

Listas em Python

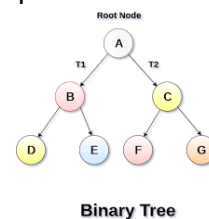
Afonso Paiva
apneto@icmc.usp.br

Fundamentos da Programação de Computadores – SME0332

1

Estrutura de Dados (E.D.)

- Forma de organizar e armazenar dados, facilitando o seu acesso e modificação
- Em Python, existem 3 tipos principais de E.D.:
 - Listas
 - Tuplas (registro)
 - Dicionários
- Outras E.D. mais complexas (exemplo, **árvores**) podem ser construídas combinando as E.D. clássicas



2

Motivação

- **Exemplo:** cálculo do IMC entre 2 pessoas.

```
nome1 = input("Entre com o nome da pessoa1: ")
nome2 = input("Entre com o nome da pessoa2: ")
peso1 = float(input("Informe o peso da pessoa1: "))
h1 = float(input("Informe a altura da pessoa1: "))
IMC1 = peso1/h1**2
peso2 = float(input("Informe o peso da pessoa2: "))
h2 = float(input("Informe a altura da pessoa2: "))
IMC2 = peso2/h2**2
```

```
print("%s seu IMC é %.2f" % (nome1, IMC1))
print("%s seu IMC é %.2f" % (nome2, IMC2))
```

O que vimos até agora são variáveis com somente um valor e se fossem **30 pessoas?**

3

Motivação

- Existe uma variável capaz de armazenar mais de um valor de um mesmo tipo?
 - Sim, essas variáveis são conhecidas como variáveis compostas, variáveis subscritas, variáveis indexáveis, arranjos ou **vetores** (*array*)
- **Lista:**
 - E.D. linear, isto é, um arranjo sequencial de informações
 - Permite o acesso eficiente aos seus elementos em ordem sequencial



4

Vetores

- E.D. lineares e estáticas, isto é, são compostas por um número **finito** de elementos de um determinado tipo
- Os valores são acessados através de um **índice**
- Semelhante aos vetores em Geometria Analítica:

$$x = (x_1, x_2, x_3, x_4, x_5, \dots)$$

- Uma mesma variável é capaz de armazenar vários valores simultaneamente:

$x_1 = 12$	$x_2 = 34$	$x_3 = 56$	...
$x = [12, 34, 56, \dots]$			

5

Listas em Python

- Em Python, a estrutura conhecida como **lista** é mais geral do que a lista clássica em E.D.
- Na verdade, pode ser vista como uma implementação tanto de **listas** como de **vetores**
- Além de acesso sequencial, suportam também **acesso direto através de índices**

6

Listas em Python

- Em Python, os elementos (valores) de uma lista podem ser de **qualquer** tipo.
- Uma variável do tipo lista é escrita entre colchetes com os elementos separados por vírgula.

```
# Exemplos de lista em Python
valores = [1.3, 5, 7.7]
lista = ["Z", 3, 9.1, "USP", [8, 4], 2+3j]
```

7

Listas: *acesso de elementos*

- Para acessar (ler ou escrever) um elemento do vetor, basta informar o índice entre **colchetes**
 - **[idx]** : acessa um elemento de índice **idx**
 - primeiro elemento tem índice **0**
 - último elemento tem índice **-1**

```
>>> v = [23, 97, 54, 21, -14]
>>> v[1]
97
>>> v[-3]
54
>>> v[-1] == v[4]
True
```

v[0]	23	v[-5]
v[1]	97	v[-4]
v[2]	54	v[-3]
v[3]	21	v[-2]
v[4]	-14	v[-1]

8

Listas: *acesso de elementos*

- Exemplo:

```
>>> lista = [1,2,3,4]
>>> lista[4]
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```



9

Listas: *acesso de elementos*

- Uma **string** pode ser acessada como uma **lista**.

```
>>> V = "amora"
>>> V[1]
'm'
>>> V[-2]
'r'
>>> V[-1]
'a'
```

V[0]	a	V[-5]
V[1]	m	V[-4]
V[2]	o	V[-3]
V[3]	r	V[-2]
V[4]	a	V[-1]

```
>>> V[0] = 'A'
```

Uma string não pode ser modificada!

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

10

Listas: *função* **list**

- Pode ser usada para converter uma **string** em uma **lista**
- Uma lista pode ser **modificada**, já uma string **não**
- A conversão inversa, pode ser feita com o método **join** da classe **string**

```
>>> v = list("amora")
>>> v
['a', 'm', 'o', 'r', 'a']
>>> v[0] = 'A'
>>> v
['A', 'm', 'o', 'r', 'a']
>>> ''.join(v)
'Amora'
```

11

Listas: *fatias* (*slices*)

- Podemos fatiar uma lista da seguinte forma:
 - **[começo:fim]**: vai do índice **começo** até o índice **fim-1**
 - **[começo:]**: vai de **começo** até o final da lista
 - **[:fim]**: vai do início da lista até **fim-1**
 - **[:]**: retorna a lista toda
 - **Obs.:** lembrando que **fim-1** é o **penúltimo** elemento!
- **Incrementos** em fatias:
 - **[começo:fim:incremento]**: vai do índice **começo** até o índice **fim-1** (sem ultrapassá-lo), com passo **incremento**. O valor *default* é **1**.

12

Listas: *fatias (slices)*

- **Exemplos:**

```
>>> lista = ["Z",3,9.1,"USP",[8,4],2+3j]
>>> lista[1:]
[3, 9.1, 'USP', [8, 4], (2+3j)]
>>> lista[:1]
['Z']
>>> lista[1:3]
[3, 9.1]
>>> lista[0:-1]
['Z', 3, 9.1, 'USP', [8, 4]]
```

13

Listas: *fatias (slices)*

- **Exemplos:**

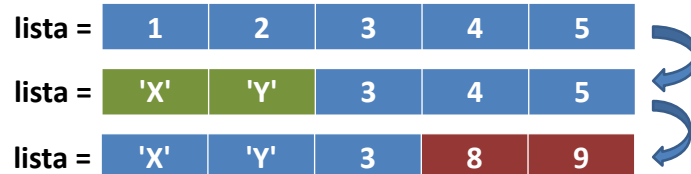
```
>>> lista = [1,2,3,4,5,6]
>>> lista[0:-1:2]
[1, 3, 5]
>>> lista[5:0:-1]
[6, 5, 4, 3, 2]
>>> lista[1:3]
[2, 5]
>>> lista[::-1]
[6, 5, 4, 3, 2, 1]
```

14

Listas: *atribuição a fatias*

- A **atribuição** a uma fatia requer que o valor atribuído seja uma sequência (uma lista ou uma string, por exemplo)
- A atribuição **substitui** os elementos da fatia pelos da sequência

```
>>> lista = [1,2,3,4,5]
>>> lista[:2] = ['X','Y']
>>> nova_lista = [8,9]
>>> lista[3:] = nova_lista
```



15

Listas: *atribuição a fatias*

- **Exemplos:**

```
>>> lista = [1,2,3,4,5]
>>> lista[1:3] = [[7]]
>>> lista
[1, [7], 4, 5]
>>> lista[1:-1] = [8,9]
>>> lista
[1, 8, 9, 5]
>>> lista[:3] = "xyz"
>>> lista
['x', 'y', 'z', 5]
```

16

Listas: *atribuição a fatias*

- Exemplos:

```
>>> lista
['x', 'y', 'z', 5]
>>> lista[:3] = "a,b,c"
>>> lista
['a', ' ', 'b', ' ', 'c', 5]
>>> lista[:2] = 1,2,3
>>> lista
[1, 2, 3, 'b', ' ', 'c', 5]
```

Note que a lista vai sendo modificada!

17

Listas: *atribuição a fatias*

- Exemplos:

```
>>> L = [1,2,3,4,5]
>>> L[0:2] = ['a','b','c']
>>> L
['a', 2, 'b', 4, 'c']
>>> L[0:2] = [8,9]
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: attempt to assign sequence of size
2 to extended slice of size 3
```

Se o incremento de uma fatia é diferente de 1,
a atribuição deve ter o mesmo número de elementos

18

Listas: *concatenação e repetição*

- Para concatenar e repetir elementos em uma lista, podemos usar os seguintes operadores:

Símbolo	Operação
+	concatenação
*	repetição
+=	concatena e atribui
*=	repete e atribui

19

Listas: *concatenação e repetição*

- **Exemplos:**

```
>>> lista = 2*[0]
>>> lista
[0, 0]
>>> lista = lista + [1, 2]
>>> lista
[0, 0, 1, 2]
>>> lista += 3*[3]
>>> lista
[0, 0, 1, 2, 3, 3, 3]
>>> lista *= 2
>>> lista
[0, 0, 1, 2, 3, 3, 3, 0, 0, 1, 2, 3, 3, 3]
```

20

Listas: *inicialização*

- Uma **lista vazia** (`[]`) pode ser criada quando não sabemos o que colocar nela.

```
vetor = [] % lista vazia
vetor[0] = 100
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```



21

Listas: *inicialização*

- Se uma lista vai ser usada como um **vetor**, isto é, vai conter uma quantidade conhecida de elementos, logo é conveniente iniciá-la.

```
>>> vetor = 8*[0]
>>> vetor
[0, 0, 0, 0, 0, 0, 0, 0]
>>> vetor[0] = 100
>>> vetor
[100, 0, 0, 0, 0, 0, 0, 0]
```

22

Listas: *inicialização*

- No uso de E.D., às vezes é importante preencher uma posição com um valor “**inválido**”
- Uma boa opção é usar o valor especial **None**
 - Não faz parte de tipo nenhum
 - É melhor que usar **0** ou uma **lista vazia**
- Útil para criar uma lista “vazia” mas com um número conhecido de elementos

```
>>> vetor = [None] * 4
>>> vetor
[None, None, None, None]
```

23

Listas: *remoção de elementos*

- O operador **del** pode ser usado para deletar elementos de uma lista

```
>>> L = [1, 2, 3, ['ab', 'CD']]
>>> del L[2]
>>> L
[1, 2, ['ab', 'CD']]
>>> del L[2][1]
>>> L
[1, 2, ['ab']]
```

24

Listas: *operador* **in**

- Permite saber se um elemento **pertence** a uma lista
 - Serve também para **strings**

```
>>> lista = [1, 'x', 'yz']
>>> 1 in lista
True
>>> 3 in lista
False
>>> 'y' not in lista
True
>>> 'y' in lista[2]
True
>>> 'xy' not in 'xyzw'
False
```

25

Listas: *funções* **len**, **min**, **max**

- **len (lista)** : retorna o número de elementos de *lista*.
- **min (lista)** : retorna o menor elemento de *lista*.
- **max (lista)** : retorna o maior de elementos de *lista*.

```
>>> lista = [1, 2, 9, 3]
>>> len(lista)
4
>>> min(lista)
1
>>> max(lista)
9
>>> max(['a', 'ab', 'c'])
'c'
```

26

Listas: *funções* **len**, **min**, **max**

- As funções **min** e **max** podem ser usados também com vários argumentos ao invés de uma lista

```
>>> min(7,9,3,44)
3
>>> max(11,9)
11
>>> max([], [3], [1,2])
[3]
```

27

Listas: *repetição* **for**

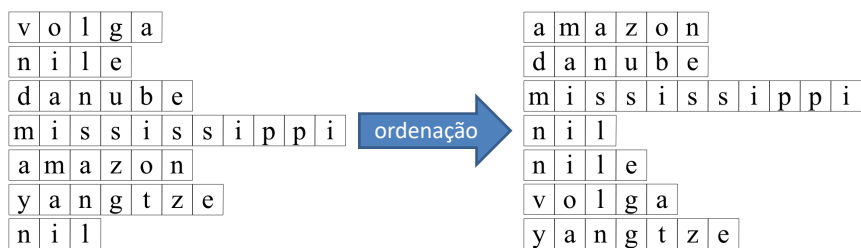
- O **for** permite iterar sobre os elementos de uma **lista**
- Forma geral: **for** *var* **in** *lista*: *instruções*
 - As *instruções* são repetidas para cada valor de *lista*
 - Durante a repetição, *var* possui o valor corrente da *lista*
- Na verdade, a função **range** cria uma lista de iteração!

```
lista = ['Oi', 3, 'x', -231]
for x in lista:
    print(x)
```

28

Listas: *comparação*

- **Ordem lexicográfica**, também conhecida como ordem do dicionário ou ordem alfabética
- **Exemplo:**
 - Dado os vetores: (7,4,6), (5,1,5), (2,4,6), (2,1,4), (5,1,7)
 - Após ordenação: (2,1,4), (2,4,6), (5,1,5), (5,1,7), (7,4,6)



29

Listas: *comparação*

- Listas são comparadas **lexicograficamente**
- Se duas listas são iguais até os k -ésimos elementos, o resultado da comparação depende da comparação entre os $(k+1)$ -ésimos elementos
- Se uma das listas tem só k elementos, então esta é considerada menor
- Duas listas são iguais se só se têm o mesmo comprimento e todos os elementos de mesma posição são iguais

30

Listas: *comparação*

- **Exemplos:**

```
>>> [1,2] < [2,3]
True
>>> [1,2] < [1,2,3]
True
>>> [1,2] != [1,2]
False
>>> min([ [1], [2,3], [3,4], [] ])
[]
>>> max([ [1], [2,3], [3,4], [] ])
[3, 4]
```

31

Listas: *a classe list*

- Assim como **string**, uma lista é na verdade um **objeto** de uma **classe** chamada **list**
- Não vimos ainda **programação orientada a objetos (POO)**, mas alguns pontos devem ser enfatizados
 - Listas possuem **métodos** que podem ser aplicados a elas
 - Um método se parece com uma função, mas são chamados de forma diferente:

`nome_objeto.método(args)`

- Para saber todos métodos de listas, escreva na **IDLE** **help(list)**

32

Listas: *métodos da classe* **list**

- **append (elemento)**
 - Acrescenta o **elemento** no fim da lista
 - Note que essa operação **altera** a lista

```
>>> lista = [1,3]
>>> lista.append(5)
>>> lista
[1, 3, 5]
>>> lista.append([7,9])
>>> lista
[1, 3, 5, [7, 9]]
```

33

Listas: *métodos da classe* **list**

- **Exemplo:** lista com 3 nomes

```
# Exemplo com erro :- (
nome.append(input("Informe o nome 1: "))
nome.append(input("Informe o nome 2: "))
nome.append(input("Informe o nome 3: "))
```

```
# Exemplo sem erro :-)
nome = []
nome.append(input("Informe o nome 1: "))
nome.append(input("Informe o nome 2: "))
nome.append(input("Informe o nome 3: "))
```

34

Listas: *métodos da classe* **list**

- **count (elemento)**
 - Retorna quantas vezes o **elemento** aparece na lista
- **extend (lista2)**
 - Adiciona os elementos de **lista2** no final da lista
 - **Altera** a lista ao invés de **retornar** a lista alterada

```
>>> lista = [1,2,3,1,2]
>>> lista.count(1)
2
>>> lista.extend([7,9])
>>> lista
[1, 2, 3, 1, 2, 7, 9]
```

35

Listas: *métodos da classe* **list**

- **index (elemento)**
 - Retorna o índice da 1ª ocorrência de **elemento** na lista

```
>>> lista = [6,2,11,41,92]
>>> lista.index(11)
2
>>> lista.index(9)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 9 is not in list
```



36

Listas: *métodos da classe* **list**

- **insert** (*índice*, *elemento*)

- Insere **elemento** na lista na posição **anterior** a indicada por **índice**

```
>>> lista = [1,2,3,4]
>>> lista.insert(2,'X')
>>> lista
[1, 2, 'X', 3, 4]
```

- Atribuições a fatias têm o mesmo efeito

```
>>> lista = [1,2,3,4]
>>> lista[2:2] = ['X']
>>> lista
[1, 2, 'X', 3, 4]
```

37

Listas: *métodos da classe* **list**

- **pop** (*índice*)

- Remove da lista o elemento na posição **índice** e o retorna
- O **índice** vale **-1** (último elemento) quando não for especificado

```
>>> lista = [1,2,3,4]
>>> lista.pop()
4
>>> lista
[1, 2, 3]
>>> lista.pop(1)
2
>>> lista
[1, 3]
```

38

Listas: *métodos da classe* **list**

- **remove (elemento)**
 - Remove da lista o 1ª elemento igual a *elemento*

```
>>> lista = ['oi', 'ai', 'ola', 'alo']
>>> lista.remove('ai')
>>> lista
['oi', 'ola', 'alo']
>>> lista.remove('aba')
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```



39

Listas: *métodos da classe* **list**

- **reverse ()**
 - Inverte a ordem dos elementos da lista
 - Cuidado, retorna **None**

```
>>> lista = [1,2,3]
>>> lista.reverse()
>>> lista
[3, 2, 1]
```

40

Exemplo: cálculo do IMC de 30 pessoas!

- Vamos criar duas listas (nomes e notas) com 30 posições
- A posição (índice) k da lista de **nomes** corresponde a posição k da lista do **IMC**

	nomes					IMC
0	João	↔	0	27.0		
1	Maria	↔	1	38.5		
...	...		...	...		
29	Pedro	↔	29	25.5		

41

Exemplo: cálculo do IMC de 30 pessoas!

```

num_pessoas = 30
nomes = []
IMC = []

for i in range(num_pessoas):
    nomes.append(input("Informe o nome: "))
    peso = float(input("O peso de " + nomes[i] + ": "))
    h = float(input("A altura de " + nomes[i] + ": "))
    IMC.append(peso/h**2)

for i in range(num_pessoas):
    print("%s seu IMC é %.2f" % (nomes[i], IMC[i]))

```

42

Exemplo: cálculo do IMC de 30 pessoas!

```
num_pessoas = 30
nomes = num_pessoas*[None]
IMC = num_pessoas*[None]

for i in range(num_pessoas):
    nomes[i] = input("Informe o nome: ")
    peso = float(input("O peso de " + nomes[i] + ": "))
    h = float(input("A altura de " + nomes[i] + ": "))
    IMC[i] = peso/h**2

for i in range(num_pessoas):
    print("%s seu IMC é %.2f " % (nomes[i],IMC[i]))
```

43

Exemplo

- Programa que retorna uma lista com todos os números ímpares no intervalo $[1, N]$.

```
lista = []
N = int(input('Digite um número: '))

for i in range(1,N+1,2):
    lista = lista + [i]

print(lista)
```

44

Exemplo

- Programa que retorna uma lista com todos os números ímpares no intervalo $[1, N]$ **na ordem decrescente**.

```
lista = []
N = int(input('Digite um número: '))

for i in range(1, N+1, 2):
    lista = [i] + lista

print(lista)
```

45

Exemplo

- Programa que retorna uma lista com todos os números ímpares no intervalo $[1, N]$ **na ordem decrescente**.

```
lista = []
N = int(input('Digite um número: '))

for i in range(1, N+1, 2):
    lista = lista + [i]

lista.reverse()
print(lista)
```

46

Exemplo

- Programa que lê uma lista de números inteiros do teclado, soma 2 aos seus elementos e imprime a lista resultante

```
sair = 'n'
lista = []
while (sair == 'n' or sair == 'N'):
    n = int(input('Entre com um numero: '))
    lista.append(n)
    sair = input('Deseja sair? (s/n): ')
print(lista)
for i in range(len(lista)):
    lista[i] += 2
print(lista)
```

47

Listas: *list comprehension*

- Estamos acostumados a representar um conjunto da seguinte forma:

Matemática

$$V = \{0,1,2,3,4,5,7,8,9,10\}$$

$$S = \{x^2 : x \in V\}$$

$$M = \{x : x \in S \text{ e } x \text{ é ímpar}\}$$

No Python

```
V = [i for i in range(11)]
S = [x**2 for x in V]
M = [x for x in S if x%2]
```

48

Exemplo

- Programa que lê uma lista de números inteiros do teclado, soma 2 aos seus elementos e imprime a lista resultante

```
lista = [int(i) for i in input().split()]
print(lista)

for i in range(len(lista)):
    lista[i] += 2

print(lista)
```

Agora sabemos converter vários números por atacado.

49

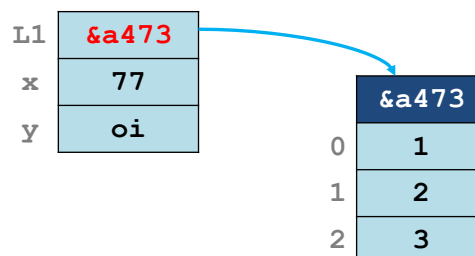
Listas: *representação em memória*

- O valor de uma variável de lista é na verdade um **endereço de memória**

No Python

```
>>> L1 = [1,2,3]
>>> x = 77
>>> y = 'oi'
```

Na Memória



50

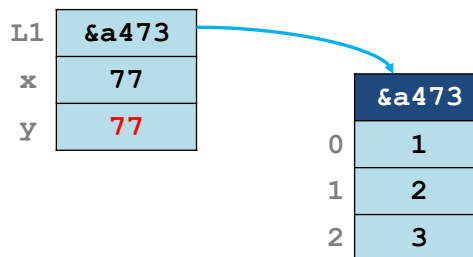
Listas: *representação em memória*

- O valor de uma variável de lista é na verdade um **endereço de memória**

No Python

```
>>> L1 = [1,2,3]
>>> x = 77
>>> y = 'oi'
>>> y = x
```

Na Mémoira



51

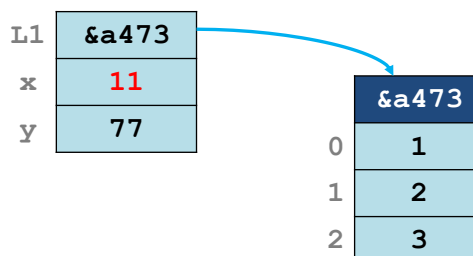
Listas: *representação em memória*

- O valor de uma variável de lista é na verdade um **endereço de memória**

No Python

```
>>> L1 = [1,2,3]
>>> x = 77
>>> y = 'oi'
>>> y = x
>>> x = 11
```

Na Mémoira



52

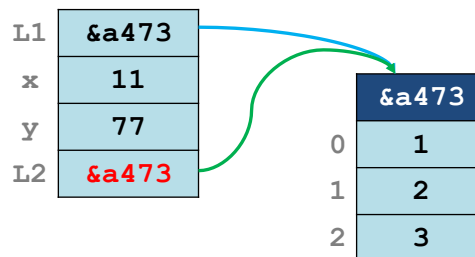
Listas: *cópia de listas*

- O valor do endereço de memória que é copiado

No Python

```
>>> L1 = [1,2,3]
>>> x = 77
>>> y = 'oi'
>>> y = x
>>> x = 11
>>> L2 = L1
```

Na Mémoire



53

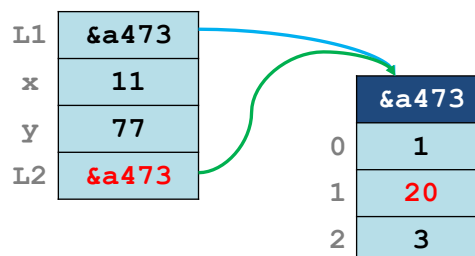
Listas: *cópia de listas*

- O valor do endereço de memória que é copiado

No Python

```
>>> L1 = [1,2,3]
>>> x = 77
>>> y = 'oi'
>>> y = x
>>> x = 11
>>> L2 = L1
>>> L2[1] = 20
```

Na Mémoire



54

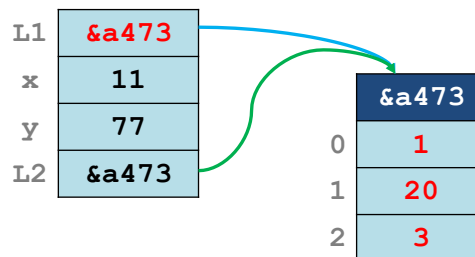
Listas: *cópia de listas*

- As duas listas apontam para o mesmo endereço → logo se uma for modificada a outra também será!

No Python

```
>>> L1 = [1,2,3]
>>> x = 77
>>> y = 'oi'
>>> y = x
>>> x = 11
>>> L2 = L1
>>> L2[1] = 20
>>> L1
[1, 20, 3]
```

Na Mémoire



Como evitar isso?

55

Listas: *métodos da classe list*

- copy()**
 - Retorna uma cópia “rasa”
 - Cuidado, se tiver listas aninhadas!
 - passagem por referência

```
>>> L1 = [1,2,3]
>>> L2 = L1.copy()
>>> L2
[1, 2, 3]
```

56

Listas: *métodos da classe* **list**

- **copy()**
 - Retorna uma cópia “rasa”
 - Pode ser feito via *slices*

```
>>> L1 = [1, 2, 3]
>>> L2 = L1[:]
>>> L2
[1, 2, 3]
```

- Usando o **módulo copy**

```
>>> import copy # carrega o módulo copy
>>> L2 = copy.copy(L1)
>>> L2
[1, 2, 3]
```

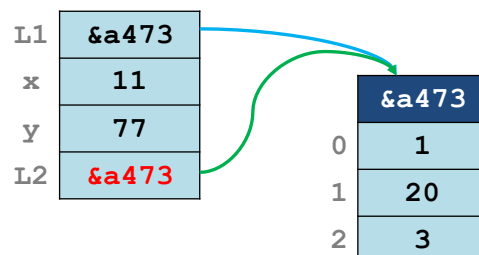
57

Listas: *cópia de listas*

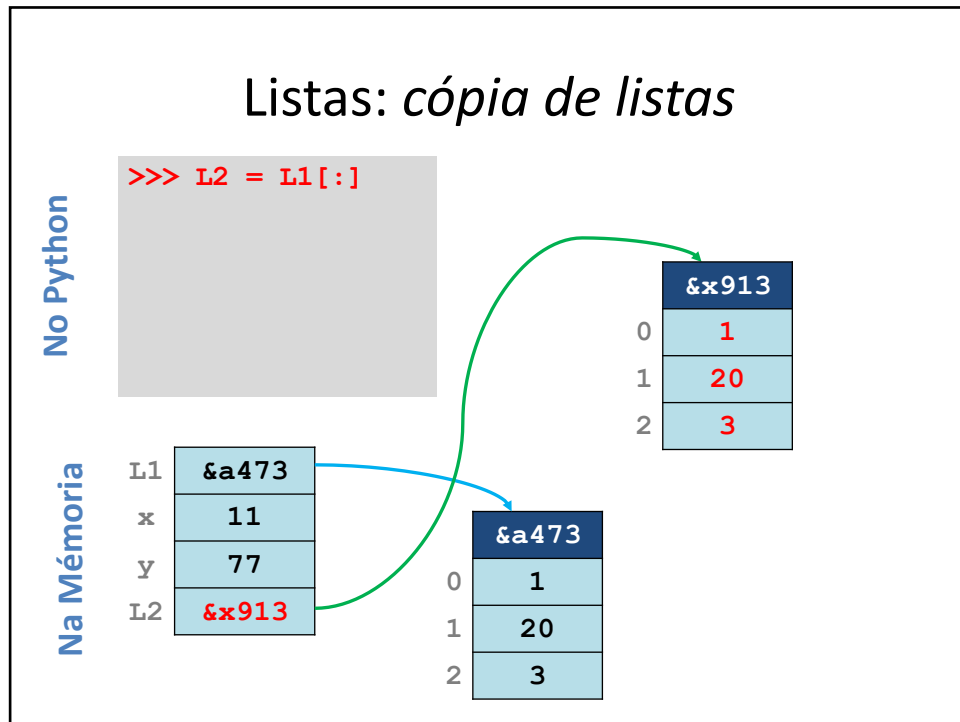
No Python

```
>>> L2 = L1[:]
```

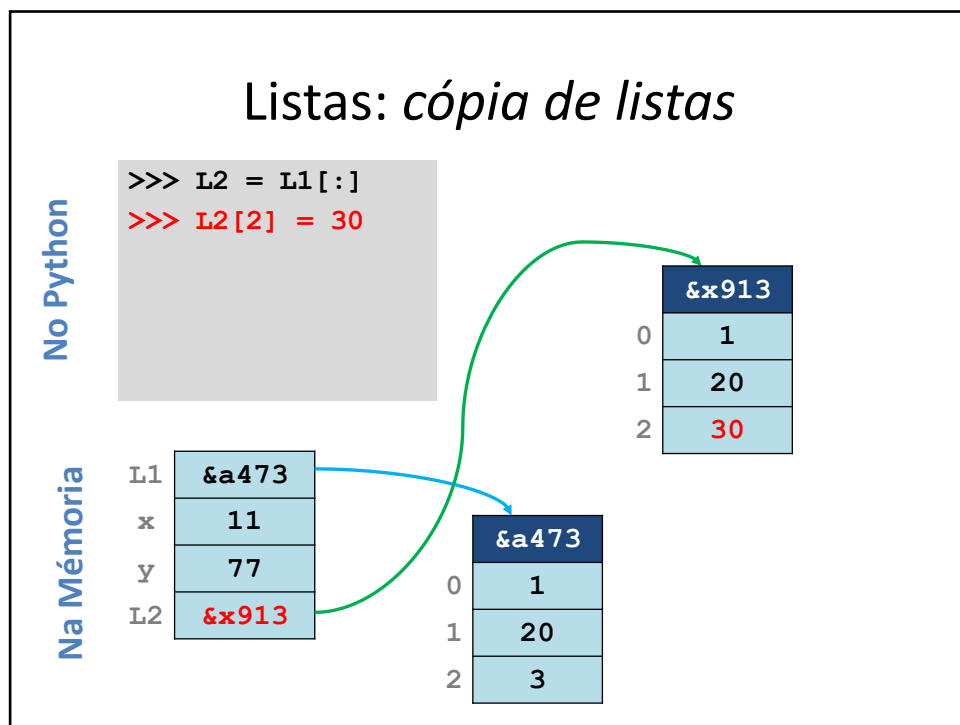
Na Memória



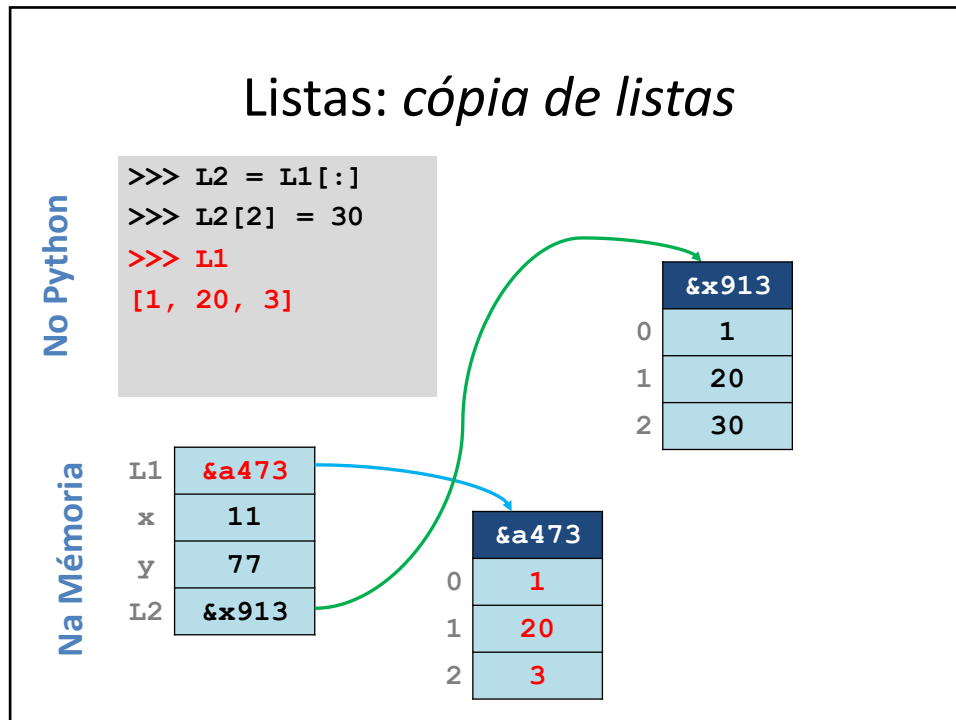
58



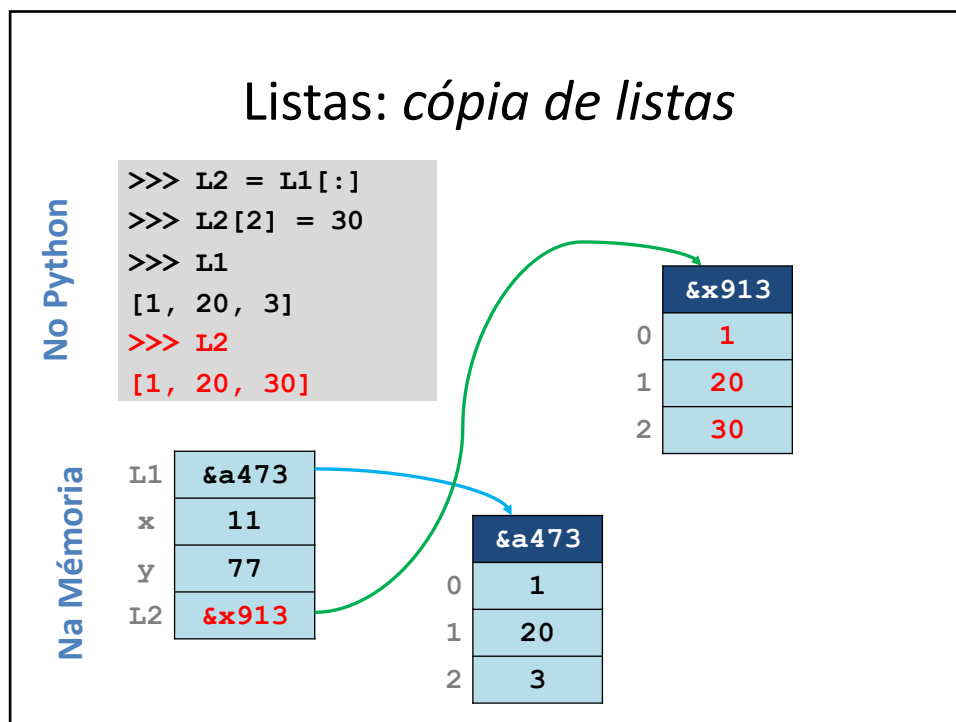
59



60



61



62

Tipos de Passagem de Parâmetro

- **Por valor:** o valor da variável na chamada é copiado para a variável
 - Alterações **não** são refletidas na variável original
- **Por referência:** o endereço de memória é copiado para a variável
 - Alterações **são** refletidas na variável original
 - é como se o mesmo “gaveteiro” fosse usado

Python usa passagem de parâmetro por valor, exceto em listas aonde é por referência.