

Python para Matemática

Mini-Notas de Aula

Pedro H A Konzen

4 de dezembro de 2024

Konzen, Pedro Henrique de Almeida

Python para Matemática: mini-notas de aula / Pedro Henrique de Almeida
Konzen. –2024. Porto Alegre.- 2024.

"Esta obra é uma edição independente feita pelo próprio autor."

1. Linguagem Python. 2. Programação estruturada. 3. Computação matricial. 4. Gráficos.

Licença
CC-BY-SA 4.0.

Conteúdo

Licença	3
Prefácio	4
1 Sobre a Linguagem	5
1.1 Instalação e Execução	5
1.1.1 Online Notebook	5
1.1.2 IDE	5
1.2 Utilização	5
2 Elementos da Linguagem	8
2.1 Classes de Objetos Básicos	8
2.2 Operações Aritméticas Elementares	9
2.3 Funções e Constantes Elementares	11
2.4 Operadores de Comparação Elementares	12
2.5 Operadores Lógicos Elementares	13
2.6 <code>set</code>	14
2.7 <code>tuple</code>	17
2.8 <code>list</code>	18
2.9 <code>dict</code>	21
3 Elementos da Programação Estruturada	23
3.1 Ramificação	23
3.1.1 <code>if</code>	23
3.1.2 <code>if-else</code>	23
3.1.3 <code>if-elif-else</code>	24
3.2 Repetição	26
3.2.1 <code>while</code>	26
3.2.2 <code>for</code>	26
3.2.3 <code>range</code>	27
3.3 Funções	28
4 Elementos da Computação Matricial	31
4.1 NumPy <code>array</code>	31
4.1.1 Inicialização de um array	32
4.1.2 Manipulação de <code>arrays</code>	33
4.1.3 Operadores Elemento-a-Elemento	35
4.2 Elementos da Álgebra Linear	37
4.2.1 Vetores	37
4.2.2 Produto Escalar e Norma	37
4.2.3 Matrizes	38
4.2.4 Inicialização de Matrizes	39
4.2.5 Multiplicação de Matrizes	40
4.2.6 Traço e Determinante de uma Matriz	41
4.2.7 Rank e Inversa de uma Matriz	41
4.2.8 Autovalores e Autovetores de uma Matriz	42
5 Gráficos	46
5.1 Gráfico de uma função	46

CONTEÚDO	3
----------	---

Notas	48
-------	----

Referências	49
-------------	----

Índice de Comandos	50
--------------------	----

Licença

Este texto é disponibilizado sob a Licença Atribuição-CompartilhaIgual 4.0 Internacional Creative Commons. Para visualizar uma cópia desta licença, visite

http://creativecommons.org/licenses/by-sa/4.0/deed.pt_BR

ou mande uma carta para Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Prefácio

O site [notaspedrok.com.br](https://www.notaspedrok.com.br) é uma plataforma que construí para o compartilhamento de minhas notas de aula. Essas anotações feitas como preparação de aulas é uma prática comum de professoras/es. Muitas vezes feitas a rabiscos em rascunhos com validade tão curta quanto o momento em que são concebidas, outras vezes, com capricho de um diário guardado a sete chaves. Notas de aula também são feitas por estudantes - são anotações, fotos, prints, entre outras formas de registros de partes dessas mesmas aulas. Essa dispersão de material didático sempre me intrigou e foi o que me motivou a iniciar o site.

Com início em 2018, o site contava com apenas três notas incipientes. De lá para cá, conforme fui expandido e revisando os materiais, o site foi ganhando acessos de vários locais do mundo, em especial, de países de língua portuguesa. No momento, conta com 13 notas de aula, além de minicursos e uma coleção de vídeos e áudios.

As notas do **Minicurso de Python para Matemática** fazem uma introdução sobre os elementos da linguagem [Python](#), da programação estruturada, da computação matricial e de gráficos. É pensada para estudantes de cursos de matemática e áreas afins.

Aproveito para agradecer a todas/os que de modo assíduo ou esporádico contribuem com correções, sugestões e críticas. ;-)

Pedro H A Konzen

<https://www.notaspedrok.com.br>

1 Sobre a Linguagem

Python é uma linguagem de programação de alto nível e multiparadigma. Ou seja, é relativamente próxima das linguagens humanas naturais, é desenvolvida para aplicações diversas e permite a utilização de diferentes paradigmas de programação (programação estruturada, orientada a objetos, orientada a eventos, paralelização, etc.).

- **Site oficial**

<https://www.python.org/>

1.1 Instalação e Execução

Para executar um código **Python** em seu computador é necessário instalar um **interpretador**. No **site oficial**, estão disponíveis para *download* interpretadores gratuitos e com licença livre para uso. Neste minicurso, vamos utilizar **Python 3**.

1.1.1 Online Notebook

Usar um **Notebook Python online** é uma forma rápida e prática de iniciar os estudos na linguagem. Rodam diretamente em nuvem e vários permitem o uso gratuito por tempo limitado. Algumas opções são:

- Deepnote - <https://deepnote.com>
- Google Colab - <https://colab.research.google.com/>
- Kaggle - <https://www.kaggle.com/>
- Paperspace Gradient - <https://www.paperspace.com/notebooks>
- SageMaker - <https://aws.amazon.com/sagemaker>

1.1.2 IDE

Usar um **ambiente integrado de desenvolvimento** (IDE, em inglês, *integrated development environment*) é a melhor forma de capturar o todo o potencial da linguagem **Python**. Algumas alternativas são:

- IDLE - <https://docs.python.org/3/library/idle.html>
- GNU Emacs - <https://www.gnu.org/software/emacs/>
- Spyder - <https://www.spyder-ide.org/>
- VS Code - <https://code.visualstudio.com/>

1.2 Utilização

A **execução de códigos Python** pode ser feita de três formas básicas:

- em modo interativo em um console/*notebook* Python;
- por execução de um código `arqnome.py` em um console/*notebook* Python;
- por execução de um código `arqnome.py` em um terminal do sistema operacional.

Exemplo 1.2.1. Consideramos o seguinte pseudocódigo.

```
s = "Ola, mundo!".  
imprime(s). (imprime a string s)
```

Vamos escrevê-lo em Python e executá-lo:

a) Em um *notebook*.

Iniciamos um *notebook* Python e digitamos o seguinte código em uma célula de entrada.

```
1 s = "Olá, Mundo!"  
2 #imprime a string s  
3 print(s)
```

Ao executarmos a célula, obtemos a saída

Olá, Mundo!

b) Em modo iterativo no console.

Iniciamos um console Python em terminal do sistema e digitamos

```
$ python3
```

Aqui, \$ é o símbolo de *prompt* de entrada que pode ser diferente a depender do seu sistema operacional. Então, digitamos

```
1 >>> s = "Olá, Mundo!"  
2 >>> print(s) #imprime a string s
```

Observamos que >>> é o símbolo de *prompt* de entrada do console Python. A saída

```
1 Olá, Mundo!
```

aparece logo abaixo da última linha de *prompt* executada. Para encerrar o console, digitamos

```
1 >>> quit()
```

c) Escrevendo o código `ola.py` e executando-o em um console/*notebook* Python.

Primeiramente, escrevemos o código em uma IDE (ou em um simples editor de texto)

```
1 s = "Olá, Mundo!"  
2 print(s) # imprime a string s
```

Uma vez salvo em `/caminho/ola.py`, executamo-lo no console/*notebook* [Python](#) com

```
1 >>> exec(open('/caminho/ola.py').read())
```

A saída é impressa logo abaixo do *prompt*/célula de entrada.

d) Escrevendo o código `ola.py` e executando-o em terminal do sistema.

Assumindo que o código já esteja salvo no arquivo `/caminho/ola.py`, podemos executá-lo em um terminal digitando

```
1 $ python3 /caminho/ola.py
```

A saída é impressa logo abaixo do *prompt* de entrada do sistema.

2 Elementos da Linguagem

2.1 Classes de Objetos Básicos

Python é uma linguagem de programação **dinâmica** em que as variáveis/objetos são declaradas/os automaticamente ao receberem um valor/dado. Por exemplo, consideramos as seguintes instruções

```
1 x = 2
2 y = x * 3.0
```

Na primeira instrução, a variável `x` recebe o valor inteiro 2 e, então, é armazenado na memória do computador como um objeto da **classe `int`** (número inteiro). Na segunda instrução, `y` recebe o valor decimal 6.0 (resultado de 2×3.0) e é armazenado como um objeto da **classe `float`** (ponto flutuante de 64-bits). Podemos verificar isso, com as seguintes instruções

```
1 print(x)

2
1 print(y)

6.0
1 print(type(x), type(y))

<class 'int'> <class 'float'>
```

Observação 2.1.1. (**Comentários e Continuação de Linha.**) Códigos Python admitem **comentários** e **continuação de linha** como no seguinte exemplo

```
1 # isto é um comentário
2 s = "isto é uma \
3 string"
4 print(s)

isto é uma string
1 type(s)

<class 'str'>
```

Observação 2.1.2. (**Notação científica.**) O Python aceita **notação científica**. Por exemplo 5.2×10^{-2} é digitado da seguinte forma

```
1 5.2e-2

0.052
```

Observação 2.1.3. (**Casting.**) Quando não há ambiguidade, pode-se fazer a conversão entre objetos de classes diferentes (*casting*). Por exemplo,

```
1 x = 1
2 print(x, type(x))

1 <class 'int'>
```



```
1 y = float(x)
2 print(y, type(y))
```

```
1.0 <class 'float'>
```

Além de objetos numéricos e *string*, Python também conta com objetos `list` (lista), `tuple` (*n*-upla) e `dict` (dicionário). Estudaremos essas classes de objetos mais adiante no minicurso.

Exercício 2.1.1. Antes de implementar, diga qual o valor de `x` após as seguintes instruções.

```
1 x = 1
2 y = x
3 y = 0
```

Justifique sua resposta e verifique-a.

Exercício 2.1.2. Implemente um código em que a(o) usuá(ri)a(o) entra com valores para as variáveis `x` e `y`. Então, os valores das variáveis são permutados entre si. Dica: use `input` para a entrada de dados.

Respostas dos Exercícios

2.1.1. 1

2.1.2.

```
1 x = float(input('x = '))
2 y = float(input('y = '))
3 z = x
4 x = y
5 y = z
6 print('x = ', x)
7 print('y = ', y)
```

2.2 Operações Aritméticas Elementares

Os operadores aritméticos elementares são:

- + **adição**
- **subtração**
- * **multiplicação**
- / **divisão**
- ** **potenciação**
- % **módulo**
- // **divisão inteira**

Exemplo 2.2.1. Estudamos a seguinte computação

```
1 2+8*3/2**2-1
```

7.0

Observamos que as operações `**` tem precedência sobre as operações `*`, `/`, `%`, `//`, as quais têm precedência sobre as operações `+`, `-`. Operações de mesma precedência seguem a ordem da esquerda para direita, conforme escritas na linha de comando. Usa-se parênteses para alterar a precedência entre as operações, por exemplo

```
1 (2+8*3)/2**2-1
```

5.5

Observação 2.2.1. (**Precedência das Operações.**) Consulte mais informações sobre a precedência de operadores em [Python Docs: Operator Precedence](#).

Exercício 2.2.1. Compute as raízes do seguinte polinômio quadrático

$$p(x) = 2x^2 - 2x - 4 \quad (1)$$

usando a fórmula de Bhaskara¹.

O operador `%` módulo computa o **resto da divisão** e o operador `//` a **divisão inteira**, por exemplo

```
1 5 % 2
```

1

```
1 5 // 2
```

2

Exercício 2.2.2. Use o [Python](#) para computar os inteiros não negativos q e r tais que

$$25 = q \cdot 3 + r, \quad (2)$$

sendo r o menor possível.

Respostas dos Exercícios

2.2.1.

```
1 a = 2.
2 b = -2.
3 c = -4.
4 dlta = b**2 - 4*a*c
5 x1 = (-b - dlta**(1./2))/(2*a)
6 x2 = (-b + dlta**(1./2))/(2*a)
7 print('x1 = ', x1)
8 print('x2 = ', x2)
```

2.2.2.

```
1 q = 25//3
2 print('q = ', q)
3 r = 25%3
4 print('r = ', r)
```

2.3 Funções e Constantes Elementares

O módulo Python `math` disponibiliza várias funções e constantes elementares.

Para usá-las, precisamos importar o módulo em nosso código

```
1 import math
```

Com isso, temos acesso a todas as definições e declarações contidas neste módulo. Por exemplo

```
1 math.pi
```

3.141592653589793

```
1 math.cos(math.pi)
```

-1.0

```
1 math.sqrt(2)
```

1.4142135623730951

```
1 math.log(math.e)
```

1.0

Observação 2.3.1. (**Função Logaritmo.**) Notamos que `math.log` é a função logaritmo natural, i.e. $\ln(x) = \log_e(x)$. A implementação Python para o logaritmo de base 10 é `math.log(x, 10)` ou, mais acurado, `math.log10`.

Exercício 2.3.1. Compute

a) $\sin\left(\frac{\pi}{4}\right)$

b) $e^{\log_3(\pi)}$

c) $\sqrt[3]{-27}$

Exercício 2.3.2. Refaça o Exercício 2.2.1 usando a função `math.sqrt` para computar a raiz quadrada do discriminante.

Respostas dos Exercícios**2.3.1.**

```
1 import math
2 # a)
```

```

3 a = math.sin(math.pi/4)
4 print('a) ', a)
5 # b)
6 b = 3*math.pi
7 print('b) ', b)
8 # c)
9 c = -(27)**(1/3)
10 print('c) ', c)

```

2.3.2.

```

1 import math
2 a = 2.
3 b = -2.
4 c = -4.
5 dlta = b**2 - 4*a*c
6 x1 = (-b - math.sqrt(dlta))/(2*a)
7 x2 = (-b + math.sqrt(dlta))/(2*a)
8 print('x1 = ', x1)
9 print('x2 = ', x2)

```

2.4 Operadores de Comparação Elementares

Os **operadores de comparação** elementares são

== igual a
!= diferente de
> maior que
< menor que
>= maior ou igual que
<= menor ou igual que

Estes operadores **retornam os valores lógicos True** (verdadeiro) ou **False** (falso).

Por exemplo, temos

```

1 x = 2
2 x + x == 4

```

True

Exercício 2.4.1. Considere a circunferência de equação

$$c: (x-1)^2 + (y+1)^2 = 1. \quad (3)$$

Escreva um código em que a(o) usuá(ri)a(o) entra com as coordenadas de um ponto $P = (x, y)$ e o código verifica se P pertence ao disco determinado por c .

Exercício 2.4.2. Antes de implementar, diga qual é o valor lógico da instrução

```
1 math.sqrt(3)**2 == 3
```

Justifique sua resposta e verifique!

Respostas dos Exercícios

2.4.1.

```
1 x = float(input('x = '))
2 y = float(input('y = '))
3 resp = (x-1.)**2 + (y+1.)**2 <= 1.
4 print('P em c?', resp)
```

2.4.2. False

2.5 Operadores Lógicos Elementares

Os **operadores lógicos** elementares são:

and **e lógico**

or **ou lógico**

not **não lógico**

Exemplo 2.5.1. (**Tabela Booleana do and.**) A tabela booleana² do and é

A	B	A and B
True	True	True
True	False	False
False	True	False
False	False	False

Por exemplo, temos

```
1 x = 2
2 (x > 1) and (x < 2)
```

False

Exercício 2.5.1. Construa as tabelas booleanas do operador or e do not.

Exercício 2.5.2. Use Python para verificar se $1.4 \leq \sqrt{2} < 1.5$.

Exercício 2.5.3. Considere um retângulo $r : ABDC$ de vértices $A = (1, 1)$ e $D = (2, 3)$. Crie um código em que a(o) usuá(ri)a informa as coordenadas de um ponto $P = (x, y)$ e o código imprime **True** ou **False** para cada um dos seguintes itens:

a) $P \in r$.

b) $P \in \partial r$.

c) $P \notin \bar{r}$.

Exercício 2.5.4. Implemente uma instrução para computar o operador xor (ou exclusivo). Dadas duas afirmações A e B, A xor B é **True** no caso de uma, e somente uma, das afirmações ser **False**, caso contrário é **False**.

Respostas dos Exercícios

2.5.1.

```
1 print('A B | A or B')
2 print(True, True, '|', True or True)
3 print(True, False, '|', True or False)
4 print(False, True, '|', False or True)
5 print(False, False, '|', False or False)
6
7 print('A | not(A)')
8 print(True, '|', not(True))
9 print(False, '|', not(False))
```

2.5.2.

```
1 import math
2 resp = 1.4 <= math.sqrt(2) < 1.5
3 print('1.4 <= math.sqrt(2) < 1.5 ?', resp)
```

2.5.3.

```
1 x = float(input('x = '))
2 y = float(input('y = '))
3 print('P em r', \
4       (1 < x < 2) and (1 < y < 3))
5 print('P na fronteira', \
6       ((x == 1 or x == 2) and (1 <= y <= 3)) or \
7       ((1 <= x <= 2) and (y == 1 or y == 3)))
8 print('P fora do fecho', \
9       (x < 1 or x > 2) or (y < 1 or y > 3))
```

2.5.4. (A or B) and not(A and B)

2.6 set

Um **set** em **Python** é uma coleção de objetos não ordenada, imutável e não admite itens duplicados. Por exemplo,

```
1 a = {1, 2, 3}
2 type(a)

<class 'set'>

1 b = set((2, 1, 3, 3))
2 print(b)
```

```
{1, 2, 3}
```

```
1 a == b
```

True

```
1 # conjunto vazio
2 e = set()
```

Acima, alocamos o conjunto $a = \{1, 2, 3\}$. Note que o conjunto b é igual a a . Observamos que o conjunto vazio deve ser construído com a instrução `set()` e não com `{}`³.

Observação 2.6.1. (**Tamanho de uma Coleção de Objetos.**) A função `len` retorna o número de elementos de uma coleção de objetos. Por exemplo,

```
1 len(a)
```

3

Operadores envolvendo conjuntos:

- diferença entre conjuntos
- | união de conjuntos
- & interseção de conjuntos
- ^ diferença simétrica

Exemplo 2.6.1. Os conjuntos

$$A = \{2, \pi, -0.25, 3, \text{'banana'}\}, \quad (4)$$

$$B = \{\text{'laranja'}, 3, \arccos(-1), -1\} \quad (5)$$

podem ser alocados como `sets`

```
1 import math
2 A = {2, math.pi, -0.25, 3, 'banana'}
3 B = {'laranja', 3, math.acos(-1), -1}
```

e, então, podemos computar:

a) $A \setminus B$

```
1 a = A - B
2 print(a)
```

```
{-0.25, 2, 'banana'}
```

b) $A \cup B$

```
1 b = A | B
2 print(b)
```

```
{-0.25, 2, 3, 3.141592653589793, 'laranja', 'banana', -1}
```

c) $A \cap B$

```
1 c = A & B
2 print(c)

{3, 3.141592653589793}
```

d) $A \Delta B = (A \setminus B) \cup (B \setminus A)$

```
1 d = A ^ B
2 print(d)

{-0.25, 2, 'laranja', 'banana', -1}
```

Exercício 2.6.1. Aloque como `set` cada um dos seguintes conjuntos:

a) O conjunto A dos números $-12 \leq n \leq 6$ e que são divisíveis por 2.

b) O conjunto B dos números $-12 < n \leq 6$ e que são divisíveis por 3.

Então, compute o subconjunto de A e B que contém apenas os números divisíveis por 2 e 3.

Observação 2.6.2. (**Compreensão de sets.**) `Python` disponibiliza a sintaxe de compreensão de `sets`. Por exemplo,

```
1 C = {x for x in A if type(x) == str}
2 print(C)

{'banana'}
```

Exercício 2.6.2. Considere o conjunto

$$Z = \{-4, -3, -2, -1, 0, 1, 2, 3, 4\}. \quad (6)$$

Faça um código `Python` para extrair o subconjunto \mathcal{P} dos números pares do conjunto Z . Depois, modifique-o para extrair o subconjunto \mathcal{I} dos números ímpares. Dica: use de compreensão de `sets`.

Respostas dos Exercícios

2.6.1.

```
1 A = {-12, -10, -8, -6, \
2     -4, -2, 0, 2, 4, 6}
3 B = {-12, -9, -6, -3, \
4     0, 3, 6}
5 C = A & B
6 print('C = ', C)
```

2.6.2.

```
1 Z = {-4, -3, -2, -1, \
2     0, 1, 2, 3, 4}
```



```
3 P = {p for p in Z if p % 2 == 0}
4 print('P = ', P)
```

2.7 tuple

Em Python, **tuple** é uma coleção ordenada e imutável de objetos. Por exemplo, na sequência alocamos um par, uma tripla e uma quadrupla ordenada usando tuples.

```
1 a = (1, 2)
2 print(a, type(a))
```

(1, 2) <class 'tuple'>

```
1 b = -1, 1, 0
2 print(b, len(b))
```

(-1, 1, 0) 3

```
1 c = (0.5, 'laranja', {2, -1}, 2)
2 print(c)
```

(0.5, 'laranja', {2, -1}, 2)

Os elementos de um **tuple** são indexados, o índice 0 corresponde ao primeiro elemento, o índice 1 ao segundo elemento e assim por diante. Desta forma é possível o acesso direto a um elemento de um **tuple** usando-se sua posição. Por exemplo,

```
1 print(c[2])
```

{2, -1}

Pode-se também extrair uma fatia (um subconjunto) usando-se a notação `:`. Por exemplo,

```
1 d = c[1:3]
2 print(d)
```

('laranja', {2, -1})

Operadores básicos:

+ **concatenação**

```
1 a = (1, 2) + (3, 4, 5)
2 print(a)
```

(1, 2, 3, 4, 5)

* **repetição**

```
1 b = (1, 2) * 2
```

(1, 2, 1, 2)

```

    in pertencimento
1 c = 1 in (-1, 0, 1, 2)

True

```

Exercício 2.7.1. Use `sets` para alocar os conjuntos

$$A = \{-1, 0, 2\}, \quad (7)$$

$$B = \{2, 3, 5\}. \quad (8)$$

Então, compute o produto cartesiano $A \times B = \{(a, b) : a \in A, b \in B\}$. Qual o número de elementos da $A \times B$? Dica: use a sintaxe de compreensão de `sets` (consulte a Observação 2.6.2).

Exercício 2.7.2. Aloque o gráfico discreto da função⁴ $f(x) = x^2$ para $x = 0, \frac{1}{2}, 1, 2$. Dica: use a sintaxe de compreensão de conjuntos (consulte a Observação 2.6.2).

Respostas dos Exercícios

2.7.1.

```

1 A = {-1, 0, 2}
2 B = {2, 3, 5}
3 P = {(a,b) for a in A for b in B}
4 print('P = ', P)

```

2.7.2.

```

1 X = {0., 0.5, 1., 2.}
2 G = {(x, x**2) for x in X}
3 print('G = ', G)

```

2.8 list

Um `list` é uma coleção de objetos **indexada** e **mutável**. Por exemplo,

```

1 x = [-1, 2, -3, -5]
2 print(x, type(x))

[-1, 2, -3, -5] <class 'list'>

```

```

1 y = [1, 1, 'oi', 2.5]
2 print(y)

```

```
[1, 1, 'oi', 2.5]
```

```

1 vazia = []
2 print(len(vazia))
3 print(len(y))

```

0

4

Os elementos de um `list` são indexados de forma análoga a um `tuple`, o índice 0 corresponde ao primeiro elemento, o índice 1 ao segundo elemento e assim por diante. Bem como, o índice `-1` corresponde ao último elemento, o `-2` ao penúltimo e segue. Por exemplo,

```
1 x[-1] = 3.14
2 print('x[0] = ', x[0])
3 print(x = ', x)
```

```
x[0] = 1
x = [-1, 2, -3, 3.14]
```

```
1 x[:3] = [10, -20]
2 print(x)
```

```
[10, -20, -3, 3.14]
```

Os operadores básicos de concatenação e de repetição também estão disponíveis para um `list`. Por exemplo,

```
1 x = [1,2] + [3, 4, 5]
2 print(x)
3 y = [1,2]*2
4 print(y)
```

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

Observação 2.8.1. `list` conta com várias funções prontas para a execução de diversas tarefas práticas como, por exemplo, inserir/deletar itens, contar ocorrências, ordenar itens, etc. Consulte na web [Python Docs: More on Lists](#).

Observação 2.8.2. (*Alocação versus Cópia.*) Estudamos o seguinte exemplo

```
1 x = [2, 3, 1]
2 y = x
3 y[1] = 0
4 print('x = ', x)
```

```
x = [2, 0, 1]
```

Em Python, dados têm identificação única. Logo, neste exemplo, `x` e `y` apontam para o mesmo endereço de memória. Modificar `y` é também modificar `x` e vice-versa. Para desassociar `y` de `x`, `y` precisa receber uma cópia de `x`, como segue

```
1 x = [2, 3, 1]
2 print('id(x) = ', id(x))
3 y = x.copy()
4 print('id(y) = ', id(y))
5 y[1] = 0
6 print('x = ', x)
7 print('y = ', y)
```

```
id(x) = 140476759980864
```

```
id(y) = 140476760231360
x = [2, 3, 1]
y = [2, 0, 1]
```

Observação 2.8.3. (**Anexar ou Estender.**) Um `list` tem tamanho dinâmico, permitindo a anexação de um novo item ou sua extensão. A anexação de um item pode ser feita com o método `list.append`, enquanto que a extensão é feita com `list.extend`. Por exemplo, com o `list.append` temos

```
1 l = [1, 2]
2 l.append((3, 4))
3 print(l)
```

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

Enquanto, que com o `list.extend` obtemos

```
1 l = [1, 2]
2 l.extend((3, 4))
3 print(l)
```

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

Exercício 2.8.1. A solução de

$$x^2 - 2 = 0 \quad (9)$$

pode ser aproximada pela iteração do método babilônico

$$x_0 = 1, \quad (10)$$

$$x_{i+1} = \frac{1}{2} \left(x_i + \frac{2}{x_i} \right) \quad (11)$$

para $i = 0, 1, 2, \dots$. Aloque uma lista com as quatro primeiras iteradas, i.e. $[x_0, x_1, x_2, x_3, x_4]$. Dica: use `list.append`.

Exercício 2.8.2. Aloque cada um dos seguintes vetores como um `list`:

$$x = (-1, 3, -2), \quad (12)$$

$$y = (4, -2, 0). \quad (13)$$

Então, compute

a) $x + y$

b) $x \cdot y$

Dica: use uma compreensão de lista e os métodos `zip` e `sum`.

Exercício 2.8.3. Uma matriz pode ser alocada como um encadeamento de `lists`. Por exemplo, a matriz

$$M = \begin{bmatrix} 1 & -2 \\ 2 & 3 \end{bmatrix} \quad (14)$$

pode ser alocada como a seguinte `list`

```
1 M = [[1, -2], [2, 3]]
2 M

[[1, -2], [2, 3]]
```

Use `list` para alocar a matriz

$$A = \begin{bmatrix} 1 & -2 & 1 \\ 8 & 0 & -7 \\ 3 & -1 & -2 \end{bmatrix} \quad (15)$$

e o vetor

$$x = (2, -3, 1), \quad (16)$$

então compute Ax .

Respostas dos Exercícios

2.8.1.

```
1 x = [1] # x0
2 x.append(0.5*(x[-1] + 2./x[-1])) # x1
3 x.append(0.5*(x[-1] + 2./x[-1])) # x2
4 x.append(0.5*(x[-1] + 2./x[-1])) # x3
5 x.append(0.5*(x[-1] + 2./x[-1])) # x4
6 print('x = ', x)
```

2.8.2.

```
1 # a)
2 x = [-1, 3, -2]
3 y = [4, -2, 0]
4 xpy = [xy[0]+xy[1] for xy in zip(x,y)]
5 print('x + y = ', xpy)
6
7 # b)
8 dxy = sum([xy[0]*xy[1] for xy in zip(x,y)])
9 print('x . y = ', dxy)
```

2.8.3.

```
1 A = [[1, -2, 1],
2       [8, 0, -7],
3       [3, -1, -2]]
4 x = [2, -3, 1]
5 Ax = [sum([aix[0]*aix[1] for aix in zip(ai, x)]) for
6        ai in A]
6 print('Ax = ', Ax)
```

2.9 dict

Um `dict` é um mapeamento de objetos (um dicionário), em que cada item é um par `chave:valor`. Por exemplo,

```
1 a = {'nome': 'triangulo', 'perímetro': 3.2}
2 print(a, type(a))
```

```
{'nome': 'triangulo', 'perímetro': 3.2} <class 'dict'>
```

O acesso a um item do dicionário pode ser feito por sua chave, por exemplo,

```
1 a['nome'] = 'triângulo'
2 print(a['nome'])
```

```
'triângulo'
```

Pode-se adicionar um novo par, simplesmente, atribuindo valor a uma nova chave. Por exemplo,

```
1 a['vértices'] = {'A': (0,0), 'B': (3,0), 'C': (0,4)}
2 print('vértice B =', a['vértices']['B'])
```

```
vértice B = (3,0)
```

Exercício 2.9.1. Considere a função afim

$$f(x) = 3 - x. \quad (17)$$

Implemente um dicionário para alocar a raiz da função, a interseção com o eixo y e seu coeficiente angular.

Exercício 2.9.2. Considere a função quadrática

$$g(x) = x^2 - x - 2 \quad (18)$$

Implemente um dicionário para alocar suas raízes, vértice e interseção com o eixo y .

Respostas dos Exercícios

2.9.1.

```
1 f_dict = {'raiz': 3.,
2           'y_intercep': 3.,
3           'coef_angular': -1.}
4 print('raiz = ', f_dict['raiz'])
5 print('y_intercep = ', f_dict['y_intercep'])
6 print('coef_angular = ', f_dict['coef_angular'])
```

2.9.2.

```
1 g_dict = {'raízes': (-1., 2.),
2           'vértice': 0.5,
3           'y_intercep': 2.}
4 print('raiz = ', g_dict['raízes'])
5 print('y_intercep = ', g_dict['y_intercep'])
6 print('vértice = ', g_dict['vértice'])
```

3 Elementos da Programação Estruturada

Na programação estruturada, os comandos de programação são executados em sequência, um novo comando só iniciado após o término do processamento do comando anterior. Em Python, cada linha consiste em um comando, o programa tem início na primeira linha e término na última linha do código. Instruções de ramificação permitem a seleção *on-the-fly* de blocos de comandos, enquanto que instruções de repetição permitem a execução repetida de um bloco. A definição de função permite a criação de um sub-código (sub-programa) do código.

3.1 Ramificação

Uma estrutura de ramificação é uma instrução `if-[elif-...-elif-else]` para a tomada de decisões durante a execução de um programa.

3.1.1 if

Por exemplo, o código abaixo computa as raízes reais do polinômio

$$p(x) = ax^2 + bx + c, \quad (19)$$

com a , b e c alocados no início do código.

```
1 import math as m
2 a = 1.
3 b = -1.
4 c = -2.
5 dlta = b**2 - 4.*a*c
6 if (dlta >= 0.):
7     x1 = (-b - m.sqrt(dlta))/(2.*a)
8     x2 = (-b + m.sqrt(dlta))/(2.*a)
9     print('x1 =', x1)
10    print('x2 =', x2)
```

```
x1 = -1.0
x2 = 2.0
```

Neste código, o bloco de comandos (linhas 7-10) só é executado, se o discriminante do polinômio seja não-negativo. Verifique! Troque os valores de a , b e c de forma que p tenha raízes complexas.

Observação 3.1.1. (Indentação.) Nas linhas 7-10 do código anterior, a indentação dos comandos é obrigatória. O bloco de comandos indentados indicam o escopo da instrução `if`.

3.1.2 if-else

Vamos modificar o código anterior, de forma que as raízes complexas sejam computadas e impressas, quando for o caso.

```
1 import math as m
2 a = 1.
```

```

3 b = -4.
4 c = 8.
5 dlta = b**2 - 4.*a*c
6 if (dlta >= 0.):
7     # raízes reais
8     x1 = (-b - m.sqrt(dlta))/(2.*a)
9     x2 = (-b + m.sqrt(dlta))/(2.*a)
10 else:
11     # raízes complexas
12     rea = -b/(2.*a)
13     img = m.sqrt(-dlta)/(2.*a)
14     x1 = rea - img*1j
15     x2 = rea + img*1j
16 print('x1 =', x1)
17 print('x2 =', x2)

```

```

x1 = (2-2j)
x2 = (2+2j)

```

Observação 3.1.2. (**Número Complexo**.) Em **Python**, números complexos podem ser alocados como objetos da classe **complex**. O número imaginário $i = \sqrt{-1}$ é denotado por **1j** e um número completo $a + bi$ por $a + b*1j$.

3.1.3 if-elif-else

A instrução **elif** é uma conjunção de uma sequência de instruções **textttelse-if**. Vamos modificar o código anterior, de forma a computar o caso de raízes reais duplas de forma própria.

```

1 import math as m
2 a = 1.
3 b = 2.
4 c = 1.
5 dlta = b**2 - 4.*a*c
6 if (dlta > 0.):
7     # raízes reais
8     x1 = (-b - m.sqrt(dlta))/(2.*a)
9     x2 = (-b + m.sqrt(dlta))/(2.*a)
10 elif (dlta == 0.):
11     x1 = x2 = -b/(2.*a)
12 else:
13     # raízes complexas
14     rea = -b/(2.*a)
15     img = m.sqrt(-dlta)/(2.*a)
16     x1 = rea - img*1j
17     x2 = rea + img*1j
18 print('x1 =', x1)
19 print('x2 =', x2)

```

```

x1 = -1.0
x2 = -1.0

```


Exercício 3.1.1. Desenvolva um código para computar a raiz do polinômio

$$f(x) = ax + b \quad (20)$$

com dados a e b . O código deve lidar com todos os casos possíveis, a saber:

- a) única raiz ($a \neq 0$).
- b) infinitas raízes ($a = b = 0$).
- c) não existe raiz ($a = 0$ e $b \neq 0$).

Exercício 3.1.2. Desenvolva um código em que dados três pontos A , B e C no plano, verifique se ABC determina um triângulo. Caso afirmativo, classifique-o como um triângulo equilátero, isósceles ou escaleno.

Respostas dos Exercícios

3.1.1.

```
1 # params
2 a = 2.
3 b = 1.
4 # raiz
5 if (a != 0.):
6     x = -b/a
7     print('raiz única x = ', x)
8 elif (b == 0.):
9     print('infinitas raízes x')
10 else:
11     print('não existe raiz')
```

3.1.2.

```
1 import math
2 # pts
3 A = (0., 0.)
4 B = (3., 0.)
5 C = (3., 4.)
6 # compr. lados
7 lado_1 = math.sqrt((B[0]-A[0])**2 + (B[1]-A[1])**2)
8 lado_2 = math.sqrt((C[0]-B[0])**2 + (C[1]-B[1])**2)
9 lado_3 = math.sqrt((C[0]-A[0])**2 + (C[1]-A[1])**2)
10 # triangulo?
11 if (lado_1 + lado_2 > lado_3) and \
12     (lado_1 + lado_3 > lado_2) and \
13     (lado_2 + lado_3 > lado_1):
14     print('ABC é triângulo:')
15     # equilátero?
16     if lado_1 == lado_2 == lado_3:
17         print('\tequilátero')
18     elif (lado_1 != lado_2 != lado_3):
```

```
19     print('\tescaleno')
20 else:
21     print('\tisósceles')
22 else:
23     print('ABC não é triângulo')
```

3.2 Repetição

Estruturas de repetição são instruções que permitem que a execução repetida de um bloco de comandos. São duas instruções disponíveis `while` e `for`.

3.2.1 while

A instrução `while` permite a repetição de um bloco de comandos, enquanto uma dada condição for verdadeira.

Por exemplo, o seguinte código computa e imprime os elementos da sequência de Fibonacci⁵, enquanto forem menores que 10.

```
1 n = 1
2 print(n)
3 m = 1
4 print(m)
5 while (n+m < 10):
6     s = m
7     m += n
8     n = s
9     print(m)
```

Verifique!

Observação 3.2.1. (Instruções de Controle.) As instruções de controle `break`, `continue` são bastante úteis em várias situações. A primeira, encerra as repetições e, a segunda, pula para uma nova repetição.

Exercício 3.2.1. Use `while` para imprimir os dez primeiros números ímpares.

Exercício 3.2.2. Uma aplicação do Método Babilônico⁶ para a aproximação da solução da equação $x^2 - 2 = 0$, consiste na iteração

$$x_0 = 1, \quad (21)$$

$$x_{i+1} = \frac{x_i}{2} + \frac{1}{x_i}, \quad i = 0, 1, 2, \dots \quad (22)$$

Faça um código com `while` para computar aproximação x_i , tal que $|x_{i+1} - x_i| < 10^{-7}$.

3.2.2 for

A instrução `for` permite a execução iterada de um bloco de comandos. Dado um objeto iterável, a cada laço um novo item do objeto é tomado. Por exemplo,

o seguinte código computa e imprime os primeiros 6 elementos da sequência de Fibonacci.

```
1 n = 1
2 print(f'1: {n}')
3 m = 1
4 print(f'2: {m}')
5 for i in [3,4,5,6]:
6     s = m
7     m += n
8     n = s
9     print(f'{i}: {m}')
```

Verifique!

3.2.3 range

A função `range([start,]stop[,sep])` é particularmente útil na construção de instruções `for`. Ela cria um objeto de classe iterável de `start` (incluído) a `stop` (excluído), de elementos igualmente separados por `sep`. Por padrão, `start=0`, `sep=1` caso omitidos. Por exemplo, o código anterior por ser reescrito como segue.

```
1 n = 1
2 print(f'1: {n}')
3 m = 1
4 print(f'2: {m}')
5 for i in range(3,7):
6     s = m
7     m += n
8     n = s
9     print(f'{i}: {m}')
```

Verifique!

Exercício 3.2.3. Com n dado, desenvolva um código para computar o valor da soma harmônica

$$\sum_{k=1}^n \frac{1}{k} = \frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n}. \quad (23)$$

Exercício 3.2.4. Desenvolva um código para computar o fatorial de um dado número natural n . Dica: use `math.factorial` para verificar seu código.

Respostas dos Exercícios

3.2.1.

```
1 i = 1
2 c = 0
3 while (c < 10):
```

```
4 print(i)
5 i += 2
6 c += 1
```

3.2.2.

```
1 import math
2 x0 = 1.
3 x = x0/2 + 1/x0
4 while (math.fabs(x - x0) >= 1e-7):
5     x0 = x
6     x = x0/2 + 1/x0
7 print(x)
```

3.2.3.

```
1 n = 10
2 s = 0.
3 for i in range(1, n+1):
4     s += 1./i
5 print(s)
```

3.2.4.

```
1 n = 5
2 fact = 1
3 for i in range(1, n+1):
4     fact *= i
5 print(fact)
```

3.3 Funções

Em Python, uma função é definida pela instrução `def`. Por exemplo, o seguinte código implementa a função

$$f(x) = 2x - 3 \quad (24)$$

e imprime o valor de $f(2)$.

```
1 def f(x):
2     y = 2*x - 3
3     return x
4
5 z = f(2)
6 print(f'f(2) = {z}')
```

`f(2) = 2`

Observação 3.3.1. Para funções pequenas, pode-se utilizar a instrução `lambda` de funções anônimas. Por exemplo,

```
1 f = lambda x: 2*x - 3
2 print(f'f(3) = {f(3)}')
```

$f(3) = 3$

Exemplo 3.3.1. (Função com Parâmetros.) O seguinte código, implementa o polinômio de primeiro grau

$$p(x) = ax + b, \quad (25)$$

com parâmetros predeterminados $a = 1$ e $b = 0$ (função identidade).

```
1 def p(x, a=1., b=0.):
2     y = a*x + b
3     return y
4
5 print('p(2) =', p(2.))
6 print('p(2, 3, -5) =', p(2., 3., -5.))
```

Exercício 3.3.1. Implemente uma função para computar as raízes de um polinômio de grau 2 $p(x) = ax^2 + bx + c$.

Exercício 3.3.2. Implemente uma função que computa o produto escalar de dois vetores

$$x = (x_0, x_1, \dots, x_{n-1}), \quad (26)$$

$$y = (y_0, y_1, \dots, y_{n-1}). \quad (27)$$

Dica: considere que os vetores são alocados com `lists`.

Exercício 3.3.3. Implemente uma função que computa o determinante de matrizes 2×2 . Dica: considere que a matriz está alocada com um `list` encadeado.

Exercício 3.3.4. Implemente uma função que computa a multiplicação matriz - vetor Ax , com A matriz 2×2 e x um vetor de dois elementos. Assuma que a matriz e o vetor estão alocados usando `list`.

Respostas dos Exercícios

3.3.1.

```
1 import math
2 def raiz_poli2(a, b, c):
3     dlta = b**2 - 4.*a*c
4     if (dlta > 0.):
5         x1 = (-b + math.sqrt(dlta))/(2.*a)
6         x2 = (-b - math.sqrt(dlta))/(2.*a)
7     elif (dlta == 0.):
8         x1 = -b/(2.*a)
9         x2 = x1
10    else:
11        x1 = None
12        x2 = None
13    return x1, x2
```

3.3.2.

```
1 def dot(x, y):  
2     s = 0.  
3     for xi, yi in zip(x,y):  
4         s += xi*yi  
5     return s
```

3.3.3.

```
1 def det(A):  
2     d = A[0][0]*A[1][1] \  
3         - A[0][1]*A[1][0]  
4     return d
```

3.3.4.

```
1 def matVet(A, x):  
2     n = len(A)  
3     y = [0.]*n  
4     for i in range(n):  
5         for j in range(n):  
6             y[i] += A[i][j]*x[j]  
7     return y
```

4 Elementos da Computação Matricial

Nesta seção, vamos explorar a **NumPy (Numerical Python)**, biblioteca para **tratamento numérico de dados**. Ela é extensivamente utilizada nos mais diversos campos da ciência e da engenharia. Aqui, vamos nos restringir a introduzir algumas de suas ferramentas para a computação matricial.

Usualmente, a biblioteca é importada como segue

```
1 import numpy as np
```

4.1 NumPy array

Um **numpy.array** é uma **tabela de valores** (vetor, matriz ou multidimensional) e contém informação sobre os dados brutos, indexação e como interpretá-los. **Os elementos são todos do mesmo tipo** (diferente de uma lista **Python**), referenciados pela propriedade **dtype**. A **indexação** dos elementos pode ser feita por um **tuple** de inteiros não negativos, por booleanos, por outro **numpy.array** ou por números inteiros. O **ndim** de um **numpy.array** é seu **número de dimensões** (chamadas de **axes**⁷). O **numpy.ndarray.shape** é um **tuple** de inteiros que fornece seu **tamanho (número de elementos) em cada dimensão**. Sua inicialização pode ser feita usando-se listas simples ou encadeadas. Por exemplo,

```
1 a = np.array([1, 3, -1, 2])
2 print(a)
```

```
[ 1  3 -1  2]
```

```
1 a.dtype
```

```
dtype('int64')
```

```
1 a.shape
```

```
(4,)
```

```
1 a[2]
```

```
-1
```

```
1 a[1:3]
```

```
array([ 3, -1])
```

temos um **numpy.array** de números inteiros com quatro elementos dispostos em um único **axis** (eixo). Podemos interpretá-lo como uma representação de um vetor linha ou coluna, i.e.

$$a = (1, 3, -1, 2) \quad (28)$$

vetor coluna ou a^T vetor linha.

Outro exemplo,

```
1 a = np.array([[1.0, 2, 3],
2               [-3, -2, -1]])
3 a.dtype
```

```
dtype('float64')
```

```
1 a.shape
```

```
(2, 3)
```

```
1 a[1,1]
```

```
-2.0
```

temos um `numpy.array` de números decimais (`float`) dispostos em um arranjo com dois **axes** (eixos). O primeiro **axis** tem tamanho 2 e o segundo tem tamanho 3. Ou seja, podemos interpretá-lo como uma matriz de duas linhas e três colunas. Podemos fazer sua representação algébrica como

$$a = \begin{bmatrix} 1 & 2 & 3 \\ -3 & -2 & -1 \end{bmatrix} \quad (29)$$

Exercício 4.1.1. Use `numpy.array` para alocar:

a) o vetor

$$v = (-5, \pi, \sin(\pi/3)) \quad (30)$$

b) a matriz

$$A = \begin{bmatrix} -1 & \frac{1}{3} \\ 2 & \sqrt{2} \\ e^{-1} & -3 \end{bmatrix} \quad (31)$$

4.1.1 Inicialização de um array

O `NumPy` conta com úteis funções de inicialização de `numpy.array`. Vejam algumas das mais frequentes:

- `numpy.zeros`: inicializa um `numpy.array` com todos seus elementos iguais a zero.

```
1 np.zeros(2)
```

```
array([0., 0.])
```

- `numpy.ones`: inicializa um `numpy.array` com todos seus elementos iguais a 1.

```
1 np.ones((3,2), dtype='int')
```

```
array([[1, 1],
       [1, 1],
       [1, 1]])
```


- `numpy.empty`: inicializa um `numpy.array` sem alocar valores para seus elementos⁸.

```
1 np.empty(3)

array([4.9e-324, 1.5e-323, 2.5e-323])
```

- `numpy.arange`: inicializa um `numpy.array` com uma sequência de elementos⁹.

```
1 np.arange(1,6,2)

array([1, 3, 5])
```

- `numpy.linspace`(a, b[, num=n]): inicializa um `numpy.array` como uma sequência de elementos que começa em a, termina em b (incluídos) e contém n elementos igualmente espaçados.

```
1 np.linspace(0, 1, num=5)

array([0. , 0.25, 0.5 , 0.75, 1.  ])
```

Exercício 4.1.2. Aloque a matriz escalar

$$A = \begin{bmatrix} -2 & 0 & 0 \\ 0 & -2 & 0 \\ 0 & 0 & -2 \end{bmatrix} \quad (32)$$

como um `numpy.array`.

Exercício 4.1.3. Construa um `numpy.array` para alocar uma partição uniforme com 11 pontos do intervalo $[0, 1]$. Ou seja, um arranjo $\mathbf{x} = (x_1, x_2, \dots, x_n)$, de elementos $x_i = (i - 1)h$, com passo $h = 1/(n - 1)$.

4.1.2 Manipulação de arrays

Outras duas funções importantes no tratamento de `arrays` são:

- `numpy.reshape`: permite a alteração da forma de um `numpy.array`.

```
1 a = np.array([-2,-1])
2 print(a)
```

```
[-2 -1]
```

```
1 b = a.reshape(2,1)
2 print(b)
```

```
[[ -2]
 [ -1]]
```

O `numpy.reshape` também permite a utilização de um coringa `-1` que será dinamicamente determinado de forma obter-se uma estrutura adequada. Por exemplo,

```
1 a = np.array([[1,2],[3,4]])
2 print(a)
```

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

```
1 b = a.reshape((-1,1))
2 print(b)
```

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

- **numpy.transpose**: computa a transposta de uma matriz.

```
1 a = np.array([[1,2],[3,4]])
2 print(a)
```

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

```
1 b = a.transpose()
2 print(b)
```

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

- **numpy.concatenate**: concatena arrays.

```
1 a = np.array([1,2])
2 b = np.array([2,3])
3 c = np.concatenate((a,b))
4 print(c)
```

```
[1 2 2 3]
```

```
1 a = a.reshape((1,-1))
2 b = b.reshape((1,-1))
3 d = np.concatenate((a,b), axis=0)
4 print(d)
```

```
[[1 2]
 [2 3]]
```

Exercício 4.1.4. Aloque o seguinte vetor como um **numpy.array**

$$\mathbf{a} = (2, 3, -1, 1, 4, -5). \quad (33)$$

Então, use o método **numpy.reshape** para, a partir de **b**, alocar a matriz

$$A = \begin{bmatrix} -1 & 1 \\ 4 & 5 \end{bmatrix} \quad (34)$$

como um **numpy.array**.

Exercício 4.1.5. Tendo em vista que

$$A = \begin{bmatrix} \frac{\sqrt{3}}{2} & -\frac{1}{2} \\ \frac{1}{2} & \frac{\sqrt{3}}{2} \end{bmatrix} \quad (35)$$

é uma matriz ortogonal¹⁰, compute A^{-1} .

Exercício 4.1.6. Considere o seguinte sistema de equações

$$\begin{aligned} 2x_1 - x_2 &= 3 \\ x_1 + 3x_2 &= -2 \end{aligned} \quad (36)$$

Use `numpy.array` para alocar:

1. a matriz de coeficientes deste sistema.
2. o vetor dos termos constantes deste sistema.
3. a matriz estendida deste sistema.

4.1.3 Operadores Elemento-a-Elemento

Os operadores aritméticos disponível no `Python` atuam elemento-a-elemento nos `numpy.arrays`. Por exemplo,

```
1 a = np.array([1, 2])
2 b = np.array([2, 3])
3 a+b
```

```
array([3, 5])
```

```
1 a-b
```

```
array([-1, -1])
```

```
1 b*a
```

```
array([2, 6])
```

```
1 a**b
```

```
array([1, 8])
```

```
1 2*b
```

```
array([4, 6])
```

O `NumPy` também conta com várias funções matemáticas elementares que operam elemento-a-elemento em `arrays`. Por exemplo,

```
1 a = np.array([np.pi, np.sqrt(2)])
2 a
```

```
array([3.14159265, 1.41421356])
```

```
1 np.sin(a)
array([1.22464680e-16, 9.87765946e-01])
1 np.exp(a)
array([23.14069263, 4.11325038])
```

Exercício 4.1.7. Compute os valores da função cosseno para os elementos do vetor

$$\theta = (0., 30^\circ, 45^\circ, 60^\circ, 90^\circ). \quad (37)$$

Respostas dos Exercícios

4.1.1.

```
1 import numpy as np
2 # a)
3 v = np.array([-5., np.pi, np.sin(np.pi/3)])
4 print('v = ', v)
5 # b)
6 A = np.array([[ -1., 1./3],
7               [ 2., np.sqrt(2)],
8               [np.exp(-1.), -3.]])
9 print('A = \n', A)
```

4.1.2.

```
1 import numpy as np
2 A = -2*np.ones((3,3))
3 print('A = \n', A)
```

4.1.3.

```
1 import numpy as np
2 x = np.linspace(0., 1., 11)
3 print('x = ', x)
```

4.1.4.

```
1 import numpy as np
2 a = np.array([2, 3, -1, 1, 4, 5])
3 A = a.reshape((3,-1))
```

4.1.5.

```
1 import numpy as np
2 A = np.array([[np.sqrt(3)/2, -1./2],
3               [1./2, np.sqrt(3)/2]])
4 Ainv = A.transpose()
```

4.1.6.

```

1 import numpy as np
2 # a)
3 A = np.array([[2, -1],
4               [1, 3]])
5 # b)
6 b = np.array([3, -2])
7 # c)
8 E = np.concatenate((A, b.reshape(-1,1)), axis=1)

```

4.1.7.

```

1 import numpy as np
2 thta = np.array([0., np.pi/6, np.pi/4,
3                  np.pi/3, np.pi/2])
4 y = np.cos(thta)

```

4.2 Elementos da Álgebra Linear

O `numpy` conta com um módulo de álgebra linear, usualmente importado com

```
1 import numpy.linalg as npla
```

4.2.1 Vetores

Um vetor podem ser alocado usando um `numpy.array` de um eixo (dimensão). Por exemplo,

$$x = (2, -1), \quad (38)$$

$$y = (3, 1, \pi) \quad (39)$$

podem ser alocados com

```

1 x = np.array([2, -1])
2 print(x)

```

[2 -1]

e

```

1 y = np.array([3, 1, np.pi])
2 print(y)

```

[3. 1. 3.14159265]

Exercício 4.2.1. Aloque cada um dos seguintes vetores como um `numpy.array`:

a) $x = (1.2, -3.1, 4)$

c) $z = (\pi, \sqrt{2}, e^{-2})$

4.2.2 Produto Escalar e Norma

Dados dois vetores

$$x = (x_0, x_1, \dots, x_{n-1}), \quad (40)$$

$$y = (y_0, y_1, \dots, y_{n-1}), \quad (41)$$

define-se o **produto escalar** por

$$x \cdot y = x_0 y_0 + x_1 y_1 + \dots + x_{n-1} y_{n-1}. \quad (42)$$

Com o NumPy, podemos computá-lo com a função `numpy.dot`. Por exemplo,

```
1 x = np.array([-1, 0, 2])
2 y = np.array([0, 1, 1])
3 d = np.dot(x, y)
4 print(d)
```

2

A norma l_2 de um vetor é definida por

$$\|x\|_2 = \sqrt{\sum_{i=0}^{n-1} x_i^2}. \quad (43)$$

O NumPy conta com o método `numpy.linalg.norm` para computá-la. Por exemplo,

```
1 nrm = np.linalg.norm(y)
2 print(nrm)
```

4.457533443631058

Exercício 4.2.2. Faça um código para computar o produto escalar $x \cdot y$ sendo

$$x = (1.2, \ln(2), 4), \quad (44)$$

$$y = (\pi^2, \sqrt{3}, e) \quad (45)$$

4.2.3 Matrizes

Uma matriz pode ser alocada como um `numpy.array` de dois eixos (dimensões). Por exemplo, as matrizes

$$A = \begin{bmatrix} 2 & -1 & 7 \\ 3 & 1 & 0 \end{bmatrix}, \quad (46)$$

$$B = \begin{bmatrix} 4 & 0 \\ 2 & 1 \\ -8 & 6 \end{bmatrix} \quad (47)$$

podem ser alocadas como segue

```
1 A = np.array([[2, -1, 7],
2               [3, 1, 0]])
3 print(A)
```

```
[[ 2 -1  7]
 [ 3  1  0]]
```

e

```
1 B = np.array([[4,0],
2               [2,1],
3               [-8,6]])
4 print(B)
```

```
[[ 4  0]
 [ 2  1]
 [-8  6]]
```

Como já vimos, o **NumPy** conta com operadores elemento-a-elemento que podem ser utilizados na álgebra envolvendo **arrays**, logo também aplicáveis a matrizes (consulte a Subseção 4.1.3). Na sequência, vamos introduzir outras operações próprias deste tipo de objeto.

Exercício 4.2.3. Aloque cada uma das seguintes matrizes como um **numpy.array**:

a)

$$A = \begin{bmatrix} -1 & 2 \\ 2 & -4 \\ 6 & 0 \end{bmatrix} \quad (48)$$

b) $B = A^T$

Exercício 4.2.4. Seja

```
1 A = np.array([[2,1],[1,1],[-3,-2]])
```

Determine o formato (**shape**) dos seguintes **arrays**:

a) $A[:,0]$

b) $A[:,0:1]$

c) $A[1:3,0]$

d) $A[1:3,0:1]$

e) $A[1:3,0:2]$

4.2.4 Inicialização de Matrizes

Além das inicializações de **arrays** já estudadas na Subseção 4.1.1, temos mais algumas que são particularmente úteis no caso de matrizes.

- **numpy.eye**(n): retorna a matriz identidade $n \times n$.

```
1 I = np.eye(3)
2 print(I)
```

```
[[1.  0.  0.]
```

```
[0. 1. 0.],
[0. 0. 1.]]
```

- `numpy.diag`: extrai a diagonal ou constrói um `numpy.array` diagonal.

```
1 D = np.diag([1,2,3])
2 print(D)
```

```
[[1, 0, 0],
 [0, 2, 0],
 [0, 0, 3]]
```

Exercício 4.2.5. Aloque a matriz dos coeficientes e o vetor dos termos constantes do seguinte sistema de equações

$$\begin{aligned} x_1 &= 0 \\ -x_{i-1} + 2x_i - x_{i+1} &= h^2 f_i \\ x_n &= 0 \end{aligned} \quad (49)$$

onde $f_i = \pi^2 \sin(\pi x_i)$, $x_i = (i-1)h$, $h = 1/(n-1)$, $n=5$.

4.2.5 Multiplicação de Matrizes

A multiplicação da matriz $A = [a_{ij}]_{i,j=0}^{n-1,l-1}$ pela matriz $B = [b_{ij}]_{i,j=0}^{l-1,m-1}$ é a matriz $C = AB = [c_{ij}]_{i,j=0}^{n-1,m-1}$ tal que

$$c_{ij} = \sum_{k=0}^{l-1} a_{ik} b_{k,j} \quad (50)$$

O `numpy` tem a função `numpy.matmul` para computar a multiplicação de matrizes. Por exemplo, a multiplicação das matrizes dadas em (46) e (47), computamos

```
1 C = np.matmul(A,B)
2 print(C)
```

```
[[ -50  41],
 [ 14   1]]
```

Observação 4.2.1 (`matmul`, `*`, `@`). É importante notar que `numpy.matmul(A,B)` é a multiplicação de matrizes, enquanto que `*` consiste na multiplicação elemento a elemento. Alternativamente a `numpy.matmul(A,B)` pode-se usar `A @ B`.

Exercício 4.2.6. Aloque as matrizes

$$C = \begin{bmatrix} 1 & 2 & -1 \\ 3 & 2 & 1 \\ 0 & -2 & -3 \end{bmatrix} \quad (51)$$

$$D = \begin{bmatrix} 2 & 3 \\ 1 & -1 \\ 6 & 4 \end{bmatrix} \quad (52)$$

$$E = \begin{bmatrix} 1 & 2 & 1 \\ 0 & -1 & 3 \end{bmatrix} \quad (53)$$

Então, se existirem, compute e forneça as dimensões das seguintes matrizes

- a) CD
- b) $D^T E$
- c) $D^T C$
- d) DE

4.2.6 Traço e Determinante de uma Matriz

O `numpy` tem a função `numpy.ndarray.trace` para computar o **traço** de uma matriz (soma dos elementos de sua diagonal). Por exemplo,

```
1 A = np.array([[ -1, 2, 0], [2, 3, 1], [1, 2, -3]])
2 print('tr(A) = ', A.trace())
```

```
tr(A) = -1
```

Já, o **determinante** é fornecido no módulo `numpy.linalg`. Por exemplo,

```
1 A = np.array([[ -1, 2, 0], [2, 3, 1], [1, 2, -3]])
2 print('det(A) = ', np.linalg.det(A))
```

```
det(A) = 25.000000000000007
```

Exercício 4.2.7. Compute a solução do seguinte sistema de equações

$$\begin{aligned} x_1 - x_2 + x_3 &= -2 \\ 2x_1 + 2x_2 + x_3 &= 5 \\ -x_1 - x_2 + 2x_3 &= -5 \end{aligned} \quad (54)$$

pelo método de Cramer¹¹.

4.2.7 Rank e Inversa de uma Matriz

O **rank** de uma matriz é o número de linhas ou colunas linearmente independentes. O `numpy` conta com a função `numpy.linalg.matrix_rank` para computá-lo. Por exemplo,

```
1 np.linalg.matrix_rank(np.eye(3))
```

```
3
```

```
1 A = np.array([[1, 2, 3], [-1, 1, -1], [0, 3, 2]])
2 np.linalg.matrix_rank(A)
```

```
2
```

O método `numpy.linalg.inv` pode ser usado para computar a **inversa de uma matriz** *full rank*. Por exemplo,

```

1 A = np.array([[1, 2, 3],
2               [-1, 1, -1],
3               [1, 3, 2]])
4 Ainv = np.linalg.inv(A)
5 print('Ainv @ A = \n', Ainv @ A)

Ainv @ A =
[[ 1.00000000e+00 -2.22044605e-16 -8.88178420e-16]
 [ 0.00000000e+00  1.00000000e+00  0.00000000e+00]
 [ 0.00000000e+00 -2.22044605e-16  1.00000000e+00]]

```

Exercício 4.2.8. Compute, se possível, a matriz inversa de cada uma das seguintes matrizes

$$B = \begin{bmatrix} 2 & -1 \\ -2 & 1 \end{bmatrix} \quad (55)$$

$$C = \begin{bmatrix} -2 & 0 & 1 \\ 3 & 1 & -1 \\ 2 & 1 & 0 \end{bmatrix} \quad (56)$$

Verifique suas respostas.

4.2.8 Autovalores e Autovetores de uma Matriz

Um **auto-par** (λ, v) de uma matriz A , λ um escalar chamado de **autovalor** e $v \neq 0$ é um vetor chamado de **autovetor**, é tal que

$$A\lambda = \lambda v. \quad (57)$$

O **numpy** tem a função **numpy.linalg.eig** para computar os auto-pares de uma matriz. Por exemplo,

```

1 lmbda, v = npla.eig(np.eye(3))
2 print('autovalores = \n', lmbda)
3 print('autovetores = \n', v)

```

```

autovalores =
[1.  1.  1.]
autovetores =
[[1.  0.  0.]
 [0.  1.  0.]
 [0.  0.  1.]]

```

Observamos que a função retorna um **tuple** de **numpy.arrays**, sendo que o primeiro contém os autovalores (repetidos conforme suas multiplicidades) e o segundo item é a matriz dos autovetores (dispostos nas colunas).

Exercício 4.2.9. Compute os auto-pares da matriz

$$A = \begin{bmatrix} 1 & 3 & 2 \\ 3 & 2 & -1 \\ 2 & -1 & 1 \end{bmatrix}. \quad (58)$$

Então, verifique se, de fato, $Av = \lambda v$ para cada auto-par (λ, v) computado.

Respostas dos Exercícios

4.2.1.

```
1 import numpy as np
2 # a)
3 x = np.array([1.2, -3.1, 4])
4 print('x = ', x)
5 # b)
6 z = np.array([np.pi, np.sqrt(2.), np.exp(-2)])
7 print('z = ', z)
```

4.2.2.

```
1 import numpy as np
2 x = np.array([1.2, np.log(2), 4])
3 y = np.array([np.pi**2, np.sqrt(3), np.e])
4 d = np.dot(x,y)
```

4.2.3.

```
1 import numpy as np
2 # a)
3 A = np.array([[ -1,  2],
4               [ 2, -4],
5               [ 6,  0]])
6 # b)
7 B = A.transpose()
```

4.2.4.

```
1 import numpy as np
2 # a)
3 A = np.array([[2, 1],
4               [1, 1],
5               [-3, -2]])
6 print('a)', A[:,0].shape)
7 print('b)', A[:,0:1].shape)
8 print('c)', A[1:3,0].shape)
9 print('d)', A[1:3,0:1].shape)
10 print('e)', A[1:3,0:2].shape)
```

4.2.5.

```
1 import numpy as np
2 n = 5
3 h = 1./(n-1)
4 x = np.linspace(0., 1., n)
5 b = np.pi**2*np.sin(np.pi*x)
6 A = np.diag(2*np.ones(n)) + \
7     np.diag(-np.ones(n-1), k=-1) + \
```

```

8     np.diag(-np.ones(n-1), k=1)
9 A[0,0] = 1.
10 A[0,1] = 0.
11 A[n-1,n-2] = 0.
12 A[n-1,n-1] = 1.
13 print('A = \n', A)
14 print('b = \n', b)

```

4.2.6.

```

1 import numpy as np
2 C = np.array([[1, 2, -1],
3              [3, 2, 1],
4              [0, -2, -3]])
5 D = np.array([[2, 3],
6              [1, -1],
7              [6, 4]])
8 E = np.array([[1, 2, 1],
9              [0, -1, 3]])
10 print('a) CD = \n', C@D)
11 print("b) não existe D'E")
12 print("c) D'C = \n", C.T@C)
13 print("d) DE = \n", D@E)

```

4.2.7.

```

1 import numpy as np
2 import numpy.linalg as npla
3 # matriz dos coefs
4 A = np.array([[1, -1, 1],
5              [2, 2, 1],
6              [-1, -1, 2]])
7 # vetor dos termos consts
8 b = np.array([-2, 5, -5])
9 # mat aux A1
10 A1 = A.copy()
11 A1[:,0] = b
12 # sol x1
13 x1 = npla.det(A1)/npla.det(A)
14 print('x1 = ', x1)
15 # mat aux A2
16 A2 = A.copy()
17 A2[:,1] = b
18 # sol x2
19 x2 = npla.det(A2)/npla.det(A)
20 print('x2 = ', x2)
21 # mat aux A3
22 A3 = A.copy()
23 A3[:,2] = b
24 # sol x3
25 x3 = npla.det(A3)/npla.det(A)

```

```
26 print('x3 = ', x3)
```

4.2.8.

```
1 import numpy as np
2 import numpy.linalg as npla
3
4 def inv(A):
5     if (npla.matrix_rank(A) == A.shape[1]):
6         return npla.inv(A)
7     else:
8         print('Matriz não invertível.')
9         return None
10
11 B = np.array([[2, -1],
12              [-2, 1]])
13 print('inv(B) = \n', inv(B))
14
15 A = np.array([[-2, 0, 1],
16              [3, 1, -1],
17              [2, 1, 0]])
18 print('inv(A) = \n', inv(A))
```

4.2.9.

```
1 import numpy as np
2 import numpy.linalg as npla
3 A = np.array([[1, 3, 2],
4              [3, 2, -1],
5              [2, -1, 1]])
6 lmbda, v = np.linalg.eig(A)
7 # testando os auto-pares
8 print(npla.norm(A @ v[:, 0] - lmbda[0] * v[:, 0]) < 1
9       e-10)
9 print(npla.norm(A @ v[:, 1] - lmbda[1] * v[:, 1]) < 1
10       e-10)
10 print(npla.norm(A @ v[:, 2] - lmbda[2] * v[:, 2]) < 1
11        e-10)
```

5 Gráficos

A `matplotlib` é uma biblioteca `Python` livre e gratuita para a visualização de dados. É muito utilizada para a criação de gráficos estáticos, animados ou iterativos. Aqui, vamos introduzir alguma de suas ferramentas básicas para gráficos.

Usualmente, importamos a biblioteca com

```
1 import matplotlib.pyplot as plt
```

5.1 Gráfico de uma função

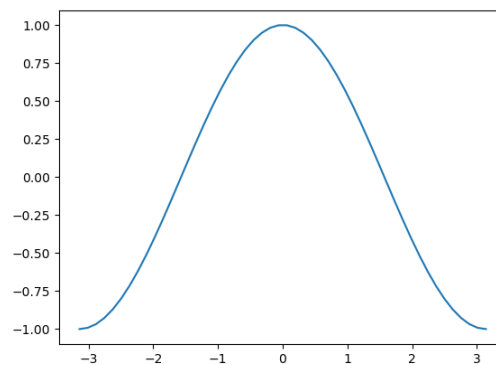


Figura 1: Esboço do gráfico da função $y = \cos(x)$ no intervalo $[-\pi, \pi]$.

A função `matplotlib.pyplot.plot(x,y)` pode ser usada para criarmos gráficos, onde `x` e `y` são `numpy.arrays` que fornecem os pontos cartesianos $\{(x_i, y_i)\}_i$ a serem plotados. Por exemplo,

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 x = np.linspace(-np.pi, np.pi)
4 y = np.cos(x)
5 plt.plot(x,y)
6 plt.show()
```

produz um esboço do gráfico da função $y = \cos(x)$ no intervalo $[-\pi, \pi]$. Consulte a Figura 1.

Observação 5.1.1. `matplotlib` é uma poderosa ferramenta para a visualização de gráficos. Consulte a galeria de exemplos no seu site oficial

<https://matplotlib.org/stable/gallery/index.html>

Exercício 5.1.1. Crie um esboço do gráfico de cada uma das seguintes funções no intervalo indicado:

- a) $y = \cos(x)$, $[0, 2\pi]$
- b) $y = x^2 - x + 1$, $[-2, 2]$
- c) $y = \operatorname{tg}\left(\frac{\pi}{2}x\right)$, $(-1, 1)$

Respostas dos Exercícios

5.1.1.

a)

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 x = np.linspace(0, 2*np.pi)
4 y = np.cos(x)
5 plt.plot(x, y, ls='--')
6 plt.show()
```

b)

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 x = np.linspace(-2, 2)
4 plt.plot(x, x**2-x+1, color='red')
5 plt.grid()
6 plt.show()
```

c)

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 x = np.linspace(-1, 1)
4 y = np.tan(np.pi/2*x)
5 plt.plot(x, y)
6 plt.ylim(-10, 10)
7 plt.xlabel('x')
8 plt.ylabel('y')
9 plt.grid()
10 plt.show()
```

Notas

¹Bhaskara Akaria, 1114 - 1185, matemático e astrônomo indiano. Fonte: [Wikipédia: Bhaskara II](#).

²George Boole, 1815 - 1864, matemático britânico. Fonte: [Wikipédia: George Boole](#).

³Isso constrói um dicionário vazio, como estudaremos logo mais.

⁴O gráfico de uma função restrita a um conjunto A é o conjunto $G(f)|_A = \{(x, y) : x \in A, y = f(x)\}$.

⁵Leonardo Fibonacci, 1170 - 1250, matemático italiano. Fonte: [Wikipédia: Leonardo Fibonacci](#).

⁶Matemática Babilônica, matemática desenvolvida na Mesopotâmia, desde os Sumérios até a queda da Babilônia em 539 a.C.. Fonte: [Wikipédia](#).

⁷**axes**, do inglês, plural de *axis*, eixo.

⁸Atenção! No momento da alocação, os valores dos elementos serão dinâmicos conforme “lixo” da memória.

⁹Similar à função Python **range**.

¹⁰ A é dita matriz ortogonal, quando $A^{-1} = A^T$.

¹¹Gabriel Cramer, 1704 - 1752, matemático suíço. Fonte: [Wikipédia: Gabriel Cramer](#).

Referências

- [1] Banin, S.L.. Python 3 - Conceitos e Aplicações - Uma Abordagem Didática, Saraiva: São Paulo, 2021. ISBN: 978-8536530253.
- [2] NumPy Developers. NumPy documentation, versão 1.26, disponível em <https://numpy.org/doc/stable/>.
- [3] Ribeiro, J.A.. Introdução à Programação e aos Algoritmos, LTC: São Paulo, 2021. ISBN: 978-8521636410.
- [4] Hunter, J.; Dale, D.; Firing, E.; Droettboom, M. & Matplotlib development team. NumPy documentation, versão 3.8.3, disponível em <https://matplotlib.org/stable/>.
- [5] Python Software Foundation. Python documentation, versão 3.12.2, disponível em <https://docs.python.org/3/>.
- [6] Wazlawick, R.. Introdução a Algoritmos e Programação com Python - Uma Abordagem Dirigida por Testes, Grupo GEN: São Paulo, 2021. ISBN: 978-8595156968.

Índice de Comandos

False, 12–14
True, 12–14
break, 26
continue, 26
dict, 9, 21
float, 8
for, 26
input, 9
int, 8
len, 15
list, 9, 18–21, 29
 .append, 20
 .extend, 20
math, 11
 .factorial, 27
 .log10, 11
 .log, 11
 .sqrt, 11
matplotlib, 46
 .pyplot
 .plot, 46
numpy, 37, 40–42
 .arange, 33
 .array, 31–35, 37–40, 42, 46
 .concatenate, 34
 .diag, 40
 .dot, 38
 .empty, 33
 .eye, 39
 .linalg, 41
 .eig, 42
 .inv, 41
 .matrix_rank, 41
 .norm, 38
 .linspace, 33
 .matmul, 40
 .ndarray
 .shape, 31
 .trace, 41
 .ones, 32
 .reshape, 33, 34
 .transpose, 34
 .zeros, 32
range, 27
set, 14, 16
sum, 20
tuple, 9, 17, 19, 42
while, 26
zip, 20