Notas de aula para OBI
Programação em Python

Encontro 4 - Vetores

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

Instituto Federal de Santa Catarina
Campus Florianópolis

Índice

Sumário

Definição de 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

Definição de 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 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.

Definição de 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 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.

Definição de 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 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.

Definição de 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 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.

Definição de 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 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.

Criando vetores

Criando vetores 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
```

Para percorrer um vetor há diferentes formas. Execute o código abaixo:

Versão 1

```
# -*- coding : utf -8 -*-
vC = [1 , 3.4 , "A" , " IFSC " ]
for i in range (0 ,4):
    print ( vC [ i ])
```

Criando vetores

Criando vetores 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
```

Para percorrer um vetor há diferentes formas. Execute o código abaixo:

Versão 1

```
# -*- coding : utf -8 -*-
vC = [1 , 3.4 , "A" , " IFSC " ]
for i in range (0 ,4):
    print ( vC [ i ])
```

Criando vetores

Criando vetores 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
```

Para percorrer um vetor há diferentes formas. Execute o código abaixo:

Versão 1

```
# -*- coding : utf -8 -*-
vC = [1 , 3.4 , "A" , " IFSC " ]
for i in range (0 ,4):
    print ( vC [ i ])
```

Criando vetores

Criando vetores 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
```

Para percorrer um vetor há diferentes formas. Execute o código abaixo:

Versão 1

```
# -*- coding : utf -8 -*-
vC = [1 , 3.4 , "A" , " IFSC " ]
for i in range (0 ,4):
    print ( vC [ i ])
```

Criando vetores

Criando vetores 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
```

Para percorrer um vetor há diferentes formas. Execute o código abaixo:

Versão 1

```
# -*- coding : utf -8 -*-
vC = [1 , 3.4 , "A" , " IFSC " ]
for i in range (0 ,4):
      print ( vC [ i ])
```

Sumário

Percorrendo vetores

Execute cada uma das versões abaixo:

Versão 2

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

Versão 3

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

- Neste caso, *i* não é mais o índice do vetor, mas sim um elemento do vetor, iterado no comando *for*.

Percorrendo vetores

Execute cada uma das versões abaixo:

Versão 2

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

Versão 3

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

- Neste caso, *i* não é mais o índice do vetor, mas sim um elemento do vetor, iterado no comando *for*.

Percorrendo vetores

Execute cada uma das versões abaixo:

Versão 2

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

Versão 3

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

- Neste caso, *i* não é mais o índice do vetor, mas sim um elemento do vetor, iterado no comando *for*.

Percorrendo vetores

Execute cada uma das versões abaixo:

Versão 2

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

Versão 3

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

- Neste caso, *i* não é mais o índice do vetor, mas sim um elemento do vetor, iterado no comando *for*.

Percorrendo vetores

Execute cada uma das versões abaixo:

Versão 2

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

Versão 3

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

- Neste caso, *i* não é mais o índice do vetor, mas sim um elemento do vetor, iterado no comando *for*.

Percorrendo vetores

Execute cada uma das versões abaixo:

Versão 2

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

Versão 3

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

- Neste caso, *i* não é mais o índice do vetor, mas sim um elemento do vetor, iterado no comando *for*.

Percorrendo vetores

Execute cada uma das versões abaixo:

Versão 2

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

Versão 3

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

- Neste caso, *i* não é mais o índice do vetor, mas sim um elemento do vetor, iterado no comando *for*.

Sumário

Alterando um elemento do vetor

É possível alterar os elementos do vetor, 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, 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, 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, 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, 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

Sumário

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(0,'B')
for i in vC :
    print ( i )
```

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(0,'B')
for i in vC :
      print ( i )
```

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(0,'B')
for i in vC :
      print ( i )
```

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(0,'B')
for i in vC :
      print ( i )
```

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.

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.

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.

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.

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.

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.

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.

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.

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)
```

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)
```

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

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

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

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

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

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

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 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)
```

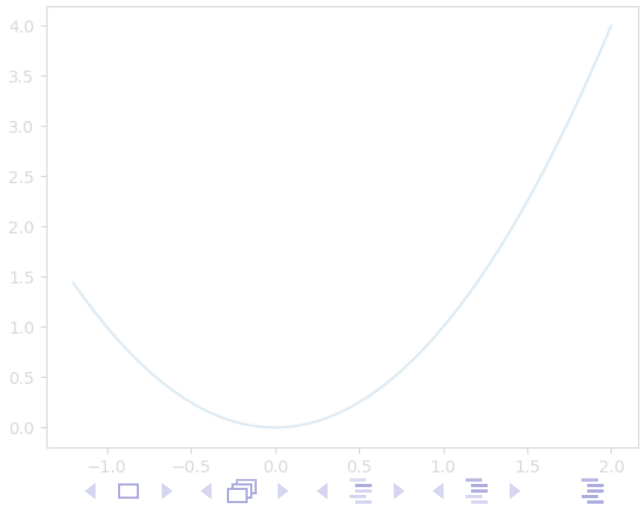
Nesta função são criados números entre início e fim (ambos inclusos), com uma quantidade de termos requerida, ou seja, a função calcula o incremento necessário para incluir a quantidade de termos entre os valores inicio e fim.

`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

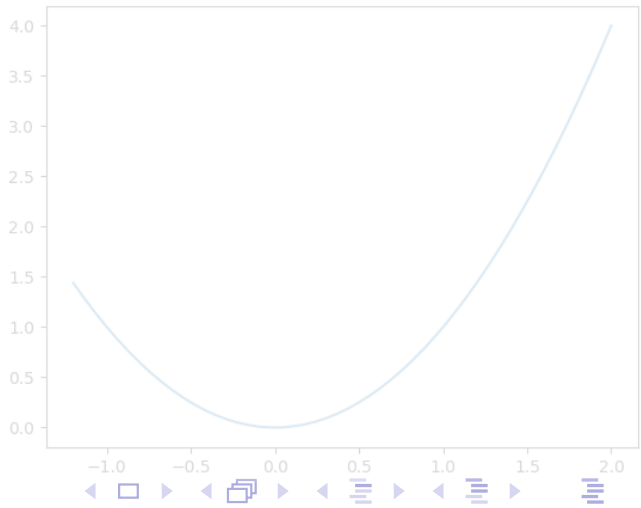
```
import numpy as np
A = np.linspace(inicio, fim, quantidade de termos)
```

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

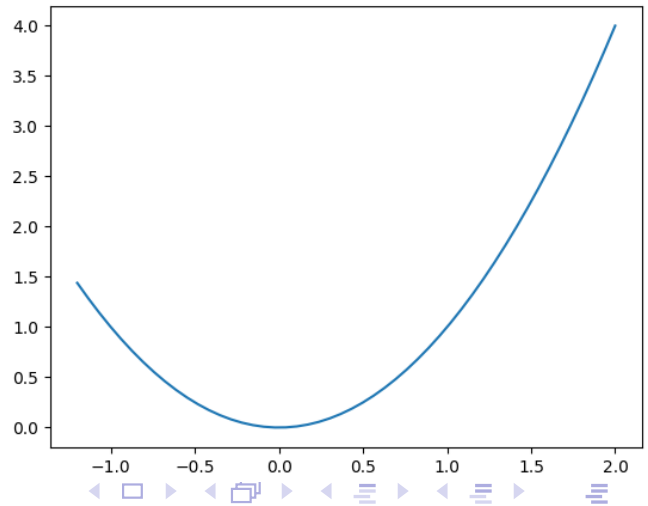
```
import numpy as np
A = np.linspace(inicio, fim, quantidade de termos)
```

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 23 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 23 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 23 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 23 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 23 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.