Curso livre de programação Linguagem Python

Encontro 4 - Vetores Parte 1

Prof. Louis Augusto
louis.augusto@ifsc.edu.br

Instituto Federal de Santa Catarina
Campus São José

Índice

Sumário

O que é um vetor

Um vetor é uma sequência de dados ocupando posições consecutivas de memória.

Exemplo de um vetor com 7 posições

0	1	2	3	4	5	6
12	15	65	-12	“casa azul”	True	45968

- Note que o vetor inicia na posição 0, como na maioria das linguagens de programação!
- Um vetor pode ser de qualquer tipo: números inteiros, reais, strings etc.
- Em Python, um vetor (lista ou tuple) pode conter tipos diferentes. Por exemplo, nas quatro primeiras posições é possível armazenar valores inteiros e nas três últimas é possível que os valores sejam inteiros, flutuante, strings ou mesmo valores booleanos.
- É possível remover termos, inserir termos, juntar vetores, inverter etc. Várias operações possíveis.

O que é um vetor

Um vetor é uma sequência de dados ocupando posições consecutivas de memória.

Exemplo de um vetor com 7 posições

0	1	2	3	4	5	6
12	15	65	-12	“casa azul”	True	45968

- Note que o vetor inicia na posição 0, como na maioria das linguagens de programação!
- Um vetor pode ser de qualquer tipo: números inteiros, reais, strings etc.
- Em Python, um vetor (lista ou tupla) pode conter tipos diferentes. Por exemplo, nas quatro primeiras posições é possível armazenar valores inteiros e nas três últimas é possível que os valores sejam em ponto flutuante, inteiros, strings ou mesmo valores booleanos.
- É possível remover termos, incluir termos, juntar vetores diferentes. Há várias operações possíveis.

O que é um vetor

Um vetor é uma sequência de dados ocupando posições consecutivas de memória.

Exemplo de um vetor com 7 posições

0	1	2	3	4	5	6
12	15	65	-12	“casa azul”	True	45968

- Note que o vetor inicia na posição 0, como na maioria das linguagens de programação!
- Um vetor pode ser de qualquer tipo: números inteiros, reais, strings etc.
- Em Python, um vetor (lista ou tupla) pode conter tipos diferentes. Por exemplo, nas quatro primeiras posições é possível armazenar valores inteiros e nas três últimas é possível que os valores sejam em ponto flutuante, inteiros, strings ou mesmo valores booleanos.
- É possível remover termos, incluir termos, juntar vetores diferentes. Há várias operações possíveis.

O que é um vetor

Um vetor é uma sequência de dados ocupando posições consecutivas de memória.

Exemplo de um vetor com 7 posições

0	1	2	3	4	5	6
12	15	65	-12	“casa azul”	True	45968

- Note que o vetor inicia na posição 0, como na maioria das linguagens de programação!
- Um vetor pode ser de qualquer tipo: números inteiros, reais, strings etc.
- Em Python, um vetor (lista ou tupla) pode conter tipos diferentes. Por exemplo, nas quatro primeiras posições é possível armazenar valores inteiros e nas três últimas é possível que os valores sejam em ponto flutuante, inteiros, strings ou mesmo valores booleanos.
- É possível remover termos, incluir termos, juntar vetores diferentes. Há várias operações possíveis.

O que é um vetor

Um vetor é uma sequência de dados ocupando posições consecutivas de memória.

Exemplo de um vetor com 7 posições

0	1	2	3	4	5	6
12	15	65	-12	“casa azul”	True	45968

- Note que o vetor inicia na posição 0, como na maioria das linguagens de programação!
- Um vetor pode ser de qualquer tipo: números inteiros, reais, strings etc.
- Em Python, um vetor (lista ou tupla) pode conter tipos diferentes. Por exemplo, nas quatro primeiras posições é possível armazenar valores inteiros e nas três últimas é possível que os valores sejam em ponto flutuante, inteiros, strings ou mesmo valores booleanos.
- É possível remover termos, incluir termos, juntar vetores diferentes. Há várias operações possíveis.

O que é um vetor

Um vetor é uma sequência de dados ocupando posições consecutivas de memória.

Exemplo de um vetor com 7 posições

0	1	2	3	4	5	6
12	15	65	-12	“casa azul”	True	45968

- Note que o vetor inicia na posição 0, como na maioria das linguagens de programação!
- Um vetor pode ser de qualquer tipo: números inteiros, reais, strings etc.
- Em Python, um vetor (lista ou tupla) pode conter tipos diferentes. Por exemplo, nas quatro primeiras posições é possível armazenar valores inteiros e nas três últimas é possível que os valores sejam em ponto flutuante, inteiros, strings ou mesmo valores booleanos.
- É possível remover termos, incluir termos, juntar vetores diferentes. Há várias operações possíveis.

Quais os tipos de vetores existentes

Há três tipos de vetores em Python:
Vetores não tipados:

- Lista - lista mutável. Os elementos da lista podem ser alterados, inclusive em relação ao tipo. É definida com colchetes. `lA = [1,2]`
- Tupla - lista imutável. Uma vez criada a lista ela não pode ser alterada. É definida com parênteses. `tA = (1,2)`

e vetores tipados:

- Array - lista mutável de valores tipados. É definida com parênteses e colchetes. `numeros = array('i', [1,2,3])`, em que `i` é uma referência para número inteiro, e logo a seguir vem a lista somente com inteiros. Precisa ser chamado o módulo array:

```
from array import array
numeros = array('i', [1,2,3])
```

A vantagem do array sobre a lista é o tempo de processamento, e deve sempre ser usada quando se trabalhar com um único tipo de dado. Depois de iniciado, todos os métodos aplicáveis a lista são aplicáveis a array.

Quais os tipos de vetores existentes

Há três tipos de vetores em Python:
Vetores não tipados:

- Lista - lista mutável. Os elementos da lista podem ser alterados, inclusive em relação ao tipo. É definida com colchetes. `lA = [1,2]`
- Tupla - lista imutável. Uma vez criada a lista ela não pode ser alterada. É definida com parênteses. `tA = (1,2)`

e vetores tipados:

- Array - lista mutável de valores tipados. É definida com parênteses e colchetes. `numeros = array('i', [1,2,3])`, em que `i` é uma referência para número inteiro, e logo a seguir vem a lista somente com inteiros. Precisa ser chamado o módulo array:

```
from array import array
numeros = array('i', [1,2,3])
```

A vantagem do array sobre a lista é o tempo de processamento, e deve sempre ser usada quando se trabalhar com um único tipo de dado. Depois de iniciado, todos os métodos aplicáveis a lista são aplicáveis a array.

Quais os tipos de vetores existentes

Há três tipos de vetores em Python:
Vetores não tipados:

- Lista - lista mutável. Os elementos da lista podem ser alterados, inclusive em relação ao tipo. É definida com colchetes. `lA = [1,2]`
- Tupla - lista imutável. Uma vez criada a lista ela não pode ser alterada. É definida com parênteses. `tA = (1,2)`

e vetores tipados:

- Array - lista mutável de valores tipados. É definida com parênteses e colchetes. `numeros = array('i', [1,2,3])`, em que `i` é uma referência para número inteiro, e logo a seguir vem a lista somente com inteiros. Precisa ser chamado o módulo array:

```
from array import array
numeros = array('i', [1,2,3])
```

A vantagem do array sobre a lista é o tempo de processamento, e deve sempre ser usada quando se trabalhar com um único tipo de dado. Depois de iniciado, todos os métodos aplicáveis a lista são aplicáveis a array.

Criando listas

Criando listas em Python:

```
# -*- coding : utf -8 -*-
vA = [] # Vetor vazio e de tamanho 0
vB = [ None ] * 5 # Vetor vazio de tamanho 5
vC = [1 , 3.4 , "A" , " IFSC " ]  #vetor de
           # tamanho 4 e com tipos diferentes

 print ( vA ) # Imprime o vetor vA
 print ( vB ) # Imprime o vetor vB
 print ( vC ) # Imprime o vetor vC
```

Se quisermos encontrar o índice para a primeira posição de um determinado valor no vetor usamos a funçao `index()`.
Exp: pos = `vC.index(3.4)`. Se não houver o valor no vetor a função retorna `ValueError`.
Se quisermos saber quantas vezes um elemento de um vetor aparece usamos o comando `count()`: `vC.count(3.4)`

Criando listas

Criando listas em Python:

```
# -*- coding : utf -8 -*-
vA = [] # Vetor vazio e de tamanho 0
vB = [ None ] * 5 # Vetor vazio de tamanho 5
vC = [1 , 3.4 , "A" , " IFSC " ]  #vetor de
           # tamanho 4 e com tipos diferentes

 print ( vA ) # Imprime o vetor vA
 print ( vB ) # Imprime o vetor vB
 print ( vC ) # Imprime o vetor vC
```

Se quisermos encontrar o índice para a primeira posição de um determinado valor no vetor usamos a funçao `index()`.
Exp: pos = `vC.index(3.4)`. Se não houver o valor no vetor a função retorna `ValueError`.
Se quisermos saber quantas vezes um elemento de um vetor aparece usamos o comando `count()`: `vC.count(3.4)`

Criando listas

Criando listas em Python:

```
# -*- coding : utf -8 -*-
vA = [] # Vetor vazio e de tamanho 0
vB = [ None ] * 5 # Vetor vazio de tamanho 5
vC = [1 , 3.4 , "A" , " IFSC " ]  #vetor de
            # tamanho 4 e com tipos diferentes

 print ( vA ) # Imprime o vetor vA
 print ( vB ) # Imprime o vetor vB
 print ( vC ) # Imprime o vetor vC
```

Se quisermos encontrar o índice para a primeira posição de um determinado valor no vetor usamos a funçao `index()`.
Exp: pos = `vC.index(3.4)`. Se não houver o valor no vetor a função retorna `ValueError`.
Se quisermos saber quantas vezes um elemento de um vetor aparece usamos o comando `count()`: `vC.count(3.4)`

Criando listas

Criando listas em Python:

```
# -*- coding : utf -8 -*-
vA = [] # Vetor vazio e de tamanho 0
vB = [ None ] * 5 # Vetor vazio de tamanho 5
vC = [1 , 3.4 , "A" , " IFSC " ]  #vetor de
            # tamanho 4 e com tipos diferentes

 print ( vA ) # Imprime o vetor vA
 print ( vB ) # Imprime o vetor vB
 print ( vC ) # Imprime o vetor vC
```

Se quisermos encontrar o índice para a primeira posição de um determinado valor no vetor usamos a funçao `index()`.
Exp: pos = `vC.index(3.4)`. Se não houver o valor no vetor a função retorna `ValueError`.
Se quisermos saber quantas vezes um elemento de um vetor aparece usamos o comando `count()`: `vC.count(3.4)`

Criando tuplas

Criando tuplas em Python:

```
# -*- coding : utf -8 -*-

tA = () # Tupla vazia, inútil porque não pode
    #sofrer alteração
tB = (1 , 3.4 , "A" , " IFSC " )  #tupla de
    #tamanho 4 e com tipos diferentes
print ( tA ) # Imprime a tupla tA
print ( tB ) # Imprime a tupla tB
```

Para saber se um vetor está vazio, pode ser tupla ou lista, basta usar o comando if sobre o nome da variável.

```
valores = []
if valores:
    print(f"Os valores abaixo estão no vetor: {valores}")
else:
    print("Vetor vazio")
```

Criando tuplas

Criando tuplas em Python:

```
# -*- coding : utf -8 -*-

tA = () # Tupla vazia, inútil porque não pode
    #sofrer alteração
tB = (1 , 3.4 , "A" , " IFSC " )  #tupla de
    #tamanho 4 e com tipos diferentes
print ( tA ) # Imprime a tupla tA
print ( tB ) # Imprime a tupla tB
```

Para saber se um vetor está vazio, pode ser tupla ou lista, basta usar o comando if sobre o nome da variável.

```
valores = []
if valores:
    print(f"Os valores abaixo estão no vetor: {valores}")
else:
    print("Vetor vazio")
```

Criando tuplas

Criando tuplas em Python:

```
# -*- coding : utf -8 -*-

tA = () # Tupla vazia, inútil porque não pode
    #sofrer alteração
tB = (1 , 3.4 , "A" , " IFSC " )  #tupla de
    #tamanho 4 e com tipos diferentes
print ( tA ) # Imprime a tupla tA
print ( tB ) # Imprime a tupla tB
```

Para saber se um vetor está vazio, pode ser tupla ou lista, basta usar o comando if sobre o nome da variável.

```
valores = []
if valores:
    print(f"Os valores abaixo estão no vetor: {valores}")
else:
    print("Vetor vazio")
```

Criando arrays

Vimos que para trabalhar com arrays devemos chamar a biblioteca array, que é parte da biblioteca padrão do python.
Precisamos chamar inicialmente a bibilioteca `array`

```
from array import array
numeros = array('i', (1,2,3,2))
```

Para maiores informações: ▸ Link
Um array funciona utilizando os tipos numéricos definidos em `C`, a saber:

Código de tipo	Tipo em C	Tipo em Python
`'b'`	signed char	int
`'B'`	unsigned char	int
`'u'`	wchar_t	Caractere unicode
`'h'`	signed short	int
`'H'`	unsigned short	int
`'i'`	signed int	int

Código de tipo	Tipo em C	Tipo em Python
`'I'`	unsigned int	int
`'l'`	signed long	int
`'L'`	unsigned long	int
`'q'`	signed long long	int
`'Q'`	unsigned long long	int
`'f'`	float	float
`'d'`	double	float

Sumário

Buscas em listas

A biblioteca padrão do Python possui o comando **in**, que serve a dois propósitos:

- fazer uma busca num vetor.
- percorrer um vetor (conjuntamente com o comando **for**).

O primeiro propósito do comando *in* é realizar uma busca num vetor por um elemento em particular. O comando **in** tem como resposta um booleano, ou seja, é sempre *True* ou *False*.

Considere o código:

```
frutas = ['maçã', 'abacate', 'açaí', 'pêra']
print('maçã' in frutas)
print(not 'cajá' in frutas)#mesmo que print('cajá' not in frutas)
```

que têm como resposta: `True`

O mais comum de se fazer é colocar um elemento no vetor se não estiver contido:

```
if 'laranja' not in frutas:
    frutas.append("laranja")
print(frutas)
```

que tem como resposta:

```
['maçã', 'abacate', 'açaí',
 'pêra', 'laranja']
```

Buscas em listas

A biblioteca padrão do Python possui o comando **in**, que serve a dois propósitos:

- fazer uma busca num vetor.
- percorrer um vetor (conjuntamente com o comando **for**).

O primeiro propósito do comando *in* é realizar uma busca num vetor por um elemento em particular. O comando **in** tem como resposta um booleano, ou seja, é sempre *True* ou *False*.

Considere o código:

```
frutas = ['maçã', 'abacate', 'açaí', 'pêra']
print('maçã' in frutas)
print(not 'cajá' in frutas)#mesmo que print('cajá' not in frutas)
```

que têm como resposta: `True`

O mais comum de se fazer é colocar um elemento no vetor se não estiver contido:

```
if 'laranja' not in frutas:
    frutas.append("laranja")
print(frutas)
```

que tem como resposta:

```
['maçã', 'abacate', 'açaí',
 'pêra', 'laranja']
```

Buscas em listas

A biblioteca padrão do Python possui o comando **in**, que serve a dois propósitos:

- fazer uma busca num vetor.
- percorrer um vetor (conjuntamente com o comando **for**).

O primeiro propósito do comando *in* é realizar uma busca num vetor por um elemento em particular. O comando **in** tem como resposta um booleano, ou seja, é sempre *True* ou *False*.

Considere o código:

```
frutas = ['maçã', 'abacate', 'açaí', 'pêra']
print('maçã' in frutas)
print(not 'cajá' in frutas)#mesmo que print('cajá' not in frutas)
```

que têm como resposta: `True`

O mais comum de se fazer é colocar um elemento no vetor se não estiver contido:

```
if 'laranja' not in frutas:
    frutas.append("laranja")
print(frutas)
```

que tem como resposta:

```
['maçã', 'abacate', 'açaí',
 'pêra', 'laranja']
```

Buscas em listas

A biblioteca padrão do Python possui o comando **in**, que serve a dois propósitos:

- fazer uma busca num vetor.
- percorrer um vetor (conjuntamente com o comando **for**).

O primeiro propósito do comando *in* é realizar uma busca num vetor por um elemento em particular. O comando **in** tem como resposta um booleano, ou seja, é sempre *True* ou *False*.

Considere o código:

```
frutas = ['maçã', 'abacate', 'açaí', 'pêra']
print('maçã' in frutas)
print(not 'cajá' in frutas)#mesmo que print('cajá' not in frutas)
```

que têm como resposta: `True`

O mais comum de se fazer é colocar um elemento no vetor se não estiver contido:

```
if 'laranja' not in frutas:
    frutas.append("laranja")
print(frutas)
```

que tem como resposta:

```
['maçã', 'abacate', 'açaí',
    'pêra', 'laranja']
```

Sumário

Percorrendo vetores

A segunda função do comando **in** é de percorrer um vetor.

Comando **in** para percorrimento

```
vC = [1 , 3.4 , 'A' , " IFSC " ]
for i in vC :
    print ( i )
```

Neste caso a variável **i**, que não tem tipo definido, receberá a cada iteração um elemento de `vC`. Trocando a 3ª linha por `print(type(i))` vamos obter como resposta os tipos da lista `vC`.

O mais comum é utilizar a lista obtida do comando `range`, da seguinte forma:

```
vC = [1 , 3.4 , "A" , " IFSC " ]
for i in range (0,len(vC)):
    print ( vC [ i ])
```

O método `len(vC)` retorna o tamanho do vetor vC, no caso retorna 4. A lista obtida de `range` será estudada ainda neste tópico.
OBS: O procedimento para tuplas é idêntico para o de listas.

Percorrendo vetores

A segunda função do comando **in** é de percorrer um vetor.

Comando **in** para percorrimento

```
vC = [1 , 3.4 , 'A' , " IFSC " ]
for i in vC :
    print ( i )
```

Neste caso a variável **i**, que não tem tipo definido, receberá a cada iteração um elemento de `vC`. Trocando a 3ª linha por `print(type(i))` vamos obter como resposta os tipos da lista `vC`.

O mais comum é utilizar a lista obtida do comando `range`, da seguinte forma:

```
vC = [1 , 3.4 , "A" , " IFSC " ]
for i in range (0,len(vC)):
    print ( vC [ i ])
```

O método `len(vC)` retorna o tamanho do vetor vC, no caso retorna 4. A lista obtida de `range` será estudada ainda neste tópico.
OBS: O procedimento para tuplas é idêntico para o de listas.

O comando *enumerate*

Utilizamos *enumerate* quando precisamos de um contador durante a execução do laço em um vetor. Por exemplo:

```
moedas = ["BRL", "USD", "EUR"]
for moeda in moedas:
    print(moeda)
```

Teremos como resposta as moedas listadas no vetor. Se quisermos enumerar as moedas, fazemos:

```
moedas = ["BRL", "USD", "EUR"]
for i, moeda in enumerate(moedas):
    print(i, moeda)
```

vamos encontrar a saída:

0 BRL
1 USD
2 EUR

O comando *enumerate*

Utilizamos *enumerate* quando precisamos de um contador durante a execução do laço em um vetor. Por exemplo:

```
moedas = ["BRL", "USD", "EUR"]
for moeda in moedas:
    print(moeda)
```

Teremos como resposta as moedas listadas no vetor. Se quisermos enumerar as moedas, fazemos:

```
moedas = ["BRL", "USD", "EUR"]
for i, moeda in enumerate(moedas):
    print(i, moeda)
```

vamos encontrar a saída:

0 BRL
1 USD
2 EUR

O comando *enumerate*

Utilizamos *enumerate* quando precisamos de um contador durante a execução do laço em um vetor. Por exemplo:

```
moedas = ["BRL", "USD", "EUR"]
for moeda in moedas:
    print(moeda)
```

Teremos como resposta as moedas listadas no vetor. Se quisermos enumerar as moedas, fazemos:

```
moedas = ["BRL", "USD", "EUR"]
for i, moeda in enumerate(moedas):
    print(i, moeda)
```

vamos encontrar a saída:

0 BRL
1 USD
2 EUR

O comando *enumerate*

Utilizamos *enumerate* quando precisamos de um contador durante a execução do laço em um vetor. Por exemplo:

```
moedas = ["BRL", "USD", "EUR"]
for moeda in moedas:
    print(moeda)
```

Teremos como resposta as moedas listadas no vetor. Se quisermos enumerar as moedas, fazemos:

```
moedas = ["BRL", "USD", "EUR"]
for i, moeda in enumerate(moedas):
    print(i, moeda)
```

vamos encontrar a saída:
0 BRL
1 USD
2 EUR

Sumário

Alterando um elemento do vetor

É possível alterar os elementos do vetor (lista ou array), no caso de lista inclusive trocando o tipo.

Execute o código abaixo:

```
# -*- coding : utf -8-*-
vC = [1 , 3.4 , 'A' , " IFSC " ]
vC [2] = 555
vC [0] = " Python "
vC [1] = 3>2
for i in vC :
    print ( i )
```

Crie um programa que leia uma linha contendo vários números inteiros separados por espaço, armazene em um vetor e altere cada elemento multiplicando o mesmo por 3. Por fim, imprima cada elemento multiplicado por 3 em uma linha e separados por espaço.

Entrada	Saída
1 2 3 4 5	3 6 9 12 15

Alterando um elemento do vetor

É possível alterar os elementos do vetor (lista ou array), no caso de lista inclusive trocando o tipo.

Execute o código abaixo:

```
# -*- coding : utf -8-*-
vC = [1 , 3.4 , 'A' , " IFSC " ]
vC [2] = 555
vC [0] = " Python "
vC [1] = 3>2
for i in vC :
    print ( i )
```

Crie um programa que leia uma linha contendo vários números inteiros separados por espaço, armazene em um vetor e altere cada elemento multiplicando o mesmo por 3. Por fim, imprima cada elemento multiplicado por 3 em uma linha e separados por espaço.

Entrada	Saída
1 2 3 4 5	3 6 9 12 15

Alterando um elemento do vetor

É possível alterar os elementos do vetor (lista ou array), no caso de lista inclusive trocando o tipo.

Execute o código abaixo:

```
# -*- coding : utf -8-*-
vC = [1 , 3.4 , 'A' , " IFSC " ]
vC [2] = 555
vC [0] = " Python "
vC [1] = 3>2
for i in vC :
      print ( i )
```

Crie um programa que leia uma linha contendo vários números inteiros separados por espaço, armazene em um vetor e altere cada elemento multiplicando o mesmo por 3. Por fim, imprima cada elemento multiplicado por 3 em uma linha e separados por espaço.

Entrada	Saída
1 2 3 4 5	3 6 9 12 15

Alterando um elemento do vetor

É possível alterar os elementos do vetor (lista ou array), no caso de lista inclusive trocando o tipo.

Execute o código abaixo:

```
# -*- coding : utf -8-*-
vC = [1 , 3.4 , 'A' , " IFSC " ]
vC [2] = 555
vC [0] = " Python "
vC [1] = 3>2
for i in vC :
      print ( i )
```

Crie um programa que leia uma linha contendo vários números inteiros separados por espaço, armazene em um vetor e altere cada elemento multiplicando o mesmo por 3. Por fim, imprima cada elemento multiplicado por 3 em uma linha e separados por espaço.

Entrada	Saída
1 2 3 4 5	3 6 9 12 15

Alterando um elemento do vetor

É possível alterar os elementos do vetor (lista ou array), no caso de lista inclusive trocando o tipo.

Execute o código abaixo:

```
# -*- coding : utf -8-*-
vC = [1 , 3.4 , 'A' , " IFSC " ]
vC [2] = 555
vC [0] = " Python "
vC [1] = 3>2
for i in vC :
        print ( i )
```

Crie um programa que leia uma linha contendo vários números inteiros separados por espaço, armazene em um vetor e altere cada elemento multiplicando o mesmo por 3. Por fim, imprima cada elemento multiplicado por 3 em uma linha e separados por espaço.

Entrada	Saída
1 2 3 4 5	3 6 9 12 15

Adicionando ou removendo um elemento do vetor

É possível adicionar novos elementos no vetor. Execute o código abaixo:

```
# -*- coding : utf -8-*-
vC = [1 , 3.4 , 'A' , " IFSC " ]
vC.insert(0,56) #Adiciona na posição 0 o inteiro 56.
vC.insert(3,'B')
vC.append(11)
for i in vC :
    print ( i )
```

O comando `insert` insere um item na posição especificada.

A forma abstrata é: `NomeVetor.insert(posicao, item)`

Para inserir itens no final usamos `NomeVetor.append(item)`.

É possível remover elementos do vetor.

```
# -*- coding : utf -8-*-
vC = [1 , 3.4 , 'A' , " IFSC " ]
vC.remove('A')
vC.remove(3.4)
for i in vC :
    print ( i )
```

Adicionando ou removendo um elemento do vetor

É possível adicionar novos elementos no vetor. Execute o código abaixo:

```
# -*- coding : utf -8-*-
vC = [1 , 3.4 , 'A' , " IFSC " ]
vC.insert(0,56) #Adiciona na posição 0 o inteiro 56.
vC.insert(3,'B')
vC.append(11)
for i in vC :
      print ( i )
```

O comando `insert` insere um item na posição especificada.

A forma abstrata é: `NomeVetor.insert(posicao, item)`

Para inserir itens no final usamos `NomeVetor.append(item)`.

É possível remover elementos do vetor.

```
# -*- coding : utf -8-*-
vC = [1 , 3.4 , 'A' , " IFSC " ]
vC.remove('A')
vC.remove(3.4)
for i in vC :
      print ( i )
```

Adicionando ou removendo um elemento do vetor

É possível adicionar novos elementos no vetor. Execute o código abaixo:

```
# -*- coding : utf -8-*-
vC = [1 , 3.4 , 'A' , " IFSC " ]
vC.insert(0,56) #Adiciona na posição 0 o inteiro 56.
vC.insert(3,'B')
vC.append(11)
for i in vC :
      print ( i )
```

O comando `insert` insere um item na posição especificada.

A forma abstrata é: `NomeVetor.insert(posicao, item)`

Para inserir itens no final usamos `NomeVetor.append(item)`.

É possível remover elementos do vetor.

```
# -*- coding : utf -8-*-
vC = [1 , 3.4 , 'A' , " IFSC " ]
vC.remove('A')
vC.remove(3.4)
for i in vC :
      print ( i )
```

Adicionando ou removendo um elemento do vetor

É possível adicionar novos elementos no vetor. Execute o código abaixo:

```
# -*- coding : utf -8-*-
vC = [1 , 3.4 , 'A' , " IFSC " ]
vC.insert(0,56) #Adiciona na posição 0 o inteiro 56.
vC.insert(3,'B')
vC.append(11)
for i in vC :
      print ( i )
```

O comando `insert` insere um item na posição especificada.

A forma abstrata é: `NomeVetor.insert(posicao, item)`

Para inserir itens no final usamos `NomeVetor.append(item)`.

É possível remover elementos do vetor.

```
# -*- coding : utf -8-*-
vC = [1 , 3.4 , 'A' , " IFSC " ]
vC.remove('A')
vC.remove(3.4)
for i in vC :
      print ( i )
```

Adicionando ou removendo um elemento do vetor

É possível adicionar novos elementos no vetor. Execute o código abaixo:

```
# -*- coding : utf -8-*-
vC = [1 , 3.4 , 'A' , " IFSC " ]
vC.insert(0,56) #Adiciona na posição 0 o inteiro 56.
vC.insert(3,'B')
vC.append(11)
for i in vC :
      print ( i )
```

O comando `insert` insere um item na posição especificada.
A forma abstrata é: `NomeVetor.insert(posicao, item)`
Para inserir itens no final usamos `NomeVetor.append(item)`.
É possível remover elementos do vetor.

```
# -*- coding : utf -8-*-
vC = [1 , 3.4 , 'A' , " IFSC " ]
vC.remove('A')
vC.remove(3.4)
for i in vC :
      print ( i )
```

Adicionando ou removendo um elemento do vetor

Cuidado: O uso do método vetor.remove(*item*) não remove todas as ocorrências de *item* do vetor. Execute o script abaixo:

```
# -*- coding : utf -8-*-
vC = [1 , 3.4 , 'A' , " IFSC ", 'A' ]
vC.remove('A')
for i in vC :
    print ( i )
```

Para remover todas as ocorrências de 'A' devemos usar um laço.

```
# -*- coding : utf -8-*-
vC = [1 , 3.4 , 'A' , " IFSC ", 'A' ]
while 'A' in vC:
    vC.remove('A')
for i in vC :
    print ( i )
```

Adicionando ou removendo um elemento do vetor

Cuidado: O uso do método vetor.remove(*item*) não remove todas as ocorrências de *item* do vetor. Execute o script abaixo:

```
# -*- coding : utf -8-*-
vC = [1 , 3.4 , 'A' , " IFSC ", 'A' ]
vC.remove('A')
for i in vC :
        print ( i )
```

Para remover todas as ocorrências de 'A' devemos usar um laço.

```
# -*- coding : utf -8-*-
vC = [1 , 3.4 , 'A' , " IFSC ", 'A' ]
while 'A' in vC:
    vC.remove('A')
for i in vC :
    print ( i )
```

Adicionando ou removendo um elemento do vetor

Cuidado: O uso do método vetor.remove(*item*) não remove todas as ocorrências de *item* do vetor. Execute o script abaixo:

```
# -*- coding : utf -8-*-
vC = [1 , 3.4 , 'A' , " IFSC ", 'A' ]
vC.remove('A')
for i in vC :
        print ( i )
```

Para remover todas as ocorrências de 'A' devemos usar um laço.

```
# -*- coding : utf -8-*-
vC = [1 , 3.4 , 'A' , " IFSC ", 'A' ]
while 'A' in vC:
    vC.remove('A')
for i in vC :
    print ( i )
```

Adicionando ou removendo um elemento do vetor

Cuidado: O uso do método vetor.remove(*item*) não remove todas as ocorrências de *item* do vetor. Execute o script abaixo:

```
# -*- coding : utf -8-*-
vC = [1 , 3.4 , 'A' , " IFSC ", 'A' ]
vC.remove('A')
for i in vC :
      print ( i )
```

Para remover todas as ocorrências de 'A' devemos usar um laço.

```
# -*- coding : utf -8-*-
vC = [1 , 3.4 , 'A' , " IFSC ", 'A' ]
while 'A' in vC:
      vC.remove('A')
for i in vC :
      print ( i )
```

Removendo um elemento do vetor pelo índice

Para remover um elemento pelo índice usamos o comando **del** ou o método **pop**.

Uso do comando del:

```
# -*- coding : utf -8-*-
vC = [1 , 3.4 , 'A' , " IFSC ", 'A' ]
del vC[1]
for i in vC :
    print ( i )
```

Observe que o termo removido foi 3.4, pois a posição 1 é ocupada pelo segundo elemento. Poderíamos remover uma faixa, usando `del vC[1:3]`, e os elementos de posição 1 e 2 seriam removidos.

Para remover todos os termos de um vetor usamos o método vetor.clear(). Inclua no script acima as linhas:

```
vC.clear()
print("Tamanho de vC: ", len(vC))
```

e verifique que o tamanho do vetor passa a ser zero.

Removendo um elemento do vetor pelo índice

Para remover um elemento pelo índice usamos o comando **del** ou o método **pop**.

Uso do comando del:

```
# -*- coding : utf -8-*-
vC = [1 , 3.4 , 'A' , " IFSC ", 'A' ]
del vC[1]
for i in vC :
        print ( i )
```

Observe que o termo removido foi 3.4, pois a posição 1 é ocupada pelo segundo elemento. Poderíamos remover uma faixa, usando `del vC[1:3]`, e os elementos de posição 1 e 2 seriam removidos.

Para remover todos os termos de um vetor usamos o método vetor.clear(). Inclua no script acima as linhas:

```
vC.clear()
print("Tamanho de vC: ", len(vC))
```

e verifique que o tamanho do vetor passa a ser zero.

Removendo um elemento do vetor pelo índice

Para remover um elemento pelo índice usamos o comando **del** ou o método **pop**.

Uso do comando del:

```
# -*- coding : utf -8-*-
vC = [1 , 3.4 , 'A' , " IFSC ", 'A' ]
del vC[1]
for i in vC :
        print ( i )
```

Observe que o termo removido foi 3.4, pois a posição 1 é ocupada pelo segundo elemento. Poderíamos remover uma faixa, usando `del vC[1:3]`, e os elementos de posição 1 e 2 seriam removidos.

Para remover todos os termos de um vetor usamos o método vetor.clear(). Inclua no script acima as linhas:

```
vC.clear()
print("Tamanho de vC: ", len(vC))
```

e verifique que o tamanho do vetor passa a ser zero.

Removendo um elemento do vetor pelo índice

Para remover um elemento pelo índice usamos o comando **del** ou o método **pop**.

Uso do comando del:

```
# -*- coding : utf -8-*-
vC = [1 , 3.4 , 'A' , " IFSC ", 'A' ]
del vC[1]
for i in vC :
        print ( i )
```

Observe que o termo removido foi 3.4, pois a posição 1 é ocupada pelo segundo elemento. Poderíamos remover uma faixa, usando `del vC[1:3]`, e os elementos de posição 1 e 2 seriam removidos.

Para remover todos os termos de um vetor usamos o método vetor.clear(). Inclua no script acima as linhas:

```
vC.clear()
print("Tamanho de vC: ", len(vC))
```

e verifique que o tamanho do vetor passa a ser zero.

Removendo um elemento do vetor pelo índice

Para remover um elemento pelo índice usamos o comando **del** ou o método **pop**.

Uso do comando del:

```
# -*- coding : utf -8-*-
vC = [1 , 3.4 , 'A' , " IFSC ", 'A' ]
del vC[1]
for i in vC :
        print ( i )
```

Observe que o termo removido foi 3.4, pois a posição 1 é ocupada pelo segundo elemento. Poderíamos remover uma faixa, usando `del vC[1:3]`, e os elementos de posição 1 e 2 seriam removidos.

Para remover todos os termos de um vetor usamos o método vetor.clear(). Inclua no script acima as linhas:

```
vC.clear()
print("Tamanho de vC: ", len(vC))
```

e verifique que o tamanho do vetor passa a ser zero.

Removendo um elemento do vetor pelo índice

Para remover um elemento pelo índice usamos o comando **del** ou o método **pop**.

Uso do comando del:

```
# -*- coding : utf -8-*-
vC = [1 , 3.4 , 'A' , " IFSC ", 'A' ]
del vC[1]
for i in vC :
        print ( i )
```

Observe que o termo removido foi 3.4, pois a posição 1 é ocupada pelo segundo elemento. Poderíamos remover uma faixa, usando `del vC[1:3]`, e os elementos de posição 1 e 2 seriam removidos.

Para remover todos os termos de um vetor usamos o método vetor.clear(). Inclua no script acima as linhas:

```
vC.clear()
print("Tamanho de vC: ", len(vC))
```

e verifique que o tamanho do vetor passa a ser zero.

Removendo um elemento do vetor pelo índice

Para remover um elemento pelo índice usamos o comando **del** ou o método **pop**.

Uso do comando del:

```
# -*- coding : utf -8-*-
vC = [1 , 3.4 , 'A' , " IFSC ", 'A' ]
del vC[1]
for i in vC :
        print ( i )
```

Observe que o termo removido foi 3.4, pois a posição 1 é ocupada pelo segundo elemento. Poderíamos remover uma faixa, usando `del vC[1:3]`, e os elementos de posição 1 e 2 seriam removidos.

Para remover todos os termos de um vetor usamos o método vetor.clear(). Inclua no script acima as linhas:

```
vC.clear()
print("Tamanho de vC: ", len(vC))
```

e verifique que o tamanho do vetor passa a ser zero.

Removendo um elemento do vetor pelo índice

Para remover um elemento pelo índice usamos o comando **del** ou o método **pop**.

Uso do comando del:

```
# -*- coding : utf -8-*-
vC = [1 , 3.4 , 'A' , " IFSC ", 'A' ]
del vC[1]
for i in vC :
        print ( i )
```

Observe que o termo removido foi 3.4, pois a posição 1 é ocupada pelo segundo elemento. Poderíamos remover uma faixa, usando `del vC[1:3]`, e os elementos de posição 1 e 2 seriam removidos.

Para remover todos os termos de um vetor usamos o método vetor.clear(). Inclua no script acima as linhas:

```
vC.clear()
print("Tamanho de vC: ", len(vC))
```

e verifique que o tamanho do vetor passa a ser zero.

Removendo um elemento do vetor pelo índice

O método pop remove o elemento do vetor e, opcionalmente, guarda o valor numa variável. Se nenhuma posição for informada, pop remove o último termo do vetor.

```
1 #-*- coding:utf:8 -*-
2 vC = [1 , 3.4 ,'A' , " IFSC ",'A' ]
3 print("Vetor original: ")
4 for i in vC:
5     print(i, end = ' ')
6 print()
7 print("Remoção do termo da posição 1:")
8 x = vC.pop(1)
9 for i in vC:
10     print(i, end = ' ')
11 print()
12 print("Remoção do último termo:")
13 y = vC.pop()    #Remove o último elemento
14 for i in vC:
15     print(i, end = ' ')
16 print()
17
18 print("Elementos removidos: ", x, y)
```

Se quisermos saber quantas vezes um valor aparece numa lista, usamos o método `count`:

```
print(vC.count('A'))
```

Removendo um elemento do vetor pelo índice

O método pop remove o elemento do vetor e, opcionalmente, guarda o valor numa variável. Se nenhuma posição for informada, pop remove o último termo do vetor.

```
1 #-*- coding:utf:8 -*-
2 vC = [1 , 3.4 ,'A' , " IFSC ",'A' ]
3 print("Vetor original: ")
4 for i in vC:
5     print(i, end = ' ')
6 print()
7 print("Remoção do termo da posição 1:")
8 x = vC.pop(1)
9 for i in vC:
10    print(i, end = ' ')
11 print()
12 print("Remoção do último termo:")
13 y = vC.pop()    #Remove o último elemento
14 for i in vC:
15    print(i, end = ' ')
16 print()
17
18 print("Elementos removidos: ", x, y)
```

Se quisermos saber quantas vezes um valor aparece numa lista, usamos o método `count`:

`print(vC.count('A'))`

Concatenando listas

Considere o bloco de código:

```
valores = list(range(1,11))
anos = list(range(2020,2060,10))
print(valores)
valores.extend(anos)
print(valores)
```

À lista `valores` foi concatenada a lista `anos`, alterando seu tamanho de 10 para 14. A lista `anos` permanece existindo e inalterada.

Para criar nova lista com a concatenação usamos o operador `+`:

```
valores = list(range(1,11))
anos = list(range(2020,2060,10))
lista_concatenada = valores+anos
print(lista_concatenada)
```

Concatenando listas

Considere o bloco de código:

```
valores = list(range(1,11))
anos = list(range(2020,2060,10))
print(valores)
valores.extend(anos)
print(valores)
```

À lista `valores` foi concatenada a lista `anos`, alterando seu tamanho de 10 para 14. A lista `anos` permanece existindo e inalterada.

Para criar nova lista com a concatenação usamos o operador `+`:

```
valores = list(range(1,11))
anos = list(range(2020,2060,10))
lista_concatenada = valores+anos
print(lista_concatenada)
```

Concatenando listas

Considere o bloco de código:

```
valores = list(range(1,11))
anos = list(range(2020,2060,10))
print(valores)
valores.extend(anos)
print(valores)
```

À lista `valores` foi concatenada a lista `anos`, alterando seu tamanho de 10 para 14. A lista `anos` permanece existindo e inalterada.

Para criar nova lista com a concatenação usamos o operador `+`:

```
valores = list(range(1,11))
anos = list(range(2020,2060,10))
lista_concatenada = valores+anos
print(lista_concatenada)
```

Concatenando listas

Considere o bloco de código:

```
valores = list(range(1,11))
anos = list(range(2020,2060,10))
print(valores)
valores.extend(anos)
print(valores)
```

À lista `valores` foi concatenada a lista `anos`, alterando seu tamanho de 10 para 14. A lista `anos` permanece existindo e inalterada.

Para criar nova lista com a concatenação usamos o operador `+`:

```
valores = list(range(1,11))
anos = list(range(2020,2060,10))
lista_concatenada = valores+anos
print(lista_concatenada)
```

Sumário

Exemplos

Para colocar uma lista em ordem crescente utilizamos o comando `sort()` no final do nome da lista. A lista pode ser numérica ou alfabética, no último caso `sort()` coloca a lista em ordem alfabética.

```
mercado = ['ouro', "bitcoin", 'titulos' ]
mercado.sort()
```

Retorna a lista em ordem alfabética. Se quiséssemos a lista em ordem reversa, poderíamos usar:

`mercado.sort(reverse = True)` ou `mercado.reverse()`

O problema ocorre quando precisamos misturar palavras que iniciam com letras maiúsculas com minúsculas, como em:

`mercado = ['ouro', 'bitcoin', 'titulos', 'Dólar', 'Real'].`

Para poder ordenar vamos utilizar: `mercado.sort(key=str.casefold).`

OBS:

Não se pode usar o comando `sort()` quando os elementos de uma lista tem elementos numéricos e alfabéticos simultaneamente. Gera erro utilizar:

```
vC = [1 , 3.4 , 'A' , " IFSC ", 'A' ]
vC.sort()
```

Exemplos

Para colocar uma lista em ordem crescente utilizamos o comando `sort()` no final do nome da lista. A lista pode ser numérica ou alfabética, no último caso `sort()` coloca a lista em ordem alfabética.

```
mercado = ['ouro', "bitcoin", 'titulos' ]
mercado.sort()
```

Retorna a lista em ordem alfabética. Se quiséssemos a lista em ordem reversa, poderíamos usar:

`mercado.sort(reverse = True)` ou `mercado.reverse()`

O problema ocorre quando precisamos misturar palavras que iniciam com letras maiúsculas com minúsculas, como em:

`mercado = ['ouro', 'bitcoin', 'titulos', 'Dólar', 'Real'].`

Para poder ordenar vamos utilizar: `mercado.sort(key=str.casefold).`

OBS:

Não se pode usar o comando `sort()` quando os elementos de uma lista tem elementos numéricos e alfabéticos simultaneamente. Gera erro utilizar:

```
vC = [1 , 3.4 , 'A' , " IFSC ", 'A' ]
vC.sort()
```

Exemplos

Para colocar uma lista em ordem crescente utilizamos o comando `sort()` no final do nome da lista. A lista pode ser numérica ou alfabética, no último caso `sort()` coloca a lista em ordem alfabética.

```
mercado = ['ouro', "bitcoin", 'titulos' ]
mercado.sort()
```

Retorna a lista em ordem alfabética. Se quiséssemos a lista em ordem reversa, poderíamos usar:

`mercado.sort(reverse = True)` ou `mercado.reverse()`

O problema ocorre quando precisamos misturar palavras que iniciam com letras maiúsculas com minúsculas, como em:

`mercado = ['ouro', 'bitcoin', 'titulos', 'Dólar', 'Real'].`

Para poder ordenar vamos utilizar: `mercado.sort(key=str.casefold).`

OBS:

Não se pode usar o comando `sort()` quando os elementos de uma lista tem elementos numéricos e alfabéticos simultaneamente. Gera erro utilizar:

```
vC = [1 , 3.4 , 'A' , "IFSC", 'A' ]
vC.sort()
```

Exemplos

Para colocar uma lista em ordem crescente utilizamos o comando `sort()` no final do nome da lista. A lista pode ser numérica ou alfabética, no último caso `sort()` coloca a lista em ordem alfabética.

```
mercado = ['ouro', "bitcoin", 'titulos' ]
mercado.sort()
```

Retorna a lista em ordem alfabética. Se quiséssemos a lista em ordem reversa, poderíamos usar:

`mercado.sort(reverse = True)` ou `mercado.reverse()`

O problema ocorre quando precisamos misturar palavras que iniciam com letras maiúsculas com minúsculas, como em:

`mercado = ['ouro', 'bitcoin', 'titulos', 'Dólar', 'Real'].`

Para poder ordenar vamos utilizar: `mercado.sort(key=str.casefold).`

OBS:

Não se pode usar o comando `sort()` quando os elementos de uma lista tem elementos numéricos e alfabéticos simultaneamente. Gera erro utilizar:

```
vC = [1 , 3.4 , 'A' , " IFSC ", 'A' ]
vC.sort()
```

Sumário

classe range

A classe range em python gera uma sequência de inteiros, que pode ter seu início, fim e passo controlados.

Considere o bloco de código:

```
a = range(12)
print(a)
print(type(a))
b = list(a)
print(b)
```

que possui a resposta:

```
range(0, 12)
<class 'range'>
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

Geralmente uma sequência range é usada para fazer iterações, forçando uma variável iterar entre seus valores a cada passo.

```
vC = [12,23,1,1,2,43,32,23]
for i in range(len(vC)):
    print(vC[i])
```

Neste exemplo fazemos `i` receber o valor a cada iteração de `range(len(vC))`, que vai de 0 até o tamanho do vetor menos 1, e pedimos para que se imprima o valor guardado na posição `i` no vetor `vC`.

O escopo mais completo de range é: `range(inicio, fim, passo)`, em que `início` é o primeiro termo guardado em `range`, `passo` é o salto numérico que é dado, e o último termo é o maior inteiro possível menor que `fim`.

classe range

A classe range em python gera uma sequência de inteiros, que pode ter seu início, fim e passo controlados.
Considere o bloco de código:

```
a = range(12)
print(a)
print(type(a))
b = list(a)
print(b)
```

que possui a resposta:

```
range(0, 12)
<class 'range'>
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

Geralmente uma sequência range é usada para fazer iterações, forçando uma variável iterar entre seus valores a cada passo.

```
vC = [12,23,1,1,2,43,32,23]
for i in range(len(vC)):
    print(vC[i])
```

Neste exemplo fazemos `i` receber o valor a cada iteração de `range(len(vC))`, que vai de 0 até o tamanho do vetor menos 1, e pedimos para que se imprima o valor guardado na posição `i` no vetor `vC`.

O escopo mais completo de range é: `range(inicio, fim, passo)`, em que `início` é o primeiro termo guardado em `range`, `passo` é o salto numérico que é dado, e o último termo é o maior inteiro possível menor que `fim`.

classe range

A classe range em python gera uma sequência de inteiros, que pode ter seu início, fim e passo controlados.
Considere o bloco de código:

```
a = range(12)
print(a)
print(type(a))
b = list(a)
print(b)
```

que possui a resposta:

```
range(0, 12)
<class 'range'>
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

Geralmente uma sequência range é usada para fazer iterações, forçando uma variável iterar entre seus valores a cada passo.

```
vC = [12,23,1,1,2,43,32,23]
for i in range(len(vC)):
    print(vC[i])
```

Neste exemplo fazemos `i` receber o valor a cada iteração de `range(len(vC))`, que vai de 0 até o tamanho do vetor menos 1, e pedimos para que se imprima o valor guardado na posição `i` no vetor `vC`.

O escopo mais completo de range é: `range(inicio, fim, passo)`, em que `início` é o primeiro termo guardado em `range`, `passo` é o salto numérico que é dado, e o último termo é o maior inteiro possível menor que `fim`.

classe range

A classe range em python gera uma sequência de inteiros, que pode ter seu início, fim e passo controlados.
Considere o bloco de código:

```
a = range(12)
print(a)
print(type(a))
b = list(a)
print(b)
```

que possui a resposta:

```
range(0, 12)
<class 'range'>
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

Geralmente uma sequência range é usada para fazer iterações, forçando uma variável iterar entre seus valores a cada passo.

```
vC = [12,23,1,1,2,43,32,23]
for i in range(len(vC)):
    print(vC[i])
```

Neste exemplo fazemos `i` receber o valor a cada iteração de `range(len(vC))`, que vai de 0 até o tamanho do vetor menos 1, e pedimos para que se imprima o valor guardado na posição `i` no vetor `vC`.

O escopo mais completo de range é: `range(inicio, fim, passo)`, em que `início` é o primeiro termo guardado em `range`, `passo` é o salto numérico que é dado, e o último termo é o maior inteiro possível menor que `fim`.

classe range

A classe range em python gera uma sequência de inteiros, que pode ter seu início, fim e passo controlados.
Considere o bloco de código:

```
a = range(12)
print(a)
print(type(a))
b = list(a)
print(b)
```

que possui a resposta:

```
range(0, 12)
<class 'range'>
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

Geralmente uma sequência range é usada para fazer iterações, forçando uma variável iterar entre seus valores a cada passo.

```
vC = [12,23,1,1,2,43,32,23]
for i in range(len(vC)):
    print(vC[i])
```

Neste exemplo fazemos `i` receber o valor a cada iteração de `range(len(vC)`, que vai de 0 até o tamanho do vetor menos 1, e pedimos para que se imprima o valor guardado na posição `i` no vetor `vC`.

O escopo mais completo de range é: `range(inicio, fim, passo)`, em que `início` é o primeiro termo guardado em `range`, `passo` é o salto numérico que é dado, e o último termo é o maior inteiro possível menor que `fim`.

classe range

A classe range em python gera uma sequência de inteiros, que pode ter seu início, fim e passo controlados.
Considere o bloco de código:

```
a = range(12)
print(a)
print(type(a))
b = list(a)
print(b)
```

que possui a resposta:

```
range(0, 12)
<class 'range'>
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

Geralmente uma sequência range é usada para fazer iterações, forçando uma variável iterar entre seus valores a cada passo.

```
vC = [12,23,1,1,2,43,32,23]
for i in range(len(vC)):
    print(vC[i])
```

Neste exemplo fazemos `i` receber o valor a cada iteração de `range(len(vC)`, que vai de 0 até o tamanho do vetor menos 1, e pedimos para que se imprima o valor guardado na posição `i` no vetor `vC`.

O escopo mais completo de range é: `range(inicio, fim, passo)`, em que `início` é o primeiro termo guardado em `range`, `passo` é o salto numérico que é dado, e o último termo é o maior inteiro possível menor que `fim`.

classe range

O Python todavia não foi projetado para trabalhar com variáveis tipadas como foi a linguagem C. Python foi feito para trabalhar da seguinte forma:

Python

```
vC = [12,23.5,'casa',1,2]
for i in vC:
    print(i)
```

Linguagem C

```
int numeros[5] = {10, 20, 30, 40, 50};
for(int i=0; i<5; ++i) {
printf("%i\n", i)}
```

Neste caso, em Python, a variável `i` recebe a cada iteração o valor que estiver na lista `vC`, que pode ser de tipo diferente se não for um array.

Em linguagem C tudo tem de ser tipado, desde a definição do array até a impressão do valor dentro do array.

Esta facilidade do Python tem um preço, caso não se use a classe range não há como saber qual a posição que está o vetor no momento da iteração. Para isto foi criado o comando `enumerate(vetor)`.

```
vC = [12,23.5,'casa',1,2]
for i in enumerate(vC):
    print(i)
```

```
(0, 12)
(1, 23.5)
(2, 'casa')
(3, 1)
(4, 2)
```

classe range

O Python todavia não foi projetado para trabalhar com variáveis tipadas como foi a linguagem C. Python foi feito para trabalhar da seguinte forma:

Python

```
vC = [12,23.5,'casa',1,2]
for i in vC:
    print(i)
```

Linguagem C

```
int numeros[5] = {10, 20, 30, 40, 50};
for(int i=0; i<5; ++i) {
printf("%i\n", i)}
```

Neste caso, em Python, a variável `i` recebe a cada iteração o valor que estiver na lista `vC`, que pode ser de tipo diferente se não for um array.

Em linguagem C tudo tem de ser tipado, desde a definição do array até a impressão do valor dentro do array.

Esta facilidade do Python tem um preço, caso não se use a classe range não há como saber qual a posição que está o vetor no momento da iteração. Para isto foi criado o comando `enumerate(vetor)`.

```
vC = [12,23.5,'casa',1,2]
for i in enumerate(vC):
    print(i)
```

```
(0, 12)
(1, 23.5)
(2, 'casa')
(3, 1)
(4, 2)
```

classe range

O Python todavia não foi projetado para trabalhar com variáveis tipadas como foi a linguagem C. Python foi feito para trabalhar da seguinte forma:

Python

```
vC = [12,23.5,'casa',1,2]
for i in vC:
    print(i)
```

Linguagem C

```
int numeros[5] = {10, 20, 30, 40, 50};
for(int i=0; i<5; ++i) {
printf("%i\n", i)}
```

Neste caso, em Python, a variável `i` recebe a cada iteração o valor que estiver na lista `vC`, que pode ser de tipo diferente se não for um array.

Em linguagem C tudo tem de ser tipado, desde a definição do array até a impressão do valor dentro do array.

Esta facilidade do Python tem um preço, caso não se use a classe range não há como saber qual a posição que está o vetor no momento da iteração. Para isto foi criado o comando `enumerate(vetor)`.

```
vC = [12,23.5,'casa',1,2]
for i in enumerate(vC):
    print(i)
```

```
(0, 12)
(1, 23.5)
(2, 'casa')
(3, 1)
(4, 2)
```

classe range

O Python todavia não foi projetado para trabalhar com variáveis tipadas como foi a linguagem C. Python foi feito para trabalhar da seguinte forma:

Python

```
vC = [12,23.5,'casa',1,2]
for i in vC:
    print(i)
```

Linguagem C

```
int numeros[5] = {10, 20, 30, 40, 50};
for(int i=0; i<5; ++i) {
printf("%i\n", i)}
```

Neste caso, em Python, a variável `i` recebe a cada iteração o valor que estiver na lista `vC`, que pode ser de tipo diferente se não for um array.

Em linguagem C tudo tem de ser tipado, desde a definição do array até a impressão do valor dentro do array.

Esta facilidade do Python tem um preço, caso não se use a classe range não há como saber qual a posição que está o vetor no momento da iteração. Para isto foi criado o comando `enumerate(vetor)`.

```
vC = [12,23.5,'casa',1,2]
for i in enumerate(vC):
    print(i)
```

```
(0, 12)
(1, 23.5)
(2, 'casa')
(3, 1)
(4, 2)
```

classe range

O Python todavia não foi projetado para trabalhar com variáveis tipadas como foi a linguagem C. Python foi feito para trabalhar da seguinte forma:

Python

```
vC = [12,23.5,'casa',1,2]
for i in vC:
    print(i)
```

Linguagem C

```
int numeros[5] = {10, 20, 30, 40, 50};
for(int i=0; i<5; ++i) {
printf("%i\n", i)}
```

Neste caso, em Python, a variável `i` recebe a cada iteração o valor que estiver na lista `vC`, que pode ser de tipo diferente se não for um array.

Em linguagem C tudo tem de ser tipado, desde a definição do array até a impressão do valor dentro do array.

Esta facilidade do Python tem um preço, caso não se use a classe range não há como saber qual a posição que está o vetor no momento da iteração. Para isto foi criado o comando `enumerate(vetor)`.

```
vC = [12,23.5,'casa',1,2]
for i in enumerate(vC):
    print(i)
```

```
(0, 12)
(1, 23.5)
(2, 'casa')
(3, 1)
(4, 2)
```

classe range

O Python todavia não foi projetado para trabalhar com variáveis tipadas como foi a linguagem C. Python foi feito para trabalhar da seguinte forma:

Python

```
vC = [12,23.5,'casa',1,2]
for i in vC:
    print(i)
```

Linguagem C

```
int numeros[5] = {10, 20, 30, 40, 50};
for(int i=0; i<5; ++i) {
printf("%i\n", i)}
```

Neste caso, em Python, a variável `i` recebe a cada iteração o valor que estiver na lista `vC`, que pode ser de tipo diferente se não for um array.

Em linguagem C tudo tem de ser tipado, desde a definição do array até a impressão do valor dentro do array.

Esta facilidade do Python tem um preço, caso não se use a classe range não há como saber qual a posição que está o vetor no momento da iteração. Para isto foi criado o comando `enumerate(vetor)`.

```
vC = [12,23.5,'casa',1,2]
for i in enumerate(vC):
    print(i)
```

```
(0, 12)
(1, 23.5)
(2, 'casa')
(3, 1)
(4, 2)
```

classe range

O Python todavia não foi projetado para trabalhar com variáveis tipadas como foi a linguagem `C`. Python foi feito para trabalhar da seguinte forma:

Python

```
vC = [12,23.5,'casa',1,2]
for i in vC:
    print(i)
```

Linguagem C

```
int numeros[5] = {10, 20, 30, 40, 50};
for(int i=0; i<5; ++i) {
printf("%i\n", i)}
```

Neste caso, em Python, a variável `i` recebe a cada iteração o valor que estiver na lista `vC`, que pode ser de tipo diferente se não for um array.

Em linguagem `C` tudo tem de ser tipado, desde a definição do array até a impressão do valor dentro do array.

Esta facilidade do Python tem um preço, caso não se use a classe range não há como saber qual a posição que está o vetor no momento da iteração. Para isto foi criado o comando `enumerate(vetor)`.

```
vC = [12,23.5,'casa',1,2]
for i in enumerate(vC):
    print(i)
```

```
(0, 12)
(1, 23.5)
(2, 'casa')
(3, 1)
(4, 2)
```

Enumerate

O comando `enumerate(vetor)` retorna uma tupla contendo a posição do item do vetor e o conteúdo. Por esta razão é comum encontrarmos construções como a abaixo:

```
vC = [12,23.5,'casa',1,2,7,9]
for indice, valor in enumerate(vC):
    print(indice, valor)
    if indice==4:
        break
```

```
0 12
1 23.5
2 casa
3 1
4 2
```

Desta forma desempacotamos a tupla em cada iteração numa variável específica, uma guardando o índice e outra o valor.
Um recurso adicional é poder iniciar o índice num valor arbitrado, por exemplo:

```
vC = [12,23.5,'casa',1,2]
for indice, valor in enumerate(vC,-1):
    print(indice, valor)
```

```
-1 12
0 23.5
1 casa
2 1
3 2
```

Apesar de pouco usual verificamos que podemos iniciar o índice por 5, ou qualquer outro inteiro que se queira iniciar a contagem, inclusive negativo.

Sumário

Métodos para compreensão de listas

São métodos para definir listas de forma rápida.

Considere o código abaixo:

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

que tem a resposta: `[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]`

Uma compreensão de lista tem a forma:
`lista=[expressão for membro in iterável]`.

Uma `expressão` pode ser:

- Um membro que está sendo iterado.
- Uma chamada a um método.
- Expressão válida que retorne um valor.

No caso `2*i` é uma expressão que envolve a variável `i` e retorna um valor, a partir de então se coloca para iterar a variável `i`. A variável `i` receberá os valores $0, 1, \cdots, 9$ e alimentará a lista com valor `2*i`.

Métodos para compreensão de listas

São métodos para definir listas de forma rápida.

Considere o código abaixo:

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

que tem a resposta: `[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]`

Uma compreensão de lista tem a forma:
`lista=[expressão for membro in iterável]`.

Uma `expressão` pode ser:

- Um membro que está sendo iterado.
- Uma chamada a um método.
- Expressão válida que retorne um valor.

No caso `2*i` é uma expressão que envolve a variável `i` e retorna um valor, a partir de então se coloca para iterar a variável `i`. A variável `i` receberá os valores $0, 1, \cdots, 9$ e alimentará a lista com valor `2*i`.

Métodos para compreensão de listas

São métodos para definir listas de forma rápida.

Considere o código abaixo:

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

que tem a resposta: `[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]`

Uma compreensão de lista tem a forma:
`lista=[expressão for membro in iterável]`.

Uma `expressão` pode ser:

- Um membro que está sendo iterado.
- Uma chamada a um método.
- Expressão válida que retorne um valor.

No caso `2*i` é uma expressão que envolve a variável `i` e retorna um valor, a partir de então se coloca para iterar a variável `i`. A variável `i` receberá os valores $0, 1, \cdots, 9$ e alimentará a lista com valor `2*i`.

Métodos para compreensão de listas

São métodos para definir listas de forma rápida.

Considere o código abaixo:

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

que tem a resposta: `[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]`

Uma compreensão de lista tem a forma:
`lista=[expressão for membro in iterável]`.

Uma `expressão` pode ser:

- Um membro que está sendo iterado.
- Uma chamada a um método.
- Expressão válida que retorne um valor.

No caso `2*i` é uma expressão que envolve a variável `i` e retorna um valor, a partir de então se coloca para iterar a variável `i`. A variável `i` receberá os valores $0, 1, \cdots, 9$ e alimentará a lista com valor `2*i`.

Métodos para compreensão de listas

São métodos para definir listas de forma rápida.

Considere o código abaixo:

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

que tem a resposta: `[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]`

Uma compreensão de lista tem a forma:
`lista=[expressão for membro in iterável]`.

Uma `expressão` pode ser:

- Um membro que está sendo iterado.
- Uma chamada a um método.
- Expressão válida que retorne um valor.

No caso `2*i` é uma expressão que envolve a variável `i` e retorna um valor, a partir de então se coloca para iterar a variável `i`. A variável `i` receberá os valores $0, 1, \cdots, 9$ e alimentará a lista com valor `2*i`.

Métodos para compreensão de listas

São métodos para definir listas de forma rápida.

Considere o código abaixo:

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

que tem a resposta: `[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]`

Uma compreensão de lista tem a forma:
`lista=[expressão for membro in iterável]`.

Uma `expressão` pode ser:

- Um membro que está sendo iterado.
- Uma chamada a um método.
- Expressão válida que retorne um valor.

No caso `2*i` é uma expressão que envolve a variável `i` e retorna um valor, a partir de então se coloca para iterar a variável `i`. A variável `i` receberá os valores $0, 1, \cdots, 9$ e alimentará a lista com valor `2*i`.

Métodos para compreensão de listas

São métodos para definir listas de forma rápida.

Considere o código abaixo:

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

que tem a resposta: `[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]`

Uma compreensão de lista tem a forma:
`lista=[expressão for membro in iterável]`.

Uma `expressão` pode ser:

- Um membro que está sendo iterado.
- Uma chamada a um método.
- Expressão válida que retorne um valor.

No caso `2*i` é uma expressão que envolve a variável `i` e retorna um valor, a partir de então se coloca para iterar a variável `i`. A variável `i` receberá os valores $0, 1, \cdots, 9$ e alimentará a lista com valor `2*i`.

Métodos para compreensão de listas

São métodos para definir listas de forma rápida.

Considere o código abaixo:

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

que tem a resposta: `[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]`

Uma compreensão de lista tem a forma:
`lista=[expressão for membro in iterável]`.

Uma `expressão` pode ser:

- Um membro que está sendo iterado.
- Uma chamada a um método.
- Expressão válida que retorne um valor.

No caso `2*i` é uma expressão que envolve a variável `i` e retorna um valor, a partir de então se coloca para iterar a variável `i`. A variável `i` receberá os valores $0, 1, \cdots, 9$ e alimentará a lista com valor `2*i`.

Métodos para compreensão de listas

Voltando à compreensão de lista:

`lista=[expressão for membro in iterável].`

O `iterável` é a lista, ou set, ou a sequência, ou gerador, ou qualquer outro objeto que possa retornar uma lista de objetos, um por vez.

Em `nova_lista = [2*i for i in range(10)]` usou-se o gerador `range` para gerar uma lista de 10 elementos, de 0 até 9. Cada vez que `range` gera um valor, este valor é passado para `i`, que é incluído na lista de acordo com a `expressão 2*i`.

O `membro` pode ser o objeto ou o valor dentro do iterável, observe a compreensão abaixo:

```
nomes = ['Larissa', "Rafael", "Marcos", "Joao"]
nomes_apr = [nome+" APROVADO" for nome in nomes]
print(nomes_apr)
```

e a saída:

```
['Larissa APROVADO', 'Rafael APROVADO', 'Marcos APROVADO', 'Joao APROVADO']
```

Métodos para compreensão de listas

Voltando à compreensão de lista:
`lista=[expressão for membro in iterável]`.

O `iterável` é a lista, ou set, ou a sequência, ou gerador, ou qualquer outro objeto que possa retornar uma lista de objetos, um por vez.

Em `nova_lista = [2*i for i in range(10)]` usou-se o gerador `range` para gerar uma lista de 10 elementos, de 0 até 9. Cada vez que `range` gera um valor, este valor é passado para `i`, que é incluído na lista de acordo com a `expressão 2*i`.

O `membro` pode ser o objeto ou o valor dentro do iterável, observe a compreensão abaixo:

```
nomes = ['Larissa', "Rafael", "Marcos", "Joao"]
nomes_apr = [nome+" APROVADO" for nome in nomes]
print(nomes_apr)
```

e a saída:

```
['Larissa APROVADO', 'Rafael APROVADO', 'Marcos APROVADO', 'Joao APROVADO']
```

Métodos para compreensão de listas

Voltando à compreensão de lista:
`lista=[expressão for membro in iterável]`.

O `iterável` é a lista, ou set, ou a sequência, ou gerador, ou qualquer outro objeto que possa retornar uma lista de objetos, um por vez.

Em `nova_lista = [2*i for i in range(10)]` usou-se o gerador `range` para gerar uma lista de 10 elementos, de 0 até 9. Cada vez que `range` gera um valor, este valor é passado para `i`, que é incluído na lista de acordo com a `expressão 2*i`.

O `membro` pode ser o objeto ou o valor dentro do iterável, observe a compreensão abaixo:

```
nomes = ['Larissa', "Rafael", "Marcos", "Joao"]
nomes_apr = [nome+" APROVADO" for nome in nomes]
print(nomes_apr)
```

e a saída:

```
['Larissa APROVADO', 'Rafael APROVADO', 'Marcos APROVADO', 'Joao APROVADO']
```

Métodos para compreensão de listas

Voltando à compreensão de lista:
`lista=[expressão for membro in iterável]`.

O `iterável` é a lista, ou set, ou a sequência, ou gerador, ou qualquer outro objeto que possa retornar uma lista de objetos, um por vez.

Em `nova_lista = [2*i for i in range(10)]` usou-se o gerador `range` para gerar uma lista de 10 elementos, de 0 até 9. Cada vez que `range` gera um valor, este valor é passado para `i`, que é incluído na lista de acordo com a `expressão 2*i`.

O `membro` pode ser o objeto ou o valor dentro do iterável, observe a compreensão abaixo:

```
nomes = ['Larissa', "Rafael", "Marcos", "Joao"]
nomes_apr = [nome+" APROVADO" for nome in nomes]
print(nomes_apr)
```

e a saída:

```
['Larissa APROVADO', 'Rafael APROVADO', 'Marcos APROVADO', 'Joao APROVADO']
```

Métodos para compreensão de listas

Voltando à compreensão de lista:
`lista=[expressão for membro in iterável]`.

O `iterável` é a lista, ou set, ou a sequência, ou gerador, ou qualquer outro objeto que possa retornar uma lista de objetos, um por vez.

Em `nova_lista = [2*i for i in range(10)]` usou-se o gerador `range` para gerar uma lista de 10 elementos, de 0 até 9. Cada vez que `range` gera um valor, este valor é passado para `i`, que é incluído na lista de acordo com a `expressão 2*i`.

O `membro` pode ser o objeto ou o valor dentro do iterável, observe a compreensão abaixo:

```
nomes = ['Larissa', "Rafael", "Marcos", "Joao"]
nomes_apr = [nome+" APROVADO" for nome in nomes]
print(nomes_apr)
```

e a saída:

```
['Larissa APROVADO', 'Rafael APROVADO', 'Marcos APROVADO', 'Joao APROVADO']
```

Métodos para compreensão de listas

Considere o bloco de código:

```
nomes = ['Larissa', "Rafael", "Marcos", "Joao"]
def aprovar_pessoa(nome):
    return nome+" Aprovado"
```

Podemos utilizar na compreensão de lista:

- Uma função como expressão:

```
nomes = [aprovar_pessoa(nome) for nome in nomes]
print(nomes)
```

- Uma condicional na expressão:

```
nomes2 = [aprovar_pessoa(nome) for nome in nomes if (nome!='Rafael')]
print(nomes2)
```

  ou com números:

```
numeros = [i for i in range(20) if i not in ([1,5,15,19]and(0,2,18))]
print(numeros)
```

Métodos para compreensão de listas

Considere o bloco de código:

```
nomes = ['Larissa', "Rafael", "Marcos", "Joao"]
def aprovar_pessoa(nome):
    return nome+" Aprovado"
```

Podemos utilizar na compreensão de lista:

- Uma função como expressão:

  ```
  nomes = [aprovar_pessoa(nome) for nome in nomes]
  print(nomes)
  ```

- Uma condicional na expressão:

  ```
  nomes2 = [aprovar_pessoa(nome) for nome in nomes if (nome!='Rafael')]
  print(nomes2)
  ```

  ou com números:

  ```
  numeros = [i for i in range(20) if i not in ([1,5,15,19]and(0,2,18))]
  print(numeros)
  ```

Métodos para compreensão de listas

Considere o bloco de código:

```
nomes = ['Larissa', "Rafael", "Marcos", "Joao"]
def aprovar_pessoa(nome):
    return nome+" Aprovado"
```

Podemos utilizar na compreensão de lista:

- Uma função como expressão:

  ```
  nomes = [aprovar_pessoa(nome) for nome in nomes]
  print(nomes)
  ```

- Uma condicional na expressão:

  ```
  nomes2 = [aprovar_pessoa(nome) for nome in nomes if (nome!='Rafael')]
  print(nomes2)
  ```

  ou com números:

  ```
  numeros = [i for i in range(20) if i not in ([1,5,15,19]and(0,2,18))]
  print(numeros)
  ```

Métodos para compreensão de listas

Podemos usar uma função na condicional. Se quisermos somente pares entre 0 e 20 na lista:

```
def eh_par(num):
    if num%2:
        return False
    else:
        return True
num2 = [i for i in range(20) if eh_par(i)]
print(num2)
```

Se quisermos números ímpares ao contrário:

```
num2 = [i for i in range(20) if not(eh_par(i))]
print(num2)
```

Podemos inclusive inserir uma condicional mais completa, com else:

```
Participantes = ['Larissa', "Rafael", "Marcos", "Joao", 'Carla', "Cassio",
                 "Mariana", "Salua"]
Selecionados = ["Salua", "Cassio", "Marcos"]
Resultado = [(nome+" Selecionado" if nome in Selecionados else
              nome +" Excluido" ) for nome in Participantes]
```

Métodos para compreensão de listas

Podemos usar uma função na condicional. Se quisermos somente pares entre 0 e 20 na lista:

```
def eh_par(num):
    if num%2:
        return False
    else:
        return True
num2 = [i for i in range(20) if eh_par(i)]
print(num2)
```

Se quisermos números ímpares ao contrário:

```
num2 = [i for i in range(20) if not(eh_par(i))]
print(num2)
```

Podemos inclusive inserir uma condicional mais completa, com else:

```
Participantes = ['Larissa', "Rafael", "Marcos", "Joao", 'Carla', "Cassio",
                 "Mariana", "Salua"]
Selecionados = ["Salua", "Cassio", "Marcos"]
Resultado = [(nome+" Selecionado" if nome in Selecionados else
              nome +" Excluido" ) for nome in Participantes]
```

Métodos para compreensão de listas

Podemos usar uma função na condicional. Se quisermos somente pares entre 0 e 20 na lista:

```
def eh_par(num):
    if num%2:
        return False
    else:
        return True
num2 = [i for i in range(20) if eh_par(i)]
print(num2)
```

Se quisermos números ímpares ao contrário:

```
num2 = [i for i in range(20) if not(eh_par(i))]
print(num2)
```

Podemos inclusive inserir uma condicional mais completa, com else:

```
Participantes = ['Larissa', "Rafael", "Marcos", "Joao", 'Carla', "Cassio",
                 "Mariana", "Salua"]
Selecionados = ["Salua", "Cassio", "Marcos"]
Resultado = [(nome+" Selecionado" if nome in Selecionados else
              nome +" Excluido" ) for nome in Participantes]
```

Métodos para compreensão de listas

Não é possível, todavia, fazer mais que uma condicional, com elif, por exemplo. Se quiséssemos distribuir medalhas, teríamos que fazer:

```
Participantes = ['Larissa', "Rafael", "Marcos", "Joao", 'Carla', "
                 "Mariana", "Salua"]
Ouro = ["Salua", "Cassio", "Marcos"]
Prata = ["Mariana"]
Bronze = ["Larissa"]
Resultado2 = []

for nome in Participantes:
    if nome in Ouro:
        nome += " Ouro"
    elif nome in Prata:
        nome+= " Prata"
    elif nome in Bronze:
        nome+=" Bronze"
    else:
        nome+=" não medalhou"
    Resultado2.append(nome)
print(Resultado2)
```

Em um caso como este não se consegue aninhar elif, podemos somente ter um else.

Métodos para compreensão de listas

Não é possível, todavia, fazer mais que uma condicional, com elif, por exemplo. Se quiséssemos distribuir medalhas, teríamos que fazer:

```
Participantes = ['Larissa', "Rafael", "Marcos", "Joao", 'Carla', "
                 "Mariana", "Salua"]
Ouro = ["Salua", "Cassio", "Marcos"]
Prata = ["Mariana"]
Bronze = ["Larissa"]
Resultado2 = []

for nome in Participantes:
    if nome in Ouro:
        nome += " Ouro"
    elif nome in Prata:
        nome+= " Prata"
    elif nome in Bronze:
        nome+=" Bronze"
    else:
        nome+=" não medalhou"
    Resultado2.append(nome)
print(Resultado2)
```

Em um caso como este não se consegue aninhar elif, podemos somente ter um else.

Métodos para compreensão de listas

Não é possível, todavia, fazer mais que uma condicional, com elif, por exemplo. Se quiséssemos distribuir medalhas, teríamos que fazer:

```
Participantes = ['Larissa', "Rafael", "Marcos", "Joao", 'Carla', "
                 "Mariana", "Salua"]
Ouro = ["Salua", "Cassio", "Marcos"]
Prata = ["Mariana"]
Bronze = ["Larissa"]
Resultado2 = []

for nome in Participantes:
    if nome in Ouro:
        nome += " Ouro"
    elif nome in Prata:
        nome+= " Prata"
    elif nome in Bronze:
        nome+=" Bronze"
    else:
        nome+=" não medalhou"
    Resultado2.append(nome)
print(Resultado2)
```

Em um caso como este não se consegue aninhar elif, podemos somente ter um else.

Aplicação com Any e All

As funções `any` e `all` servem para verificação de itens em vetores python.

all verifica se todos os itens são verdadeiros.

any verifica se pelo menos algum item é verdadeiro.

```
resultado_encontrado = [False, False, True, False, False]
resultado_esperado = [True,True,True,True,True]
if any(resultado_encontrado):
    print("Any 1 Sim")
else:
    print("Any 1 Não")

if all(resultado_encontrado):
    print("All 2 Sim")
else:
    print("All 2 Não")

if all(resultado_esperado):
    print("All 3 Sim")
else:
    print("All 3 Não")
```

Aplicação com Any e All

As funções `any` e `all` servem para verificação de itens em vetores python.

all verifica se todos os itens são verdadeiros.

any verifica se pelo menos algum item é verdadeiro.

```
resultado_encontrado = [False, False, True, False, False]
resultado_esperado = [True,True,True,True,True]
if any(resultado_encontrado):
    print("Any 1 Sim")
else:
    print("Any 1 Não")

if all(resultado_encontrado):
    print("All 2 Sim")
else:
    print("All 2 Não")

if all(resultado_esperado):
    print("All 3 Sim")
else:
    print("All 3 Não")
```

Aplicação com Any e All

As funções `any` e `all` servem para verificação de itens em vetores python.

all verifica se todos os itens são verdadeiros.

any verifica se pelo menos algum item é verdadeiro.

```
resultado_encontrado = [False, False, True, False, False]
resultado_esperado = [True,True,True,True,True]
if any(resultado_encontrado):
    print("Any 1 Sim")
else:
    print("Any 1 Não")

if all(resultado_encontrado):
    print("All 2 Sim")
else:
    print("All 2 Não")

if all(resultado_esperado):
    print("All 3 Sim")
else:
    print("All 3 Não")
```

Aplicação com Any e All

As funções `any` e `all` servem para verificação de itens em vetores python.

all verifica se todos os itens são verdadeiros.

any verifica se pelo menos algum item é verdadeiro.

```
resultado_encontrado = [False, False, True, False, False]
resultado_esperado = [True,True,True,True,True]
if any(resultado_encontrado):
    print("Any 1 Sim")
else:
    print("Any 1 Não")

if all(resultado_encontrado):
    print("All 2 Sim")
else:
    print("All 2 Não")

if all(resultado_esperado):
    print("All 3 Sim")
else:
    print("All 3 Não")
```

Aplicação com Any e All

Podemos usar compreensão de listas para acelerar um processo de verificação.

Suponha que se queira verificar se um cliente de um banco entrou em algum momento.

```
Saldo = [10,100,-19, 230, 490, 1000]
#Verifica se houve algum valor negativo
if any([item<0 for item in Saldo]):
    print("Houve saldo negativo")
else:
    print("Não houve saldo negativo")

if all([item>=0 for item in Saldo]):
    print("Não houve negativo")
else:
    print("Houve saldo negativo")
```

Exemplos

Tente fazer os exemplos:

- Ex1: Usando compreensão de lista, criar a lista `[2,4,6,8,10]`
- Ex2: Usando a lista abaixo como base:

```
cores = ["vermelho", "azul", "verde", "amarelo", "rosa", "preto"]
```

use compreensão de lista para criar a lista:

```
cores = ["1 - vermelho", "2 - azul", "3 - verde",
    "4 - amarelo", "5 - rosa", "6 - preto"]
```

- Ex3: Usando a lista a seguir como base:

```
participantes = ["joel", "jessica", "maria", "cris",
                 "Larissa", "rafael", "marcus", "john"]
pagamento_realizado = ["joel", "jessica", "maria", "cris"]
```

concatene a palavra `PAGO` aos nomes da lista `participantes` caso o nome esteja na lista `pagamento_realizado`.

Exemplos

Tente fazer os exemplos:

- Ex1: Usando compreensão de lista, criar a lista `[2,4,6,8,10]`
- Ex2: Usando a lista abaixo como base:

```
cores = ["vermelho", "azul", "verde", "amarelo", "rosa", "preto"]
```

use compreensão de lista para criar a lista:

```
cores = ["1 - vermelho", "2 - azul", "3 - verde",
    "4 - amarelo", "5 - rosa", "6 - preto"]
```

- Ex3: Usando a lista a seguir como base:

```
participantes = ["joel", "jessica", "maria", "cris",
                 "Larissa", "rafael", "marcus", "john"]
pagamento_realizado = ["joel", "jessica", "maria", "cris"]
```

concatene a palavra `PAGO` aos nomes da lista `participantes` caso o nome esteja na lista `pagamento_realizado`.

Exemplos

Tente fazer os exemplos:

Ex1: Usando compreensão de lista, criar a lista `[2,4,6,8,10]`

Ex2: Usando a lista abaixo como base:

```
cores = ["vermelho", "azul", "verde", "amarelo", "rosa", "preto"]
```

use compreensão de lista para criar a lista:

```
cores = ["1 - vermelho", "2 - azul", "3 - verde",
    "4 - amarelo", "5 - rosa", "6 - preto"]
```

Ex3: Usando a lista a seguir como base:

```
participantes = ["joel", "jessica", "maria", "cris",
                 "Larissa", "rafael", "marcus", "john"]
pagamento_realizado = ["joel", "jessica", "maria", "cris"]
```

concatene a palavra `PAGO` aos nomes da lista `participantes` caso o nome esteja na lista `pagamento_realizado`.

Solução dos exemplos

```
from pprint import*
lista = [2*(i+1) for i in range(5)]
print(lista)
cores = ["vermelho", "azul", "verde", "amarelo", "rosa", "preto"]
lista2 = [ str(i+1)+" - "+cores[i]  for i in range(6)]
pprint(lista2)
#Forma alternativa
lista3 = [str(cores.index(cor)+1)+' - '+ cor for cor in cores]
pprint(lista3)
participantes = ["joel", "jessica", "maria", "cris",
                 "Larissa", "rafael", "marcus", "john"]
pagamento_realizado = ["joel", "jessica", "maria", "cris"]
Pagos = [(nome+" PAGO" if nome in pagamento_realizado else nome)
         for nome in participantes]
pprint(Pagos)
```

Sumário

Utilização de map

O comando map altera uma lista, evitando o uso de loop para fazer uma alteração ou verificação.

Considere, por exemplo, o vetor:

```
nomes = ['Larissa', 'Rafael', 'Marcos','Joao']
```

O comando `map` insere por referência uma função que será aplicada a todos os itens da lista. Queremos inserir a palavra "APROVADO" aos itens da lista. Apresentamos duas formas a partir da função `aprovar_pessoa`:

```
def aprovar_pessoa(nome):
    return nome+" APROVADO"
```

Primeiro através de um laço:

```
for i in range(len(nomes)):
    nomes[i] = aprovar_pessoa(nomes[i])
```

Neste caso altera-se o próprio vetor `nomes`. Preferindo um novo:

```
Nomes_Aprovados = []
for i in range(len(nomes)):
    Nomes_Aprovados.append(aprovar_pessoa(nomes[i]))
```

Utilização de map

O comando map altera uma lista, evitando o uso de loop para fazer uma alteração ou verificação.
Considere, por exemplo, o vetor:

```
nomes = ['Larissa', 'Rafael', 'Marcos','Joao']
```

O comando `map` insere por referência uma função que será aplicada a todos os itens da lista. Queremos inserir a palavra "APROVADO" aos itens da lista. Apresentamos duas formas a partir da função `aprovar_pessoa`:

```
def aprovar_pessoa(nome):
    return nome+" APROVADO"
```

Primeiro através de um laço:

```
for i in range(len(nomes)):
    nomes[i] = aprovar_pessoa(nomes[i])
```

Neste caso altera-se o próprio vetor `nomes`. Preferindo um novo:

```
Nomes_Aprovados = []
for i in range(len(nomes)):
    Nomes_Aprovados.append(aprovar_pessoa(nomes[i]))
```

Utilização de map

O comando map altera uma lista, evitando o uso de loop para fazer uma alteração ou verificação.
Considere, por exemplo, o vetor:

```
nomes = ['Larissa', 'Rafael', 'Marcos','Joao']
```

O comando `map` insere por referência uma função que será aplicada a todos os itens da lista. Queremos inserir a palavra "APROVADO" aos itens da lista. Apresentamos duas formas a partir da função `aprovar_pessoa`:

```
def aprovar_pessoa(nome):
    return nome+" APROVADO"
```

Primeiro através de um laço:

```
for i in range(len(nomes)):
    nomes[i] = aprovar_pessoa(nomes[i])
```

Neste caso altera-se o próprio vetor `nomes`. Preferindo um novo:

```
Nomes_Aprovados = []
for i in range(len(nomes)):
    Nomes_Aprovados.append(aprovar_pessoa(nomes[i]))
```

Utilização de map

O comando map altera uma lista, evitando o uso de loop para fazer uma alteração ou verificação.
Considere, por exemplo, o vetor:

```
nomes = ['Larissa', 'Rafael', 'Marcos','Joao']
```

O comando `map` insere por referência uma função que será aplicada a todos os itens da lista. Queremos inserir a palavra "APROVADO" aos itens da lista. Apresentamos duas formas a partir da função `aprovar_pessoa`:

```
def aprovar_pessoa(nome):
    return nome+" APROVADO"
```

Primeiro através de um laço:

```
for i in range(len(nomes)):
    nomes[i] = aprovar_pessoa(nomes[i])
```

Neste caso altera-se o próprio vetor `nomes`. Preferindo um novo:

```
Nomes_Aprovados = []
for i in range(len(nomes)):
    Nomes_Aprovados.append(aprovar_pessoa(nomes[i]))
```

Utilização de map

Uso do comando `map`

Uma segunda forma é utilizar o comando `map`, que **recebe por referência** a função e aplica a todos os itens do vetor.

Primeiro caso: Criando novo vetor

```
situacao = list(map(aprovar_pessoa,nomes))
print(situacao)
print(nomes)
```

Segundo caso: Alterando o próprio vetor

```
nomes = list(map(aprovar_pessoa,nomes))
print(nomes)
```

Observe que `map` recebe por referência a função, isto é, sem os parênteses. O resultado é:

```
['Larissa APROVADO', 'Rafael APROVADO', 'Marcos APROVADO', 'Joao APROVADO']
['Larissa', 'Rafael', 'Marcos', 'Joao']
```

Utilização de map

Uso do comando `map`

Uma segunda forma é utilizar o comando `map`, que **recebe por referência** a função e aplica a todos os itens do vetor.

Primeiro caso: Criando novo vetor

```
situacao = list(map(aprovar_pessoa,nomes))
print(situacao)
print(nomes)
```

Segundo caso: Alterando o próprio vetor

```
nomes = list(map(aprovar_pessoa,nomes))
print(nomes)
```

Observe que `map` recebe por referência a função, isto é, sem os parênteses.
O resultado é:

```
['Larissa APROVADO', 'Rafael APROVADO', 'Marcos APROVADO', 'Joao APROVADO']
['Larissa', 'Rafael', 'Marcos', 'Joao']
```

Utilização de map

Uso do comando `map`

Uma segunda forma é utilizar o comando `map`, que **recebe por referência** a função e aplica a todos os itens do vetor.

Primeiro caso: Criando novo vetor

```
situacao = list(map(aprovar_pessoa,nomes))
print(situacao)
print(nomes)
```

Segundo caso: Alterando o próprio vetor

```
nomes = list(map(aprovar_pessoa,nomes))
print(nomes)
```

Observe que `map` recebe por referência a função, isto é, sem os parênteses. O resultado é:

```
['Larissa APROVADO', 'Rafael APROVADO', 'Marcos APROVADO', 'Joao APROVADO']
['Larissa', 'Rafael', 'Marcos', 'Joao']
```

Utilização de map

Uso do comando `map`

Uma segunda forma é utilizar o comando `map`, que **recebe por referência** a função e aplica a todos os itens do vetor.

Primeiro caso: Criando novo vetor

```
situacao = list(map(aprovar_pessoa,nomes))
print(situacao)
print(nomes)
```

Segundo caso: Alterando o próprio vetor

```
nomes = list(map(aprovar_pessoa,nomes))
print(nomes)
```

Observe que `map` recebe por referência a função, isto é, sem os parênteses.

O resultado é:

```
['Larissa APROVADO', 'Rafael APROVADO', 'Marcos APROVADO', 'Joao APROVADO']
['Larissa', 'Rafael', 'Marcos', 'Joao']
```

Utilização de map

Uso do comando `map`

Uma segunda forma é utilizar o comando `map`, que **recebe por referência** a função e aplica a todos os itens do vetor.

Primeiro caso: Criando novo vetor

```
situacao = list(map(aprovar_pessoa,nomes))
print(situacao)
print(nomes)
```

Segundo caso: Alterando o próprio vetor

```
nomes = list(map(aprovar_pessoa,nomes))
print(nomes)
```

Observe que `map` recebe por referência a função, isto é, sem os parênteses. O resultado é:

```
['Larissa APROVADO', 'Rafael APROVADO', 'Marcos APROVADO', 'Joao APROVADO']
['Larissa', 'Rafael', 'Marcos', 'Joao']
```

Aplicação da classe map

Uma boa aplicação de `map` ocorre quando queremos converter todos os tipos recebidos em arquivo, ou via terminal em uma só linha, para float ou int.

```
A = input('Entre com uma quantidade de numeros: ').split()
print(A)
A = list(map(float, A))
print(A)
```

O programa desta forma pede que o usuário insira uma quantidade qualquer de números no terminal (e somente após pressionar `enter`) e depois o código faz a conversão para `float`, usando a função `float()`.

```
Entre com uma quantidade de numeros: 45.3 45 78.98 -42.9 0.02
['45.3', '45', '78.98', '-42.9', '0.02']
[45.3, 45.0, 78.98, -42.9, 0.02]
```

Uma alternativa seria:

```
A = input('Entre com uma quantidade de numeros: ').split()
for i in range(len(A)):
    A[i] = float(A[i])
print(A)
```

Aplicação da classe map

Uma boa aplicação de `map` ocorre quando queremos converter todos os tipos recebidos em arquivo, ou via terminal em uma só linha, para float ou int.

```
A = input('Entre com uma quantidade de numeros: ').split()
print(A)
A = list(map(float, A))
print(A)
```

O programa desta forma pede que o usuário insira uma quantidade qualquer de números no terminal (e somente após pressionar `enter`) e depois o código faz a conversão para `float`, usando a função `float()`.

```
Entre com uma quantidade de numeros: 45.3 45 78.98 -42.9 0.02
['45.3', '45', '78.98', '-42.9', '0.02']
[45.3, 45.0, 78.98, -42.9, 0.02]
```

Uma alternativa seria:

```
A = input('Entre com uma quantidade de numeros: ').split()
for i in range(len(A)):
    A[i] = float(A[i])
print(A)
```

Aplicação da classe map

Uma boa aplicação de `map` ocorre quando queremos converter todos os tipos recebidos em arquivo, ou via terminal em uma só linha, para float ou int.

```
A = input('Entre com uma quantidade de numeros: ').split()
print(A)
A = list(map(float, A))
print(A)
```

O programa desta forma pede que o usuário insira uma quantidade qualquer de números no terminal (e somente após pressionar `enter`) e depois o código faz a conversão para `float`, usando a função `float()`.

```
Entre com uma quantidade de numeros: 45.3 45 78.98 -42.9 0.02
['45.3', '45', '78.98', '-42.9', '0.02']
[45.3, 45.0, 78.98, -42.9, 0.02]
```

Uma alternativa seria:

```
A = input('Entre com uma quantidade de numeros: ').split()
for i in range(len(A)):
    A[i] = float(A[i])
print(A)
```

Aplicação da classe map

Uma boa aplicação de `map` ocorre quando queremos converter todos os tipos recebidos em arquivo, ou via terminal em uma só linha, para float ou int.

```
A = input('Entre com uma quantidade de numeros: ').split()
print(A)
A = list(map(float, A))
print(A)
```

O programa desta forma pede que o usuário insira uma quantidade qualquer de números no terminal (e somente após pressionar `enter`) e depois o código faz a conversão para `float`, usando a função `float()`.

```
Entre com uma quantidade de numeros: 45.3 45 78.98 -42.9 0.02
['45.3', '45', '78.98', '-42.9', '0.02']
[45.3, 45.0, 78.98, -42.9, 0.02]
```

Uma alternativa seria:

```
A = input('Entre com uma quantidade de numeros: ').split()
for i in range(len(A)):
    A[i] = float(A[i])
print(A)
```

Aplicação da classe filter

A classe `filter` difere da classe `map` porque exige que se utilize uma função que retorna `True` para algo que se deseja filtrar.

O resultado é uma variável tipo filter, que pode ser convertida para lista, assim como map.

Considere que uma pintura clássica para ser antiguidade deve ser anterior a 1900. Vamos separá-las do vetor pinturas abaixo. Observe que pinturas é um vetor de vetores.

```
pinturas = [
    ['Picasso', 'Les demoiselles', 1907],
    ['Monet', "Lagoa dos lirios d'água", 1899],
    ['Renoir', "Duas irmãs", 1881] ,
    ['Tarcila', 'Abaporu', 1928]  ]
def antiguidade(pintura):
    if pintura[2]<1900:
        return True
    else:
        return False
print("Filter")
print(list(filter(antiguidade, pinturas)))
print("Map")
print(list(map(antiguidade, pinturas)))
```

A classe `filter` criou um novo vetor de vetores, filtrado pela função `antiguidades`. Já `map` vai separar em verdadeiro ou falso.

Aplicação da classe filter

A classe `filter` difere da classe `map` porque exige que se utilize uma função que retorna `True` para algo que se deseja filtrar.

O resultado é uma variável tipo filter, que pode ser convertida para lista, assim como map.

Considere que uma pintura clássica para ser antiguidade deve ser anterior a 1900. Vamos separá-las do vetor pinturas abaixo. Observe que pinturas é um vetor de vetores.

```
pinturas = [
    ['Picasso', 'Les demoiselles', 1907],
    ['Monet', "Lagoa dos lirios d'água", 1899],
    ['Renoir', "Duas irmãs", 1881] ,
    ['Tarcila', 'Abaporu', 1928]  ]
def antiguidade(pintura):
    if pintura[2]<1900:
        return True
    else:
        return False
print("Filter")
print(list(filter(antiguidade, pinturas)))
print("Map")
print(list(map(antiguidade, pinturas)))
```

A classe `filter` criou um novo vetor de vetores, filtrado pela função `antiguidades`. Já `map` vai separar em verdadeiro ou falso.

Aplicação da classe filter

A classe `filter` difere da classe `map` porque exige que se utilize uma função que retorna `True` para algo que se deseja filtrar.

O resultado é uma variável tipo filter, que pode ser convertida para lista, assim como map.

Considere que uma pintura clássica para ser antiguidade deve ser anterior a 1900. Vamos separá-las do vetor pinturas abaixo. Observe que pinturas é um vetor de vetores.

```
pinturas = [
    ['Picasso', 'Les demoiselles', 1907],
    ['Monet', "Lagoa dos lirios d'água", 1899],
    ['Renoir', "Duas irmãs", 1881] ,
    ['Tarcila', 'Abaporu', 1928]  ]
def antiguidade(pintura):
    if pintura[2]<1900:
        return True
    else:
        return False
print("Filter")
print(list(filter(antiguidade, pinturas)))
print("Map")
print(list(map(antiguidade, pinturas)))
```

A classe `filter` criou um novo vetor de vetores, filtrado pela função `antiguidades`. Já `map` vai separar em verdadeiro ou falso.

Aplicação da classe filter

A classe `filter` difere da classe `map` porque exige que se utilize uma função que retorna `True` para algo que se deseja filtrar.

O resultado é uma variável tipo filter, que pode ser convertida para lista, assim como map.

Considere que uma pintura clássica para ser antiguidade deve ser anterior a 1900. Vamos separá-las do vetor pinturas abaixo. Observe que pinturas é um vetor de vetores.

```
pinturas = [
    ['Picasso', 'Les demoiselles', 1907],
    ['Monet', "Lagoa dos lirios d'água", 1899],
    ['Renoir', "Duas irmãs", 1881] ,
    ['Tarcila', 'Abaporu', 1928]  ]
def antiguidade(pintura):
    if pintura[2]<1900:
        return True
    else:
        return False
print("Filter")
print(list(filter(antiguidade, pinturas)))
print("Map")
print(list(map(antiguidade, pinturas)))
```

A classe `filter` criou um novo vetor de vetores, filtrado pela função `antiguidades`. Já `map` vai separar em verdadeiro ou falso.

Sumário

Aplicação da classe set

o tipo `set` é mutável, pode receber, remover e alterar elementos, com a diferença que os elementos são únicos no vetor.

A classe `set` é definida entre chaves, mas pode ser convertida de lista.

```
numeros = [2,2,5,8]     #O número 2 aparece duplicado
set_numeros = set(numeros) #Pode alterar a ordenação original.
print(set_numeros)
frutas = {'maca', 'uva', 'banana', 'maca', 'morango'}
print(frutas)
```

Métodos de sets:

```
a = {2,2,5,8}
b = {2,2,3,9}
print(a.intersection(b))
print(a.symmetric_difference(b))
print(a.union(b))
print(a.intersection(b))
```

```
{2}
{3, 5, 8, 9}
{2, 3, 5, 8, 9}
{2}
```

Alguns os métodos para sets: Link e Link

Cuidado: para criar um set vazio usamos `set_vazio=set()`, e não `set_vazio={}`, pois neste caso teríamos um dicionário (aula posterior).

Aplicação da classe set

o tipo `set` é mutável, pode receber, remover e alterar elementos, com a diferença que os elementos são únicos no vetor.

A classe `set` é definida entre chaves, mas pode ser convertida de lista.

```
numeros = [2,2,5,8]     #O número 2 aparece duplicado
set_numeros = set(numeros) #Pode alterar a ordenação original.
print(set_numeros)
frutas = {'maca', 'uva', 'banana', 'maca', 'morango'}
print(frutas)
```

Métodos de sets:

```
a = {2,2,5,8}
b = {2,2,3,9}
print(a.intersection(b))
print(a.symmetric_difference(b))
print(a.union(b))
print(a.intersection(b))
```

```
{2}
{3, 5, 8, 9}
{2, 3, 5, 8, 9}
{2}
```

Alguns os métodos para sets: Link e Link

Cuidado: para criar um set vazio usamos `set_vazio=set()`, e não `set_vazio={}`, pois neste caso teríamos um dicionário (aula posterior).

Aplicação da classe set

o tipo `set` é mutável, pode receber, remover e alterar elementos, com a diferença que os elementos são únicos no vetor.

A classe `set` é definida entre chaves, mas pode ser convertida de lista.

```
numeros = [2,2,5,8]     #O número 2 aparece duplicado
set_numeros = set(numeros) #Pode alterar a ordenação original.
print(set_numeros)
frutas = {'maca', 'uva', 'banana', 'maca', 'morango'}
print(frutas)
```

Métodos de sets:

```
a = {2,2,5,8}
b = {2,2,3,9}
print(a.intersection(b))
print(a.symmetric_difference(b))
print(a.union(b))
print(a.intersection(b))
```

```
{2}
{3, 5, 8, 9}
{2, 3, 5, 8, 9}
{2}
```

Alguns os métodos para sets: Link e Link

Cuidado: para criar um set vazio usamos `set_vazio=set()`, e não `set_vazio={}`, pois neste caso teríamos um dicionário (aula posterior).

Aplicação da classe set

o tipo `set` é mutável, pode receber, remover e alterar elementos, com a diferença que os elementos são únicos no vetor.

A classe `set` é definida entre chaves, mas pode ser convertida de lista.

```
numeros = [2,2,5,8]     #O número 2 aparece duplicado
set_numeros = set(numeros) #Pode alterar a ordenação original.
print(set_numeros)
frutas = {'maca', 'uva', 'banana', 'maca', 'morango'}
print(frutas)
```

Métodos de sets:

```
a = {2,2,5,8}
b = {2,2,3,9}
print(a.intersection(b))
print(a.symmetric_difference(b))
print(a.union(b))
print(a.intersection(b))
```

```
{2}
{3, 5, 8, 9}
{2, 3, 5, 8, 9}
{2}
```

Alguns os métodos para sets: Link e Link

Cuidado: para criar um set vazio usamos `set_vazio=set()`, e não `set_vazio={}`, pois neste caso teríamos um dicionário (aula posterior).

Aplicação da classe set

o tipo `set` é mutável, pode receber, remover e alterar elementos, com a diferença que os elementos são únicos no vetor.

A classe `set` é definida entre chaves, mas pode ser convertida de lista.

```
numeros = [2,2,5,8]     #O número 2 aparece duplicado
set_numeros = set(numeros) #Pode alterar a ordenação original.
print(set_numeros)
frutas = {'maca', 'uva', 'banana', 'maca', 'morango'}
print(frutas)
```

Métodos de sets:

```
a = {2,2,5,8}
b = {2,2,3,9}
print(a.intersection(b))
print(a.symmetric_difference(b))
print(a.union(b))
print(a.intersection(b))
```

```
{2}
{3, 5, 8, 9}
{2, 3, 5, 8, 9}
{2}
```

Alguns os métodos para sets: Link e Link

Cuidado: para criar um set vazio usamos `set_vazio=set()`, e não `set_vazio={}`, pois neste caso teríamos um dicionário (aula posterior).

Aplicação da classe set

o tipo `set` é mutável, pode receber, remover e alterar elementos, com a diferença que os elementos são únicos no vetor.

A classe `set` é definida entre chaves, mas pode ser convertida de lista.

```
numeros = [2,2,5,8]     #O número 2 aparece duplicado
set_numeros = set(numeros) #Pode alterar a ordenação original.
print(set_numeros)
frutas = {'maca', 'uva', 'banana', 'maca', 'morango'}
print(frutas)
```

Métodos de sets:

```
a = {2,2,5,8}
b = {2,2,3,9}
print(a.intersection(b))
print(a.symmetric_difference(b))
print(a.union(b))
print(a.intersection(b))
```

```
{2}
{3, 5, 8, 9}
{2, 3, 5, 8, 9}
{2}
```

Alguns os métodos para sets: ▸ Link e ▸ Link

Cuidado: para criar um set vazio usamos `set_vazio=set()`, e não `set_vazio={}`, pois neste caso teríamos um dicionário (aula posterior).

Sumário

Exemplos

1. Crie um programa que leia uma linha contendo vários números inteiros separados por espaço e imprima a média com duas casas decimais.

Entrada	Saída
54 85 21 47 87	58.80

2. Crie um programa que leia uma linha contendo vários números inteiros separados por espaço e imprima a o maior deles.

Entrada	Saída
5 7 8 14 20 4	20

3. Beecrowd problema 2791
4. Beecrowd problema 2162
5. Beecrowd problema 2540 (tratamento de exceções - introdução ao próximo encontro.)

Exemplos

1. Crie um programa que leia uma linha contendo vários números inteiros separados por espaço e imprima a média com duas casas decimais.

Entrada	Saída
54 85 21 47 87	58.80

2. Crie um programa que leia uma linha contendo vários números inteiros separados por espaço e imprima a o maior deles.

Entrada	Saída
5 7 8 14 20 4	20

3. Beecrowd problema 2791
4. Beecrowd problema 2162
5. Beecrowd problema 2540 (tratamento de exceções - introdução ao próximo encontro.)

Exemplos

1. Crie um programa que leia uma linha contendo vários números inteiros separados por espaço e imprima a média com duas casas decimais.

Entrada	Saída
54 85 21 47 87	58.80

2. Crie um programa que leia uma linha contendo vários números inteiros separados por espaço e imprima a o maior deles.

Entrada	Saída
5 7 8 14 20 4	20

3. Beecrowd problema 2791
4. Beecrowd problema 2162
5. Beecrowd problema 2540 (tratamento de exceções - introdução ao próximo encontro.)

Exemplos

1. Crie um programa que leia uma linha contendo vários números inteiros separados por espaço e imprima a média com duas casas decimais.

Entrada	Saída
54 85 21 47 87	58.80

2. Crie um programa que leia uma linha contendo vários números inteiros separados por espaço e imprima a o maior deles.

Entrada	Saída
5 7 8 14 20 4	20

3. Beecrowd problema 2791
4. Beecrowd problema 2162
5. Beecrowd problema 2540 (tratamento de exceções - introdução ao próximo encontro.)

Exemplos

1. Crie um programa que leia uma linha contendo vários números inteiros separados por espaço e imprima a média com duas casas decimais.

Entrada	Saída
54 85 21 47 87	58.80

2. Crie um programa que leia uma linha contendo vários números inteiros separados por espaço e imprima a o maior deles.

Entrada	Saída
5 7 8 14 20 4	20

3. Beecrowd problema 2791
4. Beecrowd problema 2162
5. Beecrowd problema 2540 (tratamento de exceções - introdução ao próximo encontro.)

Exemplos

1. Crie um programa que leia uma linha contendo vários números inteiros separados por espaço e imprima a média com duas casas decimais.

Entrada	Saída
54 85 21 47 87	58.80

2. Crie um programa que leia uma linha contendo vários números inteiros separados por espaço e imprima a o maior deles.

Entrada	Saída
5 7 8 14 20 4	20

3. Beecrowd problema 2791
4. Beecrowd problema 2162
5. Beecrowd problema 2540 (tratamento de exceções - introdução ao próximo encontro.)

Exemplos

1. Crie um programa que leia uma linha contendo vários números inteiros separados por espaço e imprima a média com duas casas decimais.

Entrada	Saída
54 85 21 47 87	58.80

2. Crie um programa que leia uma linha contendo vários números inteiros separados por espaço e imprima a o maior deles.

Entrada	Saída
5 7 8 14 20 4	20

3. Beecrowd problema 2791
4. Beecrowd problema 2162
5. Beecrowd problema 2540 (tratamento de exceções - introdução ao próximo encontro.)

Sumário

Características gerais do numpy

Numpy em parte existe para compensar a lentidão de programas em Python para quantidades largas de dados, evitando o uso livre de variáveis não tipadas.

Numpy permite:

- Criação de `ndarray`, que são vetores e matrizes tipados, rápidos de operar computacionalmente.
- Funções matemáticas para operar os vetores, como exponenciais e logaritmos.
- Ferramentas para leitura e escrita de arquivos em disco, e mapeamento de memória.
- Ferramentas de Álgebra Linear, números aleatórios, transformada de Fourier etc.
- Uma api para conexão com a linguagem *C*/*C*++.

Características gerais do numpy

Numpy em parte existe para compensar a lentidão de programas em Python para quantidades largas de dados, evitando o uso livre de variáveis não tipadas.
Numpy permite:

- Criação de `ndarray`, que são vetores e matrizes tipados, rápidos de operar computacionalmente.
- Funções matemáticas para operar os vetores, como exponenciais e logaritmos.
- Ferramentas para leitura e escrita de arquivos em disco, e mapeamento de memória.
- Ferramentas de Álgebra Linear, números aleatórios, transformada de Fourier etc.
- Uma api para conexão com a linguagem $C/C++$.

Características gerais do numpy

Numpy em parte existe para compensar a lentidão de programas em Python para quantidades largas de dados, evitando o uso livre de variáveis não tipadas.
Numpy permite:

- Criação de `ndarray`, que são vetores e matrizes tipados, rápidos de operar computacionalmente.
- Funções matemáticas para operar os vetores, como exponenciais e logaritmos.
- Ferramentas para leitura e escrita de arquivos em disco, e mapeamento de memória.
- Ferramentas de Álgebra Linear, números aleatórios, transformada de Fourier etc.
- Uma api para conexão com a linguagem $C/C++$.

Características gerais do numpy

Numpy em parte existe para compensar a lentidão de programas em Python para quantidades largas de dados, evitando o uso livre de variáveis não tipadas.
Numpy permite:

- Criação de `ndarray`, que são vetores e matrizes tipados, rápidos de operar computacionalmente.
- Funções matemáticas para operar os vetores, como exponenciais e logaritmos.
- Ferramentas para leitura e escrita de arquivos em disco, e mapeamento de memória.
- Ferramentas de Álgebra Linear, números aleatórios, transformada de Fourier etc.
- Uma api para conexão com a linguagem $C/C++$.

Características gerais do numpy

Numpy em parte existe para compensar a lentidão de programas em Python para quantidades largas de dados, evitando o uso livre de variáveis não tipadas.
Numpy permite:

- Criação de `ndarray`, que são vetores e matrizes tipados, rápidos de operar computacionalmente.
- Funções matemáticas para operar os vetores, como exponenciais e logaritmos.
- Ferramentas para leitura e escrita de arquivos em disco, e mapeamento de memória.
- Ferramentas de Álgebra Linear, números aleatórios, transformada de Fourier etc.
- Uma api para conexão com a linguagem $C/C++$.

Características gerais do numpy

Numpy em parte existe para compensar a lentidão de programas em Python para quantidades largas de dados, evitando o uso livre de variáveis não tipadas.
Numpy permite:

- Criação de `ndarray`, que são vetores e matrizes tipados, rápidos de operar computacionalmente.
- Funções matemáticas para operar os vetores, como exponenciais e logaritmos.
- Ferramentas para leitura e escrita de arquivos em disco, e mapeamento de memória.
- Ferramentas de Álgebra Linear, números aleatórios, transformada de Fourier etc.
- Uma api para conexão com a linguagem $C/C++$.

Características específicas do numpy

Numpy é uma ferramenta útil para trabalhar com ciência de dados, pois permite:

- Operações rápidas de troca de dados (de string para float por exemplo).
- Aplicações de propriedades a todos os elementos do ndarray sem necessidade de uso de `for` ou `while`.
- Expressões condicionais automáticas, sem precisar usar cadeias `if-elif-else`.
- Numpy armazena os dados internamente em blocos contíguos de memória, e por isso usa menos recursos do que vetores nativos do Python, que aceitam dados de múltiplos tipos.

OBS: Numpy não é a melhor ferramenta para uso de dados em forma tabular, para isto a ferramenta `pandas` é muito mais apropriada.

Características específicas do numpy

Numpy é uma ferramenta útil para trabalhar com ciência de dados, pois permite:

- Operações rápidas de troca de dados (de string para float por exemplo).
- Aplicações de propriedades a todos os elementos do ndarray sem necessidade de uso de `for` ou `while`.
- Expressões condicionais automáticas, sem precisar usar cadeias `if-elif-else`.
- Numpy armazena os dados internamente em blocos contíguos de memória, e por isso usa menos recursos do que vetores nativos do Python, que aceitam dados de múltiplos tipos.

OBS: Numpy não é a melhor ferramenta para uso de dados em forma tabular, para isto a ferramenta `pandas` é muito mais apropriada.

Características específicas do numpy

Numpy é uma ferramenta útil para trabalhar com ciência de dados, pois permite:

- Operações rápidas de troca de dados (de string para float por exemplo).
- Aplicações de propriedades a todos os elementos do ndarray sem necessidade de uso de `for` ou `while`.
- Expressões condicionais automáticas, sem precisar usar cadeias `if-elif-else`.
- Numpy armazena os dados internamente em blocos contíguos de memória, e por isso usa menos recursos do que vetores nativos do Python, que aceitam dados de múltiplos tipos.

OBS: Numpy não é a melhor ferramenta para uso de dados em forma tabular, para isto a ferramenta `pandas` é muito mais apropriada.

Características específicas do numpy

Numpy é uma ferramenta útil para trabalhar com ciência de dados, pois permite:

- Operações rápidas de troca de dados (de string para float por exemplo).
- Aplicações de propriedades a todos os elementos do ndarray sem necessidade de uso de `for` ou `while`.
- Expressões condicionais automáticas, sem precisar usar cadeias `if-elif-else`.
- Numpy armazena os dados internamente em blocos contíguos de memória, e por isso usa menos recursos do que vetores nativos do Python, que aceitam dados de múltiplos tipos.

OBS: Numpy não é a melhor ferramenta para uso de dados em forma tabular, para isto a ferramenta `pandas` é muito mais apropriada.

Características específicas do numpy

Numpy é uma ferramenta útil para trabalhar com ciência de dados, pois permite:

- Operações rápidas de troca de dados (de string para float por exemplo).
- Aplicações de propriedades a todos os elementos do ndarray sem necessidade de uso de `for` ou `while`.
- Expressões condicionais automáticas, sem precisar usar cadeias `if-elif-else`.
- Numpy armazena os dados internamente em blocos contíguos de memória, e por isso usa menos recursos do que vetores nativos do Python, que aceitam dados de múltiplos tipos.

OBS: Numpy não é a melhor ferramenta para uso de dados em forma tabular, para isto a ferramenta `pandas` é muito mais apropriada.

Características específicas do numpy

Numpy é uma ferramenta útil para trabalhar com ciência de dados, pois permite:

- Operações rápidas de troca de dados (de string para float por exemplo).
- Aplicações de propriedades a todos os elementos do ndarray sem necessidade de uso de `for` ou `while`.
- Expressões condicionais automáticas, sem precisar usar cadeias `if-elif-else`.
- Numpy armazena os dados internamente em blocos contíguos de memória, e por isso usa menos recursos do que vetores nativos do Python, que aceitam dados de múltiplos tipos.

OBS: Numpy não é a melhor ferramenta para uso de dados em forma tabular, para isto a ferramenta `pandas` é muito mais apropriada.

Características específicas do numpy

Vamos a uma comparação de velocidade funções built-in do python e do numpy.

Para diferenciar a função de geração de inteiros, numpy usa arange ao invés de range.

Com isto em mente, considere os códigos, que geram os mesmos resultados:

```
import numpy as np
n = 1000000

arr1 = np.arange(1000000)
arr2 = arr1*2
for i in range(3):
    print(arr2[i])
```

```
import numpy as np
n = 1000000
arr1 = list(range(n))
arr2 =[int]*n
for i in range(n):
    arr2[i] = arr1[i]*2
for i in range(3):
    print(arr2[i])
```

e as respectivas saidas, pedindo tempo de execução

```
0
2
4

real    0m0,130s
user    0m0,327s
sys     0m0,323s
```

```
0
2
4

real    0m0,285s
user    0m0,473s
sys     0m0,334s
```

Características específicas do numpy

Vamos a uma comparação de velocidade funções built-in do python e do numpy.

Para diferenciar a função de geração de inteiros, numpy usa arange ao invés de range.

Com isto em mente, considere os códigos, que geram os mesmos resultados:

```
import numpy as np
n = 1000000

arr1 = np.arange(1000000)
arr2 = arr1*2
for i in range(3):
    print(arr2[i])
```

```
import numpy as np
n = 1000000
arr1 = list(range(n))
arr2 =[int]*n
for i in range(n):
    arr2[i] = arr1[i]*2
for i in range(3):
    print(arr2[i])
```

e as respectivas saidas, pedindo tempo de execução

```
0
2
4

real    0m0,130s
user    0m0,327s
sys     0m0,323s
```

```
0
2
4

real    0m0,285s
user    0m0,473s
sys     0m0,334s
```

Características específicas do numpy

Vamos a uma comparação de velocidade funções built-in do python e do numpy.

Para diferenciar a função de geração de inteiros, numpy usa arange ao invés de range.

Com isto em mente, considere os códigos, que geram os mesmos resultados:

```python
import numpy as np
n = 1000000

arr1 = np.arange(1000000)
arr2 = arr1*2
for i in range(3):
    print(arr2[i])
```

```python
import numpy as np
n = 1000000
arr1 = list(range(n))
arr2 =[int]*n
for i in range(n):
    arr2[i] = arr1[i]*2
for i in range(3):
    print(arr2[i])
```

e as respectivas saidas, pedindo tempo de execução

```
0
2
4

real    0m0,130s
user    0m0,327s
sys     0m0,323s
```

```
0
2
4

real    0m0,285s
user    0m0,473s
sys     0m0,334s
```

Características específicas do numpy

Vamos a uma comparação de velocidade funções built-in do python e do numpy.

Para diferenciar a função de geração de inteiros, numpy usa arange ao invés de range.

Com isto em mente, considere os códigos, que geram os mesmos resultados:

```python
import numpy as np
n = 1000000

arr1 = np.arange(1000000)
arr2 = arr1*2
for i in range(3):
    print(arr2[i])
```

```python
import numpy as np
n = 1000000
arr1 = list(range(n))
arr2 =[int]*n
for i in range(n):
    arr2[i] = arr1[i]*2
for i in range(3):
    print(arr2[i])
```

e as respectivas saidas, pedindo tempo de execução

```
0
2
4

real    0m0,130s
user    0m0,327s
sys     0m0,323s
```

```
0
2
4

real    0m0,285s
user    0m0,473s
sys     0m0,334s
```

Sumário

Função arange do numpy

A função built-in do python `range(m,n,passo)` gera uma sucessão de inteiros entre os valores `m` e `n` (`n` exclusive).
O passo precisa ser um número inteiro.
Observe que o código:

```
for i in range(-6,33,3):
    print(i, end = " ")
print()
```

nos dá a saída `-6 -3 0 3 6 9 12 15 18 21 24 27 30 .`

Já o código

```
for i in range(-2,8,0.5):
    print(i, end = " ")
print()
```

gera um erro.

E o código

```
import numpy as np
for i in np.arange(-2,8,0.5):
    print(i, end = " ")
print()
```

tem como saída

`[-2,-1.5, ... , 7.5]`

Função arange do numpy

A função built-in do python `range(m,n,passo)` gera uma sucessão de inteiros entre os valores `m` e `n` (`n` exclusive).
O passo precisa ser um número inteiro.

Observe que o código:

```
for i in range(-6,33,3):
    print(i, end = " ")
print()
```

nos dá a saída `-6 -3 0 3 6 9 12 15 18 21 24 27 30 .`

Já o código

```
for i in range(-2,8,0.5):
    print(i, end = " ")
print()
```

gera um erro.

E o código

```
import numpy as np
for i in np.arange(-2,8,0.5):
    print(i, end = " ")
print()
```

tem como saída

`[-2,-1.5, ... , 7.5]`

Função arange do numpy

A função built-in do python `range(m,n,passo)` gera uma sucessão de inteiros entre os valores `m` e `n` (`n` exclusive).
O passo precisa ser um número inteiro.
Observe que o código:

```
for i in range(-6,33,3):
    print(i, end = " ")
print()
```

nos dá a saída `-6 -3 0 3 6 9 12 15 18 21 24 27 30 .`

Já o código

```
for i in range(-2,8,0.5):
    print(i, end = " ")
print()
```

gera um erro.

E o código

```
import numpy as np
for i in np.arange(-2,8,0.5):
    print(i, end = " ")
print()
```

tem como saída

`[-2,-1.5, ... , 7.5]`

Função arange do numpy

A função built-in do python `range(m,n,passo)` gera uma sucessão de inteiros entre os valores `m` e `n` (`n` exclusive).
O passo precisa ser um número inteiro.
Observe que o código:

```
for i in range(-6,33,3):
    print(i, end = " ")
print()
```

nos dá a saída `-6 -3 0 3 6 9 12 15 18 21 24 27 30 .`

Já o código

```
for i in range(-2,8,0.5):
    print(i, end = " ")
print()
```

gera um erro.

E o código

```
import numpy as np
for i in np.arange(-2,8,0.5):
    print(i, end = " ")
print()
```

tem como saída

`[-2,-1.5, ... , 7.5]`

Função arange do numpy

A função built-in do python `range(m,n,passo)` gera uma sucessão de inteiros entre os valores `m` e `n` (`n` exclusive).
O passo precisa ser um número inteiro.
Observe que o código:

```
for i in range(-6,33,3):
    print(i, end = " ")
print()
```

nos dá a saída `-6 -3 0 3 6 9 12 15 18 21 24 27 30 .`

Já o código

```
for i in range(-2,8,0.5):
    print(i, end = " ")
print()
```

gera um erro.

E o código

```
import numpy as np
for i in np.arange(-2,8,0.5):
    print(i, end = " ")
print()
```

tem como saída

`[-2,-1.5, ... , 7.5]`

Sumário

Métodos básicos

Temos várias formas de iniciar ndarrays (vetores do numpy):

1. A partir dos vetores built-in do python:

   ```
   data1 = [6, 7.5, 8, 0, 1]
   arr1 = np.array(data1)
   ```

   O problema aqui é que há dados inteiros e em ponto flutuante, e como numpy é tipado, ocorre uma conversão para dados em ponto flutuante.

   Usando a diretiva `dtype` podemos usar um tipo específico:

   ```
   arr1 = np.array([1, 2, 3], dtype=np.float64)
   ```

2. vetor de zeros:

   ```
   A = np.zeros(10)
   ```

   Cria um vetor com 10 posições nulas.

3. vetor de dados vazios:

   ```
   A=np.empty(10)
   ```

   Cuidado neste caso, porque o vetor criado pode vir ou não com termos nulos.

4. Vetor de dados unitários:

   ```
   A=np.ones(10)
   ```

Métodos básicos

Temos várias formas de iniciar ndarrays (vetores do numpy):

1. A partir dos vetores built-in do python:

   ```
   data1 = [6, 7.5, 8, 0, 1]
   arr1 = np.array(data1)
   ```

   O problema aqui é que há dados inteiros e em ponto flutuante, e como numpy é tipado, ocorre uma conversão para dados em ponto flutuante.

   Usando a diretiva `dtype` podemos usar um tipo específico:

   ```
   arr1 = np.array([1, 2, 3], dtype=np.float64)
   ```

2. vetor de zeros:

   ```
   A = np.zeros(10)
   ```

   Cria um vetor com 10 posições nulas.

3. vetor de dados vazios:

   ```
   A=np.empty(10)
   ```

   Cuidado neste caso, porque o vetor criado pode vir ou não com termos nulos.

4. Vetor de dados unitários:

   ```
   A=np.ones(10)
   ```

Métodos básicos

Temos várias formas de iniciar ndarrays (vetores do numpy):

1. A partir dos vetores built-in do python:

   ```
   data1 = [6, 7.5, 8, 0, 1]
   arr1 = np.array(data1)
   ```

   O problema aqui é que há dados inteiros e em ponto flutuante, e como numpy é tipado, ocorre uma conversão para dados em ponto flutuante.
   Usando a diretiva `dtype` podemos usar um tipo específico:

   ```
   arr1 = np.array([1, 2, 3], dtype=np.float64)
   ```

2. vetor de zeros:

   ```
   A = np.zeros(10)
   ```

   Cria um vetor com 10 posições nulas.

3. vetor de dados vazios:

   ```
   A=np.empty(10)
   ```

   Cuidado neste caso, porque o vetor criado pode vir ou não com termos nulos.

4. Vetor de dados unitários:

   ```
   A=np.ones(10)
   ```

Métodos básicos

Temos várias formas de iniciar ndarrays (vetores do numpy):

1. A partir dos vetores built-in do python:

   ```
   data1 = [6, 7.5, 8, 0, 1]
   arr1 = np.array(data1)
   ```

   O problema aqui é que há dados inteiros e em ponto flutuante, e como numpy é tipado, ocorre uma conversão para dados em ponto flutuante.
   Usando a diretiva `dtype` podemos usar um tipo específico:

   ```
   arr1 = np.array([1, 2, 3], dtype=np.float64)
   ```

2. vetor de zeros:

   ```
   A = np.zeros(10)
   ```

   Cria um vetor com 10 posições nulas.

3. vetor de dados vazios:

   ```
   A=np.empty(10)
   ```

   Cuidado neste caso, porque o vetor criado pode vir ou não com termos nulos.

4. Vetor de dados unitários:

   ```
   A=np.ones(10)
   ```

Métodos básicos

Temos várias formas de iniciar ndarrays (vetores do numpy):

1. A partir dos vetores built-in do python:

   ```
   data1 = [6, 7.5, 8, 0, 1]
   arr1 = np.array(data1)
   ```

   O problema aqui é que há dados inteiros e em ponto flutuante, e como numpy é tipado, ocorre uma conversão para dados em ponto flutuante.
   Usando a diretiva `dtype` podemos usar um tipo específico:

   ```
   arr1 = np.array([1, 2, 3], dtype=np.float64)
   ```

2. vetor de zeros:

   ```
   A = np.zeros(10)
   ```

   Cria um vetor com 10 posições nulas.

3. vetor de dados vazios:

   ```
   A=np.empty(10)
   ```

   Cuidado neste caso, porque o vetor criado pode vir ou não com termos nulos.

4. Vetor de dados unitários:

   ```
   A=np.ones(10)
   ```

Métodos especiais

Uma função parecida com a `arange` para criar arrays é a função

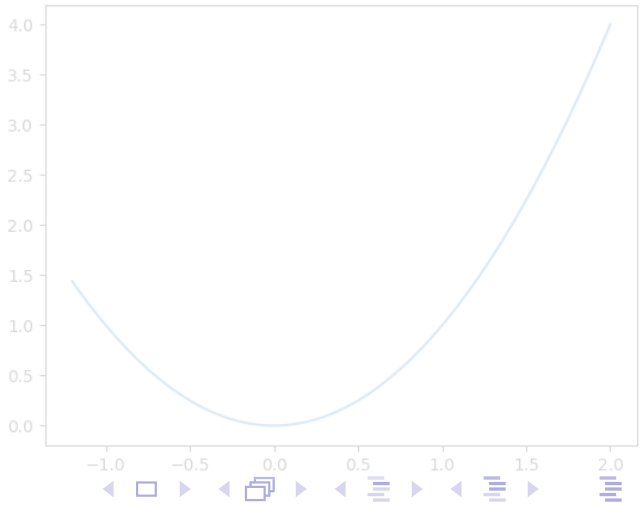
```
import numpy as np
A = np.linspace(inicio, fim, quantidade de termos)
```

Nesta função são criados números entre início e fim (ambos inclusos), com uma quantidade de termos requerida, ou seja, a função calcula o incremento necessário para incluir a quantidade de termos entre os valores inicio e fim.
`print(np.linspace(0,1.2,13))` gera a saída:

```
[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.  1.1 1.2]
```

Muito útil para construir gráficos usando matplotlib.

```
import matplotlib.pyplot as plt
x = np.linspace(-1.2,2,50)
y = x**2
plt.plot(x,y)
plt.show()
```



Métodos especiais

Uma função parecida com a `arange` para criar arrays é a função

```
import numpy as np
A = np.linspace(inicio, fim, quantidade de termos)
```

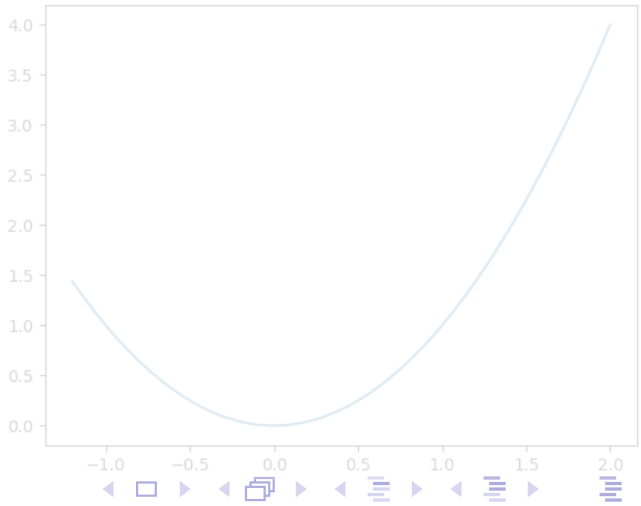
Nesta função são criados números entre início e fim (ambos inclusos), com uma quantidade de termos requerida, ou seja, a função calcula o incremento necessário para incluir a quantidade de termos entre os valores inicio e fim.

`print(np.linspace(0,1.2,13))` gera a saída:

```
[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.  1.1 1.2]
```

Muito útil para construir gráficos usando matplotlib.

```
import matplotlib.pyplot as plt
x = np.linspace(-1.2,2,50)
y = x**2
plt.plot(x,y)
plt.show()
```



Métodos especiais

Uma função parecida com a `arange` para criar arrays é a função

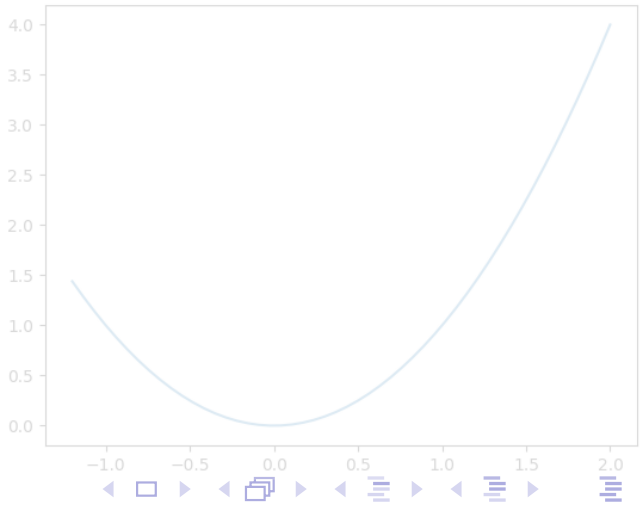
```
import numpy as np
A = np.linspace(inicio, fim, quantidade de termos)
```

Nesta função são criados números entre início e fim (ambos inclusos), com uma quantidade de termos requerida, ou seja, a função calcula o incremento necessário para incluir a quantidade de termos entre os valores inicio e fim.
`print(np.linspace(0,1.2,13))` gera a saída:

```
[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.  1.1 1.2]
```

Muito útil para construir gráficos usando matplotlib.

```
import matplotlib.pyplot as plt
x = np.linspace(-1.2,2,50)
y = x**2
plt.plot(x,y)
plt.show()
```



Métodos especiais

Uma função parecida com a `arange` para criar arrays é a função

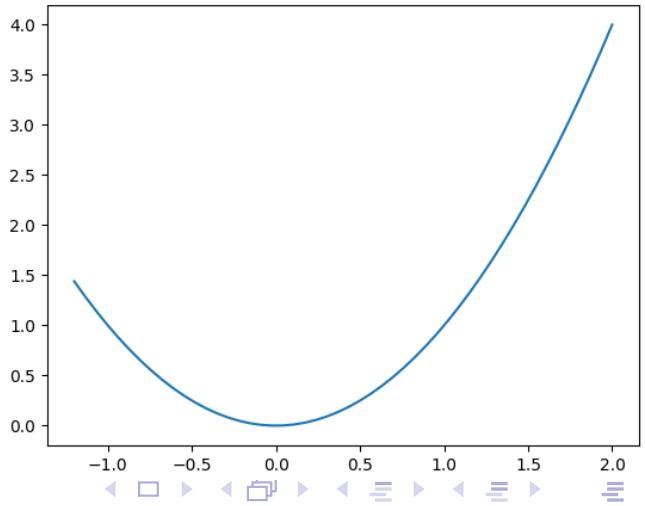
```
import numpy as np
A = np.linspace(inicio, fim, quantidade de termos)
```

Nesta função são criados números entre início e fim (ambos inclusos), com uma quantidade de termos requerida, ou seja, a função calcula o incremento necessário para incluir a quantidade de termos entre os valores inicio e fim.
`print(np.linspace(0,1.2,13))` gera a saída:

```
[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.  1.1 1.2]
```

Muito útil para construir gráficos usando matplotlib.

```
import matplotlib.pyplot as plt
x = np.linspace(-1.2,2,50)
y = x**2
plt.plot(x,y)
plt.show()
```



Tipos numéricos no numpy

Há vários tipos numéricos no numpy:

bool	Variável de 1 bit (True ou False)
int8 e uint8	Valores inteiros de 8 bits (1 byte) com ou sem sinal.
int16	Valores inteiros de 16 bits com sinal.
int32	Valores inteiros de 32 bits com sinal.
int64	Valores inteiros de 64 bits com sinal.
float16	Ponto flutuante de baixa precisão (16 bits)
float32	Ponto flutuante de média precisão (32 bits)
float64	Ponto flutuante de alta precisão (64 bits)
float128	Ponto flutuante de altíssima precisão (128 bits)
complex64	Variável complexa de média precisão.
complex128	Variável complexa de alta precisão.
complex256	Variável complexa de altíssima precisão.

Tipos numéricos no numpy

Há vários tipos numéricos no numpy:

bool	Variável de 1 bit (True ou False)
int8 e uint8	Valores inteiros de 8 bits (1 byte) com ou sem sinal.
int16	Valores inteiros de 16 bits com sinal.
int32	Valores inteiros de 32 bits com sinal.
int64	Valores inteiros de 64 bits com sinal.
float16	Ponto flutuante de baixa precisão (16 bits)
float32	Ponto flutuante de média precisão (32 bits)
float64	Ponto flutuante de alta precisão (64 bits)
float128	Ponto flutuante de altíssima precisão (128 bits)
complex64	Variável complexa de média precisão.
complex128	Variável complexa de alta precisão.
complex256	Variável complexa de altíssima precisão.

Tipos numéricos no numpy

Há vários tipos numéricos no numpy:

bool	Variável de 1 bit (True ou False)
int8 e uint8	Valores inteiros de 8 bits (1 byte) com ou sem sinal.
int16	Valores inteiros de 16 bits com sinal.
int32	Valores inteiros de 32 bits com sinal.
int64	Valores inteiros de 64 bits com sinal.
float16	Ponto flutuante de baixa precisão (16 bits)
float32	Ponto flutuante de média precisão (32 bits)
float64	Ponto flutuante de alta precisão (64 bits)
float128	Ponto flutuante de altíssima precisão (128 bits)
complex64	Variável complexa de média precisão.
complex128	Variável complexa de alta precisão.
complex256	Variável complexa de altíssima precisão.

Tipos numéricos no numpy

Há vários tipos numéricos no numpy:

bool	Variável de 1 bit (True ou False)
int8 e uint8	Valores inteiros de 8 bits (1 byte) com ou sem sinal.
int16	Valores inteiros de 16 bits com sinal.
int32	Valores inteiros de 32 bits com sinal.
int64	Valores inteiros de 64 bits com sinal.
float16	Ponto flutuante de baixa precisão (16 bits)
float32	Ponto flutuante de média precisão (32 bits)
float64	Ponto flutuante de alta precisão (64 bits)
float128	Ponto flutuante de altíssima precisão (128 bits)
complex64	Variável complexa de média precisão.
complex128	Variável complexa de alta precisão.
complex256	Variável complexa de altíssima precisão.

Tipos numéricos no numpy

Há vários tipos numéricos no numpy:

bool	Variável de 1 bit (True ou False)
int8 e uint8	Valores inteiros de 8 bits (1 byte) com ou sem sinal.
int16	Valores inteiros de 16 bits com sinal.
int32	Valores inteiros de 32 bits com sinal.
int64	Valores inteiros de 64 bits com sinal.
float16	Ponto flutuante de baixa precisão (16 bits)
float32	Ponto flutuante de média precisão (32 bits)
float64	Ponto flutuante de alta precisão (64 bits)
float128	Ponto flutuante de altíssima precisão (128 bits)
complex64	Variável complexa de média precisão.
complex128	Variável complexa de alta precisão.
complex256	Variável complexa de altíssima precisão.

Tipos numéricos no numpy

Há vários tipos numéricos no numpy:

bool	Variável de 1 bit (True ou False)
int8 e uint8	Valores inteiros de 8 bits (1 byte) com ou sem sinal.
int16	Valores inteiros de 16 bits com sinal.
int32	Valores inteiros de 32 bits com sinal.
int64	Valores inteiros de 64 bits com sinal.
float16	Ponto flutuante de baixa precisão (16 bits)
float32	Ponto flutuante de média precisão (32 bits)
float64	Ponto flutuante de alta precisão (64 bits)
float128	Ponto flutuante de altíssima precisão (128 bits)
complex64	Variável complexa de média precisão.
complex128	Variável complexa de alta precisão.
complex256	Variável complexa de altíssima precisão.

Tipos numéricos no numpy

Há vários tipos numéricos no numpy:

bool	Variável de 1 bit (True ou False)
int8 e uint8	Valores inteiros de 8 bits (1 byte) com ou sem sinal.
int16	Valores inteiros de 16 bits com sinal.
int32	Valores inteiros de 32 bits com sinal.
int64	Valores inteiros de 64 bits com sinal.
float16	Ponto flutuante de baixa precisão (16 bits)
float32	Ponto flutuante de média precisão (32 bits)
float64	Ponto flutuante de alta precisão (64 bits)
float128	Ponto flutuante de altíssima precisão (128 bits)
complex64	Variável complexa de média precisão.
complex128	Variável complexa de alta precisão.
complex256	Variável complexa de altíssima precisão.

Tipos numéricos no numpy

Há vários tipos numéricos no numpy:

bool	Variável de 1 bit (True ou False)
int8 e uint8	Valores inteiros de 8 bits (1 byte) com ou sem sinal.
int16	Valores inteiros de 16 bits com sinal.
int32	Valores inteiros de 32 bits com sinal.
int64	Valores inteiros de 64 bits com sinal.
float16	Ponto flutuante de baixa precisão (16 bits)
float32	Ponto flutuante de média precisão (32 bits)
float64	Ponto flutuante de alta precisão (64 bits)
float128	Ponto flutuante de altíssima precisão (128 bits)
complex64	Variável complexa de média precisão.
complex128	Variável complexa de alta precisão.
complex256	Variável complexa de altíssima precisão.

Tipos numéricos no numpy

Há vários tipos numéricos no numpy:

bool	Variável de 1 bit (True ou False)
int8 e uint8	Valores inteiros de 8 bits (1 byte) com ou sem sinal.
int16	Valores inteiros de 16 bits com sinal.
int32	Valores inteiros de 32 bits com sinal.
int64	Valores inteiros de 64 bits com sinal.
float16	Ponto flutuante de baixa precisão (16 bits)
float32	Ponto flutuante de média precisão (32 bits)
float64	Ponto flutuante de alta precisão (64 bits)
float128	Ponto flutuante de altíssima precisão (128 bits)
complex64	Variável complexa de média precisão.
complex128	Variável complexa de alta precisão.
complex256	Variável complexa de altíssima precisão.

Tipos numéricos no numpy

Há vários tipos numéricos no numpy:

bool	Variável de 1 bit (True ou False)
int8 e uint8	Valores inteiros de 8 bits (1 byte) com ou sem sinal.
int16	Valores inteiros de 16 bits com sinal.
int32	Valores inteiros de 32 bits com sinal.
int64	Valores inteiros de 64 bits com sinal.
float16	Ponto flutuante de baixa precisão (16 bits)
float32	Ponto flutuante de média precisão (32 bits)
float64	Ponto flutuante de alta precisão (64 bits)
float128	Ponto flutuante de altíssima precisão (128 bits)
complex64	Variável complexa de média precisão.
complex128	Variável complexa de alta precisão.
complex256	Variável complexa de altíssima precisão.

Tipos numéricos no numpy

Há vários tipos numéricos no numpy:

bool	Variável de 1 bit (True ou False)
int8 e uint8	Valores inteiros de 8 bits (1 byte) com ou sem sinal.
int16	Valores inteiros de 16 bits com sinal.
int32	Valores inteiros de 32 bits com sinal.
int64	Valores inteiros de 64 bits com sinal.
float16	Ponto flutuante de baixa precisão (16 bits)
float32	Ponto flutuante de média precisão (32 bits)
float64	Ponto flutuante de alta precisão (64 bits)
float128	Ponto flutuante de altíssima precisão (128 bits)
complex64	Variável complexa de média precisão.
complex128	Variável complexa de alta precisão.
complex256	Variável complexa de altíssima precisão.

Tipos numéricos no numpy

Há vários tipos numéricos no numpy:

bool	Variável de 1 bit (True ou False)
int8 e uint8	Valores inteiros de 8 bits (1 byte) com ou sem sinal.
int16	Valores inteiros de 16 bits com sinal.
int32	Valores inteiros de 32 bits com sinal.
int64	Valores inteiros de 64 bits com sinal.
float16	Ponto flutuante de baixa precisão (16 bits)
float32	Ponto flutuante de média precisão (32 bits)
float64	Ponto flutuante de alta precisão (64 bits)
float128	Ponto flutuante de altíssima precisão (128 bits)
complex64	Variável complexa de média precisão.
complex128	Variável complexa de alta precisão.
complex256	Variável complexa de altíssima precisão.

Conversão de tipos numéricos

É possível criar um vetor específico já com precisão desejada:

```
empty_uint8 = np.empty(8, dtype=np.uint8)
```

E podemos verificar também o tipo que está sendo usado:

```
print(empty_uint8.dtype)
```

Para trocar um tipo para outro usamos o método `astype`:

```
float_arr = arr.astype(np.float64)
print(float_arr.dtype)
```

OBS:Se for passar de ponto flutuante para inteiro, a variável recebe o valor inteiro:

```
arr1 = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])
arr2 = arr1.astype(np.int32)
print(arr2)
```

Tem como saída [3 -1 -2 0 12 10]

Conversão de tipos numéricos

É possível criar um vetor específico já com precisão desejada:

```
empty_uint8 = np.empty(8, dtype=np.uint8)
```

E podemos verificar também o tipo que está sendo usado:

```
print(empty_uint8.dtype)
```

Para trocar um tipo para outro usamos o método `astype`:

```
float_arr = arr.astype(np.float64)
print(float_arr.dtype)
```

OBS:Se for passar de ponto flutuante para inteiro, a variável recebe o valor inteiro:

```
arr1 = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])
arr2 = arr1.astype(np.int32)
print(arr2)
```

Tem como saída [3 -1 -2 0 12 10]

Conversão de tipos numéricos

É possível criar um vetor específico já com precisão desejada:

```
empty_uint8 = np.empty(8, dtype=np.uint8)
```

E podemos verificar também o tipo que está sendo usado:

```
print(empty_uint8.dtype)
```

Para trocar um tipo para outro usamos o método `astype`:

```
float_arr = arr.astype(np.float64)
print(float_arr.dtype)
```

OBS:Se for passar de ponto flutuante para inteiro, a variável recebe o valor inteiro:

```
arr1 = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])
arr2 = arr1.astype(np.int32)
print(arr2)
```

Tem como saída [3 -1 -2 0 12 10]

Conversão de tipos numéricos

É possível criar um vetor específico já com precisão desejada:

```
empty_uint8 = np.empty(8, dtype=np.uint8)
```

E podemos verificar também o tipo que está sendo usado:

```
print(empty_uint8.dtype)
```

Para trocar um tipo para outro usamos o método `astype`:

```
float_arr = arr.astype(np.float64)
print(float_arr.dtype)
```

OBS:Se for passar de ponto flutuante para inteiro, a variável recebe o valor inteiro:

```
arr1 = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])
arr2 = arr1.astype(np.int32)
print(arr2)
```

Tem como saída [3 -1 -2 0 12 10]

Sumário

Aritmética no numpy

O numpy tem vários métodos instalados para facilitar a vida do programador, tornando optativo usar for, while, if-elif-else.
Veja o código:

```
arr = np.array([2., 3., 4.])
arr2 = 1/arr
print(arr2)
```

que tem como saída: [0.5 0.33333333 0.25]
Volte ao slide 54 e perceba o que foi feito para gerar o gráfico.
Da mesma forma:

```
arr3 = arr*arr
#arr3 = arr**2
print(arr3)
```

tem como saída [4. 9. 16.]
Outras funções matemáticas importantes são: `np.sqrt(arr)` raiz quadrada, `np.exp(arr)` e^x, `log`, `log10`, `log2`, `cos`, `sin` etc.

Aritmética no numpy

O numpy tem vários métodos instalados para facilitar a vida do programador, tornando optativo usar for, while, if-elif-else.
Veja o código:

```
arr = np.array([2., 3., 4.])
arr2 = 1/arr
print(arr2)
```

que tem como saída: [0.5 0.33333333 0.25]
Volte ao slide 54 e perceba o que foi feito para gerar o gráfico.
Da mesma forma:

```
arr3 = arr*arr
#arr3 = arr**2
print(arr3)
```

tem como saída [4. 9. 16.]
Outras funções matemáticas importantes são: `np.sqrt(arr)` raiz quadrada, `np.exp(arr)` e^x, `log`, `log10`, `log2`, `cos`, `sin` etc.

Aritmética no numpy

O numpy tem vários métodos instalados para facilitar a vida do programador, tornando optativo usar for, while, if-elif-else.
Veja o código:

```
arr = np.array([2., 3., 4.])
arr2 = 1/arr
print(arr2)
```

que tem como saída: [0.5 0.33333333 0.25]
Volte ao slide 54 e perceba o que foi feito para gerar o gráfico.
Da mesma forma:

```
arr3 = arr*arr
#arr3 = arr**2
print(arr3)
```

tem como saída [4. 9. 16.]
Outras funções matemáticas importantes são: `np.sqrt(arr)` raiz quadrada, `np.exp(arr)` e^x, `log`, `log10`, `log2`, `cos`, `sin` etc.

Aritmética no numpy

O numpy tem vários métodos instalados para facilitar a vida do programador, tornando optativo usar for, while, if-elif-else.
Veja o código:

```
arr = np.array([2., 3., 4.])
arr2 = 1/arr
print(arr2)
```

que tem como saída: [0.5 0.33333333 0.25]
Volte ao slide 54 e perceba o que foi feito para gerar o gráfico.
Da mesma forma:

```
arr3 = arr*arr
#arr3 = arr**2
print(arr3)
```

tem como saída [4. 9. 16.]
Outras funções matemáticas importantes são: `np.sqrt(arr)` raiz quadrada, `np.exp(arr)` e^x, `log`, `log10`, `log2`, `cos`, `sin` etc.

Aritmética no numpy

O numpy tem vários métodos instalados para facilitar a vida do programador, tornando optativo usar for, while, if-elif-else.
Veja o código:

```
arr = np.array([2., 3., 4.])
arr2 = 1/arr
print(arr2)
```

que tem como saída: [0.5 0.33333333 0.25]
Volte ao slide 54 e perceba o que foi feito para gerar o gráfico.
Da mesma forma:

```
arr3 = arr*arr
#arr3 = arr**2
print(arr3)
```

tem como saída [4. 9. 16.]
Outras funções matemáticas importantes são: `np.sqrt(arr)` raiz quadrada, `np.exp(arr)` e^x, `log`, `log10`, `log2`, `cos`, `sin` etc.