Clonagem e Criação de Referência

Thiago Leucz Astrizi

Faculdade Municipal de Palhoça

17 de setembro de 2025

Fazendo uma Cópia de uma Lista

- Você pode fazer uma cópia de um objeto do tipo lista duplicando cada elemento em um novo objeto do tipo lista
- `copia_de_L = L[:]`
  - Equivalente a fazer um loop sobre a lista `L` e adicionar cada elemento a `copia_de_L`
  - Mas isso não faz uma cópia dos elementos de `L` que também forem listas (mas ainda veremos como fazer isso neste caso).

```
1 lista_original = [4, 5, 6]
2 lista_nova = lista_original[:]
```

Exercício de Sala

- Escreva uma função que atenda à seguinte especificação.
- Dica: Você pode fazer uma cópia para salvar os elementos. Usar `L.clear()` para esvaziar a lista e depois repovoá-la com aqueles que você salvou.

```
1  def remove_tudo(L, e):
2      """
3      ENTRADA: 'L' é uma lista.
4               'e' é um elemento que pode ou não estar na lista.
5      SAÍDA: Modifica a lista L para remover todos os seus
6             elementos que são iguais a 'e'.
7      """
8      pass
9
10 L = [1, 2, 2, 2]
11 remove_tudo(L, 2)
12 print(L) # Deve imprimir '[1]'
```

Operações em Listas: `remove`

- Para apagar elementos em um índice específico, use `del(L[indice])`
- Remova um elemento do fim da lista com `L.pop()`. Esta função retorna o elemento removido. (Ela funciona também passando um índice específico: `L.pop(3)`).
- Remova um elemento específico com `L.remove(elemento)`
  - Procura por um elemento e o remove (modificando a lista)
  - Se o elemento ocorre várias vezes, remove a primeira ocorrência
  - Se o elemento não existe, gera um erro

```
1 L = [2, 1, 3, 6, 3, 7, 0]
2 L.remove(2)   # A lista se torna: L = [1, 3, 6, 3, 7, 0]
3 L.remove(3)   # A lista se torna: L = [1, 6, 3, 7, 0]
4 del(L[1])     # A lista se torna: L = [1, 3, 7, 0]
5 a = L.pop()   # Retorna 0 e fica: L = [1, 3, 7]
```

Refazendo o Exercício com `remove`

```python
def remove_tudo(L, e):
    """
    ENTRADA: 'L' é uma lista.
             'e' é um elemento que pode ou não estar na lista.
    SAÍDA: Modifica a lista L para remover todos os seus
           elementos que são iguais a 'e'.
    """
    while e in L:
        L.remove(e)
```

Refazendo o Exercício com `remove`

Mas e se fizéssemos assim?

```
 1 def remove_tudo(L, e):
 2     """
 3     ENTRADA: 'L' é uma lista.
 4              'e' é um elemento que pode ou não estar na lista.
 5     SAÍDA: Modifica a lista L para remover todos os seus
 6            elementos que são iguais a 'e'.
 7     """
 8     for elem in L:
 9         if elem == e:
10             L.remove(e)
11
12 L = [1, 2, 2, 2]
13 remove_all(L, 2)
14 print(L) # Deve imprimir [1]
```

Refazendo o Exercício com `remove`

Mas e se fizéssemos assim?

```
 1 def remove_tudo(L, e):
 2     """
 3     ENTRADA: 'L' é uma lista.
 4              'e' é um elemento que pode ou não estar na lista.
 5     SAÍDA: Modifica a lista L para remover todos os seus
 6            elementos que são iguais a 'e'.
 7     """
 8     for elem in L:
 9         if elem == e:
10             L.remove(e)
11
12 L = [1, 2, 2, 2]
13 remove_all(L, 2)
14 print(L) # Deve imprimir [1]
```

ERRO! L imprime [1, 2]! Não é o que queríamos!

Alguns Exemplos Complicados

1. Estamos iterando sobre os índices de `L`, mas a cada iteração nós modificamos `L` e adicionamos novos elementos.
2. Estamos iterando sobre os elementos de `L`, mas a cada iteração nós modificamos `L` e adicionamos novos elementos.
3. Estamos iterando sobre os elementos de `L`, mas a cada iteração, atribuímos `L` a um novo objeto.
4. Estamos iterando sobre os elementos de `L`, mas a cada iteração nós modificamos `L` e removemos elementos.

Exemplo Complicado 4

Queremos modificar a lista L1 para remover qualquer elemento que também está em L2.

```
1 def remove_duplicatas(L1, L2):
2     for e in L1:
3         if e in L2:
4             L1.remove(e)
5
6 L1 = [10, 20, 30, 40]
7 L2 = [10, 20, 50, 60]
8 remove_duplicatas(L1, L2)
```

Exemplo Complicado 4

Queremos modificar a lista L1 para remover qualquer elemento que também está em L2.

```
1 def remove_duplicatas(L1, L2):
2     for e in L1:
3         if e in L2:
4             L1.remove(e)
5
6 L1 = [10, 20, 30, 40]
7 L2 = [10, 20, 50, 60]
8 remove_duplicatas(L1, L2)
```

- L1 termina como [20, 30, 40], e não [30, 40]. Por que?
  - Você está modificando a lista à medida que itera sobre ela.
  - Python usa um contador interno. Ele armazena o índice em que ele está na lista.
  - A lista é modificada. Mas o contador interno não é atualizado.
  - O loop nunca vê o elemento 20.

Exemplo Complicado 4

Queremos modificar a lista L1 para remover qualquer elemento que também está em L2.

```
1 def remove_duplicatas(L1, L2):
2     copia_de_L1 = L1[:]
3     for e in copia_de_L1:
4         if e in L2:
5             L1.remove(e)
6
7 L1 = [10, 20, 30, 40]
8 L2 = [10, 20, 50, 60]
9 remove_duplicatas(L1, L2)
```

Esta versão funciona! Itera sobre uma cópia. Modifica a lista original, não a cópia.

Criação de Referência

- Uma cidade pode ser conhecida por muitos nomes
- Atributos de uma cidade: litorânea, pequena, chuvosa
- Todos os apelidos apontam para a mesma cidade

Florianópolis	Litorânea	Chuvosa
Miembipe	Litorânea	Chuvosa
Desterro	Litorânea	Chuvosa

Criação de Referência

- Uma cidade pode ser conhecida por muitos nomes
- Atributos de uma cidade: litorânea, pequena, chuvosa
- Todos os apelidos apontam para a mesma cidade
  - Adicionando um atributo a um dos apelidos...

Florianópolis	Litorânea	Chuvosa	Turística
Miembipe	Litorânea	Chuvosa	
Desterro	Litorânea	Chuvosa	

Criação de Referência

- Uma cidade pode ser conhecida por muitos nomes
- Atributos de uma cidade: litorânea, pequena, chuvosa
- Todos os apelidos apontam para a mesma cidade
  - Adicionando um atributo a um dos apelidos...

Florianópolis	Litorânea	Chuvosa	Turística

Todos os nomes que se referem à mesma cidade também tem o atributo:

Miembipe	Litorânea	Chuvosa	Turística
Desterro	Litorânea	Chuvosa	Turística

Mutação e Iteração com Referências

```python
def remove_duplicatas(L1, L2):
    copia_de_L1 = L1[:]
    for e in copia_de_L1:
        if e in L2:
            L1.remove(e)

L1 = [10, 20, 30, 40]
L2 = [10, 20, 50, 60]
remove_duplicatas(L1, L2)
```

```python
def remove_duplicatas(L1, L2):
    copia_de_L1 = L1
    for e in copia_de_L1:
        if e in L2:
            L1.remove(e)

L1 = [10, 20, 30, 40]
L2 = [10, 20, 50, 60]
remove_duplicatas(L1, L2)
```

- Usar uma simples atribuição sem fazer uma cópia:
  - Cria um novo nome para a mesma lista
  - Iterar sobre o novo nome é a mesma coisa que iterar sobre a lista original. Então a versão à direita do código acima não funciona.

Quando você passa uma lista como parâmetro para função, você também está criando um novo nome para ela.

```
def remove_duplicatas(L1, L2):
    copia_de_L1 = L1
    for e in copia_de_L1:
        if e in L2:
            L1.remove(e)

La = [10, 20, 30, 40]
Lb = [10, 20, 50, 60]
remove_duplicatas(La, Lb)
```

Controlando o Processo de Cópia

- Atribuições apenas criam um novo ponteiro para o mesmo objeto

```
lista_antiga = [[1, 2], [3, 4], [5, 'fmp']]
lista_nova = lista_antiga

lista_nova[2][1] = 6
print("Nova lista:", lista_nova)
print("Lista antiga:", lista_antiga)
```

- Assim, se modificarmos um deles, modificamos o outro

Controlando o Processo de Cópia

- Suponha que queiramos copiar uma lista, não apenas criar um novo ponteiro compartilhado
- A **cópia superficial** faz isso na camada mais externa da lista
  - Equivalente a usar a sintaxe `[:]`
  - Quaisquer elementos mutáveis NÃO são copiados
- Use isso somente se sua lista contém apenas objetos imutáveis

```
import copy

lista_antiga = [[1, 2], [3, 4], [5, 6]]
lista_nova = copy.copy(lista_antiga)

lista_antiga.append([7, 8])
lista_antiga[1][1] = 9

print("Nova lista:", lista_nova)
print("Lista antiga:", lista_antiga)
```

Controlando o Processo de Cópia

- Se queremos que todas as estruturas internas sejam copiadas, precisamos de uma **cópia profunda**.
- Use uma cópia profunda quando sua lista pode conter elementos mutáveis para garantir que toda a estrutura interna seja copiada.

```
import copy

lista_antiga = [[1, 2], [3, 4], [5, 6]]
lista_nova = copy.deepcopy(lista_antiga)

lista_antiga.append([7, 8])
lista_antiga[1][1] = 9

print("Nova lista:", lista_nova)
print("Lista antiga:", lista_antiga)
```

Listas na Memória

- Separe o conceito do **objeto** e o **nome** que damos para ele.
  - Uma lista é um objeto na memória
  - Um nome de variável aponta para um objeto
- Listas são mutáveis e se comportam de maneira diferente que objetos imutáveis
- Usar o sinal de igual (=) com objetos mutáveis cria uma referência
  - Ambas as variáveis apontam para o mesmo lugar na memória
  - Qualquer variável apontando para o mesmo objeto é afetada por mudanças no objeto, mesmo que tenham sido feitas usando outro nome
- Se você quer uma cópia, precisa pedir explicitamente uma cópia para o Python
- É preciso ter em mente os **efeitos colaterais** das funções ao lidar com listas, especialmente quando você mantém mais de uma variável apontando para a mesma lista

Por que listas e tuplas?

- Se a mutação causa tantos problemas, por que precisamos de listas? Não podíamos ficar só com tuplas?
  - Eficiência—se lidamos com sequências grandes, não precisamos copiar ela toda vez que realizamos uma operação.
- Se listas fazem o mesmo que as tuplas, por que não ficamos só com as listas então?
  - Estruturas imutáveis vão ser úteis em breve para lidarmos com outros tipos de objetos.
  - Se você não quer que uma sequência seja modificada, as tuplas protegem contra isso
  - Elas podem ser mais rápidas se usadas no contexto certo.

Exemplos de Estruturas: Filas e Pilhas

- Úteis quando queremos armazenar objetos para tratá-los mais tarde.
- **Fila:** O primeiro objeto a entrar será o primeiro a sair.
- **Pilha:** O último objeto a entrar será o primeiro a sair.

```
def enfileira(L, e):
    L.append(e)

def desenfileira(L):
    return L.pop(0)

fila = []

enfileira(fila, 3)
enfileira(fila, 2)
enfileira(fila, 5)
print(desenfileira(fila))
```

```
def empilha(L, e):
    L.append(e)

def desempilha(L):
    return L.pop()

pilha = []

empilha(pilha, 3)
empilha(pilha, 2)
empilha(pilha, 5)
print(desempilha(pilha))
```