

Introdução à Programação

Aula 10: Listas (parte 2)

Prof. Eduardo Corrêa Gonçalves

02/04/2024

Sumário

Listas (Parte 2)

Fatiamento

Métodos de Lista (parte 2)

Carregando uma Lista via Teclado

Cópia Rasa x Cópia Profunda

List Comprehension

Exercícios

Fatiamento (1/3)

- **Fatiamento (*slicing*)**: operação que consiste em **gerar uma sublista** a partir de uma lista existente.
 - A definição dos elementos que vão compor a sublista é feita a partir de uma **sintaxe similar** a utilizada com a função **range()**.

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

```
familia[0:3]          # ['John', 'Yoko', 'Julian']
```

```
familia[2:4]          # ['Julian', 'Sean']
```

```
familia[:2]           # ['John', 'Yoko']
```

```
familia[2:]           # ['Julian', 'Sean']
```

```
familia[::2]           # ['John', 'Julian']
```

```
familia[1::2]          # ['Yoko', 'Sean']
```

```
familia[-3:]           # ['Yoko', 'Julian', 'Sean']
```

```
familia[:-3]           # ['John']
```

Fatiamento (2/3)

- **Sintaxe para fatiar uma lista w:** `w[início:fim:incremento]`
 - **Até 3 parâmetros** podem ser utilizados:
 - início, fim e incremento
 - Eles devem ser separados por dois pontos “:”
 - **início:** posição inicial da fatia. Caso seja omitido, o valor 0 é assumido.
 - **fim:** a fatia englobará até, mas sem incluir, o elemento na posição indicada neste parâmetro.
 - **ENTÃO MUITO CUIDADO!!!** A fatia nunca irá incluir o elemento da posição que você especificar no parâmetro fim. Ela sempre vai parar 1 valor antes. Caso seja omitido, o valor referente ao total de elementos da lista é assumido.
 - **passo:** define o “pulo” que será dado para obter o próximo elemento da fatia. Se omitido, o valor 1 será utilizado.
- **IMPORTANTE:** o fatiamento **retorna uma nova lista** e não uma visão da lista original.

Fatiamento (3/3)

- Formas mais comuns de utilizar os parâmetros:

- `lst[i:j]`: do elemento de índice i ao de índice $j-1$.
- `lst[i:]`: do elemento de índice i até o último da lista.
- `lst[:j]`: do primeiro elemento da lista (índice 0) ao elemento de índice $j-1$.
- `lst[i:j:k]`: do elemento de índice i , até, no máximo, o de índice $j-1$, utilizando o passo k .
- `lst[-k:]`: obtém os k últimos elementos da lista.
- `lst[:-k]`: em uma lista com n elementos, retornará os primeiros $n-k$ elementos.

- Também é possível fatiar da direita para esquerda, com a sintaxe: `lst[n:m:-k]`, onde $n > m$.

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

```
familia[::-1]          # ['Sean', 'Julian', 'Yoko', 'John'] => retorna a lista invertida!
```

```
familia[3:1:-1]        # ['Sean', 'Julian']
```

```
familia[2::-1]         # ['Julian', 'Yoko', 'John']
```

Métodos de Lista – parte 2 (1/3)

- As operações de **busca**, **contagem**, **inversão** e **ordenação** podem ser realizadas de maneira muito simples.
- Isso porque existem **métodos** disponíveis para executá-las.
- Como vimos na última aula, um método é uma função que você “ganha de presente” sempre que cria uma lista.
 - Você já conheceu os métodos **append()**, **pop()** e outros. Agora vamos complementar apresentando os métodos:
 - **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 *inicio* e *fim*.
 - **lista.reverse()**: inverte a ordem da lista (modifica a lista)
 - **lista.sort(reverse=False/True)**: ordena a lista em ordem ascendente de acordo com os valores dos elementos (*reverse=False*) ou descendente (*reverse=True*). O *default* (padrão) é ascendente. (modifica a lista)

Métodos de Lista – parte 2 (2/3)

- Exemplo 1:

```

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

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

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

# veja que reverse() e sort() alteram a lista sem que seja
# preciso fazer uma atribuição. Outros métodos que alteram a lista
# como append(), insert(), pop() e remove() também são assim !!!

```

Métodos de Lista – parte 2 (3/3)

- Exemplo 2:

```
notas = [95, 70, 75, 100, 70, 65, 70, 100]
```

```
# o código abaixo ocasiona erro, pois 50 não está na lista
```

```
print(notas.index(50))
```

```
# para evitar, usa-se o if com in antes do index
```

```
if 50 in notas:
```

```
    print(notas.index(50))
```

```
else:
```

```
    print('50 não faz parte notas')
```


Carga de Lista via Teclado (1/3)

- Nos exemplos apresentados até agora, nossas listas foram **definidas diretamente dentro do código** do programa, como no exemplo a seguir:

```
familia = ['John', 'Yoko', 'Julian', 'Sean']
```

- Porém, em programas reais quase sempre existe entrada de dados por parte do usuário, seja via teclado ou através de arquivo.
- Por esse motivo, apresentamos agora a **receita básica** para que você possa carregar listas a partir da digitação do usuário:

1. criar uma lista vazia

2. criar um laço para receber cada informação

- 2.1 dentro do laço, receber cada informação via input()

- 2.2 dentro do laço, inserir cada informação na lista usando o método append()

Carga de Lista via Teclado (2/3)

- O programa a seguir mostra como usar a receita na prática.
 - Ele cria uma lista contendo 5 nomes de filmes lidos via teclado.
 - Cada nome lido é inserido no fim da lista com o uso do método `append()`.
 - Digite o programa e teste-o com diferentes entradas.

```
print('Digite os nomes dos 5 filmes que você mais gosta:')
filmes = [] # 1. Cria a lista vazia
for i in range(5): # 2. Cria um laço para receber as
    informações
    titulo = input() # 2.1 Recebe uma informação
    filmes.append(titulo) # 2.2 Insere a informação na lista

print('Os seus 5 filmes favoritos são:')
print(filmes)
```

- **IMPORTANTE:** Também seria possível fazer usar o `input()` dentro do `append()`, sem carregar os dados primeiros para a variável “titulo”:
 - `filmes.append(input())`.

Carga de Lista via Teclado (3/3)

```
print('Digite os nomes dos 5 filmes que você mais gosta:')
filmes = [] # 1. Cria a lista vazia
for i in range(5): # 2. Cria um laço para receber as
    informações
        titulo = input() #2.1 Recebe uma informação
        filmes.append(titulo) # 2.2 Insere a informação na
        lista

print('Os seus 5 filmes favoritos são:')
print(filmes)
```

Exemplo de execução:

```
>>>
Digite os nomes dos 5 filmes que você mais gosta:
Edukators
Casablanca
O Filho da Noiva
O Discreto Charme da Burguesia
Orgulho e Preconceito
Os seus 5 filmes favoritos são:
['Edukators', 'Casablanca', 'O Filho da Noiva', 'O Discreto Charme da
Burguesia', 'Orgulho e Preconceito']
```

Cópia Rasa x Cópia Profunda (1/3)

- Função `id()`
 - Com o uso da função *built-in* `id()` é possível obter a **identidade** de um objeto Python.
 - É um número inteiro único e constante que perdurará durante o tempo de vida do objeto para uma dada execução do programa.
 - Você pode considerar que a identidade corresponde ao endereço do objeto em memória (embora não seja exatamente isso)

```
>>> lst1 = ['John', 'Paul']
>>> lst2 = ['George', 'Ringo']
>>>
>>> id(lst1)          #no seu computador vai dar outro valor
2327190461376
>>>
>>> id(lst2)          #no seu computador vai dar outro valor
2327191884352
>>>
```

Cópia Rasa x Cópia Profunda (2/3)

- Se você tem uma lista *a* e quiser gerar um clone *b*, **não** faça ***b = a***.
 - A atribuição criará apenas uma nova variável *b* que irá referenciar (ou apontar para) o endereço de *a*.
 - Ou seja: *a* e *b* terão o **mesmo id** e serão a mesma lista. Basicamente *b* é apenas uma **visão** de *a*.
 - Isso é chamado de **cópia rasa** (*shallow copy*).

```
>>> a = [1,2]
>>> b = a
>>>
>>> id(a)
2327191884480
>>>
>>> id(b)
2327191884480
>>>
```

Cópia Rasa x Cópia Profunda (3/3)

- Para clonar de verdade, use um dos métodos abaixo:
 - `b = a[:]` *# o resultado do fatiamento é sempre um clone*
 - `b = a.copy()` *# copy() é um método para clonar uma lista*
- Neste caso, *a* e *b* serão listas diferentes, portanto, com ids diferentes.
- Isso é chamado de **cópia profunda** (*deep copy*).

```
>>> a = [1,2]
>>> b = a[:]
>>>
>>> id(a)
2327191885184
>>>
>>> id(b)
2327191884672
>>>
```

List Comprehension (1/9)

- Conceito (1/2)

- Conforme sabemos, a ED lista não é muito prática para uso em operações matemáticas. Veja:

```
>>> a = [1, 2, 3]
>>> b = a * 2          #não multiplica cada elemento por 2, pois o asterisco é
>>> b                  #na verdade o operador de repetição
[1, 2, 3, 1, 2, 3]
```

#para resolver do jeito tradicional eu tenho que:

(i) instanciar "b;" (ii) fazer um for em "a"; (iii) fazer o cálculo e dar o append em "b".

```
>>> b = []
>>> for elemento in a:
        b.append(elemento * 2)
>>> b
[2, 4, 6]
```

#enfim, não dá para somar/subtrair/multiplicar/dividir todos os elementos de uma vez...

```
>>> c = a / 2
```

Traceback (most recent call last):

File "<pyshell#5>", line 1, in <module>

c = a/2

TypeError: unsupported operand type(s) for /: 'list' and 'int'

List Comprehension (2/9)

- **Conceito (2/2)**
 - Em algumas situações, torna-se possível contornar essa “inabilidade matemática” das listas utilizando o recurso **list comprehension**.
 - 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: $[2^0, 2^1, \dots, 2^{16}]$.
 - 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*
 - Elegante, compacto e bem mais parecido com a notação $[2^0, 2^1, \dots, 2^{16}]$

List Comprehension (3/9)

- **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 (4/9)

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

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

List Comprehension (5/9)

- **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 (6/9)

- 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 (7/9)

- 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 (8/9)

- 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 (9/9)

- 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]
```

Exercícios

- (1) A partir da lista `cores= ['amarelo', 'azul', 'branco', 'preto', 'verde', 'vermelho']`, produza as novas listas a seguir utilizando os métodos de lista ou a operação de fatiamento. **Obs.:** a lista original não pode ter o seu conteúdo modificado.
 - `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']`
- Ao final, imprima a lista cores original e todas as listas geradas.
- (2) Utilizando **list comprehension** faça um programa **com uma única linha de código** que gere uma lista *H* com os 50 primeiros termos da série:

$$H = 1 + (1 / 2) + (1 / 3) + \dots + (1 / 50).$$

Exercícios

(3)- Faça um programa que:

- i. carregue uma lista v com 6 elementos via teclado.
- ii. faça um clone w da lista gerada (deep copy).
- iii. imprima as duas listas
- iv. imprima o id da lista original v e de seu clone w.
- v. Altere o último elemento da lista w.
- vi. Imprima o id do primeiro elemento de v e do primeiro de w.
- vii. Imprima o id do último elemento de v e do último de w.