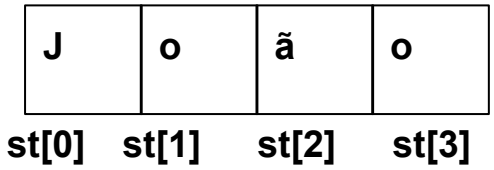
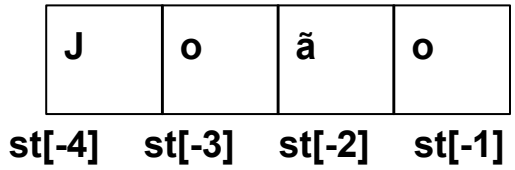
Tópico 4 - Tipos estruturados

Tuplas e Listas

Sequências

- Valores contíguos (adjacentes) e normalmente relacionados.
  - String é um exemplo de estrutura com valores contíguos: Cadeia de caracteres
- Sequências podem ser acessadas a partir do fim.





Sequências

- Conhecidas como vetores (arrays) em outras linguagens de programação.
- Há três tipos de sequências básicas em Python:
  - String (Vista anteriormente)
  - Tuplas
  - Listas

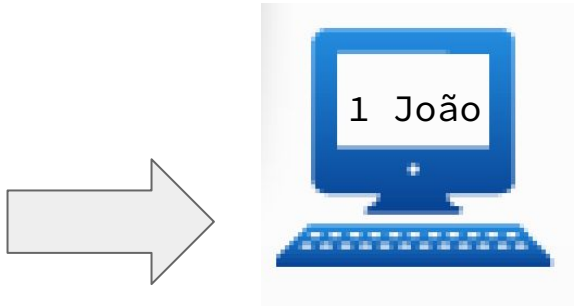
Tuplas

- Estrutura representada com ().
  - t=() - Tupla vazia
  - t=(1,2.5,"Joao")
- Pode relacionar diferentes tipos escalares ou não.
  - t=(**1**,**2.5**,**"Joao"**)
- Imutável: os valores na tupla não podem ser alterados
  - t[0]=3

Tuplas

- Valores podem ser acessados pelos índices como no caso das Strings.

```
x=t[0]
y=t[2]
print(x,y)
```



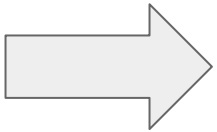
- Outros acessos possíveis por índices:

```
t[1:2]  ⟹  (2.5,) - Tupla com 1 único valor.
t[1:3]   ⟹  (2.5,"Joao")
t[0:2]  ⟹  (1,2.5)
```

Tuplas

- Pode-se concatenar tuplas

```
t=()
t=t+(1,2)
t=t+('a',)
t=t+(4.5,'!',6)
print(t)
```



```
(1,2,'a',4.5,'!',6)
```

Tuplas

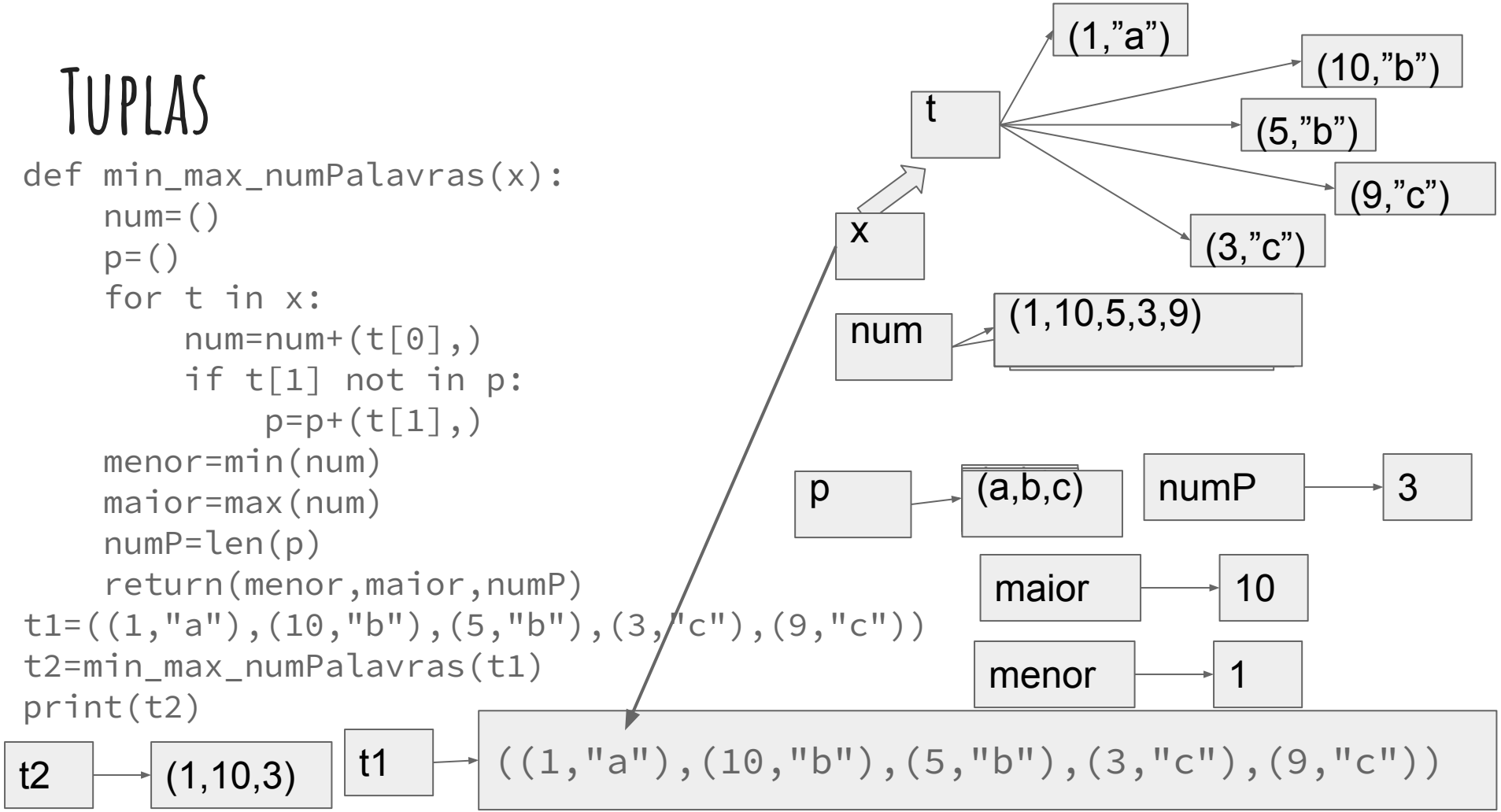
- Facilita a troca de valores

  `(x,y)=(y,x)`

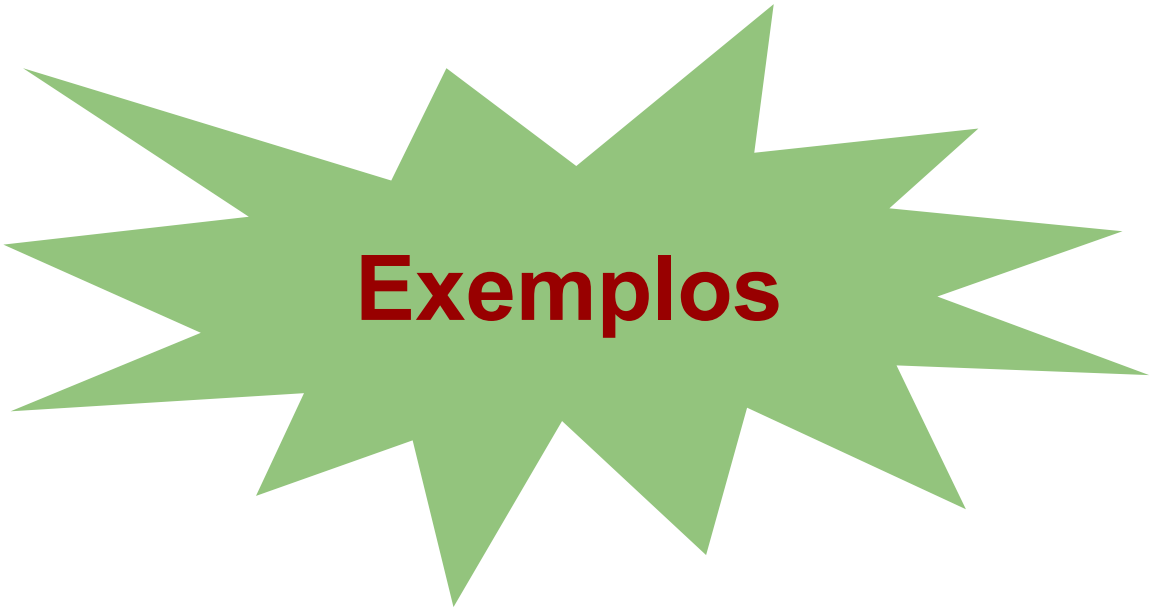
- Permite que funções retornem uma sequência de valores
- Permite realizar repetições para a coleção de valores na tupla

Tuplas

```
def min_max_numPalavras(x):
    num=()
    p=()
    for t in x:
        num=num+(t[0],)
        if t[1] not in p:
            p=p+(t[1],)
    menor=min(num)
    maior=max(num)
    numP=len(p)
    return(menor,maior,numP)
t1=((1,"a"),(10,"b"),(5,"b"),(3,"c"),(9,"c"))
t2=min_max_numPalavras(t1)
print(t2)
```

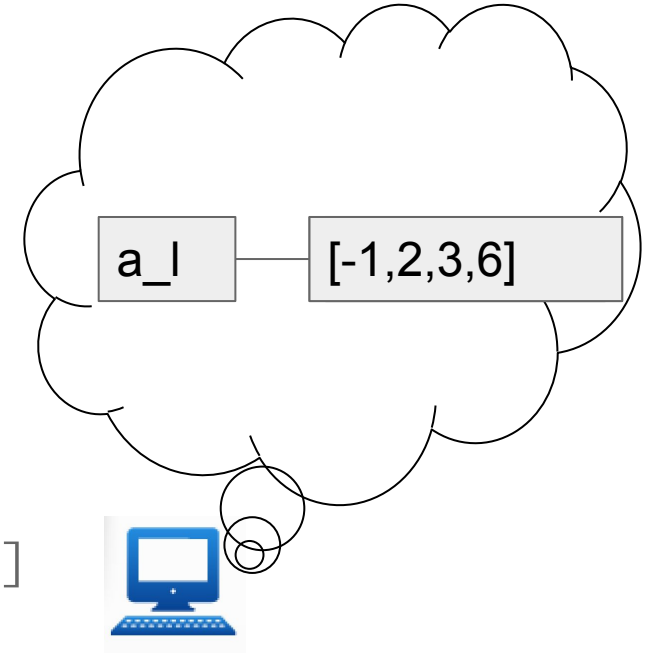


Tuplas



Listas

- Estrutura representada com [].
  - a_l=[] - Lista vazia
  - a_l=[1,2,3,[4,5]]
  - b_l=[**1**,**2.5**,**"Joao"**]
- Mutável: podem ser alteradas
  - a_l[0]=-1 ⟹ a_l=[-1,2,3,[4,5]]
  - a_l[3]=6 ⟹ a_l=[-1,2,3,6]



Listas

- Normalmente armazenam o mesmo tipo de dado
- Motivo:
  - Aplicar estruturas de repetição a partir dos valores da lista.

```
a_l=[10, 20, 30, 40, 50]
soma=0
for i in range(len(a_l)):
    soma = soma + a_l[i]
print(soma/len(a_l))
```

```
a_l=[10, 20, 30, 40, 50]
soma=0
for i in a_l:
    soma = soma + i
print(soma/len(a_l))
```

Listas

- Métodos em listas.
  - Listas são objetos em Python
  - Objetos possuem atributos e métodos
  - Atributos podem ser entendidos como parâmetros do objeto.
  - Métodos podem ser entendidos como funções associadas ao objeto.
  - Acesso ocorre como

```
<objeto>.<nomeMétodo>(<argumentos>)
```

Listas

- Métodos para incluir elementos
  - <lista>**.append(**<elemento>**)**

    a_l=[1,2,3]

    a_l.**append(**4**)** ⟹ a_l[1,2,3,4]

    Altera a lista adicionado um elemento ao final

Listas

- Métodos para incluir elementos de uma lista
  - <lista>**.extend([**e1,e2,...,eN**])**

    a_l=[1,2,3]

    a_l.**extend(**[4,5]**)** ⟹ a_l[1,2,3,4,5]

    Altera a lista adicionado elemento da lista de entrada como elementos na lista de destino

  - Não confundir com concatenar listas

    a_l1=[1,2,3] a_l2=[4,5]
    a_l3=a_l1+a_l2 ⟹ a_l3=[1,2,3,4,5]
    a_l1.extend(a_l2)⟹ a_l1=[1,2,3,4,5]

Listas

- Diferença obj.append() e obj.extend():

  a_l=[1,2,3]

  a_l.append([4,5]) ⟹ a_l[1,2,3,[4,5]]

  a_l.append(4,5) ⟹ **ERRO!! Apenas 1 valor!!**

  a_l.extend([4,5]) ⟹ a_l[1,2,3,[4,5],4,5]

Listas

- Métodos e funções para remover elementos de uma lista
  - **del**<lista>**[**índice] **- função**

    a_l=[1,2,3] del a_l[1] $\Longrightarrow$ a_l=[1,3]

    Remove elemento dado o seu índice na lista.
  - <lista>.pop() - método

    a_l=[1,2,3] x=a_l.pop() $\Longrightarrow$ a_l=[1,2] x=3

    Remove último elemento e retorna este elemento

Listas

- Métodos e funções para remover elementos de uma lista
  - <lista>.remove(elemento)

```
a_l=[81,72,23] x=a_l.remove(72)
```

$\Longrightarrow$ a_l=[81,23] x=72

Remove elemento

Remove a primeira ocorrência do elemento

Retorna erro se o elemento não está na lista

Listas

- Pode-se converter listas em strings e vice-versa.
- **list(**<string>**)**
  - Transforma uma string em lista.

    a_l=**list(**‘João’**)**⟹ a_l=[‘J’,’o’,’ã’,’o’]

- <objString>.split()
  - Retorna uma lista com a separação da string.
  - Por default, separa pelo caracter espaço, mas outros podem ser utilizados
    str = “Joao e Maria”
    x=str.split() ⟹ x=[‘Joao’,’e’,’Maria’]
    x=str.split(‘e’)⟹ x=[‘Joao ’,’ Maria’]

Listas

- <caracter>.**join(**<[lista]>**)**
  - Converte uma lista de caracteres em uma string.
  - Pode adiconar elementos específicos entre os elementos da lista de caracteres.

```
a_l = [‘@’,’#’,’!’,’x’]
x=’’.join(a_l)⟹ x=’@#!x’
x=’_’.join(a_l)⟹ x=’@_#_!_x’
x=’:’.join(a_l)⟹ x=’@:#:!:x’
```

Listas

- Outras funções e métodos
  - sorted(<lista>)

    a_l=[22,15,13,20] x=sorted(a_l) ⟹ x=[13,15,20,22]

    Retorna uma lista ordenada sem alterar a lista original

  - <lista>.sort()

    a_l=[22,15,13,20] a_l.sort() ⟹ a_l=[13,15,20,22]

    Altera a lista para ficar ordenada.

Listas

- Outras funções e métodos
  - <lista>.reverse()

a_l=[13,15,20,22] a_l.reverse()⟹a_l=[22,20,15,13]

Altera a lista invertendo a ordem.

Listas

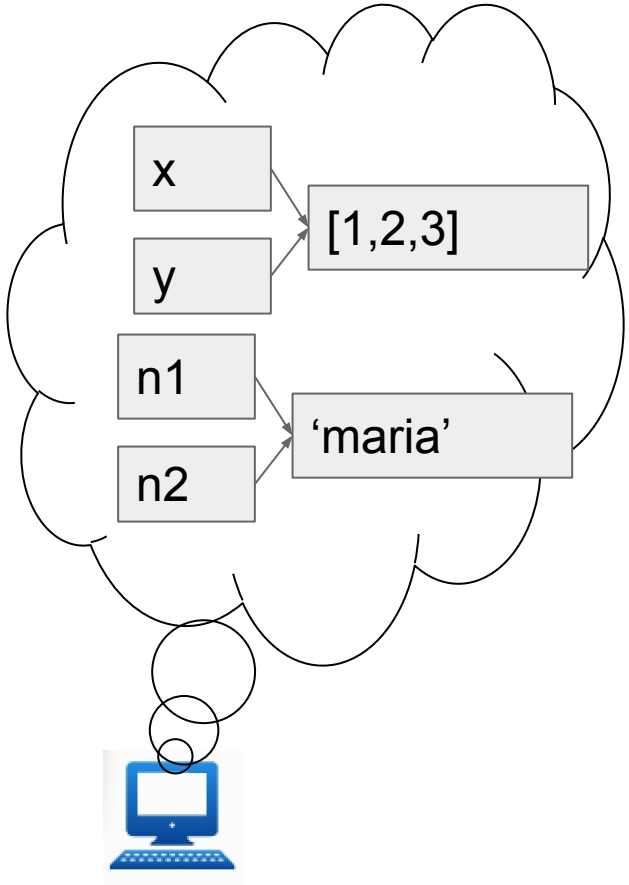
- Alias - “Aliás”- Pseudônimo

x=[1,2,3], y=x e print(x,y) - **objeto mutável**

⟹ o objeto [1,2,3] existe na memória e posso acessá-lo através da variável x e y: x=[1,2,3] y=x

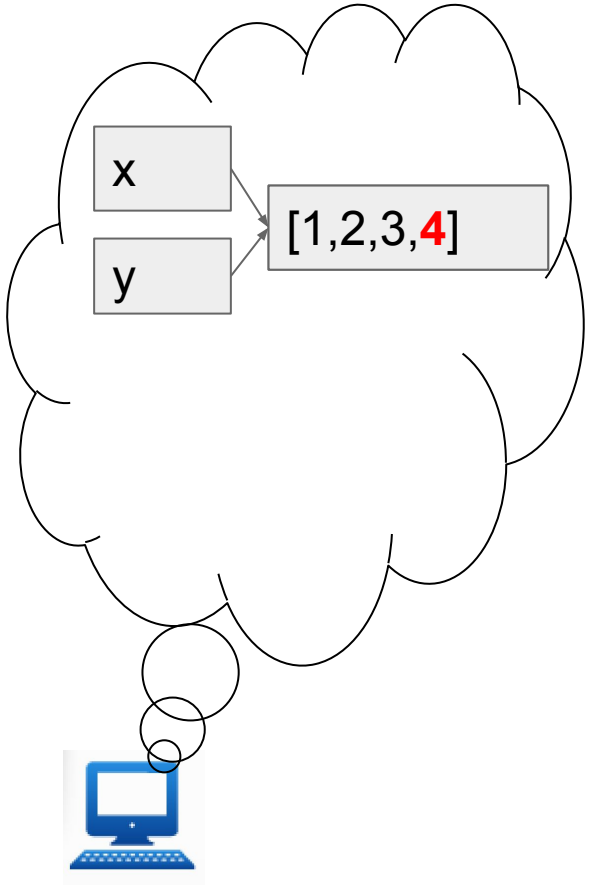
n1= ‘maria’ n2=n1 - **objeto não mutável**

⟹ o objeto ‘maria’ existe na memória e posso acessá-lo através da variável n1 e n2: n1=[1,2,3] n2=n1



Listas

- Alias - “Aliás”- Pseudônimo
  - Uma variável aponta para um objeto, sendo referenciado (acessado) pelo nome da variável.
  - Mudança em objetos mutáveis repercute em todo os alias (referências) daquele objeto:



`x=[1,2,3] y=x`

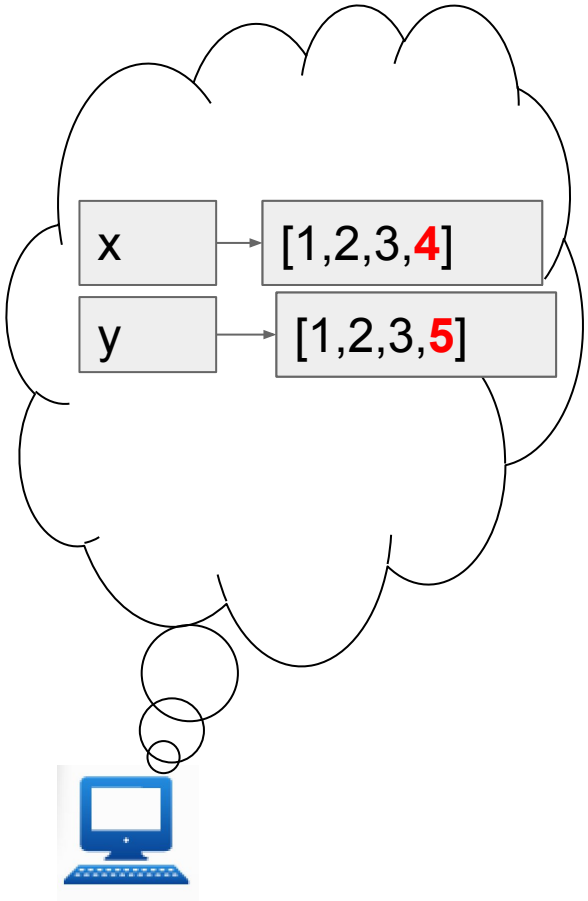
`y.append(4) print(x,y)` ⟹ o objeto muda para `[1,2,3,4]`, temos `x=[1,2,3,4]` e `y=[1,2,3,4]`

Mudança em y afeta x pois ocorre no objeto.

Listas

- Alias - “Aliás”- Pseudônimo
  - Variáveis apontando para o mesmo objeto são afetadas pelas mudanças naquele objeto.
  - **Clonar**: copiar todos os elementos da lista

```
x=[1,2,3] e y=x[:]
    ⟹ x=[1,2,3] e y=[1,2,3]
x.append(4) e y.append(5)
    ⟹ x=[1,2,3,4] e y=[1,2,3,5]
```



Strings VS tuplas vs Listas

- Tratam-se de estruturas do tipo sequências, onde podemos:
  - retornar o i-ésimo elemento: seq[i];
  - retornar o tamanho da sequência: len(seq);
  - concatenar duas sequências: seq1 + seq2;
  - repetir n vezes uma sequência: n * seq;
  - fatiar a sequência: seq[start:end];
  - avaliar(True/False), se há objeto na sequência: e in seq;
  - iterar sobre elementos da sequência: for e in seq i

Strings VS tuplas vs Listas

Type	Type of elements	Examples of literals	Mutable
`str`	characters	`''`, `'a'`, `'abc'`	No
`tuple`	any type	`()`, `(3,)`, `('abc', 4)`	No
`list`	any type	`[]`, `[3]`, `['abc', 4]`	Yes

- Listas costumam ser mais empregadas que tuplas já que são mutáveis.
- Por outro lado, uma vez que tuplas são imutáveis, evitam problemas com uso de alias

Strings VS tuplas vs Listas

Type	Type of elements	Examples of literals	Mutable
`str`	characters	`''`, `'a'`, `'abc'`	No
`tuple`	any type	`()`, `(3,)`, `('abc', 4)`	No
`list`	any type	`[]`, `[3]`, `['abc', 4]`	Yes

- Listas costumam ser mais empregadas que tuplas já que são mutáveis.
- Por outro lado, uma vez que tuplas são imutáveis, evitam problemas com uso de alias

Dicionários

- Coleção não ordenada de valores que são referenciados pelas chaves correspondentes.

  al_D={‘joao’:,5.0, ‘maria’:10.0,’pedro’:3.0}

- Trata-se de uma outro tipo de dado assim como listas e tuplas.

Dicionários

- Notação

  ```
  <nome-dicionario> = {chave1 : valor1, chave2 :
  valor2,...,chave_n : valor_n}
  ```

- Apenas valores imutáveis podem ser usados como chaves: strings, números e tuplas.

Dicionários

- Vantagens
  - Acessa itens de interesse diretamente
  - Permite usar uma única estrutura de dados ao invés de listas separadas.

Índice	Valor
0	joao
1	maria
2	pedro

Índice	Valor
0	5
1	10
2	3

Chave	Valor
joao	5
maria	10
pedro	3

Dicionários

- Criando dicionário

```
al_D = {} (Vazio)
al_D={'joao':5.0,'maria':10,'pedro':3.0}
```

- Acessando dados

```
print(al_D[‘joao’])  ⇒ 5
print(al_D[‘antonio’]) ⇒ KeyERROR
```

- Alterando e acrescentando dados

```
al_D[‘joao’] =8
al_D[‘ana’]=9.5
```

Chave	Valor
joao	5
maria	10
pedro	3

Chave	Valor
joao	**8**
maria	10
pedro	3
ana	**9.5**

Dicionários

Chave	Valor
maria	10
pedro	3
ana	**9.5**

- Removendo
  ```
  del(al_D['joao'])
  ```

- Avaliando se está no dicionário
  ```
  print('maria' in al_D) ⇒ True
  print('joao' in al_D) ⇒ Falso
  ```

- Não há ordem para chaves e valores num dicionário
  ```
  qualquer_D={1:'janeiro','joao':80.5,(1,2,3):'sequencia'}
  ```

Dicionários

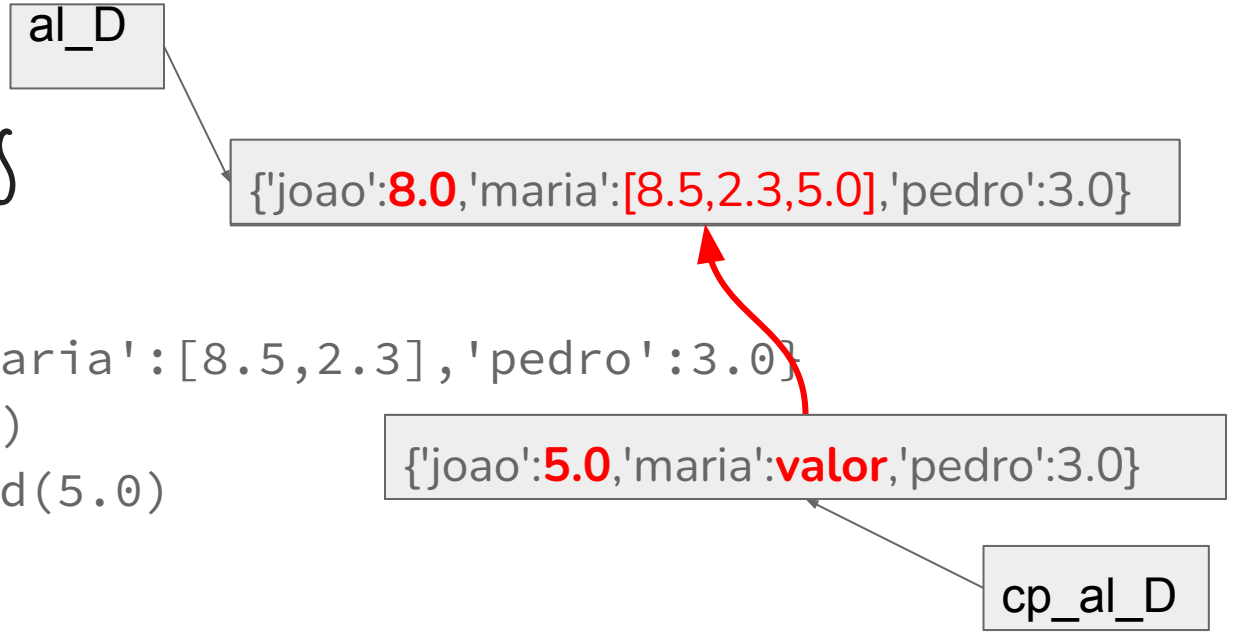
- Retornando as chaves em uma lista
  list(al_D.**keys())** ⇒
  [‘’maria’,’pedro’,’ana’]
  A ordem original não é garantida

- Retornando os valores em uma lista
  list(al_D.values())⇒ [10,3,9.5’]

- Vários métodos para dicionário - PESQUISAR!!!
  copy(),clear(), get(),has_key(), etc..

Chave	Valor
maria	10
pedro	3
ana	**9.5**

Dicionários

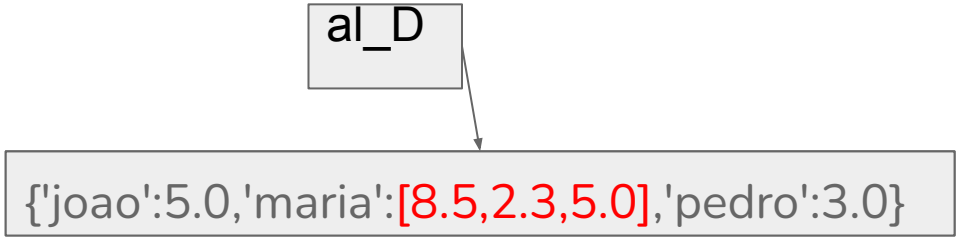


- copy()

```
al_D={'joao':5.0,'maria':[8.5,2.3],'pedro':3.0}
cp_al_D= al_D.copy()
al_D['maria'].append(5.0)
al_D['joao']=8.0
```

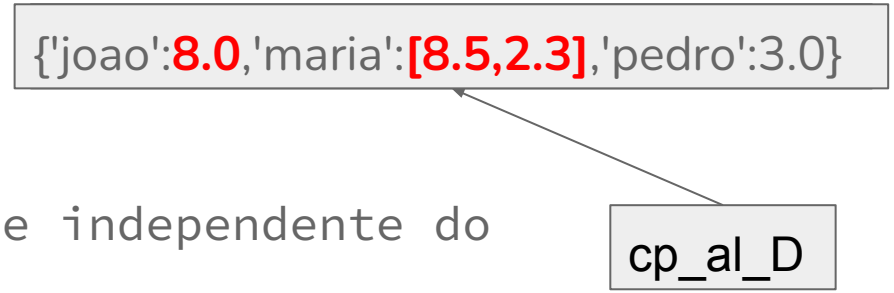
- copy() não cria uma cópia completamente independente do dicionário original
- A cópia é independente apenas para valores numéricos, literais ou imutáveis.
- A cópia não é independente para listas!!

Dicionários



- deepCopy()

```
from copy import deepCopy
al_D={'joao':5.0,'maria':[8.5,2.3],'pedro':3.0}
cp_al_D= deepcopy(al_D)
al_D['maria'].append(5.0)
cp_al_D['joao']=8.0
```



  - Cria uma cópia completamente independente do dicionário original
  - Necessidade de importar deepCopy
  - deepcopy() é função
  - copy() é método..

Exception

```
1  L=[1,2,3]
2  print(f'{L[3]=}')
3
```

```
---------------------------------------------------------
IndexError                                Traceback
<ipython-input-1-00e491ad74c6> in <cell line: 2>()
      1 L=[1,2,3]
----> 2 print(f'{L[3]=}')

IndexError: list index out of range
```

```
1  L=[]
2  print(f'{10/len(L)}')
3
```

```
---------------------------------------------------------
ZeroDivisionError                         Traceback
<ipython-input-3-c8509ceec060> in <cell line: 2>()
      1 L=[]
----> 2 print(f'{10/len(L)}')

ZeroDivisionError: division by zero
```

```
1  print('x'/10)
2
```

```
---------------------------------------------------------------
TypeError                                 Traceback (most rece
<ipython-input-4-da56615c7243> in <cell line: 1>()
----> 1 print('x'/10)

TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

Exception

- Indicam a ocorrência de um evento inesperado durante a execução do código.
- Tratam eventos cuja ocorrência não seja frequente mas que comprometem a execução do código.
- Assim, permitem aos programadores tratarem tais eventos de forma adequada.
- Os eventos podem estar relacionados a erros ou situações inesperadas.

Exception

- Formas de lidar com eventos inesperados:
  - try/exception
  - try/exception/else
  - try/finally
- try : habilita o tratamento da exceção ao conter o bloco de instruções que podem causar exceções.
- exception : determina possíveis causas para as exceções, podendo especificar um identificador para aquele tipo de erro.
- else : contém instruções a serem executadas, caso nenhuma exceção seja detectada no bloco try.
- finally : Um bloco finally define instruções a serem executadas independente da ocorrência ou não de exceções.

Raise Exception

- Permite ao desenvolvedor definir uma exceção
- O programa termina a execução:

```
raise <nome da exceção> (<argumentos>)
```

```
raise ValueError('deu ruim')
```

- O programa não termina a execução:

```
try: ....
    raise ValueError
     ....
except ValueError:
        print('deu ruim')
```

Assertion

- Utilizado para avaliar se determinada condição é necessária para a execução do código.
- Permite testar se tal condição é verdadeira. Se for falsa, dispara `AssertionError`
- No caso da condição ser falsa, uma mensagem de erro pode ser retornada.

```
assert <condição> , <mensagem>
```