

Departamento de Engenharia de Sistemas e Computação

Operações Avançadas com Listas

Métodos, clonagem de listas e listas como parâmetros

João Araujo

Algoritmos Computacionais I

Aula 18

Roteiro

- Operações em listas: `in`, `del`, métodos (Cap. 5.7)
- Clonando listas: aliasing e cópia (5.8)
- Listas como parâmetros de funções (5.9)

Aula importante

Os conceitos de hoje (*referência*, *alias*, *cópia*) explicam muitos erros "misteriosos" em programas com listas.

Pertinência e remoção

```
primos = [2, 3, 5, 7, 11, 13, 17, 19]
for i in range(3, 20, 2):
    if i in primos:
        print(i, 'está em primos.')
    else:
        print(i, 'não está em primos.')

del primos[1:3]        # apaga os índices 1 e 2
print(primos)          # [2, 7, 11, 13, 17, 19]
del primos[1]
print(primos)          # [2, 11, 13, 17, 19]
```

Fonte: code/Cap5/prog.5.23_listas_pertence.py, 5.24_listas_del.py

Métodos de lista

```
lista = [123, 1, 0, 24, 1, 14]
lista.append(34)         # acrescenta 34 ao final
lista.extend([45, 12])   # acrescenta vários itens
lista.insert(1, 345)     # insere 345 na posição 1
lista.remove(24)         # remove a primeira ocorrência de 24
lista.pop()              # retira (e devolve) o último item
print(lista.count(1))    # quantos 1 existem: 2
lista.sort()             # ordena NA PRÓPRIA lista
lista.reverse()          # inverte NA PRÓPRIA lista
print(lista.index(14))   # posição do primeiro 14
```

Fonte: code/Cap5/prog.5.25_listas_comandos.py

Métodos que *alteram* *versus* funções que *devolvem*

```
lista = [3, 1, 2]
nova = sorted(lista)     # devolve outra lista ordenada
print(lista, nova)       # [3, 1, 2] [1, 2, 3]
lista.sort()             # altera a própria lista; devolve None!
print(lista)             # [1, 2, 3]
x = lista.sort()
print(x)                 # None  <- erro clássico
```

Erro comum

`lista = lista.sort()` apaga a lista (guarda `None`). Métodos que modificam no lugar normalmente devolvem `None`.

append *versus* extend (Exercícios 5.17 e 5.18)

```
a = [1, 2, 3]
a.append([4, 5])
print(a)                # [1, 2, 3, [4, 5]]   lista dentro de lista!

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

- append(x) adiciona *um* elemento (x, seja o que for)
- extend(seq) adiciona *cada elemento* de seq
- Também: a += [4, 5] equivale a extend

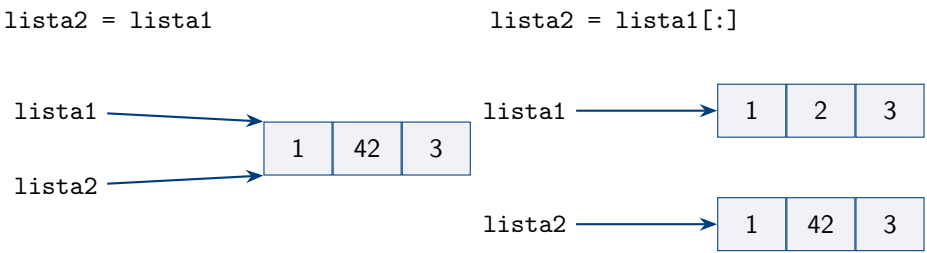
Duas variáveis, uma lista: *aliasing*

```
lista1 = [1, 2, 3]
lista2 = lista1          # NÃO copia: só dá outro nome
lista2[1] = 42
print(lista1)            # [1, 42, 3]
print(lista2)            # [1, 42, 3]
x = 30
y = x
y = 42
print('x =', x)          # 30
print('y =', y)          # 42
```

- `lista2 = lista1` faz as duas variáveis apontarem para o mesmo objeto
- Com números (imutáveis) isso não incomoda; com listas, sim!

Fonte: `code/Cap5/prog.5.28_listas_alias.py`

Variáveis são etiquetas



Esquerda: *alias*, um único objeto com duas etiquetas. Direita: *cópia*, dois objetos independentes.

Clonando listas

```
lista1 = [1, 2, 3]
lista2 = lista1[:]        # cópia por fatiamento
lista2[1] = 42
print(lista1)             # [1, 2, 3]
print(lista2)             # [1, 42, 3]
```

Outras formas de copiar:

```
lista3 = list(lista1)
lista4 = lista1.copy()
```

- Também é cópia: `lista2 = [1, 2, 3]` (criar uma lista nova com o mesmo conteúdo)

Fonte: code/Cap5/prog.5.29_listas_fatiada.py, 5.27_listas_sem_alias.py

Listas como parâmetros

```
def muda_valor(param):
    param += 'x'
    print('Dentro da função:', param)

lista = ['a', 'b', 'c']
palavra = 'abc'
muda_valor(lista)       # Dentro da função: ['a', 'b', 'c', 'x']
muda_valor(palavra)     # Dentro da função: abcx
print("lista =", lista)     # ['a', 'b', 'c', 'x']  <- mudou!
print("palavra =", palavra) # abc                   <- igual
```

- Objetos **mutáveis** (listas) podem ser alterados pela função; **imutáveis** (números, strings) não
- Resolve o mistério da Aula 11: Python passa a *referência* ao objeto

Fonte: `code/Cap6/prog.6.08_passagem_listas_strings.py`

Boa prática: não alterar o argumento sem avisar

```python
def dobra(lista):
    """Devolve uma NOVA lista com os valores dobrados."""
    return [2 * x for x in lista]

def dobra_no_lugar(lista):
    """Altera a lista recebida (retorna None)."""
    for i in range(len(lista)):
        lista[i] *= 2

a = [1, 2, 3]
b = dobra(a)            # a continua [1, 2, 3]
dobra_no_lugar(a)       # agora a = [2, 4, 6]
```

Documente na docstring se a função modifica o argumento. Na dúvida, devolva uma nova lista.

Atividade: preveja a saída

```
a = [1, 2, 3]
b = a
c = a[:]
b.append(4)
c.append(5)
print(a, b, c)
```

```
def f(l):
    l = l + [9]
def g(l):
    l.append(9)
x = [1]; f(x); print(x)
g(x); print(x)
```

Respostas: (esq.) `[1, 2, 3, 4]` `[1, 2, 3, 4]` `[1, 2, 3, 5]`; (dir.) `[1]` e depois `[1, 9]`. Em `f`, `l + [9]` cria uma *nova* lista e reatribui o parâmetro local.

Para praticar: exercícios do livro

Site do livro – Capítulo 5

- **5.17, 5.18** (`append` *vs.* `extend`), **5.19** (passo -1), **5.20** (inverter com compreensão)
- **5.21 a 5.24** (metades de uma lista), **5.25** (divisíveis por 7 e não por 3), **5.26** (média)

Enunciados e soluções comentadas em
`https://www.araujo.eng.uerj.br/livro/exercicios/cap5/`

Exercícios adicionais (sem solução): `https://www.araujo.eng.uerj.br/livro/extras/`

Resumo da aula

- Métodos: `append`, `extend`, `insert`, `remove`, `pop`, `sort`, `reverse`, `count`, `index`
- `b = a` não copia uma lista; use `a[:]`, `list(a)` ou `a.copy()`
- Funções recebem a referência: listas passadas podem ser alteradas
- Próxima aula: matrizes e tabelas de dados