(lst)):
    print("elemento {} = {} ".format(k, lst[k]))
```



```
elemento 0 = John
elemento 1 = Yoko
elemento 2 = Julian
elemento 3 = Sean
```

Propriedades e Operações (5/9)

- **Fatiando – funciona da mesma forma que a função range()**

```
lst = ["John", "Yoko", "Julian", "Sean"]

lst[0:2]        # ['John', 'Yoko']
lst[0:3]        # ['John', 'Yoko', 'Julian']
lst[2:4]        # ['Julian', 'Sean']
lst[:2]         # ['John', 'Yoko']
lst[2:]         # ['Julian', 'Sean']

lst[::2]        # ['John', 'Julian']
lst[1::2]       # ['Yoko', 'Sean']

lst[-3:]        # ['Yoko', ', 'Julian', 'Sean']
lst[:-3]        # ['John']

# ver "Meu Primeiro Livro de Python", pag. 62 para detalhes
```

Propriedades e Operações (6/9)

- **Modificando vários elementos de uma vez (modificação no estilo fatiamento):**

```
lst = ["John", "Yoko", "Julian", "Sean"]

lst[1:4] = ["Paul", "George", "Ringo"]       # ['John', 'Paul', 'George', 'Ringo']
```

- **Repetição e Concatenação**

```
a = ["John", "Paul"]
b = ["George", "Ringo"]

a * 3          # ['John', 'Paul', 'John', 'Paul', 'John', 'Paul']
a + b          # ['John', 'Paul', 'George', 'Ringo']
```

Propriedades e Operações (7/9)

- **Métodos disponíveis para lista**
  - lista.**append(*x*)**: insere elemento *x* no final da lista.
  - lista.**insert(*i, x*)**: insere elemento *x* na posição *i*.
  - lista.**pop*(i)***: remove um elemento no final (caso *i* não seja especificado) ou da posição *i* da lista (caso *i* seja especificado). O método retorna o elemento removido.
  - lista.**remove(*x*)**: remove o primeiro elemento que tiver o valor *x*.
  - lista.**clear()**: esvazia a lista
  - lista.**extend(*lista2*)**: concatena os elementos da *lista2* (ou outro iterável) ao final de *lista*. Também pode-se usar *lista* += *lista2*.
  - lista.**count(*x*)*:*** conta o número de ocorrências do elemento *x*.
  - lista.**index(*x, inicio, fim*)**: obtém o índice da primeira ocorrência do elemento *x* na lista inteira ou dentro da faixa especificada em *início* e *fim*.
  - lista.**reverse()**: inverte a ordem da lista.
  - lista.**sort(*reverse=False/True*)**: ordena a lista em ordem ascendente (*reverse=False*) ou descendente (*reverse=True*). O default é ascendente.

Propriedades e Operações (8/9)

- **Inserindo e Removendo Elementos**

```
numeros = [5, 10]

numeros.append(20)        # insere 20 no final: [5, 10, 20]
numeros.append(10)        # insere 10 no final: [5, 10, 20, 10]
numeros.insert(2,15)      # insere 15 na posição 2: [5, 10, 15, 20, 10]
numeros.insert(0,10)      # insere 10 na posição 0: [10, 5, 10, 15, 20, 10]

numeros.pop()             # remove o último elemento: [10, 5, 10, 15, 20]
numeros.pop(3)            # remove o quarto elemento: [10, 5, 10, 20]
numeros.remove(10)        # remove o primeiro 10: [5, 10, 20]

numeros.extend([40, 50])  # estende a lista: [5, 10, 20, 40, 50]

del numeros[1:3]          # com del posso apagar 1 ou mais elementos: [5, 40, 50]

numeros.clear()           # esvazia a lista: [ ]
```

Propriedades e Operações (9/9)

- **Contando o números de ocorrências de um elemento**

```
notas = [95, 70, 75, 100, 70, 65, 70, 100]
notas.count(70)          # 3  (= 3 ocorrências)
```

- **Obtendo o primeiro índice de um elemento**

```
notas.index(70)          # 1
notas.index(70, 2, 8)    # 4
notas.index(65)          # 5
```

- **Invertendo e ordenando a Lista (a lista é modificada)**

```
notas.reverse()          # [100, 70, 65, 70, 100, 75, 70, 95]
notas.sort()             # [65, 70, 70, 70, 75, 95, 100, 100]
notas.sort(reverse=True) # [100, 100, 95, 75, 70, 70, 70, 65]
```

- **Clonando uma Lista**

```
notas_clone = notas[:]   # “notas_clone” é uma nova lista em memória com
                         # o mesmo conteúdo de “notas”
```

Tarefa

(1) – Dada uma lista com 3 números, faça um programa que identifique o menor, o maior e o do meio.

(2) – Faça um programa que:

i. gere uma lista v com 6 elementos repetidos utilizando a operação de repetição.

ii. faça um clone w da lista gerada.

iii. imprima as duas listas

iv. Altere o último elemento da lista w.

v. imprima novamente as duas listas

(3) – Dada uma lista de números, imprima os dois maiores valores desconsiderando resultados repetidos.

Ex: [84,84,86,2,85,85,0,83,23,45,84,86,1,2,85] → deve retornar 86, 85

Tarefa

(4)- A partir da lista cores= ['amarelo', 'azul', 'branco', 'preto', 'verde', 'vermelho'], produza as novas listas a seguir utilizando a operação de fatiamento ou métodos de lista.

l1 = ['amarelo', 'azul']

l2 = ['azul', 'branco', 'preto']

l3 = ['vermelho', 'verde', 'preto', 'branco', 'azul', 'amarelo']

l4 = ['amarelo', 'azul', 'preto', 'verde', 'vermelho']

l5 = ['preto', 'verde', 'vermelho']

l6 = ['amarelo', 'azul', 'branco', 'preto', 'rosa', 'verde', 'vermelho']

Obs: a lista original não pode ter o seu conteúdo modificado.

Lista Bidimensional (1/5)

- **Conceito**
  - Trata-se de uma **lista de listas**, ou seja, uma lista onde cada elemento também é uma lista.

$$m = [\ [1, 2, 3], [4, 5, 6]\ ]$$

  - Também chamada de lista 2d.
  - Útil para a representação de **matrizes** no Python padrão.
    - **Obs1.**: para o Python **não** é uma matriz... É uma lista de listas, mas por conveniência, podemos “fingir” que é uma matriz.
    - **Obs2.**: também é possível criar listas 3d (lista de listas de listas), listas 4d, ou listas de maior dimensão.

Lista Bidimensional (1/4)

- **Criando Listas 2d**
  - Lista 2d é uma “lista de listas”, ou seja, uma lista onde cada elemento também é uma lista.
    - Para criar, devemos especificar uma relação de listas separadas por vírgula entre colchetes.
    - Podemos “fingir” que a lista 2d é uma matriz! Basta tratar cada lista como se fosse uma linha da matriz.

- 4 linhas e 6 colunas

```
matriz_binaria = [
    [0, 1, 0, 0, 1, 0],
    [1, 0, 0, 1, 1, 1],
    [0, 0, 0, 0, 0, 1],
    [0, 1, 1, 0, 1, 0]
]
```

- Lista 2d. com 5 linhas, onde cada uma delas possui um número diferente de colunas (*mais uma prova da flexibilidade das listas*).

```
nao_retangular = [
    [8, 0, 5, 3, 0, 9],
    [2, 6, 0],
    [ ],
    [3, 8, 3],
    [4, 7]
]
```

Lista Bidimensional (2/4)

- **Recuperando elementos pelos índices**

```
>>> m = [
   [10, 20, 30],
   [40, 50, 60],
   [70, 80, 90]
]

# o 1º elemento de m
# é uma lista de números…
>>> m[0]
[10, 20, 30]

#o primeiro elemento dessa lista é 10,
#o segundo 20 e o terceiro 30
>>> m[0][0]
10
>>> m[0][1]
20
>>> m[0][2]
30
```

```
# o 2º elemento de m
# também é uma lista de números…
>>> m[1]
[40, 50, 60]

#o primeiro elemento dessa lista é 40,
#o segundo 50 e o terceiro 60
>>> m[1][0]
40
>>> m[1][1]
50
>>> m[1][2]
60
```

- **Obs.**: infelizmente **não** há uma forma direta de processar (recuperar ou alterar) uma **coluna inteira**.
  - Isso porque, os elementos de uma coluna estão armazenados em listas diferentes.

Lista Bidimensional (3/4)

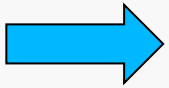
- **Modificando elementos**

```
>>> m = [
   [10, 20, 30],
   [40, 50, 60],
   [70, 80, 90]
]

#modifica o 2º elemento da 1ª linha
>>> m[0][1]  = 999

>>> m
[[10, 999, 30], [40, 50, 60], [70, 80, 90]]

#modifica toda a 2ª linha
>>> m[1]  = [-1, 100, 50]

>>> m
[[10, 999, 30], [-1, 100, 50], [70, 80, 90]]
```

Lista Bidimensional (4/4)

- **Iterando por linhas e colunas**

```
m = [
   [1, 2, 3],
   [4, 5, 6]
]

num_linhas = len(m)
for i in range(num_linhas):
    num_cols = len(m[i])
    for j in range(num_cols):
        print(m[i][j], end = " ")
    print()
```



```
1 2 3
4 5 6
```

- No exemplo, utilizamos 2 laços (um para as “linhas” e outro para as “colunas”).
- Veremos a partir dos próximos slides que o uso do recurso **list comprehension** torna mais prático o processamento de listas 2d (entre muitas outras coisas...)

List Comprehension (1/10)

- **Conceito**
  - Trata-se de uma **notação matemática** que facilita a criação e o processamento de listas.
  - **Exemplo**: Suponha que você queira criar uma lista com as potências de 2 variando de 0 a 16. Isto é: $[2^0, 2^1, ..., 2^{16}]$.
  - Forma **tradicional**:
    - lst = [1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536]
  - Usando **list comprehension**:
    - lst = [2**x **for** x **in range**(17)]
    - Literalmente: *para x variando de 0 a 16, faça cada elemento da lista igual a* 2^x
    - Solução elegante, compacta e bem mais parecida com a notação matemática $\{2^0, 2^1, ..., 2^{16}\}$

List Comprehension (2/10)

- **Sintaxe Básica**
  - List comprehension **cria uma nova lista** a partir de uma definição concisa. Abaixo a sintaxe básica do comando:
    - lst = [*operação sobre x* **for** *x* **in** *coleção-fonte*]
    - Onde:
      - ***coleção-fonte***: é uma coleção de valores utilizados como fonte de dados para a geração da lista. Pode ser um **range()** ou uma **ED iterável** (lista, tupla, conjunto, etc.).
      - ***x***: recebe cada elemento da coleção-fonte.
      - ***operação sobre x***: representa o cálculo que faremos sobre x para gerar os elementos da nova lista. No exemplo anterior, fizemos 2**x.

```
lst = [2**x for x in range(17)]
```

List Comprehension (3/10)

- **Criando Listas**
  - Mais alguns exemplos de criação de listas:

Notação Matemática	List Comprehension
$A = \{x^3 \mid 0 \leq x \leq 10\}$	A = [x**3 **for** x **in range**(11)]
B = {1/2, 1/4, …, 1/10}	B = [1/x **for** x **in range**(2,11,2)]
C = {0, 0 0, 0 0, 0 0, 0 0, 0}	C = [([0]*2) **for** linha **in range**(5)]

List Comprehension (4/10)

- **Operações**
  - Veremos agora como utilizar o recurso list comprehension para facilitar a execução de dois tipos de operações aritméticas sobre listas:

    1. Operações com escalares

    2. Operações envolvendo duas listas de tamanho compatível.

List Comprehension (5/10)

- **Operações com escalares**

```
a = [1, 2, 3, 4]

b= [x+10 for x in a]      # [11, 12, 13, 14]      (somou 10 a cada elemento)
c= [x-1 for x in a]       # [0, 1, 2, 3]          (subtraiu 1 de cada elemento)
d  = [x * 2 for x in a]   # [2, 4, 6, 8]          (multiplicou todos por 2)
e  = [x / 2 for x in a]   # [0.5, 1.0, 1.5, 2.0]  (dividiu todos por 2)

#Você pode usar qualquer função ...seja de um módulo ou alguma que você criou.
#Ex.: módulo math, função sqrt (raiz quadrada).
import math
raiz  = [math.sqrt(x) for x in a]   # [1.0, 1.4142135623730951, 1.7320508075688772, 2.0]
```

List Comprehension (6/10)

- **Operações com duas listas**

```
>>> a=[1,2,3,4]
>>> b=[100,200,300,400]

# soma os elementos que estão na mesma posição nas duas listas
>>> soma = [a[i] + b[i] for i in range(len(a))]
>>> soma
[101, 202, 303, 404]
```

List Comprehension (7/10)

- **Incluindo condicionais (testes lógicos)**
  - É possível adicionar testes lógicos para **filtrar** e **transformar** dados.
    - EXEMPLO 1 - **FILTRAR**
    - Significa descartar alguns elementos.
    - Nesse caso, o **teste lógico** deverá ser incluído **no final**.

```
# gerar lista apenas com os números ímpares entre 1 e 20
>>>D = [x for x in range(20) if x%2==1]
>>> D
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]

# gera “b” a partir de “a”, incluindo só os números entre 20 e 35
>>> a=[10,20,30,40]
>>> b = [x for x in a if x >= 20 and x <= 35]
>>> b
[20, 30]
```

List Comprehension (8/10)

- **Incluindo condicionais (testes lógicos)**
  - EXEMPLO 2 - **TRANSFORMAR**
    - Nesse caso, o **teste** deverá ser incluído **no início**.

```
# gera “b” a partir de “a” substituindo os números negativos por 0
>>> a = [-1, 3, 8, -5, 5]
>>> b = [x if x > 0 else 0 for x in a]
>>> b
[0, 3, 8, 0, 5]
```

List Comprehension (9/10)

- **Listas aninhadas (*nested lists*)**
  - Uma list comprehension pode ser aninhada dentro de outra. Isto é especialmente útil para manipular listas 2d. Neste exemplo, transformamos a matriz em um vetor.

```
>>> m = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
#pega cada linha (sublista) com "for linha in m"
#e depois cada elemento x da linha com "for x in linha".
>>> [item for linha in m for item in linha]
[1, 2, 3, 4, 5, 6, 7, 8, 9]

#dá até para obter a soma (ou máximo, mínimo, ...) dos elementos da matriz
>>> sum([item for linha in m for item in linha])
45
```

List Comprehension (10/10)

- **Listas aninhadas (*nested lists*)**
  - Abaixo uma comparação entre as resoluções sem e com list comprehension.

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

```
#sem list comprehension
v=[]
for linha in m:
    for item in linha:
        v.append(item)

print(v)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
#com list comprehension
v = [item for linha in m for item in linha]

print(v)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Resumo - Listas

- **Pontos fortes**:
  - Simples e flexível.
  - Suporta muitas operações.
  - A ED mais popular do Python.
    - *Você não consegue fazer nada em Python se não entender as listas!*
- **Pontos fracos**:
  - Pouco prática e não tão eficiente (lenta) para operações matemáticas e manipulação de matrizes.
    - Nestes caso, melhor usar a NumPy
- **Quando usar?**:
  - Sempre que for possível, já que é uma ED muito simples e “pop”!
  - Quando você trabalhar com uma coleção ordenada de itens.
  - Quando houver elementos que podem se repetir.

Tarefa

(5) - Faça um programa gere uma nova lista lst2 contendo todos os números de uma lst1 multiplicados por -1

lst1 = [1,2,3,4,5] → lst2 = [-1,-2,-3,-4,-5]

(6) – Dado um inteiro positivo *n*, implemente uma *list comprehension* capaz de calcular número harmônico H_n definido por:

$$H_n = \sum_{k=1}^{n} \frac{1}{k}$$

(7) – Dada uma matriz quadrada de ordem *n*, estruturada em uma lista 2d, calcule a soma da diagonal principal. Exemplo:

[[1,3,5,7],[1,4,6,0],[7,6,9,0],[1,2,3,4]] → 1 + 4 + 9 + 4 => 18

- Obs.: faça um programa que funcione para qualquer n.

Referências

- Corrêa, E. (2020). “Meu Primeiro Livro de Python”. V 2.0.0, edubd, 2020. (*capítulo 3*).
  - Disponível em: https://github.com/edubd/meu_primeiro_livro_de_python

- Willems, K. (2019). 18 Most Common Python List Questions. https://www.datacamp.com/community/tutorials/18-most-common-python-list-questions-learn-python

- Timmins, J. (2019). When to use a List Comprehension in Python. https://realpython.com/list-comprehension-python/