Programação Script

Listas (parte 2)

Aula 08

Prof. Felipe A. Louza

Roteiro

1. Pertinência em uma Lista
2. Inicialização de listas
3. Fatiamento (`slicing`) de listas
4. Objetos e referências
5. Clonagem de listas
6. Concatenação de listas
7. Listas e funções
8. Outras operações
9. Referências

Roteiro

Comando: `in`

Podemos testar a pertinência de um `item` em uma lista:

- Comando `in` retorna um valor booleano
- Exemplos:

```
1 frutas = ["maça", "laranja", "banana", "cereja"]
2 >>> "maça" in frutas
3 True
4 >>> "pera" not in frutas
5 True
```

Comando: `in`

Vimos o comando `in` combinado com o `range`:

```
1 for i in range(0, 10, 1):
2   print(i, end=" ")
3 print()
```

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

- Podemos testar a pertinência de um valor em um `range`:

```
1 r = range(0, 10, 2)
2 print(5 in r)
```

```
1 False
```

Comando: `in`

Vimos o comando `in` combinado com o `range`:

```
1 for i in range(0, 10, 1):
2   print(i, end=" ")
3 print()
```

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

- Podemos testar a pertinência de um valor em um `range`:

```
1 r = range(0, 10, 2)
2 print(5 in r)
```

```
1 False
```

Comando: `range`

Podemos também acessar valores em um `range`:

- Exemplo:

```
1 r = range(0, 20, 2)
2 print(r[5])
3 print(type(r[5]))
```

```
1 10
2 <class 'int'>
```

Roteiro

Inicialização de listas

Em algumas situações é preciso declarar e já atribuir valores para uma lista.

- O comando `list()` converte os valores de um `range` em uma lista:

```
1 lista = list(range(0, 100, 2))
2 print(lista)
```

```
1 [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, ..., 98]
```

Inicialização de listas

Outra opção é utilizar **compreensão de listas**.

- Dentro da lista incluímos uma construção com um laço que gerará valores iniciais para a lista.
- Exemplos:

```
>>> lista = [0 for i in range(10)]
>>> lista
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

```
>>> lista = [i for i in range(10)]
>>> lista
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Inicialização de listas

- Podemos adicionar uma condição para o `item` ser incluído na lista:

```
1 >>> lista = [i for i in range(10) if i % 2 == 0]
2 >>> lista
3 [0, 2, 4, 6, 8]
```

Inicialização de listas

- Podemos adicionar valores de uma outra lista:

```
1 frutas = ["apple", "banana", "cherry", "kiwi", "mango"]
2
3 lista = [x for x in frutas]
4 print(lista)
```

```
1 ["apple", "banana", "cherry", "kiwi", "mango"]
```

Inicialização de listas

- Podemos ter também um condição para adicionar um item:

```
1 frutas = ["apple", "banana", "cherry", "kiwi", "mango"]
2
3 #inicializa a lista apenas com palavras que contém a letra 'a'
4 lista = [x for x in frutas if "a" in x]
5 print(lista)
```

```
1 ['apple', 'banana', 'mango']
```

Inicialização de listas

- Podemos aplicar uma operação/função a um item antes que ele seja inserido na lista:

```
1 potencias = [2**i for i in range(10)]
2 print(potencias)
```

```
1 [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

- Código equivalente:

```
1 potencias = []
2 for i in range(10):
3   squares.append(2**i)
```

Inicialização de listas

- Podemos aplicar uma operação/função a um item antes que ele seja inserido na lista:

```
1 potencias = [2**i for i in range(10)]
2 print(potencias)
```

```
1 [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

- Código equivalente:

```
1 potencias = []
2 for i in range(10):
3   squares.append(2**i)
```

Inicialização de listas

- Exemplo com uma função:

```
1 import math
2
3 def area_circulo(raio):
4     return math.pi*(raio**2)
5
6 areas = [area_circulo(a) for a in range(3)]
7 print(areas)
```

```
1 [0.0, 3.141592653589793, 12.566370614359172]
```

Roteiro

Listas: `slicing`

Podemos obter uma sub-lista com a operação de fatiamento, ou `slicing`.

```
1 lista[a:b]
```

o resultado é uma outra lista com os elementos

`lista[a]`, `lista[a+1]`, `...` até `lista[b-1]`.

- Exemplo:

```
1 >>> lista1 = [10, 20, 30, 40, 50]
2
3 >>> lista2 = lista1[1:4]
4 >>> lista2
5 [20, 30, 40]
```

Listas: slicing

Podemos obter uma sub-lista com a operação de fatiamento, ou slicing.

```
1 lista[a:b]
```

o resultado é uma outra lista com os elementos

`lista[a]`, `lista[a+1]`, ... até `lista[b-1]`.

- Exemplo:

```
1 >>> lista1 = [10, 20, 30, 40, 50]
2
3 >>> lista2 = lista1[1:4]
4 >>> lista2
5 [20, 30, 40]
```

Listas: slicing

Podemos abreviar:

1. O primeiro parâmetro:

```
>>> lista1 = [10, 20, 30, 40, 50]

>>> lista2 = lista1[:3]
>>> lista2
[10, 20, 30]
```

2. O segundo parametro:

```
>>> lista1 = [10, 20, 30, 40, 50]

>>> lista2 = lista1[2:]
>>> lista2
[30, 40, 50]
```

Listas: slicing

Podemos abreviar:

1. O primeiro parâmetro:

```
>>> lista1 = [10, 20, 30, 40, 50]

>>> lista2 = lista1[:3]
>>> lista2
[10, 20, 30]
```

2. O segundo parametro:

```
>>> lista1 = [10, 20, 30, 40, 50]

>>> lista2 = lista1[2:]
>>> lista2
[30, 40, 50]
```

Listas: slicing

Podemos abreviar:

3. Os dois parâmetro:

```
1 >>> lista1 = [10, 20, 30, 40, 50]
2
3 >>> lista2 = lista1[:]
4 >>> lista2
5 [10, 20, 30, 40, 50]
```

Listas: slicing

- Qual o resultado da operação abaixo?

```
>>> lista1 = [10, 20, 30, 40, 50]

>>> lista2 = lista1[-3:-1]
>>> lista2
```

```
[30, 40]
```

Listas: slicing

- Qual o resultado da operação abaixo?

```
>>> lista1 = [10, 20, 30, 40, 50]

>>> lista2 = lista1[-3:-1]
>>> lista2
```

```
[30, 40]
```

Listas: slicing

Podemos combinar uma atribuição com o operador de fatiamento:

- Modificamos vários elementos de uma só vez.

```
1 >>> lista = ['a', 'b', 'c', 'd', 'e', 'f']
2
3 >>> lista[1:3] = ['x', 'y']
4 ['a', 'x', 'y', 'd', 'e', 'f']
```

Listas: slicing

Podemos inserir elementos em no meio de uma lista com o fatiamento:

- "Espremendo-os" em uma fatia vazia na posição desejada.

```
>>> lista =  ['a', 'd', 'f']

>>> lista[1:1] = ['b', 'c']
['a', 'b', 'c', 'd', 'f']
```

Listas: slicing

Podemos remover elementos de uma lista com o fatiamento:

- Atribuímos uma lista vazia a eles.

```
>>> lista = ['a', 'b', 'c', 'd', 'e', 'f']

>>> lista[1:3] = []
['a', 'd', 'e', 'f']
```

Roteiro

Objetos Mutáveis e Imutáveis

Em **Python** tudo é um objeto, e seu tipo especifica quais operações podem ser realizadas sobre o objeto.

- O tipo de um objeto indica se ele é **mutável** ou **imutável**.
  - Os tipos vistos anteriormente, `int, float, string, bool` são todos imutáveis.
  - Vamos ver que `listas` são **mutáveis**.

- No exemplo abaixo, a variável `a` faz referência ao objeto `int` `81`.

```
1 >>> a = 81
2 >>>
3 >>> print(a)
4 81
```

Objetos Mutáveis e Imutáveis

Em **Python** tudo é um objeto, e seu tipo especifica quais operações podem ser realizadas sobre o objeto.

- O tipo de um objeto indica se ele é **mutável** ou **imutável**.
  - Os tipos vistos anteriormente, `int`, `float`, `string`, `bool` são todos imutáveis.
  - Vamos ver que `listas` são **mutáveis**.

- No exemplo abaixo, a variável `a` faz referência ao objeto `int` `81`.

```
1 >>> a = 81
2 >>>
3 >>> print(a)
4 81
```

Objetos Mutáveis e Imutáveis

A função `id()` retorna um identificador do objeto referênciado por uma variável:

```
1 >>> a = 81
2 >>> id(a)
3 140177809498752
```

- Quando alteramos o vamos de a um novo objeto é criado.

```
1 >>> a = 82
2 >>> id(a)
3 140177809498784
```

  - Enquanto um objeto estiver sendo referênciado, ele continua na memória.

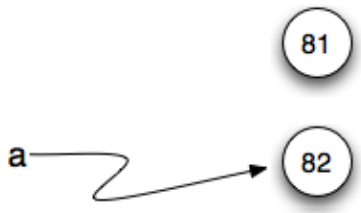
Objetos Mutáveis e Imutáveis

A função `id()` retorna um identificador do objeto referênciado por uma variável:

```
1 >>> a = 81
2 >>> id(a)
3 140177809498752
```

- Quando alteramos o vamos de `a` um novo objeto é criado.

```
1 >>> a = 82
2 >>> id(a)
3 140177809498784
```



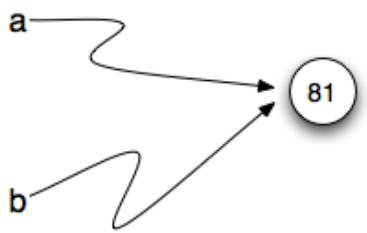
  - Enquanto um objeto estiver sendo referênciado, ele continua na memória.

Objetos Mutáveis e Imutáveis

Mais de uma variável podem referênciar o mesmo objeto:

- O operador `is` retorna `True` se as duas referências são ao mesmo objeto:

```
1 >>> a = 81
2 >>> b = 81
3 >>> id(a)
4 140177809498752
5 >>> id(b)
6 140177809498752
7
8 >>> a is b
9 True
```



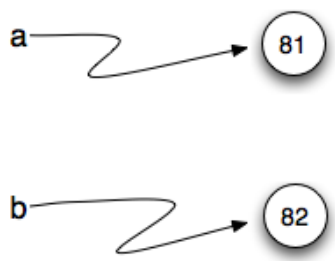
- **Python** optimiza recursos fazendo dois que se referem ao mesmo `int` se referiram ao mesmo objeto.

Objetos Mutáveis e Imutáveis

Um objeto imutável não é alterado:

- O que ocorre em `b = b+1` é a criação de um novo objeto:

```
1  >>> a = 81
2  >>> b = a
3  >>> b = b+1
4  >>> id(a)
5  140177809498752
6  >>> id(b)
7  140177809498784
8
9  >>> a is b
10 False
```



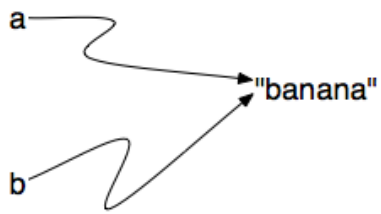
  - Enquanto um objeto estiver sendo referênciado, ele continua na memória.

Objetos Mutáveis e Imutáveis

O mesmo acontece com `strings` que são objetos imutáveis:

- No exemplo abaixo `a` e `b` referenciam o mesmo objeto.

```
1  >>> a = "banana"
2  >>> b = "banana"
3
4  >>> id(a)
5  140177546130480
6  >>> id(b)
7  140177546130480
8
9  >>> a is b
10 True
```



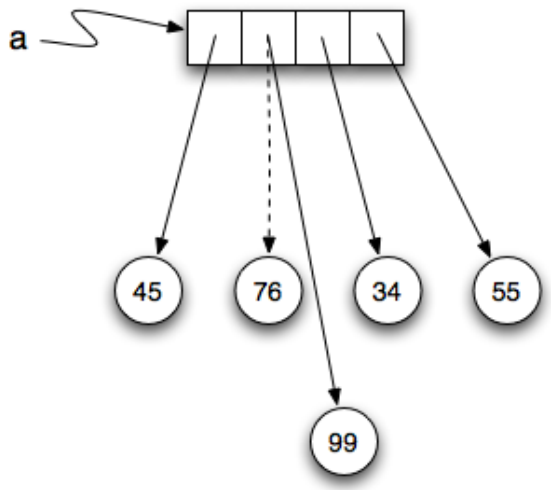
  - Não podemos alterar uma `string`, podemos referenciar um novo objeto.

Objetos Mutáveis e Imutáveis

Já `listas` são objetos **mutáveis**:

- Operações podem ser realizadas sobre uma objeto lista alterando-o (não é criado um novo objeto após a alteração).

```
1 >>> a = [45, 76, 34, 55]
2 >>> id(a)
3 140177546196864
4
5 >>> a[1] = 99
6 >>> id(a)
7 140177546196864
```



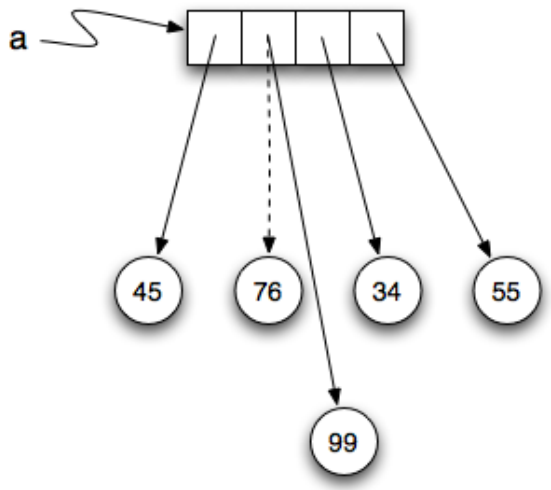
  - Seria computacionalmente caro criar um novo objeto lista para cada alteração.

Objetos Mutáveis e Imutáveis

Já `listas` são objetos **mutáveis**:

- Operações podem ser realizadas sobre uma objeto lista alterando-o (não é criado um novo objeto após a alteração).

```
1 >>> a = [45, 76, 34, 55]
2 >>> id(a)
3 140177546196864
4
5 >>> a[1] = 99
6 >>> id(a)
7 140177546196864
```



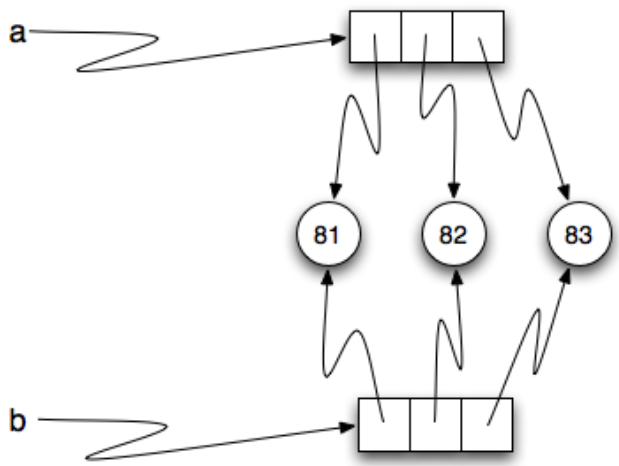
  - Seria computacionalmente caro criar um novo objeto lista para cada alteração.

Objetos Mutáveis e Imutáveis

Duas `listas` diferentes podem, por acaso, ter o mesmo conteúdo, ainda assim são objetos **diferentes**:

- O operador == compara o conteúdo das listas:

```
1 >>> a = [81, 82, 83]
2 >>> b = [81, 82, 83]
3 >>> a is b
4 False
5 >>> a == b
6 True
```




```
1 >>> id(a[0])
2 140177809497632
3 >>> id(b[0])
4 140177809497632
```

  - Podemos entender uma `lista` como uma coleção de referências para objetos.

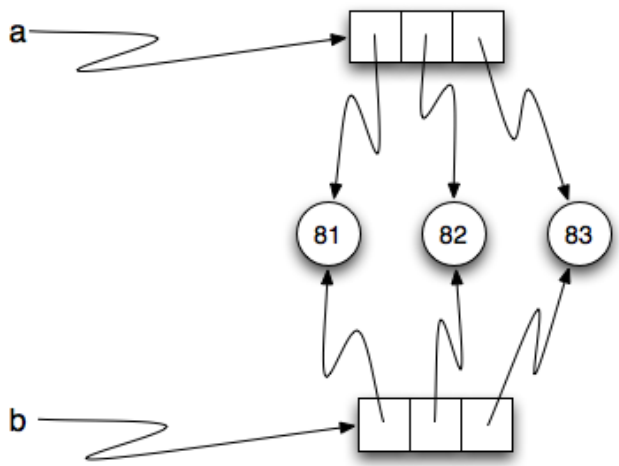
Objetos Mutáveis e Imutáveis

Duas `listas` diferentes podem, por acaso, ter o mesmo conteúdo, ainda assim são objetos **diferentes**:

- O operador == compara o conteúdo das listas:

```
>>> a = [81, 82, 83]
>>> b = [81, 82, 83]
>>> a is b
False
>>> a == b
True
```

```
>>> id(a[0])
140177809497632
>>> id(b[0])
140177809497632
```



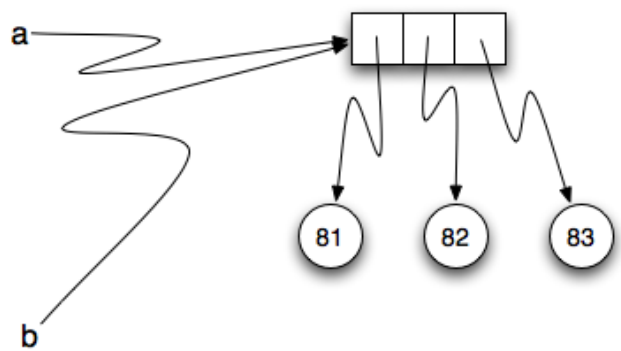
  - Podemos entender uma `lista` como uma coleção de referências para objetos.

Objetos Mutáveis e Imutáveis

Como variáveis, se atribuirmos uma lista a outra, ambas passam a fazer referência ao mesmo objeto.

- Dizemos que a lista tem apelidos (*"alias"*).

```
>>> a = [81, 82, 83]
>>> b = a
>>> a == b
True
>>> a is b
True
```




```
>>> b[0] = 5
>>> print(a)
[5, 82, 83]
>>> print(b)
[5, 82, 83]
```

- Mudanças feitas com um apelido afeta o outro.

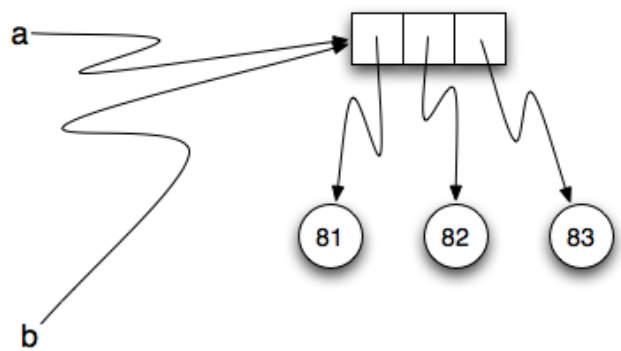
Objetos Mutáveis e Imutáveis

Como variáveis, se atribuirmos uma lista a outra, ambas passam a fazer referência ao mesmo objeto.

- Dizemos que a lista tem apelidos (*"alias"*).

```
>>> a = [81, 82, 83]
>>> b = a
>>> a == b
True
>>> a is b
True
```

```
>>> b[0] = 5
>>> print(a)
[5, 82, 83]
>>> print(b)
[5, 82, 83]
```



  - Mudanças feitas com um apelido afeta o outro.

Roteiro

Clonagem de listas

Como criar uma **cópia** em nova lista de uma lista já existente?

- Vimos que `b = a` apenas cria um *alias* da lista `a`.
- Uma possibilidade é criar uma lista vazia e adicionar (com o `append()`) os elementos de `a` em `b`

```
>>> a = [81, 82, 83]
>>> b = []
>>> for item in a:
...     b.append(item)
>>> b
[81, 82, 83]
```

```
>>> a is b
False
>>> a == b
True
```

  - Podemos alterar `b` sem modificar os valores de `a`.

Clonagem de listas

Como criar uma **cópia** em nova lista de uma lista já existente?

- Vimos que `b = a` apenas cria um *alias* da lista `a`.
- Uma possibilidade é criar uma lista vazia e adicionar (com o `append()`) os elementos de `a` em `b`

```
>>> a = [81, 82, 83]
>>> b = []
>>> for item in a:
>>>     b.append(item)
>>> b
[81, 82, 83]
```

```
>>> a is b
False
>>> a == b
True
```

  - Podemos alterar `b` sem modificar os valores de `a`.

Clonagem de listas

Como criar uma **cópia** em nova lista de uma lista já existente?

- Vimos que `b = a` apenas cria um *alias* da lista `a`.
- Uma possibilidade é criar uma lista vazia e adicionar (com o `append()`) os elementos de `a` em `b`

```
>>> a = [81, 82, 83]
>>> b = []
>>> for item in a:
>>>     b.append(item)
>>> b
[81, 82, 83]
```

```
>>> a is b
False
>>> a == b
True
```

- Podemos alterar `b` sem modificar os valores de `a`.

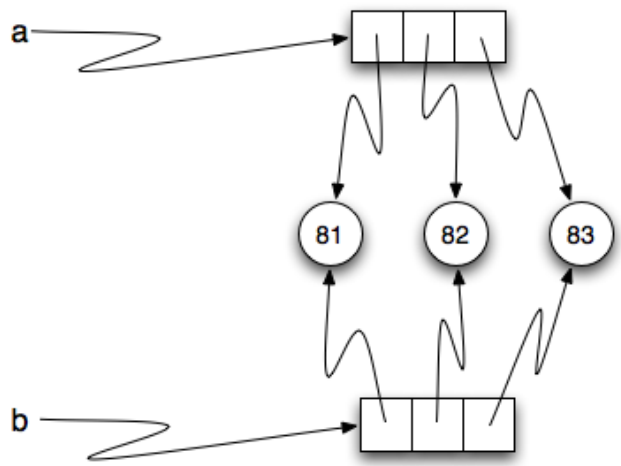
Clonagem de listas

Como criar uma **cópia** em nova lista de uma lista já existente?

- Vimos que `b = a` apenas cria um *alias* da lista `a`.
- Uma possibilidade é criar uma lista vazia e adicionar (com o `append()`) os elementos de `a` em `b`

```
>>> a = [81, 82, 83]
>>> b = []
>>> for item in a:
>>>     b.append(item)
>>> b
[81, 82, 83]
```

```
>>> a is b
False
>>> a == b
True
```



- Podemos alterar `b` sem modificar os valores de `a`.

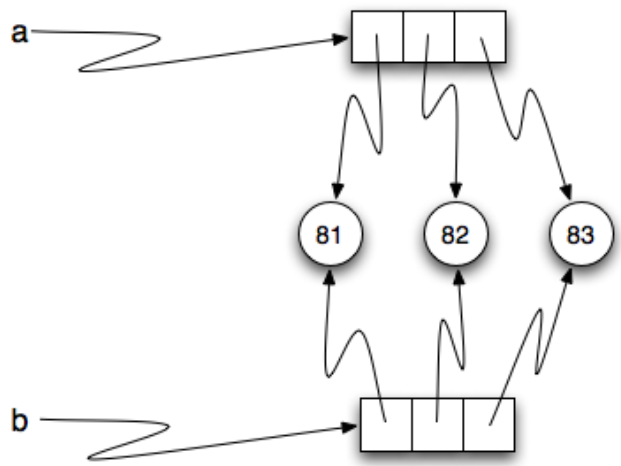
Clonagem de listas

Como criar uma **cópia** em nova lista de uma lista já existente?

- Vimos que `b = a` apenas cria um *alias* da lista `a`.
- Uma possibilidade é criar uma lista vazia e adicionar (com o `append()`) os elementos de `a` em `b`

```
>>> a = [81, 82, 83]
>>> b = []
>>> for item in a:
>>>     b.append(item)
>>> b
[81, 82, 83]
```

```
>>> a is b
False
>>> a == b
True
```



  - Podemos alterar `b` sem modificar os valores de `a`.

Clonagem de listas

Como criar uma **cópia** em nova lista de uma lista já existente?

- Outra alternativa é utilizar `fatiamento`:

```
>>> a = [81, 82, 83]
>>> b = a[:]
>>> b
[81, 82, 83]
```

```
>>> a is b
False
>>> a == b
True
```

  - Um novo objeto lista é criado (com as mesmas referências).
  - Em geral, utilizamos essa alternativa.

O método `lista.copy()` também retorna uma nova lista com cópias da lista.

Clonagem de listas

Como criar uma **cópia** em nova lista de uma lista já existente?

- Outra alternativa é utilizar `fatiamento`:

```
1 >>> a = [81, 82, 83]
2 >>> b = a[:]
3 >>> b
4 [81, 82, 83]
```

```
1 >>> a is b
2 False
3 >>> a == b
4 True
```

  - Um novo objeto lista é criado (com as mesmas referências).
  - Em geral, utilizamos essa alternativa.

O método `lista.copy()` também retorna uma nova lista com cópias da lista.

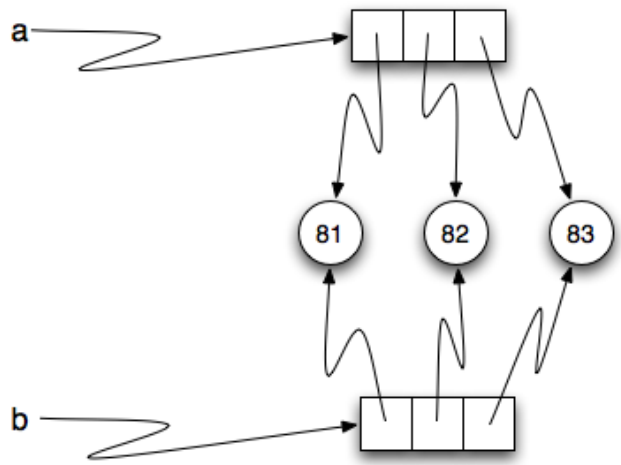
Clonagem de listas

Como criar uma **cópia** em nova lista de uma lista já existente?

- Outra alternativa é utilizar `fatiamento`:

```
1 >>> a = [81, 82, 83]
2 >>> b = a[:]
3 >>> b
4 [81, 82, 83]
```

```
1 >>> a is b
2 False
3 >>> a == b
4 True
```



  - Um novo objeto lista é criado (com as mesmas referências).
  - Em geral, utilizamos essa alternativa.

O método `lista.copy()` também retorna uma nova lista com cópias da lista.

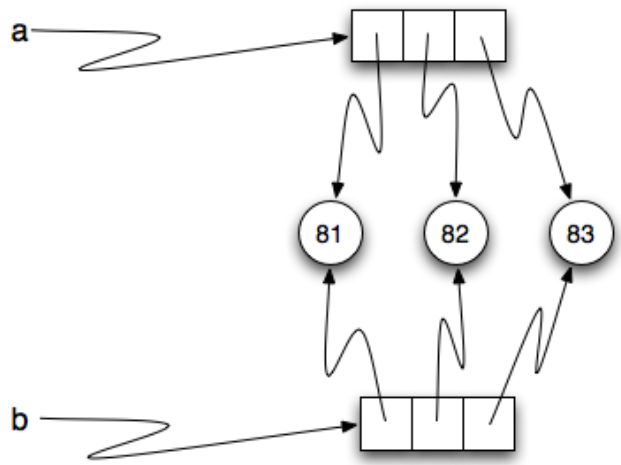
Clonagem de listas

Como criar uma **cópia** em nova lista de uma lista já existente?

- Outra alternativa é utilizar `fatiamento`:

```
>>> a = [81, 82, 83]
>>> b = a[:]
>>> b
[81, 82, 83]
```

```
>>> a is b
False
>>> a == b
True
```



  - Um novo objeto lista é criado (com as mesmas referências).
  - Em geral, utilizamos essa alternativa.

O método `lista.copy()` também retorna uma nova lista com cópias da lista.

Roteiro

Listas: concatenação

Podemos concatenar listas com o operador de soma +.

```
1 lista1 + lista2
```

- Exemplo:

```
1 >>> lista1 = [1, 2, 3]
2 >>> lista2 = ['a', 'b', 'c']
3 >>>
4 >>> x = lista1 + lista2
5 >>> x
6 [1, 2, 3, 'a', 'b', 'c']
```

```
1 >>> id(lista1)
2 4302774664
3 >>> id(lista2)
4 4302778312
5 >>> id(x)
6 4302774408
```

  - Uma nova lista é criada e referenciada por x.
  - Pode ser caro criar uma nova lista:

Listas: concatenação

Podemos concatenar listas com o operador de soma +.

```
1 lista1 + lista2
```

- Exemplo:

```
1 >>> lista1 = [1, 2, 3]
2 >>> lista2 = ['a', 'b', 'c']
3 >>>
4 >>> x = lista1 + lista2
5 >>> x
6 [1, 2, 3, 'a', 'b', 'c']
```

```
1 >>> id(lista1)
2 4302774664
3 >>> id(lista2)
4 4302778312
5 >>> id(x)
6 4302774408
```

  - Uma nova lista é criada e referenciada por x.
  - Pode ser caro criar uma nova lista:

Listas: concatenação

Podemos concatenar listas com o operador de soma +.

```
1 lista1 + lista2
```

- Exemplo:

```
1 >>> lista1 = [1, 2, 3]
2 >>> lista2 = ['a', 'b', 'c']
3 >>>
4 >>> x = lista1 + lista2
5 >>> x
6 [1, 2, 3, 'a', 'b', 'c']
```

```
1 >>> id(lista1)
2 4302774664
3 >>> id(lista2)
4 4302778312
5 >>> id(x)
6 4302774408
```

  - Uma nova lista é criada e referenciada por x.
  - Pode ser caro criar uma nova lista:

Listas: concatenação

Podemos concatenar listas com o operador de soma +.

```
1 lista1 + lista2
```

- Exemplo:

```
1 >>> lista1 = [1, 2, 3]
2 >>> lista2 = ['a', 'b', 'c']
3 >>>
4 >>> x = lista1 + lista2
5 >>> x
6 [1, 2, 3, 'a', 'b', 'c']
```

```
1 >>> id(lista1)
2 4302774664
3 >>> id(lista2)
4 4302778312
5 >>> id(x)
6 4302774408
```

  - Uma nova lista é criada e referenciada por x.
  - Pode ser caro criar uma nova lista:

Listas: concatenação

Se não for problema alterar a `lista1`, o método `extend()` é preferível à concatenação:

```
1 lista.extend(lista2)
```

- Exemplo:

```
1 >>> lista1 = [1, 2, 3]
2 >>> lista2 = ['a', 'b', 'c']
3 >>> id(lista1)
4 4302774664
5
6 >>> lista1.extend(lista2)
```

```
1 >>> id(lista1)
2 4302774664
3
4 >>> lista1
5 [1, 2, 3, 'a', 'b', 'c']
```

  - `lista1.extend(lista2)` insere os elementos de `lista2` no final da `lista1`.

Listas: concatenação

Se não for problema alterar a `lista1`, o método `extend()` é preferível à concatenação:

```
1 lista.extend(lista2)
```

- Exemplo:

```
1 >>> lista1 = [1, 2, 3]
2 >>> lista2 = ['a', 'b', 'c']
3 >>> id(lista1)
4 4302774664
5
6 >>> lista1.extend(lista2)
```

```
1 >>> id(lista1)
2 4302774664
3
4 >>> lista1
5 [1, 2, 3, 'a', 'b', 'c']
```

  - `lista1.extend(lista2)` insere os elementos de `lista2` no final da `lista1`.

Listas: concatenação

Se não for problema alterar a `lista1`, o método `extend()` é preferível à concatenação:

```
1 lista.extend(lista2)
```

- Exemplo:

```
1 >>> lista1 = [1, 2, 3]
2 >>> lista2 = ['a', 'b', 'c']
3 >>> id(lista1)
4 4302774664
5
6 >>> lista1.extend(lista2)
```

```
1 >>> id(lista1)
2 4302774664
3
4 >>> lista1
5 [1, 2, 3, 'a', 'b', 'c']
```

  - `lista1.extend(lista2)` insere os elementos de `lista2` no final da `lista1`.

Listas: concatenação

- O operador "*" faz repetições da concatenação:

```
1 lista * n
```

- Exemplo:

```
1 >>> origlist = [1, 2]
2 >>> newlist = origlist * 4
3 >>> newlist
4 [1, 2, 1, 2, 1, 2, 1, 2]
```

```
1 >>> id(origlist)
2 4302774664
3 >>> id(newlist)
4 4302774427
```

  - O resultado da operação do exemplo é o mesmo que somar (concatenar) 4 vezes a lista `lista1`.
  - Uma nova lista é criada e referenciada por `x`.

Listas: concatenação

- O operador "*" faz repetições da concatenação:

```
lista * n
```

- Exemplo:

```
>>> origlist = [1, 2]
>>> newlist = origlist * 4
>>> newlist
[1, 2, 1, 2, 1, 2, 1, 2]
```

```
>>> id(origlist)
4302774664
>>> id(newlist)
4302774427
```

  - O resultado da operação do exemplo é o mesmo que somar (concatenar) 4 vezes a lista `lista1`.
  - Uma nova lista é criada e referenciada por `x`.

Listas: concatenação

- Cuidado ao usar o operador "*" com listas e colchetes:

```
1 [lista] * n
```

- O resultado dessa operação é uma listas de listas.

```
1 >>> origlist = [45, 76, 34, 55]
2 >>>
3 >>> newlist = [origlist] * 3
```

```
1 >>> id(origlist)
2 4302774664
3 >>> id(newlist)
4 4302774427
```

```
1 >>> newlist
2 [[45, 76, 34, 55], [45, 76, 34, 55], [45, 76, 34, 55]]
```

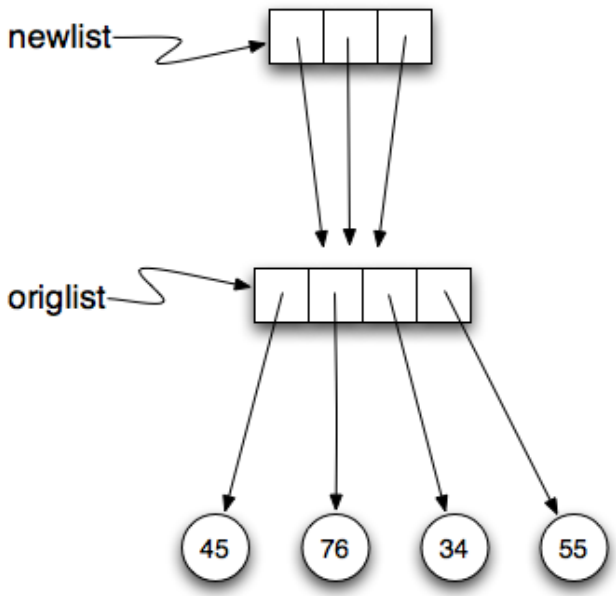
Listas: concatenação

- Cuidado ao usar o operador "*" com listas e colchetes:

```
1 [lista] * n
```

- O resultado dessa operação é uma listas de listas.

```
1 >>> origlist = [45, 76, 34, 55]
2 >>>
3 >>> newlist = [origlist] * 3
```




```
1 >>> id(origlist)
2 4302774664
3 >>> id(newlist)
4 4302774427
```

```
1 >>> newlist
2 [[45, 76, 34, 55], [45, 76, 34, 55], [45, 76, 34, 55]]
```

Listas: concatenação

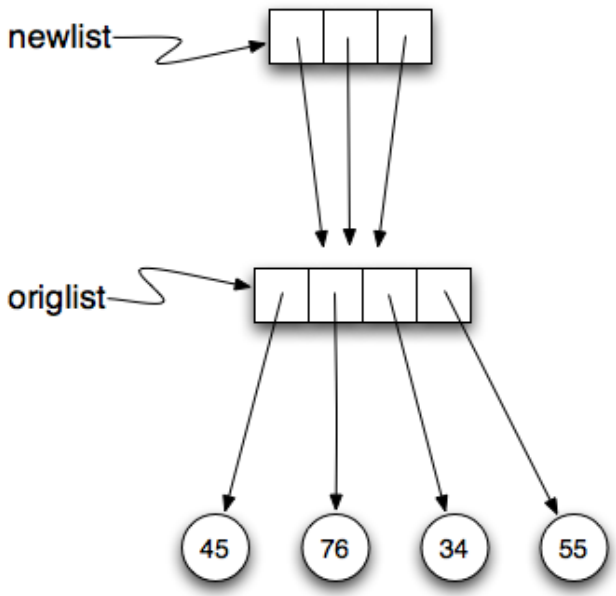
- Cuidado ao usar o operador "*" com listas e colchetes:

```
1 [lista] * n
```

- O resultado dessa operação é uma listas de listas.

```
1 >>> origlist = [45, 76, 34, 55]
2 >>>
3 >>> newlist = [origlist] * 3
```

```
1 >>> id(origlist)
2 4302774664
3 >>> id(newlist)
4 4302774427
```




```
1 >>> newlist
2 [[45, 76, 34, 55], [45, 76, 34, 55], [45, 76, 34, 55]]
```

Listas: concatenação

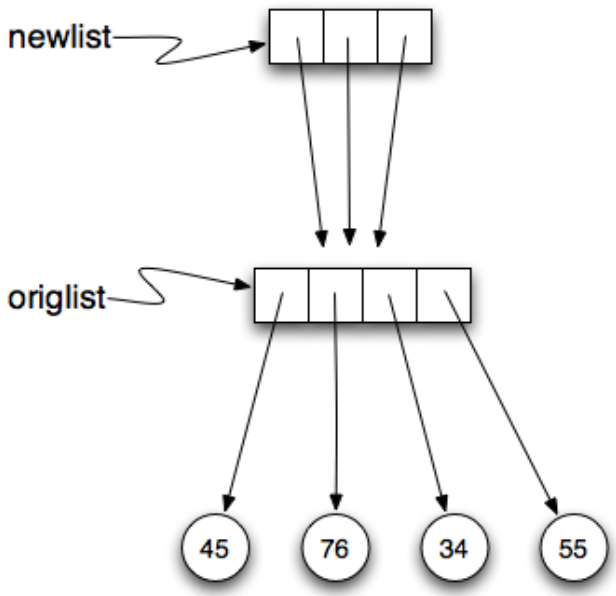
- Cuidado ao usar o operador "*" com listas e colchetes:

```
1 [lista] * n
```

- O resultado dessa operação é uma listas de listas.

```
1 >>> origlist = [45, 76, 34, 55]
2 >>>
3 >>> newlist = [origlist] * 3
```

```
1 >>> id(origlist)
2 4302774664
3 >>> id(newlist)
4 4302774427
```

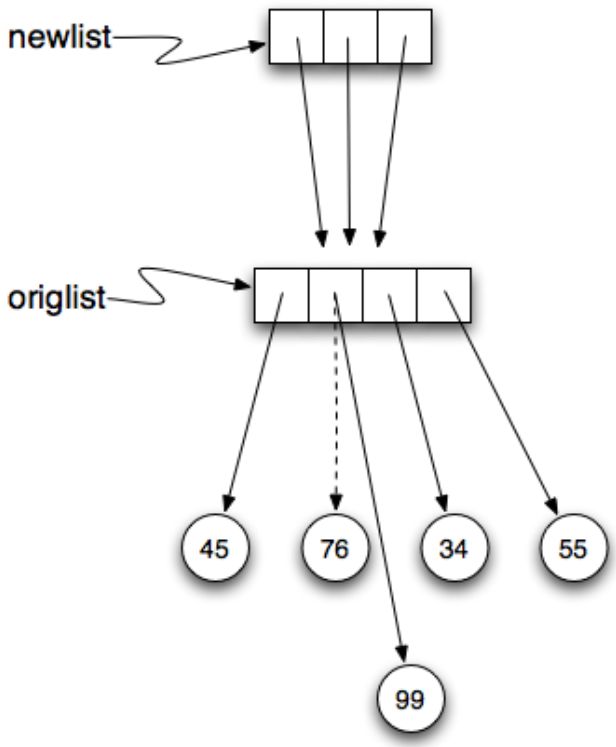



```
1 >>> newlist
2 [[45, 76, 34, 55], [45, 76, 34, 55], [45, 76, 34, 55]]
```

Listas: concatenação

- Nesse caso, alterar um elemento de `origlist` modifica *n* elementos de `newlist`:

```
1 >>> origlist = [45, 76, 34, 55]
2 >>>
3 >>> newlist = [origlist] * 3
4 >>>
5 >>> origlist[1] = 99
```

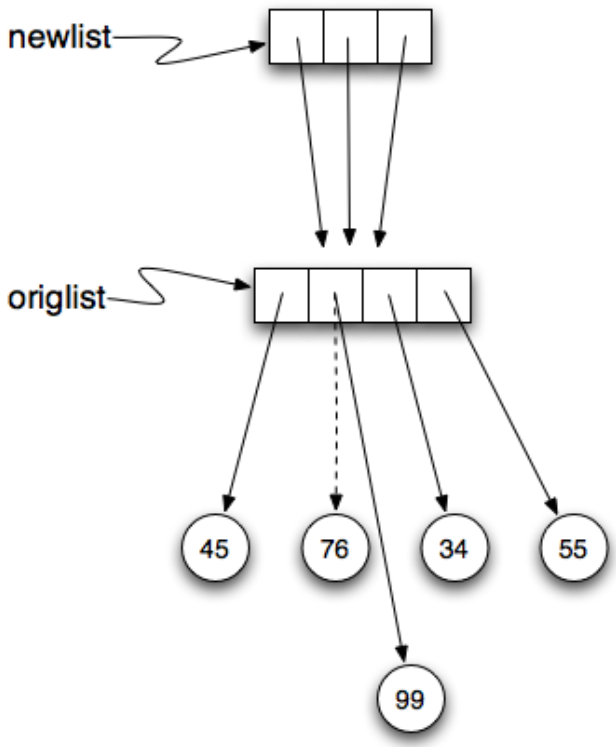



```
1 >>> newlist
2 [[45, 99, 34, 55], [45, 99, 34, 55], [45, 99, 34, 55]]
```

Listas: concatenação

- Nesse caso, alterar um elemento de `origlist` modifica *n* elementos de `newlist`:

```
1 >>> origlist = [45, 76, 34, 55]
2 >>>
3 >>> newlist = [origlist] * 3
4 >>>
5 >>> origlist[1] = 99
```




```
1 >>> newlist
2 [[45, 99, 34, 55], [45, 99, 34, 55], [45, 99, 34, 55]]
```

Roteiro

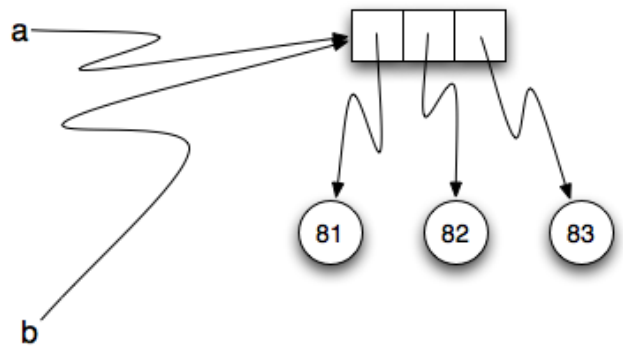
Listas em funções

Ao passar uma lista como argumento para a função estamos passando uma referência para a lista e não um cópia.

- Podemos ver o nome do parâmetro como um *"alias"* da `lista`.

```
1 def funcao(b):
2   print(b)
3   b[1] = 99
4
5 a = [81, 82, 83]
6 funcao(a)
7
8 print(a)
```

```
1 [81, 82, 83]
2 [81, 99, 83]
```



  - Mudanças feitas dentro da função alteram a lista passada como argumento.

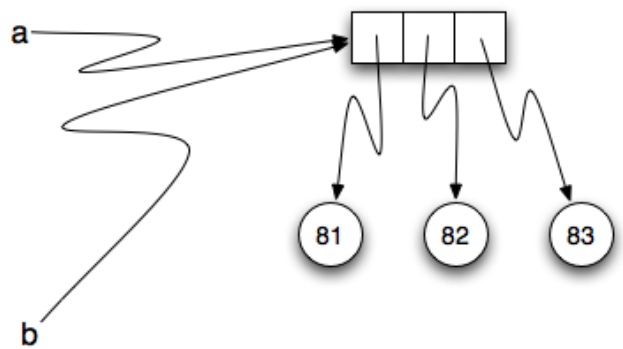
Listas em funções

Ao passar uma lista como argumento para a função estamos passando uma referência para a lista e não um cópia.

- Podemos ver o nome do parâmetro como um *"alias"* da `lista`.

```
1 def funcao(b):
2   print(b)
3   b[1] = 99
4
5 a = [81, 82, 83]
6 funcao(a)
7
8 print(a)
```

```
1 [81, 82, 83]
2 [81, 99, 83]
```



  - Mudanças feitas dentro da função alteram a lista passada como argumento.

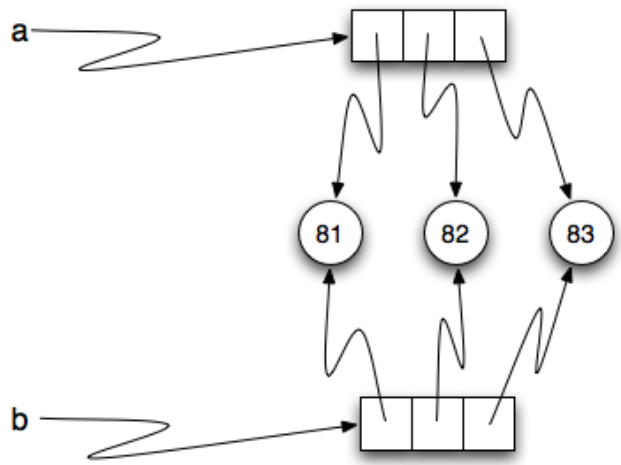
Listas em funções

Atenção:

- Se mudarmos o valor de b, estamos criando outra lista:

```
1 def funcao(b):
2   b = [81, 82, 83]
3   b[1] = 99
4   print(b)
5
6 a = [81, 82, 83]
7 funcao(a)
8
9 print(a)
```

```
1 [81, 99, 83]
2 [81, 82, 83]
```



  - Neste caso a variável b local da função é associada com uma nova lista.

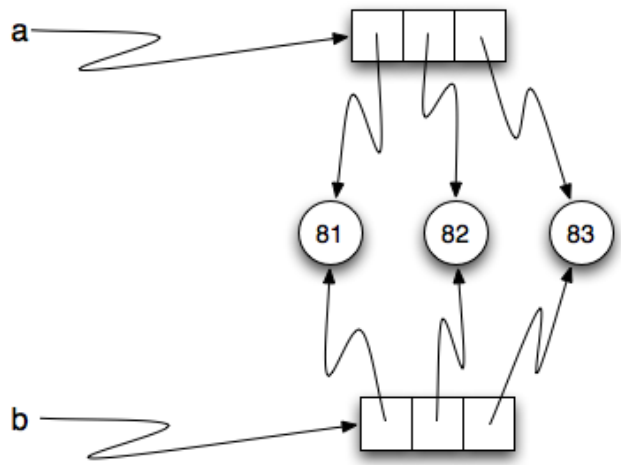
Listas em funções

Atenção:

- Se mudarmos o valor de b, estamos criando outra lista:

```
1 def funcao(b):
2   b = [81, 82, 83]
3   b[1] = 99
4   print(b)
5
6 a = [81, 82, 83]
7 funcao(a)
8
9 print(a)
```

```
1 [81, 99, 83]
2 [81, 82, 83]
```



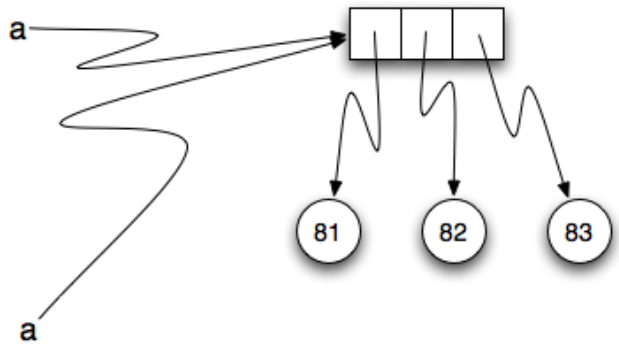
  - Neste caso a variável b local da função é associada com uma nova lista.

Listas em funções

- Podemos usar o modificador de escopo `global` para fazer referência a lista `a` dentro da função:

```
1 def funcao():
2   global a
3   print(a)
4   a[1] = 99
5
6 a = [81, 82, 83]
7 funcao()
8
9 print(a)
```

```
1 [81, 82, 83]
2 [81, 99, 83]
```



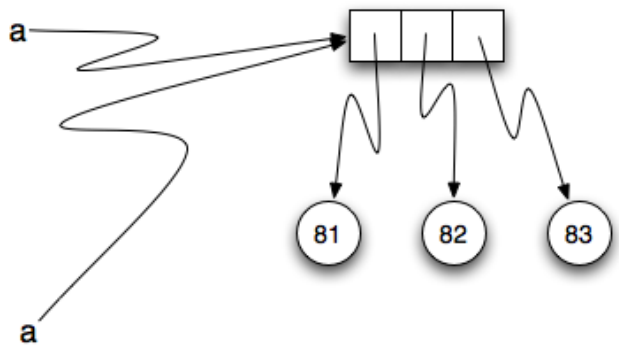
– Mudanças feitas dentro da função alteram a lista global.

Listas em funções

- Podemos usar o modificador de escopo `global` para fazer referência a lista `a` dentro da função:

```
1 def funcao():
2   global a
3   print(a)
4   a[1] = 99
5
6 a = [81, 82, 83]
7 funcao()
8
9 print(a)
```

```
1 [81, 82, 83]
2 [81, 99, 83]
```



  - Mudanças feitas dentro da função alteram a lista global.

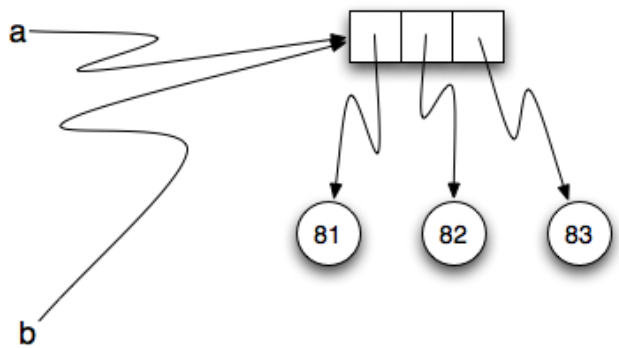
Listas em funções

Funções podem retornar uma listas:

- O valor retornado é uma referência da lista criada:

```
1  def funcao():
2    b = [81, 82, 83]
3    print(b)
4    print(id(b))
5    b[1] = 99
6    return b
7
8  a = funcao()
9
10 print(a)
11 print(id(a))
```

```
1 [81, 82, 83]
2 4302774664
3 [81, 99, 83]
4 4302774664
```



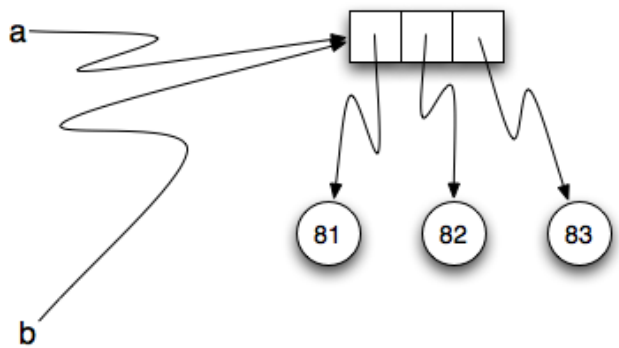
Listas em funções

Funções podem retornar uma listas:

- O valor retornado é uma referência da lista criada:

```
1  def funcao():
2    b = [81, 82, 83]
3    print(b)
4    print(id(b))
5    b[1] = 99
6    return b
7
8  a = funcao()
9
10 print(a)
11 print(id(a))
```

```
1 [81, 82, 83]
2 4302774664
3 [81, 99, 83]
4 4302774664
```



Roteiro

Resumo de operações

`lista[pos] = item`	modifica o valor de `lista[pos]` para time.
`lista.append(item)`	acrescenta item no final de uma `lista`.
`lista.insert(pos, item)`	insere item `lista[pos]`.
`del(lista[pos])`:	remove o item da `lista[pos]`.
`lista.remove(item)`:	remove a primeira ocorrência de item na `lista`.
`item = lista.pop()`:	remove o último item da `lista`.
`lista.clear()`:	remove todos os itens da `lista`.
`len(lista)`:	retorna o número de itens na `lista`.
`lista.count(item)`:	devolve o número de ocorrências de item.
`lista.index(item)`:	devolve a posição da primeira ocorrência de item.
`lista = l1+l2`:	devolve uma lista com l1 e l2 concatenadas.
`lista.extend(l2)`:	insere os elementos de l2 no final da `lista`.
`lista = l1[a:b]`:	devolve uma sub-lista com os elementos de `l1[a,b]`.

Fim

Dúvidas?

Leitura complementar

Leitura complementar:

1. https://panda.ime.usp.br/cc110/static/cc110/09-listas.html
2. https://panda.ime.usp.br/pensepy/static/pensepy/09-Listas/listas.html

Roteiro

Referências

1. Materiais adaptados dos slides do Prof. Eduardo C. Xavier, da Universidade Estadual de Campinas.