Python para Matemática

Pedro H A Konzen

26 de setembro de 2024

Conteúdo

1 Licença

2 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/

2.1 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**.

2.1.1 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`

2.1.2 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/`

2.2 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 cógido `arqnome.py` em um terminal do sistema operacional.

Exemplo 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.

```python
s = "Olá, Mundo!"
#imprime a string s
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

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

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

```
Olá, Mundo!
```

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

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

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

Primeiramente, escrevemos o código

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

em um IDE (ou em um simples editor de texto) e salvamo-lo no caminho `/caminho/ola.py`. Então, o executamos no console/*notebook* Python com

```
>>> exec(open('/pasta/codigo.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

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

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

3 Elementos da Linguagem

3.1 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

```
x = 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

```
print(x)
```

```
2
```

```
print(y)
```

```
6.0
```

```
print(type(x), type(y))
```

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

Observação 3.1. (Comentários e Continuação de Linha.) Códigos Python admitem **comentários** e **continuação de linha** como no seguinte exemplo

```
# isto é um comentário
s = "isto é uma \
string"
print(s)
```

```
isto é uma string
```

```
  type(s)
```

```
<class 'str'>
```

Observação 3.2. (Notação científica.) O Python aceita **notação científica**. Por exemplo 5.2×10^{-2} é digitado da seguinte forma

```
5.2e-2
```

```
0.052
```

Observação 3.3. (*Casting*.) Quando não há ambiguidade, pode-se fazer a conversão entre objetos de classes diferentes (*casting*). Por exemplo,

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

```
1 <class 'int'>
```

```
y = float(x)
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 3.1.1. Antes de implementar, diga qual o valor de `x` após as seguintes instruções.

```
x = 1
y = x
y = 0
```

Justifique seu resposta e verifique-a.

Exercício 3.1.2. Implemente um código em que a(o) usuária(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.

3.2 Operações Aritméticas Elementares

Os operadores aritméticos elementares são:

`+` **adição**

- `-` **subtração**
- `*` **multiplicação**
- `/` **divisão**
- `**` **potenciação**
- `%` **módulo**
- `//` **divisão inteira**

Exemplo 3.1. Estudamos a seguinte computação

```
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

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

```
5.5
```

Observação 3.4. (Precedência das Operações.) Consulte mais informações sobre a precedência de operadores em Python Docs: Operator Precedence.

Exercício 3.2.1. Compute as raízes do seguinte polinômio quadrático

$$p(x) = 2x^2 - 2x - 4 \tag{1}$$

usando a fórmula de Bhaskara[1].

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

```
5 % 2
```

```
1
```

```
5 // 2
```

```
2
```

Exercício 3.2.2. Use o Python para computar os inteiros não negativos q e r tais que

$$25 = q \cdot 3 + r, \tag{2}$$

sendo r o menor possível.

3.3 Funções e Constantes Elementares

O **módulo** Python `math` disponibiliza várias funções e constantes elementares. Para usá-las, precisamos importar o módulo em nosso código

```
import math
```

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

```
math.pi
```

```
3.141592653589793
```

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

```
-1.0
```

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

```
1.4142135623730951
```

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

```
1.0
```

Observação 3.5. (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 3.3.1. Compute

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

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

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

Exercício 3.3.2. Refaça o Exercício 3.2.1 usando a função `math.sqrt` para computar a raiz quadrada do discriminante.

3.4 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

```
x = 2
x + x == 4
```

```
True
```

Exercício 3.4.1. Considere a circunferência de equação

$$c: \; (x-1)^2 + (y+1)^2 = 1. \tag{3}$$

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

Exercício 3.4.2. Antes de implementar, diga qual é o valor lógico da instrução

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

Justifique sua resposta e verifique!

3.5 Operadores Lógicos Elementares

Os **operadores lógicos** elementares são:

`and` **e lógico**

`or` **ou lógico**

`not` **não lógico**

Exemplo 3.2. (Tabela Booleana do `and`.) A tabela booleana[2] do `and` é

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

Por exemplo, temos

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

```
False
```

Exercício 3.5.1. Construa as tabelas booleanas do operador `or` e do `not`.

Exercício 3.5.2. Use Python para verificar se $1.4 <= \sqrt{2} < 1.5$.

Exercício 3.5.3. Considere um retângulo $r: \; ABDC$ de vértices $A = (1, 1)$ e $D = (2, 3)$. Crie um código em que a(o) usuária(o) informa as coordenadas de um ponto $P = (x, y)$ e o código imprime `True` ou `False` para cada um dos seguintes itens:

1. $P \in r$.

2. $P \in \partial r$.

3. $P \notin \overline{r}$.

Exercício 3.5.4. 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`.

3.6 set

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

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

```
<class 'set'>
```

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

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

```
a == b
```

```
True
```

```
# conjunto vazio
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 `{}`[1].

Observação 3.6. (Tamanho de uma Coleção de Objetos.) A função `len` retorna o número de elementos de uma coleção de objetos. Por exemplo,

```
len(a)
```

[1]Isso constrói um dicionário vazio, como estudaremos logo mais.

```
3
```

Operadores envolvendo conjuntos:

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

Exemplo 3.3. Os conjuntos

$$A = \{2, \pi, -0.25, 3, \text{'banana'}\}, \tag{4}$$
$$B = \{\text{'laranja'}, 3, \arccos(-1), -1\} \tag{5}$$

podem ser alocados como `sets`

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

e, então, podemos computar:

a) $A \setminus B$

```python
a = A - B
print(a)
```

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

b) $A \cup B$

```python
b =  A | B
print(b)
```

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

c) $A \cap B$

```python
c = A & B
print(c)
```

```
{3, 3.141592653589793}
```

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

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

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

Exercício 3.6.1. Aloque como `set` cada um dos seguintes conjuntos:

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

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 3.7. (Compreensão de `sets`.) Python disponibiliza a sintaxe de compreensão de `sets`. Por exemplo,

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

```
{'banana'}
```

Exercício 3.6.2. Considere o conjunto

$$Z = \{-4, -3, -2, -1, 0, 1, 2, 3, 4\}. \tag{6}$$

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

3.7 `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`.

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

```
(1, 2) <class 'tuple'>
```

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

```
(-1, 1, 0) 3
```

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

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

```
{2, -1}
```

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

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

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

Operadores básicos:

`+` concatenação

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

```
(1, 2, 3, 4, 5)
```

`*` repetição

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

```
(1, 2, 1, 2)
```

in pertencimento

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

```
True
```

Exercício 3.7.1. Use sets para alocar os conjuntos

$$A = \{-1, 0, 2\}, \tag{7}$$
$$B = \{2, 3, 5\}. \tag{8}$$

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 3.7).

Exercício 3.7.2. Aloque o gráfico discreto da função[2] $f(x) = x^2$ para $x = 0, \frac{1}{2}, 1, 2$. Dica: use a sintaxe de compreensão de conjuntos (consulte a Observação 3.7).

3.8 list

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

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

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

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

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

```
vazia = []
print(len(vazia))
print(len(y))
```

```
0
4
```

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

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,

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

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

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

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

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

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

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

Observação 3.8. `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 3.9. (Alocação *versus* Cópia.) Estudamos o seguinte exemplo

```python
x = [2, 3, 1]
y = x
y[1] = 0
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-e-versa. Para desassociar y de x, y precisa receber uma cópia de x, como segue

```python
x = [2, 3, 1]
print('id(x) =', id(x))
y = x.copy()
print('id(y) =', id(y))
y[1] = 0
print('x =', x)
print('y =', y)
```

```
id(x) = 140476759980864
id(y) = 140476760231360
x = [2, 3, 1]
y = [2, 0, 1]
```

Observação 3.10. (Anexar ou Estender.) Um `list` tem tamanho dinâmico, premitindo a anexação de um novo item ou sua estensão. A anexação de um item pode ser feita com o método `list.append`, equanto que a extensão é feita com `list.extend`. Por exemplo, com o `list.append` temos

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

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

Equanto, que com o `list.extend` obtemos

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

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

Exercício 3.8.1. A solução de

$$x^2 - 2 = 0 \tag{9}$$

pode ser aproximada pela iteração[3]

$$x_0 = 1, \tag{10}$$

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

para $i = 0, 1, 2, \ldots$. 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 3.8.2. Aloque cada um dos seguintes vetores como um `list`:

$$x = (-1, 3, -2), \tag{12}$$

$$y = (4, -2, 0). \tag{13}$$

Então, compute

a) $x + y$

b) $x \cdot y$

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

Exercício 3.8.3. Uma matriz pode ser alocada como um encadeamento de `lists`. Por exemplo, a matriz

$$M = \begin{bmatrix} 1 & -2 \\ 2 & 3 \end{bmatrix} \tag{14}$$

pode ser alocada como a seguinte `list`

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

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

Use `list` para alocar a matriz

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

[3]Iteração do método babilônico. Saiba mais em Wikipédia: Raiz quadrada.

e o vetor

$$x = (2, -3, 1), \tag{16}$$

então compute Ax.

3.9 dict

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

```python
a = {'nome': 'triangulo', 'perimetro': 3.2}
print(a, type(a))
```

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

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

```python
a['nome'] = 'triângulo'
print(a[nome])
```

```
'triângulo'
```

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

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

```
vértice B = (3,0)
```

Exercício 3.9.1. Considere a função afim

$$f(x) = 3 - x. \tag{17}$$

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

Exercício 3.9.2. Considere a função quadrática

$$g(x) = x^2 - x - 2 \tag{18}$$

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

4 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.

4.1 Ramificação

Uma **estrutura de ramificação** é uma instrução para a tomada de decisões durante a execução de um programa. No Python, temos disponível a instrução `if-[elif-...-elif-else]`.

4.1.1 `if`

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

$$p(x) = ax^2 + bx + c, \tag{19}$$

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

```
import math as m
a = 1.
b = -1.
c = -2.
dlta = b**2 - 4.*a*c
if (dlta >= 0.):
  x1 = (-b - m.sqrt(dlta))/(2.*a)
  x2 = (-b + m.sqrt(dlta))/(2.*a)
  print('x1 =', x1)
  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 4.1. (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`.

4.1.2 `if-else`

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

```python
import math as m
a = 1.
b = -4.
c = 8.
dlta = b**2 - 4.*a*c
if (dlta >= 0.):
  # raízes reais
  x1 = (-b - m.sqrt(dlta))/(2.*a)
  x2 = (-b + m.sqrt(dlta))/(2.*a)
else:
  # raízes complexas
  rea = -b/(2.*a)
  img = m.sqrt(-dlta)/(2.*a)
  x1 = rea - img*1j
  x2 = rea + img*1j
print('x1 =', x1)
print('x2 =', x2)
```

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

Observação 4.2. (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`.

4.1.3 `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.

```
import math as m
a = 1.
b = 2.
c = 1.
dlta = b**2 - 4.*a*c
if (dlta > 0.):
  # raízes reais
  x1 = (-b - m.sqrt(dlta))/(2.*a)
  x2 = (-b + m.sqrt(dlta))/(2.*a)
elif (dlta == 0.):
  x1 = x2 = -b/(2.*a)
else:
  # raízes complexas
  rea = -b/(2.*a)
  img = m.sqrt(-dlta)/(2.*a)
  x1 = rea - img*1j
  x2 = rea + img*1j
print('x1 =', x1)
print('x2 =', x2)
```

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

Exercício 4.1.1. Desenvolva um código para computar a raiz do polinômio

$$f(x) = ax + b \tag{20}$$

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 raíz ($a = 0$ e $b \neq 0$).

Exercício 4.1.2. Desenvolva um código em que dados três pontos A, B e C no plano, verifique se ABC determia um triângulo. Caso afirmativo, classifique-o como um triângulo equilátero, isósceles ou escaleno.

4.2 Repetição

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

4.2.1 `while`

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

Por exemplo, o seguinte código computa e imprimi os elementos da sequência de Fibonacci[3], enquanto forem menores que 10.

```
n = 1
print(n)
m = 1
print(m)
while (n+m < 10):
  s = m
  m += n
  n = s
  print(m)
```

Verifique!

Observação 4.3. (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 4.2.1. Use `while` para imprimir os dez primeiros números ímpares.

Exercício 4.2.2. Uma aplicação do Método Babilônico[4] para a aproximação

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

da solução da equação $x^2 - 2 = 0$, consiste na iteração

$$x_0 = 1, \tag{21}$$

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

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

4.2.2 `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.

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

Verifique!

4.2.3 `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.

```
n = 1
print(f'1: {n}')
m = 1
print(f'2: {m}')
```

```
for i in range(3,7):
  s = m
  m += n
  n = s
  print(f'{i}: {m}')
```

Verifique!

Exercício 4.2.3. Com n dado, desenvolva um código para computar o valor da soma harmônica

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

Exercício 4.2.4. Desenvolva um código para computar o fatorial de um dado número natural n. Dica: use `math.factorial` para verificar seu código.

4.3 Funções

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

$$f(x) = 2x - 3 \tag{24}$$

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

```
  def f(x):
  y = 2*x - 3
  return x

z = f(2)
print(f'f(2) = {z}')
```

```
f(2) = 2
```

Observação 4.4. Para funções pequenas, pode-se utilizar a instrução `lambda` de funções anônimas. Por exemplo,

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

```
f(3) = 3
```

Exemplo 4.1. (Função com Parâmetros.) O seguinte código, implementa o polinômio de primeiro grau

$$p(x) = ax + b, \tag{25}$$

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

```
def p(x, a=1., b=0.):
  y = a*x + b
  return y

print('p(2) =', p(2.))
print('p(2, 3, -5) =', p(2., 3., -5.))
```

Exercício 4.3.1. Implemente uma função para computar as raízes de um polinômio de grau 2 $p(x) = ax^2 + bx + c$.

Exercício 4.3.2. Implemente uma função que computa o produto escalar de dois vetores

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

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

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

Exercício 4.3.3. Implemente uma função que computa o determinante de matrizes 2×2. Dica: considere que a matriz está alocada com um `list` encadeado.

Exercício 4.3.4. Implemente uma função que computa a multiplicação matriz - vetor Ax, com A matriz 2×2 e x um vetor de dois elementos.

5 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

```python
import numpy as np
```

5.1 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`[5]). 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,

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

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

```python
a.dtype
```

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

```python
a.shape
```

```
(4,)
```

```python
a[2]
```

```
-1
```

```python
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.

$$a = (1, 3, -1, 2) \tag{28}$$

[5]Do inglês, plural de *axis*, eixo.

vetor coluna ou a^T vetor linha.

Outro exemplo,

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

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

```
a.shape
```

```
(2, 3)
```

```
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

$$a = \begin{bmatrix} 1 & 2 & 3 \\ -3 & -2 & -1 \end{bmatrix} \tag{29}$$

5.1.1 Inicialização de um array

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

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

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

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

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

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

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

```
        [1, 1],
        [1, 1]])
```

- `np.empty()`: inicializa um `numpy.array` sem alocar valores para seus elementos[6].

```
np.empty(3)
```

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

- `np.arange()`: inicializa um `numpy.array` com uma sequência de elementos[7].

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

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

- `np.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.

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

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

5.1.2 Manipulação de arrays

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

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

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

```
array([-2, -1])
```

```
a.reshape(2,1)
```

```
array([[-2],
       [-1]])
```

[6] Atenção! Os valores dos elementos serão dinâmicos conforme “lixo” da memória.
[7] Similar a função Python `range`.

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

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

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

```
a.reshape((-1,1))
```

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

- `arr.transpose()`: computa a transposta de uma matriz.

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

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

```
a.transpose()
```

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

- `np.concatenate()`: concatena `arrays`.

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

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

```
a = a.reshape((1,-1))
a.ndim
```

```
2
```

```
b = b.reshape((1,-1))
b
```

```
array([[2, 3]])
```

```
d = np.concatenate((a,b), axis=0)
d
```

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

5.1.3 Operadores Elemento-a-Elemento

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

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

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

```
a-b
```

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

```
b*a
```

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

```
a**b
```

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

```
2*b
```

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

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

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

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

```
np.sin(a)
```

```
array([1.22464680e-16, 9.87765946e-01])
```

```
np.exp(a)
```

```
array([23.14069263,  4.11325038])
```

Observação 5.1. O NumPy contém um série de outras funções práticas para a manipulação de `arrays`. Consulte NumPy: the absolute basics for beginners.

5.2 Elementos da Álgebra Linear

O `numpy` conta com um módulo de álgebra linear

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

5.2.1 Vetores

Um vetor podem ser representado usando um `numpy.array` de um eixo (dimensão) ou um com dois eixos, caso se queira diferenciá-lo entre um vetor linha ou coluna. Por exemplo, os vetores

$$a = (2, -1, 7), \tag{30}$$

$$b = (3, 1, 0)^T \tag{31}$$

podem ser alocados com

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

Caso queira-se que x siga um arranjo em coluna, pode-se modificado como segue

```
a = a.reshape((-1,1))
a
```

```
array([[ 2],
       [-1],
       [ 7]])
```

Como já vimos, o NumPy conta com operadores elemento-a-elemento que podem ser utilizados na álgebra envolvendo `arrays`, logo também aplicáveis a vetores (consulte a Subseção 5.1.3). Vamos, aqui, introduzir outras operações próprias deste tipo de objeto.

Exercício 5.2.1. Aloque cada um dos seguintes vetores como um `numpy.array`:

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

b) $y = x^T$

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

5.2.2 Produto Escalar e Norma

Dados dois vetores,

$$x = (x_0, x_1, \ldots, x_{n-1}), \tag{32}$$
$$y = (y_0, y_1, \ldots, y_{n-1}) \tag{33}$$

define-se o **produto escalar** por

$$x \cdot y = x_0y_0 + x_1y_1 + \cdots + x_{n-1}y_{n-1} \tag{34}$$

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

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

```
-2
```

A norma (euclidiana) de um vetor é definida por

$$\|x\| = \sqrt{\sum_{i=0}^{n-1} x_i^2}. \tag{35}$$

O NumPy conta com a função `np.linalg.norm()` para computá-la. Por exemplo,

```
npla.norm(y)
```

```
1.7320508075688772
```

Exercício 5.2.2. Faça um código para computar o produto escalar $x \cdot y$ sendo

$$x = (1.2, \ln(2), 4), \tag{36}$$
$$y = (\pi^2, \sqrt{3}, e) \tag{37}$$

5.2.3 Matrizes

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

$$A = \begin{bmatrix} 2 & -1 & 7 \\ 3 & 1 & 0 \end{bmatrix}, \tag{38}$$
$$B = \begin{bmatrix} 4 & 0 \\ 2 & 1 \\ -8 & 6 \end{bmatrix} \tag{39}$$

podem ser alocadas como segue

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

```
array([[ 2, -1,  7],
       [ 3,  1,  0]])
```

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

```
array([[ 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 5.1.3). Vamos, aqui, introduzir outras operações próprias deste tipo de objeto.

Exercício 5.2.3. Aloque cada uma das seguintes matrizes como um `numpy.array`:

a)
$$A = \begin{bmatrix} -1 & 2 \\ 2 & -4 \\ 6 & 0 \end{bmatrix} \tag{40}$$

b) $B = A^T$

Exercício 5.2.4. Seja

```
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]`

5.2.4 Inicialização de Matrizes

Além das inicializações de `arrays` já estudadas na Subseção 5.1.1, temos mais algumas que são particularmente úteis no caso de matrizes.

- `np.eye(n)`: retorna a matriz identidade $n \times n$

```
np.eye(3)
```

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

- `np.diag(v)`: retorna uma matriz diagonal formada pela `list v`

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

```
array([[1, 0, 0],
       [0, 2, 0],
```

```
        [0, 0, 3]])
```

Exercício 5.2.5. Aloque a matriz escalar $C = [c_{ij}]_{i,j=0}^{99}$, sendo $c_{ii} = \pi$ e $c_{ij} = 0$ para $i \neq j$.

5.2.5 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

$$c_{ij} = \sum_{k=0}^{l-1} a_{ik} b_{k,j} \tag{41}$$

O numpy tem a função numpy.matmul para computar a multiplicação de matrizes. Por exemplo, a multiplicação das matrizes dadas em (38) e (39), computamos

```
C = np.matmul(A,B)
C
```

```
array([[-50,  41],
[ 14,   1]])
```

Observação 5.2. (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 5.2.6. Aloque as matrizes

$$C = \begin{bmatrix} 1 & 2 & -1 \\ 3 & 2 & 1 \\ 0 & -2 & -3 \end{bmatrix} \tag{42}$$

$$D = \begin{bmatrix} 2 & 3 \\ 1 & -1 \\ 6 & 4 \end{bmatrix} \tag{43}$$

$$E = \begin{bmatrix} 1 & 2 & 1 \\ 0 & -1 & 3 \end{bmatrix} \tag{44}$$

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

a) CD

b) $D^T E$

c) $D^T C$

d) DE

5.2.6 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,

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

```
-1
```

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

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

```
25.000000000000007
```

Exercício 5.2.7. Compute e verifique os traços e os determinantes das seguintes matrizes

$$C = \begin{bmatrix} -2 & 3 \\ 1 & 4 \end{bmatrix} \tag{45}$$

$$D = \begin{bmatrix} 3 & 1 & -1 \\ 1 & 0 & 2 \\ 4 & 2 & -1 \end{bmatrix} \tag{46}$$

5.2.7 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,

```
npla.matrix_rank(np.eye(3))
```

```
3
```

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

```
2
```

A inversa de uma matriz **full rank** pode ser computada com a função `numpy.linalg.inv`. Por exemplo,

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

```
3
```

```
Ainv = np.linalg.inv(A)
np.matmul(A,Ainv)
```

```
array([[ 1.00000000e+00,  0.00000000e+00,  0.00000000e+00],
       [ 1.11022302e-16,  1.00000000e+00,  2.22044605e-16],
       [-2.22044605e-16,  0.00000000e+00,  1.00000000e+00]])
```

Exercício 5.2.8. Compute, se possível, a matriz inversa de cada uma das seguintes matrizes

$$B = \begin{bmatrix} 2 & -1 \\ -2 & 1 \end{bmatrix} \tag{47}$$

$$C = \begin{bmatrix} -2 & 0 & 1 \\ 3 & 1 & -1 \\ 2 & 1 & 0 \end{bmatrix} \tag{48}$$

Verifique suas respostas.

5.2.8 Autovalores e Autovetores de uma Matriz

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

$$A\lambda = \lambda v. \tag{49}$$

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

```
npla.eig(np.eye(3))
```

```
(array([1., 1., 1.]), array([[1., 0., 0.],
[0., 1., 0.],
[0., 0., 1.]]))
```

Observamos que a função uma dupla, sendo o primeiro item um `numpy.array` contendo os autovalores (repetidos conforme suas multiplicidades) e o segundo item é a matriz dos autovetores, onde estes são suas colunas.

Exercício 5.2.9. Compute os auto-pares da matriz

$$A = \begin{bmatrix} 1 & 3 & 2 \\ 3 & 2 & -1 \\ 2 & -1 & 1 \end{bmatrix}. \tag{50}$$

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

6 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.

Para utilizá-la, é necessário instalá-la. Pacotes de instalação estão disponíveis para os principais sistemas operacionais, consule a sua loja de *apps* ou Matplotlib Installation. Para importá-la, usamos

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

Observação 6.1. Se você está usando um console Python remoto, você pode querer adicionar a seguinte linha de comando para que os gráficos sejam visualizados no próprio console.

```
%matplotlib inline
```

Gráficos bidimensionais podem ser criados com a função `matplotlib.pyplot.plot(x,y)`, onde `x` e `y` são `arrays` que fornecem os pontos cartesianos (x_i, y_i) a serem plotados. Por exemplo,

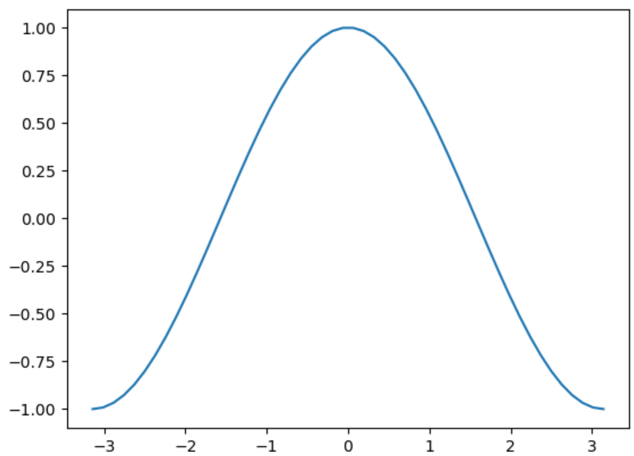

Figura 1: Esboço do gráfico da função $y = \operatorname{sen}(x)$ no intervalo $[-\pi, \pi]$.

```
import matplotlib.pyplot as plt
x = np.linspace(-np.pi, np.pi)
y = np.cos(x)
plt.plot(x,y)
plt.show()
```

produz o seguinte esboço do gráfico da função $y = \operatorname{sen}(x)$ no intervalo $[-\pi, \pi]$. Consulte a Figura 1.

Observação 6.2. 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 6.0.1. 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) $y = \operatorname{tg}\left(\frac{\pi}{2}x\right)$, $(-1, 1)$

Notas

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

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

[3]Leonardo Fibonacci, 1170 - 1250, matemático italiano. Fonte: Wikipédia: Leonardo Fibonacci.

Referências

[1] Banin, S.L.. Python 3 - Conceitos e Aplicações - Uma Abordagem Didática, Saraiva: São Paulo, 2021. ISBN: 978-8536530253.

[2] NumpPy 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.