Programação I / Introdução à Programação

Capítulo 5, "Data types" (listas e tuplos)

João Pedro Pedroso

2024/2025

Última aula: listas:

- Operadores:
  - comprimento, concatenação, repetição, pertença, indexação
- Iteração
- Remoção de elementos (por índice e por valor)
- Objetos e referências
- Métodos
  - que não alteram as listas
    - `index`, `count`, `copy`
  - que alteram as listas
    - `append`, `insert`, `remove`, `sort`

Hoje:

- Listas e tuplos

Fatias de listas

(Considerando uma lista `s`)

`s[i:j:k]` elementos de i a $j-1$ com incrementos k

- incrementos negativos $\rightarrow$ percorrer a lista ao contrário: elementos $j, j-k, \ldots$ até i (exclusive).

```
>>> vogais = ['a','e','i','o','u']
>>> vogais[::2]      # índices pares
['a','i','u']
>>> vogais[1::2]     # índices ímpares
['e','o']
>>> vogais[4:1:-1]
['u', 'o', 'i']
>>> vogais[::-1]     # inverter a lista
['u', 'o', 'i', 'e', 'a']
```

Função `range()`: gerar sequências de inteiros

- `range(n)` → [0, 1, ... n-1]

```
>>> list(range(7))
[0, 1, 2, 3, 4, 5, 6]
```

- `range(m,n)` → [m, m+1, ..., n-1]

```
>>> list(range(2,7))
[2, 3, 4, 5, 6]
```

- `range(m,n,k)` → `r[i] = m + k*i`, onde `i >= 0` e
  - `r[i] < n` para `k>0`

```
>>> list(range(2,7,3))
[2, 5]
```

  - `r[i] > n` para `k<0`

```
>>> list(range(7,2,-3))
[7, 4]
```

  - se o sinal de k for diferente do de $n - m$ → sequência vazia:

```
>>> list(range(7,2,3))
[]
```

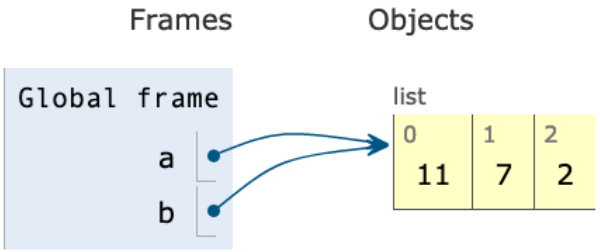
Operador `in`: teste de pertença

- com *listas:* `x in y` é `True` se `x` for um *elemento* de `y`
  - caso contrário é `False`
- com *strings:* `x in y` é `True` se `x` for uma *substring* de `y`
  - caso contrário é `False`
  - caso particular: `x` ser um *elemento* de `y` (como com listas)

Objetos e referências

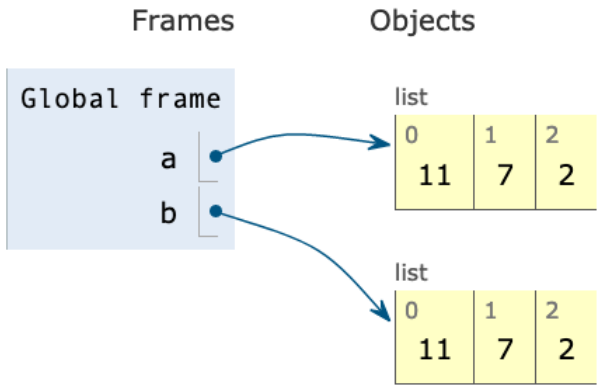
- *Aliasing*

```
>>> a = [11, 7, 2]
>>> b = a
>>> a is b
True
```



- *Cloning*:

```
>>> a = [11, 7, 2]
>>> b = list(a)  # or: b = a[:], b = a.copy()
>>> b
[11, 7, 2]
>>> a is b
False
```



Objetos e referências

Para testar se dois nomes se referem ao mesmo objeto: operador `is`

- *object identity*
- `x is y` é verdadeiro se e só se `x` e `y` são o mesmo objeto

- Nota: como *strings* são imutáveis, o Python otimiza a utilização de memória:

```
>>> a = "banana"
>>> b = "banana"
>>> a is b
True
```

Módulo copy

```
a = [[1,2,-1],
     [3,1,0],
     [0,1,-2]]

b = a.copy()  # shallow copy

# import copy              # alternative form
# c = copy.deepcopy(a)     # in lines below

from copy import deepcopy
c = deepcopy(a)
```

Qual é a diferença? → *slide* a seguir

Python 3.6
(known limitations)

```
1  a = [[1,2,-1],
2       [3,1,0],
3       [0,1,-2]]
4
5  b = a.copy()  # shallow copy
6
7  from copy import deepcopy
8  c = deepcopy(a)
```

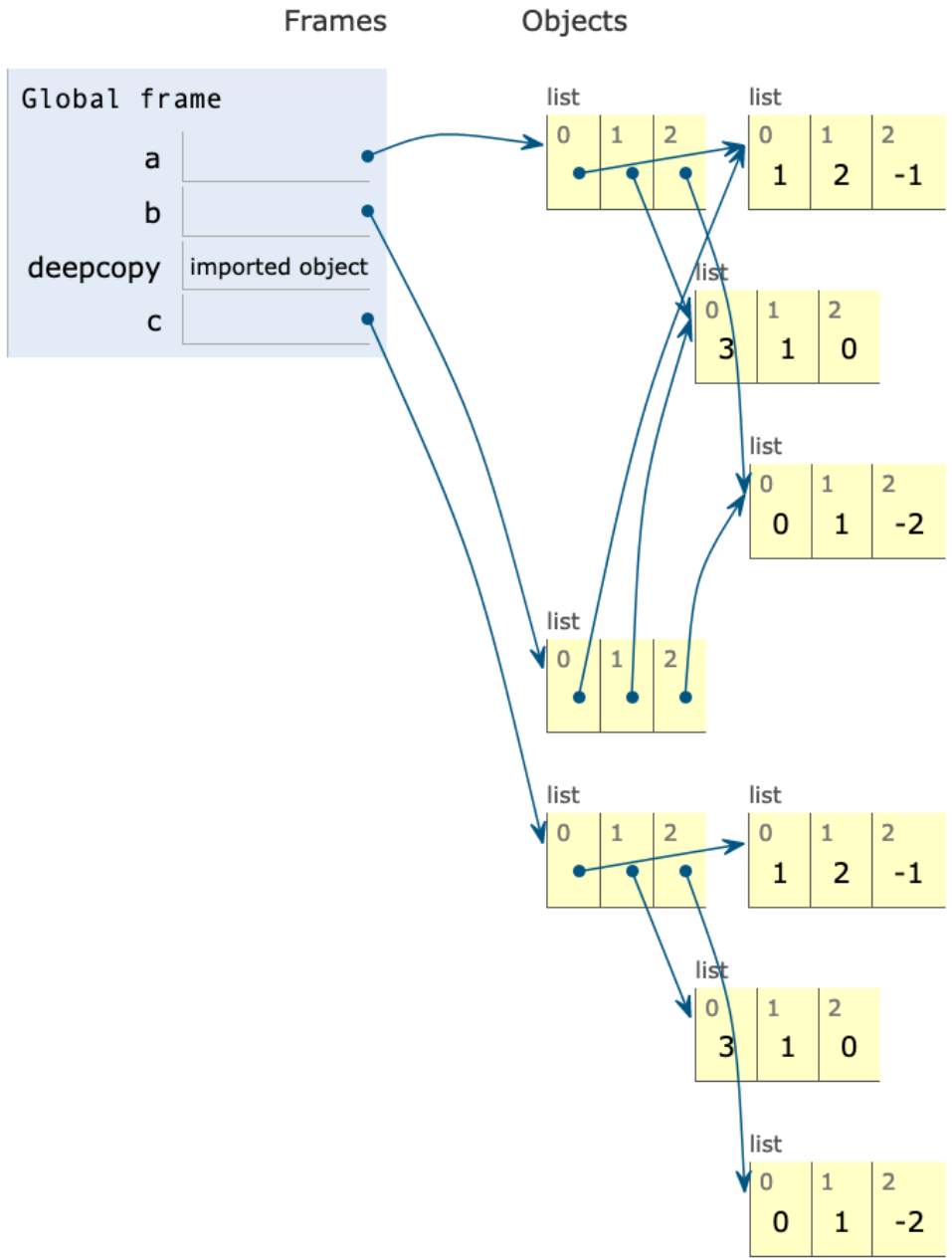
Edit this code

cuted
ute

<< First < Prev Next > Last >>

Done running (6 steps)

(NEW!)



Instrução import

Nota sobre a instrução `import`:

```
from copy import deepcopy
c = deepcopy(a)
```

é equivalente a

```
import copy
c = copy.deepcopy(a)
```

- útil quando queremos
  - importar apenas uma (ou poucas) das funcionalidades do módulo
  - permite código mais conciso

Mais métodos sobre listas: reverse, count, extend, index, remove

```
>>> mylist = [5, 27, 3]
>>> mylist.append(12)
>>> mylist.insert(1, 12) # Insert 12 at pos 1, shift other items up
>>> mylist
[5, 12, 27, 3, 12]
>>> mylist.count(12) # How many times is 12 in mylist?
2
>>> mylist.extend([5, 9, 5, 11]) # Put whole list onto end of mylist
>>> mylist
[5, 12, 27, 3, 12, 5, 9, 5, 11]
>>> mylist.index(9) # Find index of first 9 in mylist
6
>>> mylist.reverse()
>>> mylist
[11, 5, 9, 5, 12, 3, 27, 12, 5]
>>> mylist.remove(12) # Remove the first 12 in the list
>>> mylist
[11, 5, 9, 5, 3, 27, 12, 5]
>>> mylist.sort()
>>> mylist
[3, 5, 5, 5, 9, 11, 12, 27]
```

Alteração de listas *in place*

```
>>> xs = [1, 2, 3, 4, 5]
>>> for i in range(len(xs)):
...     xs[i] = xs[i]**2
...
>>> xs
[1, 4, 9, 16, 25]
```

no nosso idioma favorito:

```
xs = [1, 2, 3, 4, 5]
for (i, val) in enumerate(xs):
    xs[i] = val**2
```

Funções com listas como parâmetros

```
def double_stuff(stuff_list):
    """ Overwrite each element in a_list with double its value. """
    for (index, stuff) in enumerate(stuff_list):
        stuff_list[index] = 2 * stuff
```

Quando utilizamos:

```
>>> things = [2, 5, 9]
>>> double_stuff(things)
>>> things
[4, 10, 18]
```

Listas, funções puras e modificadores

- Funções puras não produzem *efeitos laterais*
- Um efeito lateral possível é a modificação de argumentos

```
1  def double_stuff(stuff_list):
2      for (index, stuff) in enumerate(stuff_list):
3          stuff_list[index] = 2 * stuff
```

- Verão "pura"

```
1  def double_stuff(a_list):
2      new_list = []
3      for value in a_list:
4          new_elem = 2 * value
5          new_list.append(new_elem)
6      return new_list
```

- modificador

```
>>> things = [2, 5, 9]
>>> double_stuff(things)
>>> print(things)
[4, 10, 18]
```

- função pura

```
>>> things = [2, 5, 9]
>>> more_things = double_stuff(things)
>>> print(things, more_things)
[2, 5, 9] [4, 10, 18]
```

- Modificador é mais eficiente
  - a utilizar se não precisarmos mais do valor inicial de `things`
  - suscetível de originar erros
- Vimos no exemplo da função pura o padrão de *função que retorna uma lista*

Listas e *strings*

```
>>> song = "The rain in Spain..."
>>> words = song.split()
>>> words
['The', 'rain', 'in', 'Spain...']
>>> song.split("ai")
['The r', 'n in Sp', 'n...']

>>> glue = ";"
>>> phrase = glue.join(words)
>>> phrase
'The;rain;in;Spain...'
>>> " --- ".join(words)
'The --- rain --- in --- Spain...'
>>> "".join(words) 'TherainininSpain...'
```

Listas e `range`

- `range` não calcula os valores todos quando é chamado
  - coloca o cálculo *"em espera"*
  - só quando os valores são necessários é que os calcula
  - → cálculo lazy

```
1  def f(n):
2      for i in range(101, n):
3          if (i % 21 == 0):
4              return i
```

```
>>> f(1000000000)
105
```

- compare com `for i in list(range(101, n))`
- *versões 2 e 3 do Python são diferentes!*

Tuplos

- Sequências ordenadas de elementos:

  `(e1, e2, ..., en)`
- Acesso aos elementos por índices
- Ao contrário das listas, os tuplos são imutáveis

Operações básicas

- comprimento

```
>>> len(('Pedro',12))
2
```

- concatenação

```
>>> ('Pedro',12)+('João',14)
('Pedro',12,'João',14)
```

- repetição

```
>>> 2*('Pedro',12)
('Pedro',12,'Pedro',12)
```

- pertença

```
>>> 12 in ('Pedro',12)
True
```

- iteração

```
>>> for x in ('Pedro',12):
        print(x)
Pedro
12
```

Acesso aos elementos (indexação)

```
>>> nota = ('Pedro', 12)
>>> nota[0]
'Pedro'
>>> nota[1]
12
>>> nota[0] = 'Joao'
TypeError: 'tuple' object does not support item assignment
```

Caso especial: tuplos com um só elemento

```
>>> t = (42,)
>>> t
(42,)
>>> len(t)
1
>>> t[0]
42
```

Atribuição a tuplos de variáveis

```
>>> (x,y) = (5,-7)
>>> x
5
>>> y
-7
```

Ou simplesmente:

```
>>> x,y = 5,-7
>>> x
5
>>> y
-7
```

Name swapping:

```
>>> x,y = 5,-7
>>> x,y = y,x
>>> x
-7
>>> y
5
```

Listas e tuplos combinados

Representar uma agenda como uma lista de pares *nome/email*:

```
[('Maria João', 'mj@mail.pt'),
 ('José Manuel', 'jm@mail.pt'),
 ('João Pedro', 'jp@mail.pt')]
```

Operações:

- acrescentar uma entrada (nome e email)
- procurar email pelo nome

Acrescentar uma entrada

```
def acrescentar(agenda, nome, email):
    "Acrescentar um nome e email à agenda."
    agenda.append((nome,email))
```

Procurar um nome (1)

```
def procurar(agenda, txt):
    "Procurar emails por parte do nome."
    emails = []
    for par in agenda:
        if txt in par[0]:          # txt ocorre no nome?
            emails.append(par[1]) # acrescenta email
    return emails
```

Procurar um nome (2)

```
def procurar(agenda, txt):
    "Procurar emails por parte do nome."
    emails = []
    for (nome,email) in agenda:
        if txt in nome:          # txt ocorre no nome?
            emails.append(email) # acrescenta email
    return emails
```

Exemplos

```
>>> agenda = []
>>> acrescentar(agenda, "Maria João",
                "mj@mail.pt")
>>> acrescentar(agenda,"João Pedro",
               "jp@mail.pt")

>>> procurar(agenda, "Maria")
['mj@mail.pt']

>>> procurar(agenda, "João")
['mj@mail.pt', 'jp@mail.pt']
```

Usar listas ou tuplos?

- Listas → sequências mutáveis
- Tuplos → sequências imutáveis
- Os tuplos são necessários em casos especiais:
  - *chaves de dicionários* — próximas aulas
- Podemos converter entre os dois tipos:

  `list(...)` converte para lista
  `tuple(...)` converte para tuplo

```
>>> a = [1, 2, 3]
>>> b = tuple(a)
>>> b
(1, 2, 3)
>>> c = list(b)
>>> c[1] = "halt"
>>> a, b, c
([1, 2, 3], (1, 2, 3), [1, 'halt', 3])
```

Noções estudadas

aliases
clone
data structure
delimiter
element
immutable
immutable data value
index
item
list
list traversal
modifier
mutable data value
nested list
object
pattern
promise
pure function
sequence
side effect
step size
tuple
tuple assignment

Próxima aula

- Tipos de dados do Python: *dicionários*