

Listas

Listas são cadeias de valores, isto é, armazenam mais de um valor. São estruturas ordenadas, mutáveis e heterogêneas. São ordenadas, pois cada valor tem seu índice, na ordem que estão armazenados. São mutáveis, pois é possível alterar os valores. E são heterogêneas, pois os valores podem ser de tipos diferentes. Podem ser atribuídas usando colchetes:

```
1 #declara variável e imprime a lista entre colchetes:
2 x = [10, 5, 11, 0, 3]
3 x
4
```

```
[10, 5, 11, 0, 3]
```

```
1 #imprime o elemento dentro da lista:
2 x[0]
```

```
10
```

```
1 #imprime o elemento dentro da lista:
2 x[4]
```

```
3
```

```
1 L = [1, 4.5, 'abc', 3+ 4j, [1, 2, 3, 4]]
2 L[2] = 5 #substitui elemento descrito na lista L
3 L[4] = 5 #substitui elemento descrito na lista L
4 L
```

```
[1, 4.5, 5, (3+4j), 5]
```

```
1 #Adicionar novos valores com +
2 A = [1, 2, 3]; B = [4, 5, 6]
3 C = A + B
4 C
```

```
[1, 2, 3, 4, 5, 6]
```

```
1 #adicionar novos valores com +=
2 C += [7]
3 C
```

```
[1, 2, 3, 4, 5, 6, 7]
```

```
1 #Adicionar novos valores com .append()
2 C.append('caio')
3 C
```

```
[1, 2, 3, 4, 5, 6, 7, 'caio']
```

```
1 #Incluindo outras listas no índice:
2 L = [1, 4.5, 'abc', 3 + 4j, [1, 2, 3, 4]]
3 L
```

```
[1, 4.5, 'abc', (3+4j), [1, 2, 3, 4]]
```

```
1 #imprimindo lista dentro da lista:
2 L[4]
```

```
[1, 2, 3, 4]
```

```
1 #imprimindo índice dentro da lista que está dentro da lista:
2 L[4][0]
```

```
1
```

```
1 #para imprimir atributos e métodos das listas:
2 dir(list)
```

```
[ '__add__',
  '__class__',
  '__class_getitem__',
  '__contains__',
  '__delattr__',
  '__delitem__',
  '__dir__',
```

```

'__doc__',
'__eq__',
'__format__',
'__ge__',
'__getattr__',
'__getitem__',
'__getstate__',
'__gt__',
'__hash__',
'__iadd__',
'__imul__',
'__init__',
'__init_subclass__',
'__iter__',
'__le__',
'__len__',
'__lt__',
'__mul__',
'__ne__',
'__new__',
'__reduce__',
'__reduce_ex__',
'__repr__',
'__reversed__',
'__rmul__',
'__setattr__',
'__setitem__',
'__sizeof__',
'__str__',
'__subclasshook__',
'append',
'clear',
'copy',
'count',
'extend',
'index',
'insert',
'pop',
'remove',
'reverse',
'sort']


```

1 # append() Adiciona elementos a lista:

```

2
3 x = [1, 2, 3]
4 x.append(4)
5 x
6

```

 [1, 2, 3, 4]

1 # clear() Esvazia a lista:

```

2
3 x = [1, 2, 3]
4 x.clear()
5 x

```

 []

1 #count() Retorna quantas ocorrências há do argumento passado:

```

2
3 x = [1, 2, 3, 3, 3, 3, 4]
4 x.count(3)

```

 4

1 # index() Retorna o índice da primeira ocorrência do argumento passado:

```

2
3 x = [1, 2, 3, 3, 3, 3, 4]
4 x.index(3)

```


 2

1 # insert() Insere na posição especificada um elemento:

```

2
3 x = ['a', 'c', 'd']
4 x.insert(1, 'b')
5 x

```

 ['a', 'b', 'c', 'd']

```

1 # pop() Remove e retorna o elemento da posição especificada:
2
3 x = ['a', 'b', 'c', 'd']
4 x.pop(1)
5 x

```

```
↵ ['a', 'c', 'd']
```

```

1 # remove() Remove o elemento especificado:
2
3 x = ['a', 'b', 'c']
4 x.remove('c')
5 x

```

```
↵ ['a', 'b']
```

```

1 # reverse() Inverte a ordem dos elementos:
2
3 x = [5, 4, 3, 2, 1]
4 x.reverse()
5 x

```

```
↵ [1, 2, 3, 4, 5]
```

```

1 # sort() Ordena:
2
3 x = [10, 3, 3, 10, 10, 6, 6, 4, 8, 7]
4 x.sort()
5 x

```

```
↵ [3, 3, 4, 6, 6, 7, 8, 10, 10, 10]
```

Laço for

Os conjuntos finitos de valores usados na estrutura de repetição for para iterar uma variável, também é uma lista.

```

1 # retorna os elementos da lista x
2 x = [10, 5, 11, 0, 3]
3 for i in x:
4     print(i)

```

```
↵ 10
5
11
0
3
```

```

1 # se o valor i estiver presente retorna True
2 i = 10
3 i in x

```

```
↵ True
```

```

1 # se o valor i não estiver presente retorna False
2 i = 100
3 i in x

```

```
↵ False
```

```

1 # Cria uma nova lista a partir de A somando-se 1 a cada elemento
2 A=[3, 4, 5, 10]
3 somal = list ()
4 for i in A:
5     somal += [i+1]
6 somal

```

```
↵ [4, 5, 6, 11]
```

```

1 #Forma compacta de criar uma nova lista a partir de A somando-se 1 a cada elemento
2 A=[3, 4, 5, 10]
3 somal = [i+1 for i in A]
4 somal

```

```
↵ [4, 5, 6, 11]
```

```

1 #Atribui a N os valores de A ao quadrado, mas apenas os que são pares.
2 A = [0, 0, 5, 3, 2, 7, 4, 10, 4, 7]

```

```

3 N = list()
4 for i in A:
5     if i%2 == 0:
6         N += [i*i]
7 N
8

```

```
↵ [0, 0, 4, 16, 100, 16]
```

```

1 #Forma compacta de atribui a N os valores pares de A ao quadrado
2 A = [0, 0, 5, 3, 2, 7, 4, 10, 4, 7]
3 N = [i*i for i in A if i%2 ==0]
4 N

```

```
↵ [0, 0, 4, 16, 100, 16]
```

Fatiamento

É possível acessar partes da lista por meio de fatiamentos. Para isso, há três parâmetros, separados por dois pontos:

```

1 # selecionado do elemento 0 até o elemento 1, de 1 em 1
2 X = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
3 X [0:2:1]

```

```
↵ [1, 2]
```

```

1 #selecionado do 0 até o 3 de 1 em 1
2 X [0:4:1]

```

```
↵ [1, 2, 3, 4]
```

```

1 # selecionado do 0 até o 3 de 2 em 2
2 X [0:4:2]

```

```
↵ [1, 3]
```

```

1 #passo negativo lista no sentido inverso
2 X = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
3 X[::-1]

```

```
↵ [0, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

```

1 # selecionando a partir do elemento 4 e contada de 1 em 1 ao elemento 0.
2 X[4::-1]

```

```
↵ [5, 4, 3, 2, 1]
```

```

1 # X foi fatiado primeiro de 0 a 2, com passo 1
2 X = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
3 X [:2:1]

```

```
↵ [1, 2]
```

```

1 # X fatiado de 0 a 4, com passo 1
2 X[:4]

```

```
↵ [1, 2, 3, 4]
```

```

1 # a lista toda com passo 2
2 X [::2]

```

```
↵ [1, 3, 5, 7, 9]
```

```

1 # A lista toda
2 X[:]

```

```
↵ [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
```

Tuplas

As tuplas são similares as listas, pois são **ordenadas** e **heterogêneas**. No entanto, diferentemente das listas, as tuplas são **imutáveis**. Este fato faz com que as tuplas sejam mais compactas e eficazes em termos de memória e eficiência. A sua atribuição é feita por meio de parênteses:

```

1 #mostra o nome da class
2 x = ('a', 1, 2)
3 type(x)
4
5
6

```

→ tuple

```

1 #mostra os elementos da Tupla
2 x

```

→ ('a', 1, 2)

DICA: Tuplas vazias podem ser criadas usando-se uma das duas sintaxes a seguir:

```
x = ()
```

```
x = tuple()
```

No entanto, como as tuplas são imutáveis, esse procedimento não é comum. As tuplas são imutáveis, no entanto, como são heterogêneas, se em uma tupla houver listas, os item das listas podem ser alterados normalmente, pois, apesar da tupla, as listas são mutáveis:

```

1 x = ([0, 1, 2], 2, 2)
2 x[1]

```

→ 2

```

1 #mostrando indicie que é uma lista dentro da tupla
2 x[0]

```

→ [0, 1, 2]

```

1 #alterando elemento dentro da lista que está na tupla
2 x [0][0] = 'mudei'
3 x

```

→ (['mudei', 1, 2], 2, 2)

```

1 #operadores aritméticos e relacionais se aplicam as tuplas
2 X = (1, 2, 3)
3 Y = (1, 2, 4)
4 Z = (4, 0, 0)
5 X < Y

```

→ True

```
1 X < Z
```

→ True

```
1 X + Y
```

→ (1, 2, 3, 1, 2, 4)

```
1 Z * 4
```

→ (4, 0, 0, 4, 0, 0, 4, 0, 0, 4, 0, 0)

```

1 X += (1,)
2 X

```

→ (1, 2, 3, 4, 4, 5, 2, 4, 4, 9, 1, 1, 1)

Funções

O comando `dir(tuple)` retorna os atributos e métodos das tuplas.

```
1 dir(tuple)
```

→ ['__add__',
 '__class__',
 '__class_getitem__',
 '__contains__',
 '__delattr__',
 '__dir__',

```

'__doc__',
'__eq__',
'__format__',
'__ge__',
'__getattr__',
'__getitem__',
'__getnewargs__',
'__getstate__',
'__gt__',
'__hash__',
'__init__',
'__init_subclass__',
'__iter__',
'__le__',
'__len__',
'__lt__',
'__mul__',
'__ne__',
'__new__',
'__reduce__',
'__reduce_ex__',
'__repr__',
'__rmul__',
'__setattr__',
'__sizeof__',
'__str__',
'__subclasshook__',
'count',
'index']

```

Algumas funções(métodos) da estrutura composta tupla e o que fazem:

```

1 #count() Retorna quantas ocorrências há do argumento passado:
2 x = (1, 2, 3, 3, 3, 3, 4)
3 x.count (3)

```

↩ 4

```

1 #index() Retorna o índice da primeira ocorrência do argumento passado:
2 x = (1, 2, 3, 3, 3, 3, 4)
3 x.index (4)

```

↩ 6

Dicionários

Listas e tuplas são indexadas pela ordem dos elementos, isto é, são **ordenadas**. Já os dicionários, não são. Ao invés disso, nesta estrutura, cada elemento recebe uma etiqueta. Além disso, são **mutáveis** e **heterogêneos**. Esta estrutura é a mais poderosa do Python, permitindo manipulações complexas.

Sua atribuição é feita usando-se chaves e dando etiquetas aos valores:

```

1 X = {'idade': 21, 'matricula': 10000 , 'nome': 'Joao'}
2 X

```

↩ {'idade': 21, 'matricula': 10000, 'nome': 'Joao'}

```

1 #como não ordenados, usa-se suas etiquetas para acessar elementos
2 X = {'idade': 21, 'matricula': 10000 , 'nome': 'Joao'}
3 X['matricula']

```

↩ 10000

DICA: Dicionários vazios podem ser criados usando-se uma das duas sintaxes a seguir:

```
x = {}
```

```
x = dict()
```

Funções

O comando `dir(dict)` retorna os atributos e métodos dos dicionários.

```
1 dir(dict)
```

↩ ['__class__',
'__class_getitem__',
'__contains__',
'__delattr__',
'__delitem__',
'__dir__',

```

'__doc__',
'__eq__',
'__format__',
'__ge__',
'__getattr__',
'__getitem__',
'__getstate__',
'__gt__',
'__hash__',
'__init__',
'__init_subclass__',
'__ior__',
'__iter__',
'__le__',
'__len__',
'__lt__',
'__ne__',
'__new__',
'__or__',
'__reduce__',
'__reduce_ex__',
'__repr__',
'__reversed__',
'__ror__',
'__setattr__',
'__setitem__',
'__sizeof__',
'__str__',
'__subclasshook__',
'clear',
'copy',
'fromkeys',
'get',
'items',
'keys',
'pop',
'popitem',
'setdefault',
'update',
'values']

```

Veja algumas funções(métodos) da estrutura composta dicionário e o que fazem:

```

1 #clear() Esvazia o dicionário:
2 d = {'nome': 'joao', 'idade': 20}
3 d.clear ()
4 d

```

↔ {}

```

1 # get() Retorna o valor da chave passada como argumento. Caso haja um segundo
2 # argumento, será a resposta default, isto é, caso a chave não exista no
3 # dicionário, irá retornar o default.
4 d = {'nome': 'joao', 'idade': 20}
5 d.get('idade')
6
7

```

↔ 20

```

1 d.get('teste')
2 d.get('teste', 'não encontrado')

```

↔ 'não encontrado'

Laço for no Dicionário

Dicionários também podem ser usados para iterar variáveis no laço for. Caso seja utilizado apenas uma variável para ser iterada, a variável irá receber as chaves.

```

1 d = {'nome ': 'Joao ', 'idade ': 20, 'matricula ': 1}
2 for i in d:
3     print(i)

```

↔ nome
idade
matricula

Caso sejam utilizadas duas variáveis e usado o método items, a primeira receberá a chave e a segunda o valor:

```
1 d = {'nome ': 'Caio ', 'idade ': 24, 'matricula ': 76}
2 for i,c in d.items():
3     print('i = ', i, 'é = ', c)
```

```
↵ i = nome  é = Caio
   i = idade  é = 24
   i = matricula  é = 76
```

Strings As strings foram abordadas anteriormente como tipos primitivos de dados, mas são também cadeias de caracteres. Portanto, podem ser pensadas como uma estrutura composta. São sequências **ordenadas**, **imutáveis** e **homogêneas**. Ordenadas, pois cada elemento, isto é, cada letra possui seu índice, na ordem que estão armazenados. Imutáveis, pois não podem ser alteradas. Homogêneas, pois só aceitam um tipo de dado.

```
1 P = 'Palavra'
2 P[0]
```

```
↵ 'P'
```

```
1 P[-1]
```

```
↵ 'a'
```

```
1 P[5]
```

```
↵ 'r'
```

Também podem ser usadas para iterações em laços do tipo for.

```
1 #será impresso letra por letra de P, "Palavra".
2 P = 'Palavra'
3 for i in P:
4     print(i)
```

```
↵ P
   a
   l
   a
   v
   r
   a
```

O operador "+", ao ser usado em Strings, realiza concatenação, assim como em listas. No entanto, outros operadores também podem ser úteis:

```
1 A = 'abc'; B = 'def'
2 A + B
```

```
↵ 'abcdef'
```

```
1 A < B
```

```
↵ True
```

```
1 ('HA-' * 10 + 'He')
```

```
↵ 'HA-HA-HA-HA-HA-HA-HA-HA-HA-HA-He'
```


Tabela 7.4.1: Operadores em Strings

Tipo	Operador	Descrição
Aritmético	+	Concatenação
Aritmético	*	Múltiplas concatenações
Relacional	==	Igual a
Relacional	!=	Diferente de
Relacional	<>	Diferente de
Relacional	>	Maior(alfabeticamente) que
Relacional	>=	Maior(alfabeticamente) ou igual a
Relacional	<	Menor(alfabeticamente) que
Relacional	<=	Menor(alfabeticamente) ou igual a

DICA: Os operadores relacionais, nos casos de menor ou maior comparam se uma string é alfabeticamente menor ou maior, isto é, se na ordem alfabética vem antes ou depois da outra.

Funções

O comando `dir(str)` retorna os atributos e métodos das listas.

```
1 dir(str)
['__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__rmod__',
 '__rmul__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'capitalize',
 'casefold',
 'center',
 'count',
 'encode',
 'endswith',
 'expandtabs',
 'find',
 'format',
 'format_map',
 'index',
 'isalnum',
 'isalpha',
 'isascii',
 'isdecimal',
 'isdigit',
 'isidentifier',
```

```
strip',  
'swapcase',  
'title',  
'translate',  
'upper',  
'zfill']
```

Veja algumas funções (métodos) da estrutura composta string e o que fazem:

```
1 #capitalize() Retorna a string com  
2 #o primeiro caractere em maiúsculo e todos os caracteres em minúsculo:  
3 F = 'veja esta frase'  
4 F.capitalize ()
```

```
↔ 'Veja esta frase'
```

```
1 #count() Retorna quantas ocorrências há do argumento passado:  
2 a = 'palavra'  
3 a.count('a')
```

```
↔ 3
```

```
1 #index() Retorna o índice da primeira ocorrência do argumento passado:  
2 a = 'palavra'  
3 a.index('a')
```

```
↔ 1
```

```
1 #isalnum() Retorna verdadeiro  
2 #se todos os caracteres da frase forem alfanuméricos:  
3 A = 'abc11+'  
4 B = 'abc11'  
5 A.isalnum ()
```

```
↔ False
```

```
1 B.isalnum()
```

```
↔ True
```

```
1 #isalpha() Retorna verdadeiro  
2 #se todos os caracteres da frase forem alfabéticos:  
3 A = 'abc11'  
4 B = 'abc'  
5 A.isalpha ()
```

```
↔ False
```

```
1 B.isalpha ()
```

```
↔ True
```

```
1 #isdecimal() Retorna verdadeiro  
2 #se todos os caracteres da frase representarem um número decimal:  
3 A = '10.9'  
4 B = '5'  
5 A.isdecimal ()
```

```
↔ False
```

```
1 B.isdecimal()
```

```
↔ True
```

```
1 #islower() Retorna verdadeiro se todos os caracteres da frase forem minúsculos:  
2 A = 'abCD'  
3 B = 'abcd'  
4 A.islower ()
```

```
↔ False
```

```
1 B.islower ()
```

```
↔ True
```

```
1 #isnumeric() Retorna verdadeiro
2 #se todos os caracteres da frase forem numéricos:
3 A = 'ab12'
4 B = '1234'
5 A. isnumeric ()
```

False

```
1 B. isnumeric ()
```

True

```
1 #isupper() Retorna verdadeiro se todos os caracteres da frase forem minúsculos:
2 A = 'abCD'
3 B = 'ABCD'
4 A. isupper ()
```

False

```
1 B.isupper()
```

True

```
1 #lower() Retorna a string com todos os caracteres em minúsculo:
2 F = 'UMA FRASE'
3 F.lower ()
```

'uma frase'

```
1 #replace() Retorna o string trocando a primeira string do argumento pela segunda:
2 F = 'Veja uma frase'
3 F. replace (' ', '-')
```

'Veia-uma-frase'

```
1 F. replace ('uma', 'alguma')
```

'Veia alguma frase'

```
1 # split() Retorna a string separada de acordo com o separador, passado com
2 # argumento. Por padrão o separador é ' ':
3 F = 'veja uma frase'
4 F.split ()
```

['veja', 'uma', 'frase']

```
1 F.split('a')
```

['vej', ' um', ' fr', 'se']

```
1 #upper() Retorna a string com todos os caracteres em maiúsculo:
2 F = 'uma frase'
3 F.upper ()
```

'UMA FRASE'

Exercícios 1

Crie um Python Script que conte quantas vezes cada nome está presente em uma lista ['nome1', 'nome2', ...] e armazena essa contagem em um dicionário {'nome1': xvezes, 'nome2': yvezes,}.

```
1 nomes = ['Ana', 'João', 'Maria', 'Ana', 'João', 'Pedro', 'Maria', 'Maria']
2 contagem = {}
3
4 for nome in nomes:
5     contagem[nome] = contagem.get(nome, 0) + 1
6
7 print(contagem)
```

{'Ana': 2, 'João': 2, 'Maria': 3, 'Pedro': 1}

Exercício 2 Crie um Python Script que realize o mesmo procedimento da questão anterior. No entanto, ao invés do conteúdo da lista nomes ser atribuído no próprio script, faça uma estrutura de repetição na qual ela leia uma string do usuário e adicione os nomes digitados por ele, um de cada vez, na lista nomes. O término da adição de nomes deve ser indicado quando o usuário inserir uma string vazia ("").

```

1 nomes = []
2 while True:
3     nome = input("Digite um nome (ou pressione Enter para sair): ")
4     if nome == "":
5         break
6     nomes.append(nome)
7
8 contagem = {}
9 for nome in nomes:
10     contagem[nome] = contagem.get(nome, 0) + 1
11
12 print(contagem)

```

```

↵ Digite um nome (ou pressione Enter para sair): caio
  Digite um nome (ou pressione Enter para sair): fernanda
  Digite um nome (ou pressione Enter para sair):
  {'caio': 1, 'fernanda': 1}

```

Exercício 3 Crie um programa que lê uma mensagem do usuário. Com esta mensagem, faça uma nova omitindo trocando todos os caracteres de nomes próprios por ''. *Exemplo: se a mensagem for 'Lucas foi ao shopping com Fernanda assistir aquele filme da Marvel', a nova mensagem deverá ser '****foi ao shopping com **** assistir aquele filme da **'*. Assuma que um nome próprio sempre começa com letra maiúscula e contém apenas letras.

```

1 mensagem = input("Digite uma frase: ")
2 palavras = mensagem.split()
3 nova_mensagem = []
4
5 for palavra in palavras:
6     if palavra[0].isupper() and palavra.isalpha():
7         nova_mensagem.append('*' * len(palavra))
8     else:
9         nova_mensagem.append(palavra)
10
11 print(" ".join(nova_mensagem))

```

```

↵ Digite uma frase: Fernanda e Caio são lindos
  ***** e **** são lindos

```

Exercício 4 Faça um programa que leia seis valores numéricos atribuindo-os à duas variáveis do tipo lista com três elementos cada. Cada variável irá representar um vetor, informe o produto escalar e o produto vetorial destes vetores.

```

1 import numpy as np
2
3 vetor1 = [int(input(f"Digite o {i+1}º número do primeiro vetor: ")) for i in range(3)]
4 vetor2 = [int(input(f"Digite o {i+1}º número do segundo vetor: ")) for i in range(3)]
5
6 produto_escalar = np.dot(vetor1, vetor2)
7 produto_vetorial = np.cross(vetor1, vetor2)
8
9 print(f"Produto Escalar: {produto_escalar}")
10 print(f"Produto Vetorial: {produto_vetorial}")

```

```

↵ Digite o 1º número do primeiro vetor: 2
  Digite o 2º número do primeiro vetor: 5
  Digite o 3º número do primeiro vetor: 6
  Digite o 1º número do segundo vetor: 2
  Digite o 2º número do segundo vetor: 5
  Digite o 3º número do segundo vetor: 6
  Produto Escalar: 65
  Produto Vetorial: [0 0 0]

```

Exercício 5 Escreva um programa que leia um número N. Em sequência, ele deve ser N números e armazená-los em uma lista.

```

1 N = int(input("Quantos números deseja inserir? "))
2 numeros = [int(input(f"Digite o {i+1}º número: ")) for i in range(N)]
3 print("Lista de números:", numeros)

```


```

↵ Quantos números deseja inserir? 5
  Digite o 1º número: 2
  Digite o 2º número: 3
  Digite o 3º número: 4
  Digite o 4º número: 5
  Digite o 5º número: 6
  Lista de números: [2, 3, 4, 5, 6]

```

Exercício 6 Com o programa acima, imprima: A soma dos itens, a média dos valores, o maior e o menor valor.

```
1 soma = sum(numeros)
2 media = soma / len(numeros)
3 maior = max(numeros)
4 menor = min(numeros)
5
6 print(f"Soma: {soma}")
7 print(f"Média: {media}")
8 print(f"Maior número: {maior}")
9 print(f"Menor número: {menor}")
```

 Soma: 20
Média: 4.0