Algoritmos e Programação I

Pedro H A Konzen

6 de fevereiro de 2024

Licença

Prefácio

Estas notas de aula fazem uma introdução a algoritmos e programação de computadores. Como ferramenta computacional de apoio, a linguagem computacional Python é utilizada.

Agradeço a todos e todas que de modo assíduo ou esporádico contribuem com correções, sugestões e críticas. :)

Pedro H A Konzen

Conteúdo

Capa **i**

Licença **ii**

Prefácio **iii**

Sumário **vii**

1 Introdução **1**

2 Linguagem de Programação **3**

2.1 Computador . . . 3
2.1.1 Linguagem de programação . . . 5
2.1.2 Instalação e execução . . . 8
2.1.3 Exercícios . . . 9
2.2 Algoritmos e Programação . . . 11
2.2.1 Fluxograma . . . 12
2.2.2 Exercícios . . . 16
2.3 Dados . . . 18
2.3.1 Identificadores . . . 19
2.3.2 Alocação de dados . . . 21
2.3.3 Exercícios . . . 23
2.4 Dados Numéricos e Operações . . . 25
2.4.1 Números Inteiros . . . 27
2.4.2 Números Decimais . . . 28
2.4.3 Números Complexos . . . 30
2.4.4 Exercícios . . . 31
2.5 Dados Booleanos . . . 36

2.5.1 Operadores de Comparação 37
2.5.2 Operadores Lógicos 39
2.5.3 Exercícios 42
2.6 Sequência de Caracteres 44
2.6.1 Formatação de *strings* 46
2.6.2 Operações com *strings* 47
2.6.3 Entrada de dados 48
2.6.4 Exercícios 49
2.7 Coleção de Dados 50
2.7.1 Conjuntos: `set` 51
2.7.2 N-uplas: `tuple` 54
2.7.3 Listas: `list` 56
2.7.4 Dicionários: `dict` 59
2.7.5 Exercícios 61

3 Programação Estruturada **67**
3.1 Estruturas de um Programa 68
3.1.1 Sequência 68
3.1.2 Ramificação 70
3.1.3 Repetição 71
3.1.4 Exercícios 74
3.2 Instruções de Ramificação 76
3.2.1 Instrução `if` 76
3.2.2 Instrução `if-else` 78
3.2.3 Instrução `if-elif` 80
3.2.4 Instrução `if-elif-else` 82
3.2.5 Múltiplos Casos 84
3.2.6 Exercícios 85
3.3 Instruções de Repetição 92
3.3.1 Instrução `while` 92
3.3.2 Instrução `for` 95
3.3.3 Exercícios 97

4 Funções **103**
4.1 Funções Predefinidas e Módulos 103
4.1.1 Funções Predefinidas 103
4.1.2 Módulos 106
4.1.3 Exercícios 108

4.2 Definindo Funções . . . 109
4.2.1 Funções com Saída de Dados . . . 111
4.2.2 Capturando Exceções . . . 114
4.2.3 Criando um Módulo . . . 116
4.2.4 Exercícios . . . 118
4.3 Passagem de Parâmetros . . . 122
4.3.1 Variáveis Globais e Locais . . . 123
4.3.2 Parâmetros com Valor Padrão . . . 124
4.3.3 Vários Parâmteros . . . 125
4.3.4 Parâmetros Arbitrários . . . 125
4.3.5 Exercícios . . . 127

5 Arranjos e Matrizes 130
5.1 Arranjos . . . 130
5.1.1 Alocação de Arranjos . . . 130
5.1.2 Indexação e Fatiamento . . . 132
5.1.3 Reordenamento dos Elementos . . . 132
5.1.4 Operações Elemento-a-Elemento . . . 134
5.1.5 Exercícios . . . 136
5.2 Vetores . . . 139
5.2.1 Funções Vetoriais . . . 141
5.2.2 Produto Interno . . . 141
5.2.3 Norma de Vetores . . . 142
5.2.4 Produto Vetorial . . . 143
5.2.5 Exercícios . . . 144
5.3 Arranjos Multidimensionais . . . 146
5.3.1 Alocação, Indexação e Fatiamento . . . 146
5.3.2 Inicialização . . . 148
5.3.3 Manipulação . . . 149
5.3.4 Operações e Funções Elementares . . . 150
5.3.5 Exercícios . . . 152
5.4 Matrizes . . . 155
5.4.1 Operações Matriciais . . . 156
5.4.2 Aplicação: Método de Cramer . . . 161
5.4.3 Exercícios . . . 163

6 Arquivos e Gráficos **169**
6.1 Arquivos . . . 169
6.1.1 Arquivo Texto . . . 169
6.1.2 Arquivo Binário . . . 172
6.1.3 Escrita e Leitura com NumPy . . . 173
6.1.4 Exercícios . . . 173
6.2 Gráficos . . . 175
6.2.1 Área Gráfica . . . 175
6.2.2 Eixos . . . 178
6.2.3 Elementos Gráficos . . . 179
6.2.4 Textos e Anotações . . . 182
6.2.5 Exercícios . . . 184

7 Orientação a Objetos **186**
7.1 Classe e Objeto . . . 186
7.1.1 Exercícios . . . 188
7.2 Herança . . . 191
7.2.1 Exercícios . . . 194

Bibliografia **198**

Capítulo 1

Introdução

Vamos começar executando nossas primeiras **linhas de código** na linguagem de programação Python. Em um **terminal** Python digitamos

```
>>> print('Olá, mundo!')
```

Observamos que `>>>` é o símbolo do `prompt de entrada` e digitamos nossa **instrução** logo após ele. Para executarmos a instrução digitada, teclamos `<ENTER>`. Uma vez executada, o terminal apresentará as seguintes informações

```
>>> print('Olá, mundo!')
Olá, mundo!
>>>
```

Pronto! O fato do símbolo de `prompt de entrada` ter aparecido novamente, indica que a instrução foi completamente executada e o terminal está pronto para executar uma nova instrução.

A **linha de comando** executada acima pede ao computador para imprimir no `prompt de saída` a frase `Olá, mundo!`. O **método** `print` contém instruções para imprimir **objetos** em um dispositivo de saída, no caso, imprime a frase na tela do computador.

Bem! Talvez imprimir no `prompt de saída` uma frase que digitamos no `prompt de entrada` possa parecer um pouco redundante no momento. Vamos considerar um outro exemplo, vamos computar a soma dos números ímpares entre 0 e 100. Podemos fazer isso como segue

```
>>> sum([i for i in range(100) if i%2 != 0])
2500
```

Oh! No momento, não se preocupe se não tenha entendido a linha de comando de entrada, ao longo dessas notas de aula isso vai ficando natural. A linha de comando de entrada usa o método `sum` para computar a soma dos elementos da **lista** de números ímpares desejada. A lista é construída de forma **iterada** e **indexada** pela **variável** `i`, para `i` no intervalo/faixa de 0 a 99, se o resto da divisão de `i` por 2 não for igual a 0. Ok! O resultado computado foi 2500.

De fato, a soma dos números ímpares de 0 a 100

$$(1, 3, 5, \ldots, 99) \tag{1.1}$$

é a soma dos 50 primeiros elementos da progressão aritmética $a_i = 1 + 2i$, $i = 0, 1, \ldots$, i.e.

$$\begin{align}
\sum_{i=0}^{49} a_i &= a_0 + a_1 + \cdots + a_{49} \tag{1.2}\\
&= 1 + 3 + \cdots + 99 \tag{1.3}\\
&= \frac{50(1+99)}{2} \tag{1.4}\\
&= 2500 \tag{1.5}
\end{align}$$

como já esperado! Em Python, esta última conta pode ser computada como segue

```
>>> 50*(1+99)/2
2500.0
```

Capítulo 2

Linguagem de Programação

2.1 Computador

Um computador[1] é um **sistema computacional** de elementos físicos (**hardware**) e elementos lógicos (**software**).

O **hardware** são suas partes mecânicas, elétricas e eletrônicas como: fonte de energia, teclado, mouse/painel tátil, monitor/tela, dispositivos de armazenagem de dados (HDD, *hard disk drive*; SSD, *solid-state drive*; RAM, *random-access memory*; etc.), dispositivos de processamento (CPU, *central processing unit*, GPU, *graphics processing unit*), conectores de dispositivos externos (microfone, caixa de som, fone de ouvido, USB, etc.), placa mãe, etc..

O **software** é toda a informação processada pelo computador, qualquer código executado e qualquer dado usado nas computações.

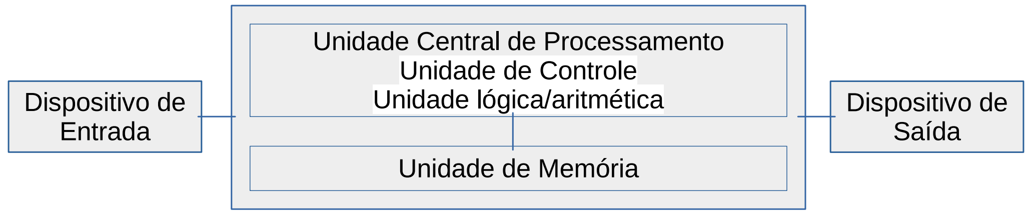


Figura 2.1: Arquitetura de computador de von Neumann.

[1] Consulte Wikipédia: Computador para uma introdução sobre a história e outras questões sobre computadores.

Os computadores que comumente utilizamos seguem a arquitetura de John von Neumann[2], que consiste em dispositivo(s) de entrada de dados, unidade(s) de processamento, unidade(s) de memória e dispositivo(s) de saída de dados (Figura 2.1).

- **Dispositivos de entrada e saída**

  São elementos do computador que permitem a comunicação humana (usuária(o)) com a máquina.

  - **Dispositivos de entrada**
    São elementos que permitem o fluxo de informação da(o) usuária(o) para a máquina. Exemplos são: teclado, mouse/painel tátil, microfone, etc.
  - **Dispositivos de saída**
    São elementos que permitem o fluxo de informação da máquina para a(o) usuária(o). Exemplos são: monitor/tela, alto-falantes, luzes espia, etc.

- **Unidade central de processamento**

  A **CPU** (do inglês, *Central Processing Unit*) é o elemento de processa as informações e é composta de **unidade de controle**, **unidade lógica e aritmética** e de **memória cache**.

  - **Unidade de controle**
    Coordena as execuções do processador: busca e decodifica instruções, lê e escreve no *cache* e controla o fluxo de dados.
  - **Unidade lógica/aritmética**
    Executa as instruções operações lógicas e aritméticas, por exemplo: executar a adição, multiplicação, testar se dois objetos são iguais, etc.
  - **Memória cache**
    Memória interna da CPU muito mais rápida que as memórias RAM e dispositivos e armazenamento HDD/SSD. É um dispositivo de memória de pequena capacidade e é utilizada como memória de curto prazo e diretamente acessada.

[2] John von Neumann, 1903 - 1957, matemático húngaro, naturalizado estadunidense. Fonte: Wikipédia.

- **Unidades de memória**

  As unidades de memória são elementos que permitem o armazenamento de dados/objetos. Como memória principal tem-se a **ROM** (do inglês, *Read Only Memory*) e a **RAM** (do inglês, *Random Access Memory*) e como memória de massa/secundária tem-se HDD, SSD, entre outras.

- **Memória ROM**

  A memória ROM é utilizada para armazenamento de dados/objetos necessários para dar início ao funcionamento do computador. Por exemplo, é onde a BIOS (dos inglês, *Basic Input/Output System*, Sistema Básico de Entrada e Saída) é armazenada. Ao ligarmos o computador este programa é iniciado e é responsável por fazer o gerenciamento inicial dos diversos dispositivos do computador e carregar o **sistema operacional** (conjunto de programas cuja função é de gerenciar os recursos do computador e controlar a execução de programas).

- **Memória RAM**

  Memória de acesso rápido utilizada para dados/objetos de uso frequente durante a execução de programas. É uma memória volátil, i.e. toda a informação guardada nela é perdida quando o computador é desligado.

- **Memória de massa/secundária**

  Memória de massa ou secundária são usadas para armazenar dados/objetos por período longo. Normalmente, são dispositivos HDD ou SSD, os dados/objetos são guardados mesmo que o computador seja desligado e contém grande capacidade de armazenagem.

Os **software** são os elementos lógicos de um sistema computacional, são programas de computadores que contém as instruções que gerenciam o **hardware** para a execução de tarefas específicas, por exemplo, imprimir um texto, gravar áudio/vídeo, resolver um problema matemático, etc. Programar é o ato de criar programas de computadores.

2.1.1 Linguagem de programação

As informações fluem no computador codificadas como registros de *bits*[3] (sequência de zeros ou uns). Há registros de instrução e de dados. Programar

[3] Usualmente de tamanho 64-*bits*.

diretamente por registros é uma tarefa muito difícil, o que levou ao surgimento de linguagens de programação. Uma **linguagem de programação**[4] é um método padronizado para escrever instruções para execução de tarefas no computador. As instruções escritas em uma linguagem são interpretadas e/ou compiladas por um software (interpretador ou compilador) da linguagem que decodifica as instruções em registros de instruções e dados, os quais são efetivamente executados na máquina.

Existem várias linguagens de programação disponíveis e elas são classificadas por diferentes características. Uma **linguagem de baixo nível** (por exemplo, Assembly) é aquela que se restringe às instruções executadas diretamente pelo processador, enquanto que uma **linguagem de alto nível** contém instruções mais complexas e abstratas. Estas contém sintaxe mais próxima da linguagem humana natural e permitem a manipulação de objetos mais abstratos. Exemplos de linguagens de alto nível são: Basic, Java, Javascript, MATLAB, PHP, R, C/C++, Python, etc.

Em geral, não existe uma melhor linguagem, cada uma tem suas características que podem ser mais ou menos adequadas conforme o programa que se deseja desenvolver. Por exemplo, para um site de internet, linguagens como Javascript e PHP são bastante úteis, mas não no desenvolvimento de modelagem matemática e computacional. Nestes casos, C/C++ é uma linguagem mais apropriada por conter várias estruturas de programação que facilitam a modelagem computacional de problemas científicos. Agora, R é uma linguagem de alto nível com diversos recursos dedicados às áreas de ciências de dados e estatística. Usualmente, utiliza-se mais de uma linguagem no desenvolvimento de programas mais avançados. A ideia é de explorar o melhor de cada linguagem na criação de programas eficientes na resolução dos problemas de interesse.

Nestas notas de aula, Python é a linguagem escolhida para estudarmos algoritmos e programação. Trata-se de uma linguagem de alto nível, **interpretada**, **dinâmica** e **mutiparadigma**. Foi lançada por Guido van Rossum[5] em 1991 e, atualmente, é desenvolvida de forma comunitária, aberta e gerenciada pela ONG Python Software Foundation. A linguagem foi projetada para priorizar a legibilidade do código. Parte da filosofia da linguagem é descrita pelo poema The Zen of Python. Pode-se lê-lo pelo *easter egg* Python:

[4]Código de programação, código de máquina ou linguagem de máquina.

[5]Guido van Rossum, 1956-, matemático e programador de computadores holandês. Fonte: Wikipédia.

```
>>> import this
```

- **Linguagem interpretada**

  Python é uma linguagem interpretada. Isso significa que o **código-fonte** escrito em linguagem Python é interpretado por um programa (interpretador Python). Ao executar-se um código, o interpretador lê uma linha do código, decodifica-a como registros para o processador que os executa. Executada uma linha, o interpretador segue para a próxima até o código ter sido completadamente executado.

- **Linguagem compilada**

  Em uma linguagem compilada, como C/C++, há um programa chamado de **compilador** (em inglês, *compiler*) e outro de **ligador** (em inglês, *linker*). O primeiro, cria um programa-objeto a partir do código e o segundo gerencia sua ligação com eventuais bibliotecas computacionais que ele possa depender. O programa-objeto (também chamado de executável) pode então ser executado pela máquina.

Em geral, a execução de um programa compilado é mais rápida que a de um código interpretado. De forma simples, isso se deve ao fato de que nesse a interpretação é feita toda de uma vez e não precisa ser refeita na execução de cada linha de código, como no segundo caso. Por outro lado, a compilação de códigos-fonte grandes pode ser bastante demorada fazendo mais sentido quando ele é compilado uma vez e o programa-objeto executado várias vezes. Além disso, linguagens interpretadas podem usar bibliotecas de programas pré-compiladas. Com isso, pode-se alcançar um bom balanceamento entre tempo de desenvolvimento e de execução do código.

O interpretador Python também pode ser usado para compilar o código para um arquivo **bytecode**, este é executado muito mais rápido do que o código-fonte em si, pois as interpretações necessárias já foram feitas. Mais adiante, vamos estudar isso de forma mais detalhada.

- **Linguagem de tipagem dinâmica**

  Python é uma linguagem de tipagem dinâmica. Nela, os dados não precisam ser explicitamente tipificados no código-fonte e o interpretador os tipifica com base em regras da própria linguagem. Ao executar operações com os dados, o interpretador pode alterar seus tipos de forma dinâmica.

- **Linguagem de tipagem estática**

  C/C++ é um exemplo de uma linguagem de tipagem estática. Em tais linguagens, os dados devem ser explicitamente tipificados no código-fonte com base nos tipos disponíveis. A retipificação pode ocorrer, mas precisa estar explicitamente definida no código.

Existem vários **paradigmas de programação** e a linguagem Python é multiparadigma, i.e. permite a utilização de mais de um no código-fonte. Exemplos de paradigmas de programação são: **estruturada**, **orientada a objetos**, **orientada a eventos**, etc.. Na maior parte destas notas de aulas, vamos estudar algoritmos para linguagens de programação estruturada. Mais ao final, vamos introduzir aspectos de linguagens orientada a objetos. Estes são paradigmas de programação fundamentais e suas estruturas são importantes na programação com demais paradigmas disponíveis em programação de computadores.

2.1.2 Instalação e execução

Python é um **software aberto**[6] e está disponível para vários sistemas operacionais (Linux, macOS, Windows, etc.) no seu site oficial

`https://www.python.org/`

Também, está disponível (gratuitamente) na loja de aplicativos dos sistemas operacionais mais usados. Esta costuma ser a forma mais fácil de instalá-lo na sua máquina, consulte a loja de seus sistema operacional. Ainda, há plataformas e IDEs[7] Python disponíveis, consulte, como por exemplo, Anaconda.

A execução de um código Python pode ser feita de várias formas.

- **Execução iterativa via terminal**

  Em terminal Python pode-se executar instruções/comandos de forma iterativa. Por exemplo:

```
>>> print('Olá, mundo!')
Olá, mundo!
>>>
```

[6]Consulte a licença de uso em `https://docs.python.org/3/license.html`.

[7]IDE, do inglês, *Integrated Development enviroment*, ambiente de desenvolvimento integrado

O símbolo `>>>` denota o **prompt de entrada**, onde uma instrução Python pode ser digitada. Após digitar, o comando é executado teclando `<ENTER>`. Caso o comando tenha alguma saída de dados, como no caso acima, esta aparecerá, por padrão, no **prompt de saída**, logo abaixo a linha de comando executada. Um novo símbolo de prompt de entrada aparece ao término da execução anterior.

- **Execução de um *script***

  Para códigos com várias linhas de instruções é mais adequado utilizar um aquivo de *script* Python. Usando-se um editor de texto ou um IDE ditam-se as linhas de comando em um arquivo `.py`. Então, *script* pode ser executado em um terminal de seu sistema operacional utilizando-se o interpretador Python. Por exemplo, assumindo que o código for salvo do arquivo `path_to_arq/arq.py`, pode-se executá-lo em um terminal do sistema com

```
$ python3 path_to_arq/arq.py
```

  IDEs para Python fornecem uma ambiente integrado, contendo um campo para escrita do código e terminal Python integrado. Consulte, por exemplo, o IDE Spyder:

  https://www.spyder-ide.org/

- **Execução em um *notebook***

  Notebooks Python são uma boa alternativa para a execução de códigos em um ambiente colaborativo/educativo. Por exemplo, Jupyter é um notebook que roda em navegadores de internet. Sua estrutura e soluções também são encontradas em notebooks online (de uso gratuito limitado) como Google Colab e Kaggle.

2.1.3 Exercícios

E.2.1.1. Verifique qual a versão do sistema operacional que está utilizado em seu computador.

E.2.1.2. Verifique os seguintes elementos de seu computador:

a) CPUs

b) Placa(s) gráfica(s)

c) Memória RAM

d) Armazenamento HDD/SSD.

E.2.1.3. Verifique como entrar na `BIOS` de seu computador. Atenção! Não faça e salve nenhuma alteração, caso não saiba o que está fazendo. Modificações na `BIOS` podem impedir que seu computador funcione normalmente, inclusive, impedir que você inicialize seu sistema operacional.

E.2.1.4. Instale Python no seu computador (caso ainda não tenha feito) e abra um terminal Python. Nele, escreva uma linha de comando que imprima no prompt de saída a frase “Olá, meu Python!”.

E.2.1.5. Instale o Spyder no seu computador (caso ainda não tenha feito) e use-o para escrever o seguinte *script*

```
import math as m
print(f'Número pi = {m.pi}')
print(f'Número de Euler e = {m.e}')
```

Também, execute o *script* diretamente em um terminal de seu sistema operacional.

E.2.1.6. Use um *notebook* Python para escrever e executar o código do exercício anterior.

Respostas

E.2.1.1. Dica: Em Linux, `$ uname --all` ou `$ cat /etc/version`.

E.2.1.2. Dica: Em Linux: `$ lshw`

E.2.1.3. Dica: cada computador tem sua forma de acessar a `BIOS`. Verifique o manual ou busque na internet pela marca e modelo de seu computador.

E.2.1.4.

```
>>> print('Olá, meu Python!')
Olá, meu Python!
>>>
```

E.2.1.6. Dica: use um notebook online Google Colab, Kaggle ou Jupyter.

2.2 Algoritmos e Programação

Programar é criar um programa (um *software*) para ser executado em computador. Para isso, escreve-se um código em uma linguagem computacional (por exemplo, em Python), o qual é interpretado/compilado para gerar o programa final. Linguagens computacionais são técnicas, utilizam uma sintaxe simples, precisa e sem ambiguidades. Ou seja, para criarmos um programa com um determinado objetivo, precisamos escrever um código computacional técnico, que siga a sintaxe da linguagem escolhida e sem ambiguidades.

Um **algoritmo** pode ser definido uma sequencia ordenada e sem ambiguidade de passos para a resolução de um problema.

Exemplo 2.2.1. O cálculo da área de um triângulo de base e altura dadas por ser feito com o seguinte algoritmo:

1. Informe o valor da base b.
2. Informe o valor da altura h.
3. $a \leftarrow \dfrac{b \cdot h}{2}$.
4. Imprima o valor de a.

Algoritmos para a programação são pensados para serem facilmente transformados em códigos computacionais. Por exemplo, o algoritmo acima pode ser escrito em Python como segue:

```python
b = float(input('Informe o valor da base.\n'))
h = float(input('Informe o valor da altura.\n'))
# cálculo da área
a = b*h/2
print(f'Área = {a}')
```

Para criar um programa para resolver um dado problema, começamos desenvolvendo um algoritmo para resolvê-lo, este algoritmo é implementado na linguagem computacional escolhida, a qual gera o programa final. Aqui, o passo mais difícil costuma ser o desenvolvimento do algoritmo. Precisamos pensar em como podemos resolver o problema de interesse em uma sequência de passos ordenada e sem ambiguidades para que possamos implementá-los em computador.

Um algoritmo deve ter as seguintes propriedades:

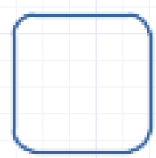
- Cada passo deve estar bem definido, i.e. não pode conter ambiguidades.
- Cada passo deve contribuir de forma efetiva na solução do problema.
- Deve ter número finito de passos que podem ser computados em um tempo finito.

Observação 2.2.1. A primeira pessoa a publicar um algoritmo para programação foi Augusta Ada King[8]. O algoritmo foi criado para computar os números de Bernoulli[9].

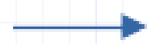
2.2.1 Fluxograma

Fluxograma é uma representação gráfica de um algoritmo. Entre outras, usam-se as seguintes formas para representar tipos de ações a serem executadas:

- **Terminal**: início ou final do algoritmo.

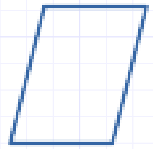


- **Linha de fluxo**: direciona para a próxima execução.

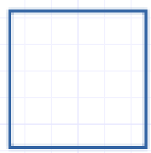


- **Entrada**: leitura de informação/dados.

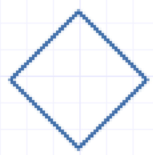
[8] Augusta Ada King, 1815 - 1852, matemática e escritora inglesa. Fonte: Wikipédia.
[9] Jacob Bernoulli, 1655-1705, matemático suíço. Fonte: Wikipédia.



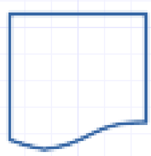
- **Processo**: ação a ser executada.



- **Decisão**: ramificação do processamento baseada em uma condição.



- **Saída**: impressão de informação/dados.



Exemplo 2.2.2. O método de Heron[10] é um algoritmo para o cálculo aproximado da raiz quadrada de um dado número x, i.e. $\sqrt{x}$. Consiste na iteração

$$s^{(0)} = \text{approx. inicial}, \tag{2.1}$$

$$s^{(i+1)} = \frac{1}{2}\left(s^{(i)} + \frac{x}{s^{(i)}}\right), \tag{2.2}$$

[10]Heron de Alexandria, 10 - 80, matemático e inventor grego. Fonte: Wikipédia.

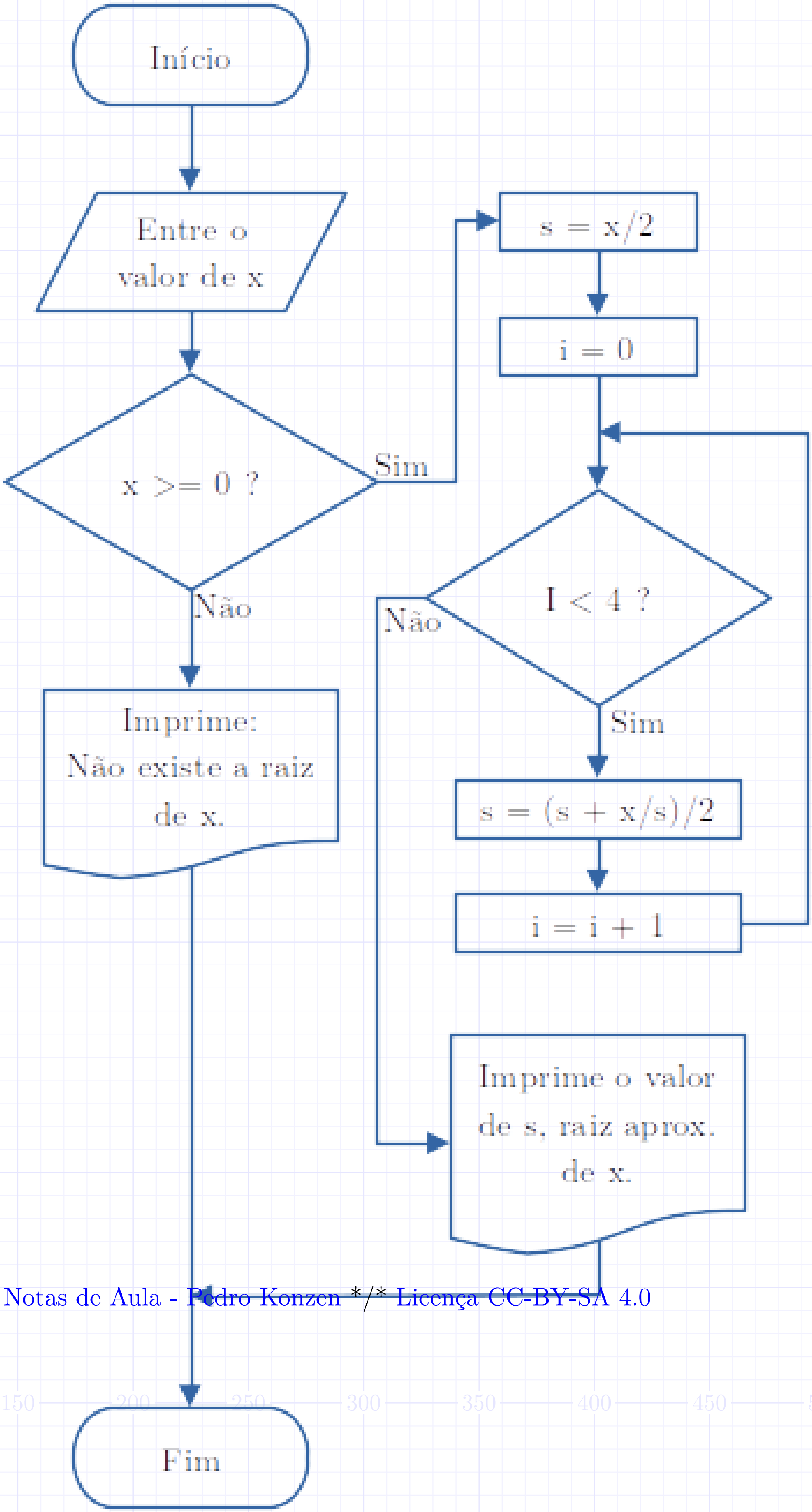
para $i = 0, 1, 2, \dots, n$, onde n é o número de iterações calculadas.

Na sequência, temos um algoritmo e seus fluxograma e código Python para computar a quarta aproximação de $\sqrt{x}$, assumindo $s^{(0)} = x/2$ como aproximação inicial.

- **Algoritmo**

  1. Entre o valor de x.
  2. Se $x \geq 0$, faça:
     (a) $s \leftarrow x/2$
     (b) Para $i = 0, 1, 2, 3$, faça:
        i. $s \leftarrow (s + x/s)/2$.
     (c) Imprime o valor de s.
  3. Senão, faça:
     (a) Imprime mensagem “Não existe!”.

- **Fluxograma**


Início
Entre o valor de x
x >= 0 ?
Sim
Não
s = x/2
i = 0
I < 4 ?
Não
Sim
Imprime: Não existe a raiz de x.
s = (s + x/s)/2
i = i + 1
Imprime o valor de s, raiz aprox. de x.
Fim

- **Código Python**

Código 2.1: metHeron.py

```
x = float(input('Entre com o valor de x: '))
if (x >= 0.):
    s = x/2
    for i in range(4):
        s = (s + x/s)/2
    print(f'Raiz aprox. de x = {s}')
else:
    print(f'Não existe!')
```

O algoritmo apresentado acima tem um *bug* (um erro)! Consulte o Exercício 2.2.9.

Algoritmos escritos em uma forma próxima de uma linguagem computacional são, também, chamados de **pseudocódigos**. Na prática, pseudocódigos e fluxogramas são usados para apresentar uma forma mais geral e menos detalhada de um algoritmo. Usualmente, sua forma detalhada é escrita diretamente em uma linguagem computacional escolhida.

2.2.2 Exercícios

E.2.2.1. Escreva um algoritmo/pseudocódigo e um fluxograma correspondente para o calcular a média aritmética entre dois números x e y dados. Como desafio, tente escrever um código Python baseado em seu algoritmo.

E.2.2.2. Escreva um algoritmo/pseudocódigo e um fluxograma correspondente para o calcular a área de um quadrado de lado l dado. Como desafio, tente escrever um código Python baseado em seu algoritmo.

E.2.2.3. Escreva um algoritmo/pseudocódigo e um fluxograma correspondente para o calcular a área de um retângulo de lados a, b dados. Como desafio, tente escrever um código Python baseado em seu algoritmo.

E.2.2.4. Escreva um algoritmo/pseudocódigo e um fluxograma correspondente para o calcular triângulo retângulo de hipotenusa h e um dos lados l dados. Como desafio, tente escrever um código Python baseado em seu algoritmo.

E.2.2.5. Escreva um algoritmo/pseudocódigo e um fluxograma correspondente para o calcular o zero de uma função afim

$$f(x) = ax + b \tag{2.3}$$

dados, os coeficientes a e b. Como desafio, tente escrever um código Python baseado em seu algoritmo.

E.2.2.6. Escreva um algoritmo/pseudocódigo e um fluxograma correspondente para o calcular as raízes reais de um polinômio quadráticos

$$p(x) = ax^2 + bx + c \tag{2.4}$$

dados, os coeficientes a, b e c. Como desafio, tente escrever um código Python baseado em seu algoritmo.

E.2.2.7. A Série Harmônica é defina por

$$\sum_{k=1}^{\infty} \frac{1}{k} := \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots \tag{2.5}$$

Escreva um algoritmo/pseudocódigo e um fluxograma corresponde para calcular o valor da série harmônica truncada em $k = n$, com n dado. Ou seja, dado n, o objetivo é calcular

$$\sum_{k=1}^{n} \frac{1}{k} := \frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n}. \tag{2.6}$$

E.2.2.8. O número de Euler[11] pode ser definido pela série

$$e := \sum_{k=0}^{\infty} \frac{1}{k!} \tag{2.7}$$

$$= \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \cdots \tag{2.8}$$

Escreva um algoritmo/pseudocódigo e um fluxograma corresponde para calcular o valor aproximado de e dado pelo truncamento da série em $k = 4$, i.e. o objetivo é de calcular

$$e \approx \sum_{k=0}^{4} \frac{1}{k!} \tag{2.9}$$

[11]Leonhard Paul Euler, 1707-1783, matemático e físico suíço. Fonte: Wikipédia.

$$= \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} \quad (2.10)$$

$$= \frac{1}{1} + \frac{1}{1} + \frac{1}{1\cdot 2} + \frac{1}{1\cdot 2\cdot 3} + \frac{1}{1\cdot 2\cdot 3\cdot 4}. \quad (2.11)$$

E.2.2.9. O algoritmo construído no Exemplo 2.2.2 tem um *bug* (um erro). Identifique o *bug* e proponha uma nova versão para corrigir o problema. Então, apresente o fluxograma da nova versão do algoritmos. Como desafio, busque implementá-lo em Python.

Respostas

E.2.2.9. Dica: o *bug* ocorre quando $x = 0$.

2.3 Dados

Informação é resultante do processamento, manipulação e organização de **dados** (altura, quantidade, volume, intensidade, densidade, etc.). Programas de computadores processam, manipulam e organizam **dados computacionais**. Os dados computacionais são representações em máquina de dados "reais". De certa forma, todo dado é uma abstração e, para ser utilizado em um programa de computador, precisa ser representado em máquina.

Cada dado manipulado em um programa é identificado por um **nome**, chamado de **identificador**. Podem ser variáveis, constantes, funções/métodos, entre outros.

- **Variável**

  Objetos de um programa que armazenam dados que podem mudar de valor durante a sua execução.

- **Constantes**

  Objetos de um programa que não mudam de valor durante a sua execução.

- **Funções e métodos**

  Subprogramas definidos e executados em um programa.

2.3.1 Identificadores

Um identificador é um nome atribuído para a identificação inequívoca de dados que são manipulados em um programa.

Exemplo 2.3.1. Vamos desenvolver um programa que computa o ponto de interseção da reta de equação

$$y = ax + b \tag{2.12}$$

com o eixo x (consulte a Figura 2.2).

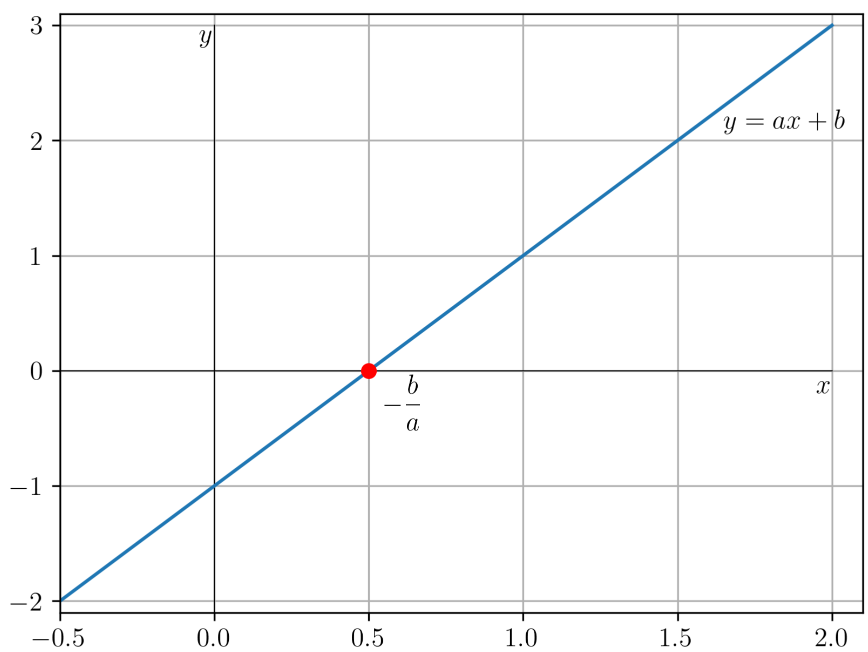


Figura 2.2: Esboço da reta de equação $y = ax + b$, com $a = 2$ e $b = -1$.

O ponto x em que a reta intercepta o eixo das abscissas é

$$x = -\frac{b}{a} \tag{2.13}$$

Assumindo que $a = 2$ e $b = -1$, segue um algoritmo para a computação.

1. Atribui o valor do **coeficiente angular**:

$$a \leftarrow 2. \tag{2.14}$$

2. Atribui o valor do **coeficiente linear**:

$$b \leftarrow -1. \tag{2.15}$$

3. Computa e armazena o valor do **ponto de interseção com o eixo** x:

$$x \leftarrow -\frac{b}{a}. \tag{2.16}$$

4. Imprime o valor de x.

No algoritmo acima, os identificados utilizados foram: a para o **coeficiente angular**, b para o **coeficiente linear** e x para o **ponto de interseção com o eixo x**.

Em Python, os identificadores são sensíveis a letras maiúsculas e minúsculas (em inglês, *case sensitive*), i.e. o identificador `nome` é diferente dos `Nome`, `NoMe` e `NOME`. Por exemplo:

```
>>> a = 7
>>> print(A)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'A' is not defined. Did you mean: 'a'?
```

Para melhorar a legibilidade de seus códigos, recomenda-se utilizar identificadores com nomes compostos que ajudem a lembrar o significado do dado a que se referem. No exemplo acima (Exemplo 2.3.1), a representa o **coeficiente angular** da reta e um identificar apropriado seria `coefAngular` ou `coef_angular`.

Identificadores não podem conter caracteres especiais (`*`, `&`, `%`, `ç`, acentuações, etc.), espaços em branco e começar com número. As seguintes convenções para identificadores com nomes compostos são recomendadas:

- **lowerCamelCase**: `nomeComposto`
- **UpperCamelCase**: `NomeComposto`

- snake: `nome_composto`

Alguns identificadores são palavras reservadas pela linguagem, pois representam dados pré-definidos nela. Veja a lista de identificadores reservados em Python Docs: Lexical Analysis: Keywords.

Exemplo 2.3.2. O algoritmo construído no Exemplo 2.3.1 pode ser implementado como segue:

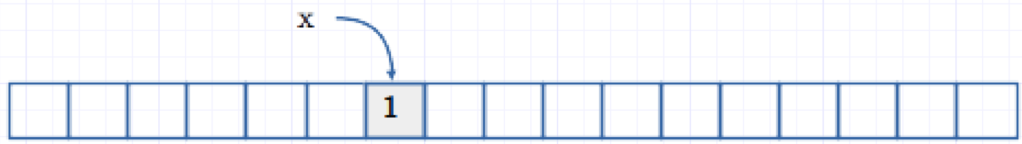
```
coefAngular = 2
coefLinear = -1
intercepEixoX = -coefLinear/coefAngular
print(intercepEixoX)
```

2.3.2 Alocação de dados

Como estudamos acima, alocamos e referenciamos dados na memória do computador usando identificadores. Em Python, ao executarmos a instrução

```
>>> x = 1
```

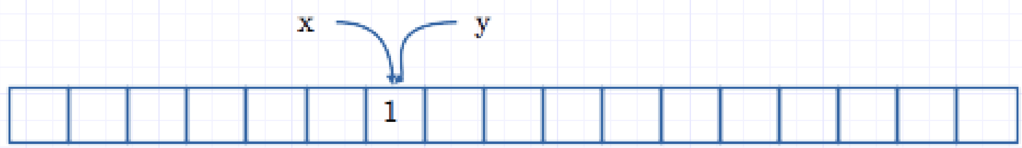
estamos criando um **objeto** na memória com valor 1 e `x` é uma referência para este dado alocado na memória. Pode-se imaginar a memória computacional como um sequência de caixinhas, de forma que `x` será a identificação da caixinha onde o valor 1 foi alocado.



Agora, quando executamos a instrução

```
>>> y = x
```

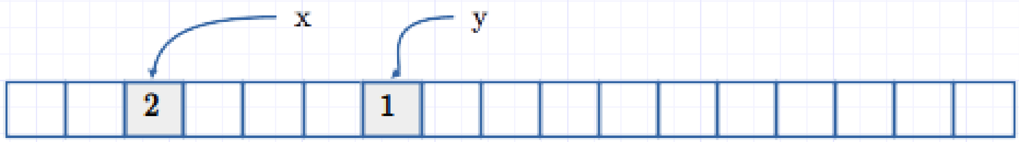
o identificador `y` passa a referenciar o mesmo local de memória de `x`.



Na sequência, se atribuirmos um novo valor para `x`

```
>>> x = 2
```

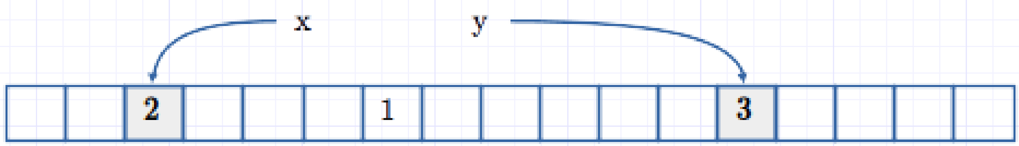
este será alocado em um novo local na memória e `x` passa a referenciar este novo local.



Ainda, se atribuirmos um novo valor para `y`

```
>>> y = 3
```

este será alocado em um novo local na memória e `y` passa a referenciar este novo local. O local de memória antigo, em que o valor 2 está alocado, passa a ficar novamente disponível para o sistema operacional.



Observação 2.3.1. O método Python `id` retorna a identidade (endereço da caixinha) de um objeto. Essa identidade deve ser única e constante para cada objeto.

```
>>> x = 1
>>> id(x)
139779845161200
>>> y = x
>>> id(y)
139779845161200
>>> x = 2
>>> id(x)
139779845161232
>>> id(y)
139779845161200
```

```
>>> y = 3
>>> id(y)
139779845161264
```

Exemplo 2.3.3. (Troca de Variáveis/Identificadores.) Em várias situações, faz-se necessário permutar dados entre dois identificadores. Sejam

```
x = 1
y = 2
```

Agora, queremos permutar os dados, ou seja, queremos que `y` tenha o valor 1 e `x` o valor 2. Podemos fazer isso utilizando uma variável auxiliar (em inglês, *buffer*).

```
z = x
x = y
y = z
```

Verifique!

2.3.3 Exercícios

E.2.3.1. Proponha identificadores adequados à linguagem Python baseados nos seguintes nomes:

a) Área

b) Perímetro do quadrado

c) Cateto+Cateto

d) Número de elementos do conjunto A

e) 77 lados

f) $f(x)$

g) x^2

h) $13x$

E.2.3.2. No Exemplo 2.2.1, apresentamos um código Python para o cálculo da área de um triângulo. Reescreva o código trocando seus identificadores por nomes mais adequados.

E.2.3.3. O seguinte código Python tem um erro:

```
x = 1
y = X + 1
```

Identifique-o e apresente uma nova versão código corrigido.

E.2.3.4. Faça uma representação gráfica da alocação de memória que ocorre para cada uma das instruções Python do Exemplo 2.3.3 na troca de variáveis. Ou seja, para a seguinte sequência de instruções:

```
x = 1
y = 2
z = x
x = y
y = z
```

E.2.3.5. No Exemplo 2.3.3 fazemos a permutação entre as variáveis `x` e `y` usando um *buffer* `z` para guardar o valor de `x`. Se, ao contrário, usarmos o *buffer* para guardar o valor de `y`, como fica o código de permutação entre as variáveis?

Respostas

E.2.3.1. a) `area`; b) `perimetroQuad`; c) `somaCatetos`; d) `numElemA`; e) `lados77`; f) `fx`; g) `x2`; h) `xv13`

E.2.3.2.

```
base = float(input('Informe o valor da base.\n'))
altura = float(input('Informe o valor da altura.\n'))
# cálculo da área
area = base * altura /2
print(f'Área = {area}')
```

E.2.3.3. Erro: variável X não foi definida.

```
x = 1
y = x + 1
```

E.2.3.5.

```
    x = 1
    y = 2
    z = y
    y = x
    x = z
    print(x, y)
    2 1
```

2.4 Dados Numéricos e Operações

Números são tipos de dados comumente manipulados em programas de computador. Números inteiros e não inteiros são tratados de forma diferente. Mas, antes de discorrermos sobre essas diferenças, vamos estudar operadores numéricos básicos.

Operações Numéricas Básicas

As seguintes operações numéricas estão disponíveis na linguagem Python:

- `+` **adição**

```
>>> 1 + 2
3
```

- `-` **subtração**

```
>>> 1 - 2
-1
```

- `*` **multiplicação**

```
>>> 2*3
6
```

- `/` **divisão**

```
>>> 5/2
2.5
```

- `//` **divisão inteira**

```
>>> 5//2
2
```

- `%` **resto da divisão**

```
>>> 5 % 2
1
```

A **ordem de precedência das operações** deve ser observada em Python. Uma expressão é executada da esquerda para a direita, mas os operadores tem a seguinte precedência[12]:

1. `**`
2. lstinline*-x* **: oposto de** x
3. `*, /, //, %`
4. `+, -`

Utilizamos parênteses para impor uma precedência diferente, i.e. expressões entre parênteses `()` são executadas antes das demais.

Exemplo 2.4.1. Estudamos a seguinte computação:

```
>>> 2+8*3/2**2-1
7.0
```

Uma pessoa desavisada poderia pensar que o resultado está errado, pois

$$2+8=10, \tag{2.17}$$
$$10\cdot 3=30, \tag{2.18}$$
$$30\div 2=15, \tag{2.19}$$
$$15^2=225, \tag{2.20}$$
$$225-1=224. \tag{2.21}$$

Ou seja, o resultado não deveria ser 224? Não, em Python, a operação de potenciação `**` tem a maior precedência, depois vem as de multiplicação `*` e

[12]Consulte a lista completa de operadores e suas precedências em Python Docs: Expressions: Operator precedence.

divisão `/` (com a mesma precedência, sendo que a mais a esquerda é executada primeiro) e, por fim, vem as de adição `+` e subtração `-` (também com a mesma precedência entre si). Ou seja, a instrução acima é computada na seguinte ordem:

$$2^2 = 4, \tag{2.22}$$
$$8 \cdot 3 = 24, \tag{2.23}$$
$$24 \div 4 = 6, \tag{2.24}$$
$$2 + 6 = 8, \tag{2.25}$$
$$8 - 1 = 7. \tag{2.26}$$

Para impormos um ordem diferente de precedência, usamos parêntese. No caso acima, escrevemos

```
>>> ((2 + 8)*3/2)**2 - 1
224.0
```

O uso de espaços entre os operandos,em geral, é arbitrário, mas conforme utilizados podem dificultar a legibilidade do código.

Exemplo 2.4.2. Consideramos a seguinte expressão

```
>>> 2 *- 3 + 2
-4
```

Essa expressão é computada na seguinte ordem:

$$-\ 3 = -3 \tag{2.27}$$
$$2 \cdot (-3) = -6 \tag{2.28}$$
$$-6 + 2 = -4 \tag{2.29}$$

Observamos que ela seria melhor escrita da seguinte forma:

```
>>> 2*-3 + 2
-4
```

2.4.1 Números Inteiros

Em Python, números inteiros são alocados por registros com um número arbitrário de *bits*. Com isso, os maior e menor números inteiros que podem ser alocados dependem da capacidade de memória da máquina. Quanto maior ou menor o número inteiro, mais *bits* são necessários para alocá-lo.

Exemplo 2.4.3. O método Python `sys.getsizeof()` retorna o tamanho de um objeto medido em *bytes* (1 *byte* = 8 *bits*).

```
>>> import sys
>>> sys.getsizeof(0)
24
>>> sys.getsizeof(1)
28
>>> sys.getsizeof(100)
28
>>> sys.getsizeof(10**9)
28
>>> sys.getsizeof(10**10)
32
>>> sys.getsizeof(10**100) #googol
72
```

O número googol 10^{100} é um número grande[13], mas 72 *bytes* não necessariamente. Um computador com 4 Gbytes[14] livres de memória, poderia armazenar um número inteiro que requer um registro de até $4,3 \times 10^9$ *bytes*.

Observação 2.4.1. O método Python type() retorna o tipo de objeto alocado. Números inteiros são objetos da classe `int`.

```
>>> type(10)
<class 'int'>
```

2.4.2 Números Decimais

No Python, números decimais são alocados pelo padrão IEEE 774 de aritmética em ponto flutuante. Em geral, são usados 64 *bits* = 8 *bytes* para alocar um número decimal. Um ponto flutuante tem a forma

$$x = \pm m \cdot 2^{c-1023}, \tag{2.30}$$

onde m é chamada de mantissa (e é um número no intervalo $[1,2)$) e $c \in [0, 2047]$ é um número inteiro chamado de característica do ponto flutuante. A mantissa usa 52 *bits*, a característica 11 bits e 1 *bit* é usado para o sinal do número.

[13] Por exemplo, o número total de partículas elementares em todo o universo observável é estimado em 10^{80}. Fonte: Wikipédia: Eddington number.

[14] 1 *Gbytes* = 1024 *Mbytes*, 1 *Mbytes* = 1024 *Kbytes*, 1 *Kbytes* = 1024 *bytes*.

```
>>> import sys
>>> sys.float_info
sys.float_info(max=1.7976931348623157e+308,
               max_exp=1024,
               max_10_exp=308,
               min=2.2250738585072014e-308,
               min_exp=-1021,
               min_10_exp=-307,
               dig=15,
               mant_dig=53,
               epsilon=2.220446049250313e-16,
               radix=2,
               rounds=1)
```

Vamos denotar `fl(x)` o número em ponto flutuante mais próximo do número decimal `x` dado. Quando digitamos

```
>>> x = 0.1
```

O valor alocado na memória da máquina não é `0.1`, mas, sim, o `fl(x)`. Normalmente, o **épsilon de máquina** $\varepsilon = 2,22 \times 10^{-16}$ é uma boa aproximação para o erro (de arredondamento) entre `x` e `fl(x)`.

Notação Científica

A **notação científica** é a representação de um dado número na forma

$$d_n \dots d_2 d_1 d_0, d_{-1} d_{-2} d_{-3} \dots \times 10^E, \tag{2.31}$$

onde d_i, $i = n, \dots, 1, 0, -1, \dots$, são algarismos da base 10. A parte à esquerda do sinal $\times$ é chamada de mantissa do número e E é chamado de expoente (ou ordem de grandeza).

Exemplo 2.4.4. O número $31,415$ pode ser representado em notação científica das seguintes formas

$$31,415 \times 10^0 = 3,1415 \times 10^1 \tag{2.32}$$
$$= 314,15 \times 10^{-1} \tag{2.33}$$
$$= 0,031415 \times 10^3, \tag{2.34}$$

entre outras tantas possibilidades.

Em Python, usa-se a letra `e` para separar a mantissa do expoente na notação científica. Por exemplo

```
>>> # 31.415 X 10^0
>>> 31.415e0
31.515
>>> # 3.1415 X 10^1
>>> 3.1415e1
31.515
>>> # 314.15 X 10^-1
>>> 314.15e-1
31.515
>>> # 0.031415 X 10^3
>>> 0.031415e3
31.415
```

No exemplo anterior (Exemplo 2.4.4), podemos observar que a representação em notação científica de um dado número não é única. Para contornar isto, introduzimos a **notação científica normalizada**, a qual tem a forma

$$d_0, d_{-1}d_{-2}d_{-3}\ldots \times 10^E, \tag{2.35}$$

com $d_0 \neq 0$[15].

Exemplo 2.4.5. O número $31,415$ representado em notação científica normalizada é $3,1415 \times 10^1$.

Em Python, podemos usar de especificação de formatação[16] para imprimir um número em notação científica normalizada. Por exemplo, temos

```
>>> x = 31.415
>>> print(f"{x:e}")
3.141500e+01
```

2.4.3 Números Complexos

Python tem números complexos como um tipo básico da linguagem. O número imaginário $i := \sqrt{-1}$ é representado por `1j`. Temos

```
>>> 1j**2
(-1+0j)
```

[15]No caso do número zero, temos $d_0 = 0$.
[16]Consulte Subseção 2.6.1 para mais informações.

Ou seja, $i^2 = -1 + 0i$. Aritmética de números completos está diretamente disponível na linguagem.

Exemplo 2.4.6. Estudamos os seguintes casos:

a) $-3i + 2i = -i$

```
>>> -3j + 2j
-1j
```

b) $(2 - 3i) + (4 + i) = 6 - 2i$

```
>>> 2-3j + 4+1j
(6-2j)
```

c) $(2 - 3i) \cdot (4 + i) = 11 - 10i$

```
>>> (2-3j)*(4+1j)
(11-10j)
```

2.4.4 Exercícios

E.2.4.1. Desenvolva um código Python para computar a interseção com o eixo das abscissas da reta de equação

$$y = 2ax - b. \tag{2.36}$$

Em seu código, aloque $a = 2$ e $b = 8$ e então compute o ponto de interseção x.

E.2.4.2. Assuma que o seguinte código Python

```
a = 2
b = 8
x = b/2*a
print("x = ", x)
```

tenha sido desenvolvido para computar o ponto de interseção com o eixo das abscissas da reta de equação

$$y = 2ax - b \tag{2.37}$$

com $a = 2$ e $b = 8$. O código acima contém um erro, qual é? Identifique-o, corrija-o e justifique sua resposta.

E.2.4.3. Desenvolva um código Python para computar a média aritmética entre dois números x e y dados.

E.2.4.4. Uma disciplina tem o seguinte critério de avaliação:

1. Trabalho: nota com peso 3.
2. Prova: nota com peso 7.

Desenvolva um código Python que compute a nota final, dadas as notas do trabalho e da prova (em escala de $0-10$) de um estudante.

E.2.4.5. Desenvolva um código Python para computar as raízes reais de uma equação quadrática

$$ax^2 + bx + c = 0. \tag{2.38}$$

Assuma dados os parâmetros $a = 2$, $b = -2$ e $c = -12$.

E.2.4.6. Encontre a quantidade de memória disponível em seu computador. Quantos *bytes* seu programa poderia alocar de dados caso conseguisse usar toda a memória disponível no momento?

E.2.4.7. Escreva os seguintes números em notação científica normalizada e entre com eles em um terminal Python:

a) 700

b) $0,07$

c) 2800000

d) $0,000019$

E.2.4.8. Escreva os seguintes números em notação decimal:

1. $2,8 \times 10^{-3}$
2. $8,712 \times 10^{4}$
3. $3,\overline{3} \times 10^{-1}$

E.2.4.9. Faça os seguintes cálculos e então verifique os resultados computando-os em Python:

1. $5 \times 10^3 + 3 \times 10^2$
2. $8,1 \times 10^{-2} - 1 \times 10^{-3}$
3. $(7 \times 10^4) \cdot (2 \times 10^{-2})$
4. $(7 \times 10^{-4}) \div (2 \times 10^2)$

E.2.4.10. Faça os seguintes cálculos e verifique seus resultados computando-os em Python:

1. $(2 - 3i) + (2 - i)$
2. $(1 + 2i) - (1 - 3i)$
3. $(2 - 3i) \cdot (-4 + 2i)$
4. $(1 - i)^3$

E.2.4.11. Desenvolva um código Python que computa a área de um quadrado de lado l dado. Teste-o com $l = 0,575$ e assegure que seu código forneça o resultado usando notação decimal.

E.2.4.12. Desenvolva um código Python que computa o comprimento da diagonal de um quadrado de lado l dado. Teste-o com $l = 2$ e assegure que seu código forneça o resultado em notação científica normalizada.

E.2.4.13. Assumindo que $a_1 \neq a_2$, desenvolva um código Python que compute o ponto (x_i, y_i) que corresponde a interseção das retas de equações

$$y = a_1 x + b_1 \tag{2.39}$$
$$y = a_2 x + b_2, \tag{2.40}$$

para a_1, a_2, b_1 e b_2 parâmetros dados. Teste-o para o caso em que $a_1 = 1$, $a_2 = -1$, $b_1 = 1$ e $b_2 = -1$. Garanta que seu código forneça a solução usando notação científica normalizada.

Respostas

E.2.4.1.

```
a = 2
b = 8
x = b/(2*a)
print("x = ", x)
```

E.2.4.2. Erro na linha 3. As operações não estão ocorrendo na precedência correta para fazer a computação desejada. Correção: `x = b/(2*a)`.

E.2.4.3.

```
x = 3
y = 9
media = (x + y)/2
print('média = ', media)
```

E.2.4.4.

```
notaTrabalho = 8.5
notaProva = 7
notaFinal = (notaTrabalho*3 + notaProva*7)/10
print('Nota final = ', notaFinal)
```

E.2.4.5.

```
a = 2
b = -2
c = -12
delta = b**2 - 4*a*c
x1 = (-b - delta**(1/2))/(2*a)
print('x1 = ', x1)
x2 = (-b + delta**(1/2))/(2*a)
print('x2 = ', x2)
```

E.2.4.6. Dica: seu sistema operacional deve ter um gerenciador de tarefas, um *software* que nos permite controlar a execução dos programas em execução. Este gerenciador muitas vezes também informa o estado de utilização da memória computacional. No `Linux`, pode-se usar o programa `top` ou o `htop`.

E.2.4.7. a) 7×10^2, `>>> 7e2`; b) 7×10^{-2}, `7e-2`; c) $2,8 \times 10^6$, `2.8e6`; d) 1.9×10^{-5}, `1.9e-5`

E.2.4.8. a) 0.0028; b) 87120; c) $0,\overline{3}$

E.2.4.9. a) $5,3 \times 10^3$;

```
>>> x = 5e3 + 3e2
>>> print(f'{x:e}')
5.300000e+03
```

b) 8×10^{-2}

```
>>> x = 8.1e-2 - 1e-3
>>> print(f'{x:e}')
```

c) $1,4 \times 10^3$

```
>>> x = 7e4 * 2e-2
>>> print(f'{x:e}')
1.400000e+03
```

d) $3,5 \times 10^{-6}$

```
>>> x = 7e-4 / 2e2
>>> print(f'{x:e}')
3.500000e-06
```

E.2.4.10. a) $3 + 7i$

```
>>> (1+8j) + (2-1j)
(3+7j)
```

b) $5i$

```
>>> (1+2j) - (1-3j)
5j
```

c) $-2 + 16i$

```
>>> (2-3j) * (-4+2j)
(-2+16j)
```

d) $-2-2i$

```
>>> (1-1j)**3
(-2-2j)
```

E.2.4.11.

```
lado = 0.575
area = lado**2
print(f'área = {area:f}')
```

E.2.4.12.

```
lado = 2
diag = lado*2**(1/2)
print(f'diagonal = {diag:e}')
```

E.2.4.13.

```
# parametros
a1 = 1
a2 = -1
b1 = 1
b2 = -1
# ponto x de interseção
x_intercep = (b2-b1)/(a1-a2)
# ponto y de interceção
y_intercep = a1*x_intercep + b1
# imprime o resultado
print(f'x_i = {x_intercep:e}')
print(f'y_i = {y_intercep:e}')
```

2.5 Dados Booleanos

Em Python, os **valores lógicos** são o `True` (verdadeiro) e o `False` (falso). Pertencem a uma subclasse dos números inteiros, com `1` correspondendo a `True` e `0` a `False`. Em referência ao matemático George Boole[17], estes dados são chamados de **booleanos**.

Normalmente, eles aparecem como resultado de expressões lógicas. Por exemplo:

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

```
>>> 2/3 < 3/4
True
>>> 7/5 > 13/9
False
```

2.5.1 Operadores de Comparação

Python possui **operadores de comparação** que retornam valores lógicos, são eles:

- `<` **menor que**

```
>>> 2 < 3
True
```

- `<=` **menor ou igual que**

```
>>> 4 <= 2**2
True
```

- `>` **maior que**

```
>>> 5 > 7
False
```

- `>=` **maior ou igual que**

```
>>> 2*5 >= 10
True
```

- `==` **igual a**

```
>>> 9**2 == 81
True
```

- `!=` **diferente de**

```
>>> 81 != 9**2
False
```

Observação 2.5.1. (Precedência de operações.) Os operadores de comparação `<`, `<=`, `>`, `>=`, `==`, `!=` tem a mesma ordem de precedência e estão abaixo da precedência dos operadores numéricos básicos.

Exemplo 2.5.1. A equação da circunferência de centro no ponto (a, b) e raio r é

$$(x - a)^2 + (y - b)^2 = r^2. \tag{2.41}$$

Um ponto (x, y) está no disco determinado pela circunferência, quando

$$(x - a)^2 + (y - b)^2 \leq r^2 \tag{2.42}$$

e está fora do disco, noutro caso.

O seguinte código verifica se o ponto dado $(x, y) = (1, 1)$ está no disco determinado pela circunferência de centro $(a, b) = (0, 0)$ e raio $r = 1$.

```
# ponto
x = 1
y = 1

# centro circunferência
a = 0
b = 0
# raio circunferência
raio = 1

# verifica se está no disco
v = (x-a)**2 + (y-b)**2 <= raio**2

# imprime resposta
print('O ponto está no disco?', v)
```

Comparação entre pontos flutuantes

Números decimais são arredondados para o número `float` (ponto flutuante) mais próximo na máquina[18]. Com isso, a comparação direta entre pontos flutuantes não é recomendada, em geral. Por exemplo,

```
>>> 0.1 + 0.2 == 0.3
False
```

Inesperadamente, este resultado é esperado na aritmética de ponto flutuante! :)

[18] Consulte a Subseção 2.4.2.

O que ocorre acima, é que ao menos um dos números (na verdade todos) não tem representação exata como ponto flutuante. Isso faz com que a soma `0.1 + 0.2` não seja exatamente computada igual a `0.3`.

O erro de arredondamento é de aproximadamente[19] 10^{-16} para cada entrada. Conforme operamos sobre pontos flutuantes este erro pode crescer. Desta forma, o mais apropriado para comparar se dois pontos flutuantes são iguais (dentro do erro de arrendamento de máquina) é verificando se a distância entre eles é menor que uma precisão desejada, por exemplo, 10^{-15}. No caso acima, podemos usar[20]:

```
>>> abs(x - 0.3) <= 1e-15
True
```

2.5.2 Operadores Lógicos

Python tem os operadores lógicos (ou **operadores booleanos**):

- `and` **e lógico**

```
>>> 3 > 4 and 3 <= 4
False
```

Tabela 2.1: Tabela verdade do `and`.

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

- `or` **ou lógico**

```
>>> 3 > 4 or 3 <= 4
True
```

[19]Épsilon de máquina $\varepsilon \approx 2,22 \times 10^{-16}$.

[20]`abs()` é um método Python para computar o valor absoluto de um número. Consulte Python Docs:Built-in Functions.

Tabela 2.2: Tabela verdade do `or`.

A	B	A or B
True	True	True
True	False	True
False	True	True
False	False	False

- `not` **negação lógica**

```
>>> not(3 < 2)
True
```

Tabela 2.3: Tabela verdade do `not`.

A	not A
True	False
False	True

Observação 2.5.2. (Precedência de operações.) Os operadores booleanos tem a seguinte ordem de precedência:

1. `not`

2. `and`

3. `or`

São executados em ordem de precedência menor que os operadores de comparação.

Exemplo 2.5.2. Sejam os discos determinados pelas circunferências

$$c_1 : (x - a_1)^2 + (y + b_1)^2 = r_1^2, \tag{2.43}$$
$$c_2 : (x - a_2)^2 + (y + b_2)^2 = r_2^2, \tag{2.44}$$

onde (a_1, b_1) e (a_2, b_2) são seus centros e r_1 e r_2 seus raios, respectivamente.

Assumindo, que a circunferência c_1 tem

$$c_1 : (a_1, b_1) = (0, 0), r_1 = 1 \tag{2.45}$$

e a circunferência c_2 tem

$$c_2 : (a_2, b_2) = (1, 1), r_2 = 1, \tag{2.46}$$

o seguinte código verifica se o ponto $(x, y) = \left(\frac{1}{2}, \frac{1}{2}\right)$ pertence a interseção dos discos determinados por c_1 e c_2.

```
# circunferência c1
a1 = 0
b1 = 0
r1 = 1

# circunferência c2
a2 = 1
b2 = 1
r2 = 1

# ponto obj
x = 0.5
y = 0.5

# está em c1?
em_c1 = (x-a1)**2 + (y-b1)**2 <= r1**2

# está em c2?
em_c2 = (x-a2)**2 + (y-b2)**2 <= r2**2

# está em c1 e c2?
resp = em_c1 and em_c2
print('O ponto está na interseção de c1 e c2?', resp)
```

Observação 2.5.3. (Ou exclusivo.) Presente em algumas linguagens, Python não tem um operador `xor` (ou exclusivo). A tabela verdade do ou exclusivo é

A	B	A xor B
True	True	False
True	False	True
False	True	True
False	False	False

A operação `xor` pode ser obtida através de expressões lógicas usando-se apenas os operadores `and`, `or` e `not`. Consulte o Exercício 2.5.6.

2.5.3 Exercícios

E.2.5.1. Compute as seguintes expressões:

a) $1 - 6 > -6$

b) $\frac{3}{2} < \frac{4}{3}$

c) $31,415 \times 10^{-1} == 3.1415$

d) $2,7128 \geq 2 + \dfrac{2}{3}$

e) $\dfrac{3}{2} + \dfrac{7}{8} \leq \dfrac{24 + 14}{16}$

E.2.5.2. Desenvolva um código que verifica se um número inteiro x dado é par. Teste-o para diferentes valores de x.

E.2.5.3. Considere um quadrado de lado l dado e uma circunferência de raio r dado. Desenvolva um código que verifique se a área do quadrado é menor que a da circunferência. Teste o seu código para diferentes valores de l e r.

E.2.5.4. Considere o plano cartesiano $x - y$. Desenvolva um código que verifique se um ponto (x, y) dado está entre a curvas $y = (x - 1)^3$ e o eixo das abscissas[21]. Verifique seu código para diferentes pontos (x, y).

E.2.5.5. Sejam A e B valores booleanos. Verifique se as seguintes expressões são verdadeiras (V) ou falsas (F):

a) `A or A == A`

b) `A and not(A) == True`

c) `A or (A and B) == A`

d) `not(A and B) == not(A) or not(B)`

[21] Eixo x.

e) `not(A or B) == not(A) or not(B)`

E.2.5.6. Sejam `A` e `B` valores booleanos dados. Escreva uma expressão lógica que emule a operação `xor` (ou exclusivo) usando apenas os operadores `and`, `or` e `not`. Dica: consulte a Observação 2.5.3.

Respostas

E.2.5.1. a) `1 - 6 > -6` b) `3/2 < 4/3` c) `31.415e-1 == 3.1415` d) `2.7128 >= 2` 2/3+ e) `3/2` 7/8 <= (24 + 14)/16+

E.2.5.2.

```
x = 3
print('É par?')
print(x % 2 == 0)
```

E.2.5.3.

```
# quadrado
ladoQuad = 1
areaQuad = ladoQuad**2

# aprox pi
pi = 3.14159

# circunferência
raioCirc = 1
areaCirc = pi * raioCirc**2

# verifica
resp = areaQuad < areaCirc
print('Área do quadrado é menor que da circunferência?')
print(resp)
```

E.2.5.4.

```
# ponto
x = 2
y = 0.5
```

```python

# y >= 0 e y <= f(x) ?
resp1 = y >= 0 and y <= (x-1)**3
# y >= f(x) e y <= 0 ?
resp2 = y >= (x-1)**3 and y <= 0

# conclusão
print("O ponto está entre as curvas?")
print(resp1 or resp2)
```

E.2.5.5. a) V; b) F; c) V; d) V; e) F

E.2.5.6. `(A or B) and not(A and B)`

2.6 Sequência de Caracteres

Dados em formato texto também são comumente manipulados em programação. Um texto é interpretado como uma cadeia/sequência de caracteres, chamada de ***string***. Para entrarmos com uma letra, palavra ou texto (um *string*), precisamos usar aspas (simples `' '` ou duplas `" "`). Por exemplo,

```python
>>> s = 'Olá, mundo!'
>>> print(s)
Olá, mundo!
>>> type(s)
<class 'str'>
```

Uma *string* é um conjunto **indexado** e **imutável** de caracteres. O primeiro caractere está na posição 0, o segundo na posição 1 e assim por diante. Por exemplo,

$$\underset{0}{\texttt{O}}\,\underset{1}{\texttt{l}}\,\underset{2}{\texttt{á}}\,\underset{3}{\texttt{,}}\,\underset{4}{\texttt{\_}}\,\underset{5}{\texttt{m}}\,\underset{6}{\texttt{u}}\,\underset{7}{\texttt{n}}\,\underset{8}{\texttt{d}}\,\underset{9}{\texttt{o}}\,\underset{10}{\texttt{!}} \tag{2.47}$$

Observamos que o espaço também é um caractere. O tamanho da *string* (número total de caracteres) pode ser obtido com o método Python `len()`, por exemplo

```python
>>> len(s)
11
```

A referência a um caractere de uma dada *string* é feito usando-se seu identificador seguido do índice de sua posição entre colchetes. Por exemplo,

```
>>> s[6]
'u'
```

Podemos, ainda, acessar fatias[22] da sequência usando o operador :[23], por exemplo,

```
>>> s[:3]
'Olá'
```

ou seja, os caracteres da posição 0 à posição 2 (um antes do índice 3). Também podemos tomar uma fatia entre posições, por exemplo,

```
>>> s[5:10]
'mundo'
```

o que nos fornece a fatia de caracteres que inicia na posição 5 e termina na posição 9. Ou ainda,

```
>>> s[6:]
'undo!'
```

Também, pode-se controlar o passo do fatiamento, por exemplo

```
>>> 'laura'[::2]
'lua'
```

Em Python, exitem diversas formas de escrever *strings*:

- **aspas simples**

```
>>> 'permitem aspas "duplas" embutidas'
'permitem aspas "duplas" embutidas'
```

- **aspas duplas**

```
>>> "permitem aspas 'simples' embutidas"
"permitem aspas 'simples' embutidas"
```

- **aspas triplas**[24]

```
>>> '''
... permitem
...   "diversas"
```

[22]Em inglês, *slice*.
[23]`x[start:stop:step]`, padrão `start=0`, `stop=len(x)`, `step=1`.
[24]`'n'` é o caractere que indica uma nova linha (em inglês, *newline*).

```
... linhas
... '''
'\npermitem\n  "diversas"\nlinhas\n'
>>> """
... permitem
...    'diversas'
... linhas
... """
"\npermitem\n  'diversas'\nlinhas\n"
```

Strings em Python usam o padrão Unicode, que nos permite manipular textos de forma muito próxima da linguagem natural. Alguns caracteres especiais úteis são:

- `'\n'` **nova linha**

```
>>> print('Uma nova\nlinha')
Uma nova
linha
```

- `'\t'` **tabulação**

```
>>> print('Uma nova\n\t linha com tabulação')
Uma nova
         linha com tabulação
```

Observação 2.6.1. (*Raw string*.) Caso seja necessário imprimir os caracteres unicode especiais `'\\n'`, `'\\t'`, entre outros, pode-se usar *raw strings*. Por exemplo,

```
>>> print(r'Aqui, o \n não quebra a linha!')
Aqui, o \n não quebra a linha!
```

2.6.1 Formatação de *strings*

Em Python, *strings* formatadas são identificadas com a letra `f` no início. Elas aceitam o uso de identificadores com valores predefinidos. Os identificadores são embutidos com o uso de chaves `{}` (*placeholder*). Por exemplo,

```
>>> nome = 'Fulane'
>>> f'Olá, {nome}!'
'Olá, Fulane!'
```

Há várias especificações de formatação disponíveis[25]:

- `'d'` **número inteiro**

```
>>> print(f'10/3 é igual a {10//3:d} e \
... resta {10%3:d}.')
10/3 é igual a 3 e resta 1.
```

- `'f'` **número decimal**

```
>>> print(f'13/7 é aproximadamente {13/7:.3f}')
13/7 é aproximadamente 1.857
```

- `'e'` **notação científica normalizada**

```
>>> print(f'103/7 é aproximadamente {103/7:.3e}')
103/7 é aproximadamente 1.471e+01
```

2.6.2 Operações com *strings*

Há uma grande variedade disponível de métodos para a manipulação de *strings* em Python (consulte Python Docs: String Methods). Alguns operadores básicos são:

- `+` **concatenação**

```
>>> s = 'Olá, mundo!'
>>> s[:5] + 'Fulane!'
'Olá, Fulane!'
```

- `*` **repetição**

```
>>> 'ha'*3
'hahaha'
```

- `in` **pertence**

```
>>> 'mar' in 'amarelo'
True
```

[25]Consulte Python Docs:String:Format Specification Mini-Language para uma lista completa.

2.6.3 Entrada de dados

O método Python `input()` pode ser usado para a entrada de *string* via teclado. Por exemplo,

```
1 >>> s = input('Digite seu nome.\n')
2 Digite seu nome.
3 Fulane
4 >>> s
5 'Fulane'
```

A instrução da linha 1 pede para que a variável `s` receba a *string* a ser digitada pela(o) usuária(o). A *string* entre parênteses é informativa, o comando `input`, imprime esta mensagem e fica aguardado que uma nova *string* seja digitada. Quando o usuário pressiona `<ENTER>`, a *string* digitada é alocada na variável `s`.

Conversão de classes de dados

A conversão entre classes de dados é possível e é feita por métodos próprios de cada classe. Por exemplo,

```
>>> # int -> str
>>> str(101)
'101'
>>> # str -> int
>>> int('23')
23
>>> # int -> float
>>> float(1)
1.0
>>> # float -> int
>>> int(-2.9)
-2
```

Atenção! Na conversão de `float` para `int`, fica-se apenas com a parte inteiro do número.

Observação 2.6.2. O método Python `input()` permite a entrada de *strings*, que podem ser convertidas para outras classes de dados. Com isso, pode-se obter a entrada via teclado destes dados.

Exemplo 2.6.1. O seguinte código, computa a área de um triângulo com base e altura fornecidas por usuária(o).

```
# entrada de dados
base = float(input('Entre com o valor da base:\n\t'))
altura = float(input('Entre com o valor da altura:\n\t'))

# cálculo da área
area = base*altura/2

# imprime a área
print(f'Área do triangulo de ')
print(f'\t base = {base:e}')
print(f'\t altura = {altura:e}')
print(f'é igual a {area:e}')
```

2.6.4 Exercícios

E.2.6.1. Aloque a palavra *traitor* em uma variável x. Use de indexação por referência para:

a) Extrair a quarta letra da palavra.

b) Extrair a *substring*[26] formada pelas quatro primeiras letras da palavra.

c) Extrair a *string* formadas pelas segunda, quarta e sexta letras (nesta ordem) da palavra.

d) Extrair a *string* formadas pelas penúltima e quarta letras (nesta ordem) da palavra.

E.2.6.2. Considere o seguinte código

```
s = 'traitor'
print(s[:3] + s[4:])
```

Sem implementá-lo, o que é impresso?

E.2.6.3. Desenvolva um contador de letras de palavras. Ou seja, crie um código que forneça o número de letras de uma palavra fornecida por um(a) usuário(a).

[26] Uma subsequência contínua de caracteres de uma *string*.

E.2.6.4. Desenvolva um código que compute a área de um quadrado de lado fornecido pela(o) usuária(o). Assuma que o lado é dado em centímetros e a área deve ser impressa em metros, usando notação decimal com 2 dígitos depois da vírgula.

E.2.6.5. Desenvolva um código que verifica se um número é divisível por outro. Ou seja, a(o) usuária entra com dois números inteiros e o código imprime verdadeiro (`True`) ou (`False`) conforme a divisibilidade de x por y.

Respostas

E.2.6.1. a) `x[3]`; b) `x[:4]`; c) `x[1::2]`; d) `[-2:2:-2]`

E.2.6.2. `trator`

E.2.6.3.

```
s = input('Digite uma palavra:\n\t')
print(f'A palavra {s} tem {len(s)} letras.')
```

E.2.6.4.

```
lado = float(input('Digite o lado (em cm) do quadrado:\n\t'))
area = lado**2/100**2
print(f'O quadrado de lado {lado:e} cm tem área {area:.2f} m.')
```

E.2.6.5.

```
x = int(input('Digite um número inteiro:\n'))
y = int(input('Digite outro número inteiro:\n'))
print(f'{x} é divisível por {y}?')
print(f'{x%y==0}')
```

2.7 Coleção de Dados

Objetos da classe de dados `int` e `float` permitem a alocação de um valor numérico por variável. Já, `string` é um coleção (sequência) de caracteres. Nesta seção, vamos estudar sobre classes de dados básicos que permitem a alocação de uma coleção de dados em uma única variável.

2.7.1 Conjuntos: `set`

Em Python, `set` é uma classe de dados para a alocação de um conjunto de objetos. Assim como na matemática, um `set` é uma coleção de itens **não indexada**, **imutável** e **não admite itens duplicados**.

A alocação de um `set` pode ser feita como no seguinte exemplo:

```
>>> a = {1, -3.7, 'amarelo'}
>>> type(a)
<class 'set'>
>>> a
{'amarelo', 1, -3.7}
```

Observamos que a ordem dos elementos é arbitrária, uma vez que `set` é uma coleeção de itens não indexada.

O método `set()` também pode ser usado para criar um conjunto. Por exemplo, o conjunto vazio pode ser criado como segue:

```
>>> b = set()
>>> type(b)
<class 'set'>
>>> b
set()
```

O método `len()` pode ser usado para obtermos o tamanho (número de elementos) de um `set`:

```
>>> len(a)
3
>>> len(b)
0
```

Operadores de comparação

Os seguintes operadores de comparação estão disponíveis para `set`s:

- `x in a` **pertencimento**

  Verifica se $x \in a$.

```
>>> a = {1, -3.7, 'amarelo'}
>>> 1 in a
True
>>> 'mar' in a
False
```

- `a == b` **igualdade**

  Verifica se $a = b$.

```
>>> a == a
True
```

- `a != b` **diferente**

  Verifica se $a \neq b$.

```
>>> b = {'amarelo', -3.7}
>>> a != b
True
```

- `a <= b` **contido em ou igual a (subconjunto)**

  Verifica se $a \subseteq b$.

```
>>> b <= a
True
```

- `<` **contido em e não igual a (subconjunto próprio)**

  Verifica se $a \subsetneq b$.

```
>>> a < a
False
>>> b < a
True
```

- `>=` **contém ou é igual a (subconjunto)**

  Verifica se $a \supset b$.

```
>>> a >= b
True
```

- `>` **contém e não é igual a (subconjunto próprio)**

  Verifica se $a \supsetneq b$.

```
>>> a > b
True
>>> b > b
False
```

Operações com conjuntos

Em Python, as seguintes operações com conjuntos estão disponíveis:

- `a | b` **união**

  Retorna o `set` equivalente a

$$a \cup b := \{x : \ x \in a \vee x \in b\} \tag{2.48}$$

```
>>> a = {1, -3.7, 'amarelo'}
>>> b = {'mar', -5}
>>> a | b
{1, 'amarelo', 'mar', -5, -3.7}
```

- `a & b` **interseção**

  Retorna o `set` equivalente a

$$a \cap b := \{x : \ x \in a \wedge x \in b\} \tag{2.49}$$

```
>>> a = {1, -3.7, 'amarelo'}
>>> b = {'mar', 1, -3.7, -5}
>>> a & b
{1, -3.7}
```

- `-` **diferença**

  Retorna o `set` equivalente a

$$a \setminus b := \{x : \ x \in a \wedge x \notin b\} \tag{2.50}$$

```
>>> a - b
{'amarelo'}
```

- `^` **diferença simétrica**

  Retorna o `set` equivalente a

$$a \Delta b := (a \setminus b) \cup (b \setminus a) \tag{2.51}$$

```
>>> a ^ b
{'amarelo', 'mar', -5}
```

2.7.2 N-uplas: `tuple`

Em Python, `tuple` é uma **sequência de objetos**, **indexada** e **imutável**. São similares as n-uplas[27] em matemática. A alocação é feita com uso de parênteses e os elementos separados por vírgula, por exemplo,

```
>>> a = (1, -3.7, 'amarelo', -5)
>>> type(a)
<class 'tuple'>
```

Indexação e fatiamento

O tamanho de um `tuple` é sua quantidade de objetos e pode ser obtido com o método `len()`, por exemplo,

```
>>> a = (1, -3.7, 'amarelo', -5, {-3,1})
>>> len(a)
5
```

Os itens são indexados como segue

$$(\underset{\substack{0\\-5}}{1}, \underset{\substack{1\\-4}}{-3.7}, \underset{\substack{2\\-3}}{\text{'amarelo'}}, \underset{\substack{3\\-2}}{-5}, \underset{\substack{4\\-1}}{\{-3, 1\}}) \tag{2.52}$$

A referência a um objeto do `tuple` pode ser feita com

```
>>> a[2]
'amarelo'
>>> a[-1]
{1, -3}
```

Analogamente a `strings`, pode-se fazer o fatiamento de `tuples` usando-se o operador `:`. Por exemplo,

```
>>> a[:2]
(1, -3.7)
>>> a[1:5:2]
(-3.7, -5)
>>> a[::-1]
({1, -3}, -5, 'amarelo', -3.7, 1)
```

[27] Pares (duplas), triplas, quadruplas ordenadas, etc.

Operações com `tuple`s

Os mesmos operadores de comparação para `set`s estão disponíveis para `tuple`s (consulte a Subseção 2.7.1). Por exemplo,

```
>>> -5 in a
True
>>> a[::-1] == a[-1:-6:-1]
True
>>> a != a[::-1]
True
>>> a[:2] < a
True
```

Observação 2.7.1. (Igualdade entre `tuple`s.) Dois `tuple`s são iguais quando contém os mesmos elementos e na mesma ordem.

Há, também, operadores para a concatenação e repetição:

- `+` **concatenação**

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

- `*` **: repetição**

```
>>> a*3
(1, 2, 1, 2, 1, 2)
```

Observação 2.7.2. ()Permutação de variáveis.) Dizemos que um código é **pythônico** quando explora a linguagem para escrevê-lo de forma sucinta e de fácil compreensão. Por exemplo, a permutação de variáveis é classicamente feita como segue

```
>>> x = 1
>>> y = 2
>>> z = x
>>> x = y
>>> y = z
>>> x, y
(2, 1)
```

Note que na última linha, um `tuple` foi criado. Ou seja, a criação de `tuples` não requer o uso de parênteses, basta colocar os objetos separados por vírgulas. Podemos explorar isso e escrevermos o seguinte código pythônico para a permutação de variáveis:

```
>>> x, y = y, x
>>> x, y
(1, 2)
```

2.7.3 Listas: `list`

Em Python, `list` é uma classe de objetos do tipo lista, é uma coleção de objetos indexada e mutável. Para a criação de uma lista, usamos colchetes `[]`:

```
>>> a = [1, -3.7, 'amarelo', -5, (-3,1)]
>>> type(a)
<class 'list'>
>>> a[2]
'amarelo'
>>> a[1::2]
[-3.7, -5]
```

Exemplo 2.7.1. (Vetores alocados como `lists`.) Sejam dados dois vetores

$$v = (v_1.v_2, v_3), \tag{2.53}$$
$$w = (w_1, w_2, w_3). \tag{2.54}$$

O produto interno $v \cdot w$ é calculado por

$$v \cdot w := v_1 w_1 + v_2 w_2 + v_3 w_3. \tag{2.55}$$

O seguinte código, aloca os vetores

$$v = (-1, 2, 1), \tag{2.56}$$
$$w = (3, -1, 4) \tag{2.57}$$

usando `lists`, computa o produto interno $v \cdot w$ e imprime o resultado.

```
v = [-1, 2, 1]
w = [3, -1, 4]
p = v[0]*w[0] \
    + v[1]*w[1] \
    + v[2]*w[2]
print(f'v.w = {p}')
```

Exemplo 2.7.2. (Matrizes e listas encadeadas.) Consideramos a matriz

$$A = \begin{bmatrix} -1 & 1 \\ 1 & 3 \end{bmatrix} \tag{2.58}$$

Podemos alocá-la por linhas pelo encadeamento de `lists`, i.e.

```
>>> A = [[-1,1],[1,3]]
>>> A
[[-1, 1], [1, 3]]
```

Com isso, podemos obter a segunda linha da matriz com

```
>>> A[1]
[1, 3]
```

Ou ainda, podemos obter o elemento da segunda linha e primeira coluna com

```
>>> A[1][0]
1
```

Observação 2.7.3. (Operadores.) Os operadores envolvendo `tuples` são análogos para `lists`. Por exemplo,

```
>>> a = [1,2]
>>> b = [3,4]
>>> a + b
[1, 2, 3, 4]
>>> 2*a
[1, 2, 1, 2]
>>> a <= a
True
```

Modificações em `lists`

`list` é uma classe de objetos mutável, i.e. permite que a coleção de objetos que a constituem seja alterada. Pode-se fazer a alteração de itens usando-se suas posições, por exemplo

```
>>> a = [1, -3.7, 'amarelo', -5, (-3,1)]
>>> a[1] = 7.5
>>> a
[1, 7.5, 'amarelo', -5, (-3, 1)]
>>> a[1:3] = ['mar', -2.47]
>>> a
[1, 'mar', -2.47, -5, (-3, 1)]
>>> a[:2] = 7
```

Tem-se disponíveis os seguintes métodos para a modificação de `lists`:

- `del` **deleta elemento(s)**

```
>>> del a[:2]
>>> a
[-2.47, -5, (-3, 1)]
```

- `.insert(i, x)` **inserção de elemento(s)**

```
>>> a.insert(1, 'azul')
>>> a
[-2.47, 'azul', -5, (-3, 1)]
```

- `.append(x)` **anexa um novo elemento**

```
>>> a.append([2,1])
>>> a
[-2.47, 'azul', -5, (-3, 1), [2, 1]]
```

- `.extend(x)` **estende com novos elementos dados**

```
>>> del a[-1]
>>> a.extend([2,1])
>>> a
[-2.47, 'azul', -5, (-3, 1), 2, 1]
>>> a += [3]
>>> a
[-2.47, 'azul', -5, (-3, 1), 2, 1, 3]
```

Observação 2.7.4. (Cópia de objetos.) Em Python, dados têm um único identificador, por isso temos

```
>>> a = [1,2,3]
>>> b = a
>>> b[1] = 4
>>> a
[1, 4, 3]
```

Para fazermos uma cópia de uma `list`, podemos usar o método `.copy()`. Com isso, temos

```
>>> a = [1,2,3]
>>> b = a.copy()
>>> b[1] = 4
>>> a
[1, 2, 3]
>>> b
[1, 4, 3]
```

2.7.4 Dicionários: `dict`

Em Python, um dicionário `dict` é uma coleção de objetos em que cada elemento está associado a uma **chave**. Como chave podemos usar qualquer dado imutável (`int`, `float`, `str`, etc.). Criamos um `dict` ao alocarmos um conjunto de chaves:valores:

```
>>> x = {'nome': 'Fulane', 'idade': 19}
>>> x
{'nome': 'Fulane', 'idade': 19}
>>> y = {3: 'número inteiro', 3.14: 'pi', 2.71: 2}
>>> y
{3: 'número inteiro', 3.14: 'pi', 2.71: 2}
>>> d = {}
>>> type(d)
<class 'dict'>
```

Observamos que `{}` cria um dicionário vazio. Acessamos um valor no `dict` referenciando-se sua chave, por exemplo

```
>>> x['idade']
19
>>> y[3]
```

```
'número inteiro'
```

Podemos obter a lista de chaves de um `dict` da seguinte forma

```
>>> list(x)
['nome', 'idade']
>>> list(y)
[3, 3.14, 2.71]
```

Exemplo 2.7.3. Consideramos o triângulo de vértices $\{(0,0),(1,0),(0,1)\}$. Alocamos um dicionário contendo os vértices do triangulo

```
>>> tria = {'A': (0,0), 'B': (1,0), 'C': (0,1)}
>>> tria
{'A': (0, 0), 'B': (1, 0), 'C': (0, 1)}
```

Para recuperarmos o valor do segundo vértice, por exemplo, digitamos

```
>>> tria['B']
(1, 0)
```

Em um `dict`, valores podem ser modificados, por exemplo,

```
>>> x['nome'] = 'Fulana'
>>> x
{'nome': 'Fulana', 'idade': 19}
```

Podemos estender um `dict` pela inserção de uma nova associação chave:valor, por exemplo

```
>>> x['altura'] = 171
>>> x
{'nome': 'Fulana', 'idade': 19, 'altura': 171}
```

Exemplo 2.7.4. No Exemplo 2.7.3, alocamos o dicionário `tria` contendo os vértices de um dado triângulo. Aora, vamos computar o comprimento de cada uma de suas arestas e alocar o resultado no próprio `dict`. A distância entre dois pontos $A=(a_1,a_2)$ e $B=(b_1,b_2)$ pode ser calculada por

$$d(A,b) := \sqrt{(b_1-a_1)^2+(a_2-b_2)^2} \tag{2.59}$$

Segue nosso código:

```python
# vértices do triangulo
tria = {'A': (0,0), 'B': (1,0), 'C': (0,1)}
# aresta AB
tria['AB'] = ((tria['B'][0] - tria['A'][0])**2 \
    + (tria['B'][1] - tria['A'][1])**2)**0.5
# aresta BC
tria['BC'] = ((tria['C'][0] - tria['B'][0])**2 \
    + (tria['C'][1] - tria['B'][1])**2)**0.5
# aresta AC
tria['AC'] = ((tria['C'][0] - tria['A'][0])**2 \
    + (tria['C'][1] - tria['A'][1])**2)**0.5
# novo dicionário
print(tria)
```

2.7.5 Exercícios

E.2.7.1. Crie um código que aloque os seguintes conjuntos

$$A = \{1, 4, 7\} \tag{2.60}$$
$$B = \{1, 3, 4, 5, 7, 8\} \tag{2.61}$$

e verifique as seguintes afirmações:

a) $A \supset B$

b) $A \subset B$

c) $B \not\supset A$

d) $A \subsetneqq B$

E.2.7.2. Crie um código que aloque os seguintes conjuntos

$$A = \{-3, -1, 0, 1, 6, 7\} \tag{2.62}$$
$$B = \{-4, 1, 3, 5, 6, 7\} \tag{2.63}$$
$$C = \{-5, -3, 1, 2, 3, 5\} \tag{2.64}$$

e, então, compute as seguintes operações:

a) $A \cap B$

b) $C \cup B$

c) $C \setminus A$

d) $B \cap (A \cup C)$

E.2.7.3. O produto cartesiano[28] de um conjunto X com um conjunto Y é o seguinte conjunto de pares ordenados

$$X \times Y := \{(x, y) : \; x \in X \wedge y \in Y\}. \tag{2.65}$$

Crie um código que aloque os conjuntos

$$X = \{-2, 1, 3\}, Y = \{5, -1, 2\} \tag{2.66}$$

e $X \times Y$. Por fim, fornece a quantidade de elementos de $X \times Y$.

E.2.7.4. A sequência de Fibonacci[29] $(f_n)_{n \in \mathcal{N}}$ é definida por

$$f_n := \begin{cases} 0 & , n = 0, \\ 1 & , n = 1, \\ f_{n-2} + f_{n-1} & , n \geq 2 \end{cases} \tag{2.67}$$

Crie um código que aloque os 6 primeiros elementos da sequência em um `list` e imprima-o.

E.2.7.5. Crie um código que usa de `lists` para alocar os seguintes vetores

$$\boldsymbol{v} = (-1, 0, 2), \tag{2.68}$$
$$\boldsymbol{w} = (3, 1, 2) \tag{2.69}$$

e computar:

[28]René Descartes, 1596 - 1650, matemático e filósofo francês. Fonte: Wikipédia.
[29]Leonardo Fibonacci, 1170 - 1250, matemático italiano. Fonte: Wikipédia.

a) $\boldsymbol{v}+\boldsymbol{w}$

b) $\boldsymbol{v}-\boldsymbol{w}$

c) $\boldsymbol{v}\cdot\boldsymbol{w}$

d) $\|\boldsymbol{v}\|$

e) $\|\boldsymbol{v}-\boldsymbol{w}\|$

E.2.7.6. Crie um código que usa de listas encadeadas para alocar a matriz

$$A=\begin{bmatrix}1 & -1\\ 2 & 3\end{bmatrix} \tag{2.70}$$

e imprima o determinante de A, i.e.

$$|A| := a_{1,1}a_{2,2}-a_{1,2}a_{2,1}. \tag{2.71}$$

E.2.7.7. Crie um código que use de listas para alocar a matriz

$$A=\begin{bmatrix}1 & -1 & 2\\ 2 & 0 & -3\\ 3 & 1 & -2\end{bmatrix} \tag{2.72}$$

e o vetor

$$\boldsymbol{x}=(-1,2,1). \tag{2.73}$$

Na sequência, compute $A\boldsymbol{x}$ e imprime o resultado.

Respostas

E.2.7.1.

```python
A = {1,4,7}
B = {1,3,4,5,7,8}
# a)
a = A >= B
print(f"a) A>=B: {a}")
# b)
b = A <= B
print(f"b) A<=B: {b}")
# c)
c = not(B >= A)
print(f"c) not(A>=B): {c}")
# d)
d = A < B
print(f"d) A<B: {d}")
```

E.2.7.2.

```python
A = {-3,-1,0,1,6,7}
B = {-4,1,3,5,6,7}
C = {-5,-3,1,2,3,5}
# a)
a = A & B
print(f"a)\n A&B = {a}")
# b)
b = C | B
print(f"b)\n A|B = {b}")
# c)
c = C - A
print(f"c)\n C-A = {c}")
# d)
d = B & (A | C)
print(f"d)\n B&(A|C) = {d}")
```

E.2.7.3.

```python
X = {-2,1,3}
Y = {5,-1,2}
XxY = {(-2,5), (-2,-1), (-2,2), \
       (1,5), (1,-1), (1,2), \
       (3,5), (3,-1), (3,2)}
print(f'#(X x Y) = {len(XxY)}')
```

E.2.7.4.

```
a = [0,1]
a.append(a[0]+a[1])
a.append(a[1]+a[2])
a.append(a[2]+a[3])
a.append(a[3]+a[4])
print(a)
```

E.2.7.5.

```
v = [-1, 0, 2]
w = [3, 1, 2]
# a)
vpw = [v[0] + w[0],
       v[1] + w[1],
       v[2] + w[2]]
print(f'a) v+w = {vpw}')
# b)
vmw = [v[0] - w[0],
       v[1] - w[1],
       v[2] - w[2]]
print(f'b) v-w = {vmw}')
# c)
vdw = v[0]*w[0] + \
      v[1]*w[1] + \
      v[2]*w[2]
print(f'c) v.w = {vdw}')
# d)
norm_v = (v[0]**2 + \
          v[1]**2 + \
          v[2]**2)**0.5
print(f'd) ||v|| = {norm_v:.2f}')
# e)
norm_vmw = (vmw[0]**2 + \
            vmw[1]**2 + \
            vmw[2]**2)**0.5
print(f'e) ||v-w|| = {norm_vmw:.2f}')
```

E.2.7.6.

```
A = [[1, -1],
     [2, 3]]
detA = A[0][0]*A[1][1] \
     - A[0][1]*A[1][0]
print(f'|A| = {detA}')
```

E.2.7.7.

```
A = [[1, -1, 2],
     [2, 0, -3],
     [3, 1, -2]]
x = [-1, 2, 1]
Ax = [A[0][0]*x[0] + A[0][1]*x[1] + A[0][2]*x[2],
      A[1][0]*x[0] + A[1][1]*x[1] + A[1][2]*x[2],
      A[2][0]*x[0] + A[2][1]*x[1] + A[2][2]*x[2]]
print(f'Ax = {Ax}')
```

Capítulo 3

Programação Estruturada

No paradigma de programação estruturada, o programa é organizado em blocos de códigos. Cada bloco tem uma entrada de dados, um processamento (execução de uma tarefa) e produz uma saída.

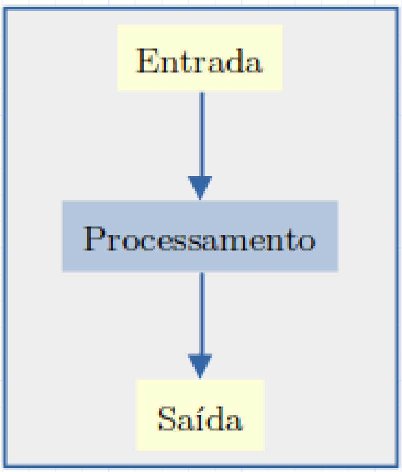


Figura 3.1: Bloco de processamento.

Blocos podem ser colocados em sequência, selecionados com base em condições lógicas, iterados ou colocados dentro de outros blocos (sub-blocos).

3.1 Estruturas de um Programa

Para escrever qualquer programa, apenas três estruturas são necessárias: **sequência**, **seleção/ramificação** e **iteração**.

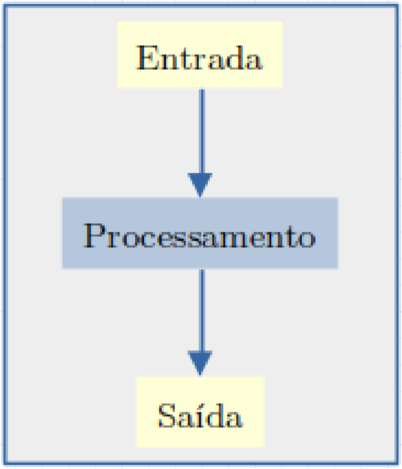


Figura 3.2: Bloco de processamento.

3.1.1 Sequência

A estrutura de **sequência** apenas significa que os blocos de programação são executados em sequência. Ou seja, a execução de um bloco começa somente após a finalização do bloco anterior.

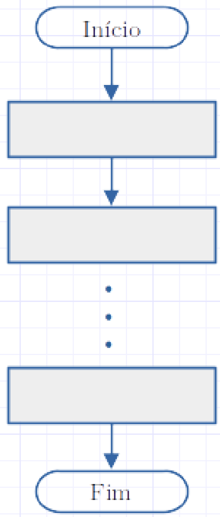


Figura 3.3: Estrutura de sequência de blocos.

Exemplo 3.1.1. O seguinte código computada a área do triângulo de base e altura informadas pela(o) usuária(o).

```
1 #início
2 
3 # bloco: entrada de dados
4 base = float(input('Digite a base:\n'))
5 altura = float(input('Digite a altura\n'))
6 
7 # bloco: computação da área
8 area = base*altura/2
9 
10 # bloco: saída de dados
11 print(f'Área = {area}')
12 
13 #fim
```

O código acima está estruturado em três blocos. O primeiro bloco (linhas 3-5) processa a entrada de dados, seu término ocorre somente após a(o) usuária(o) digitar os valores da base e da altura. Na sequência, o bloco (linhas 7-8) faz a computação da área do triângulo e aloca o resultado na variável `area`. No que este bloco termina seu processamento, é executado o último bloco (linhas 10-11), que imprime o resultado na tela.

3.1.2 Ramificação

Estruturas de ramificação permitem a seleção de um mais blocos com base em condições lógicas.

Exemplo 3.1.2. O seguinte código lê um número inteiro digitado pela(o) usuária(o) e imprime uma mensagem no caso do número digitado ser par.

```
1 #início
2 
3 # entrada de dados
4 n = int(input('Digite um número inteiro:\n'))
5 
6 # ramificação
7 if (n%2 == 0):
8     print(f'{n} é par.')
9 
10 #término
```

Observamos que, no caso do número digitado não ser par, o programa termina sem nenhuma mensagem ser impressa. Esse é um exemplo de um bloco de ramificação, a instrução de ramificação (linha 7) testa a condição de `n` ser par. Somente no caso de ser verdadeiro, a instrução de impressão (linha 8) é executada. Após e impressão o programa é encerrado. No caso de `n` não ser par, o programa é encerrado sem que a instrução da linha 8 seja executada, i.e. a mensagem não é impressa.

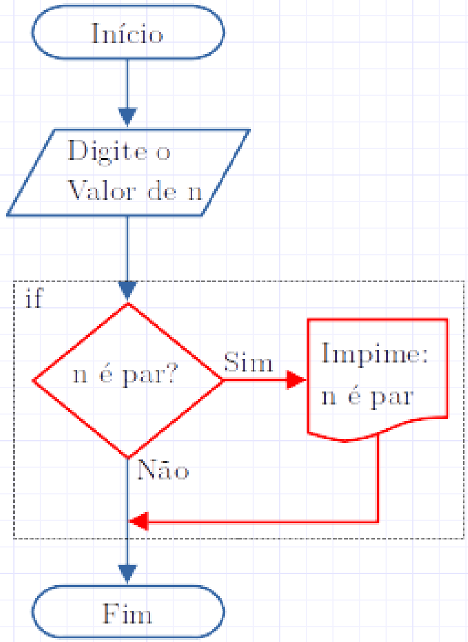


Figura 3.4: Fluxograma de uma estrutura de ramificação.

Observação 3.1.1. (Escopo e indentação.) Na linguagem Python, a indentação indica o **escopo**, i.e. o início e fim do bloco de instruções que pertencem a ramificação. No Exemplo 3.1.2, o escopo da instrução `if` é apenas a linha 8.

3.1.3 Repetição

Instruções de repetição permitem que um mesmo bloco seja processado várias vezes em sequência. Em Python, há duas instruções de repetição disponíveis: `for` e `while`.

`for`

A instrução `for` permite que um bloco seja iterado para cada elemento de uma dada coleção de dados.

Exemplo 3.1.3. O seguinte código testa a paridade de cada um dos elementos do conjunto $\{-3, -2, -1, 0, 1, 2, 3\}$.

```
1 #início
2
3 # repetição for
4 for n in {-3, -2, -1, 0, 1, 2, 3}:
5     res = (n%2 == 0)
6     print(f'{n} é par? ', res)
7
8 #término
```

A instrução de repetição `for` (linha 4), aloca em `n` um dos elementos do conjunto. Então, executa em sequência o bloco de comandos das linhas 5 e 6. De forma iterada, `n` recebe um novo elemento do conjunto e o bloco das linhas 5 e 6 é novamente executado. A repetição termina quando todos os elementos do conjunto já tiverem sido iterados. O código segue, então, para a linha 7. Não havendo mais instruções, o programa é encerrado.

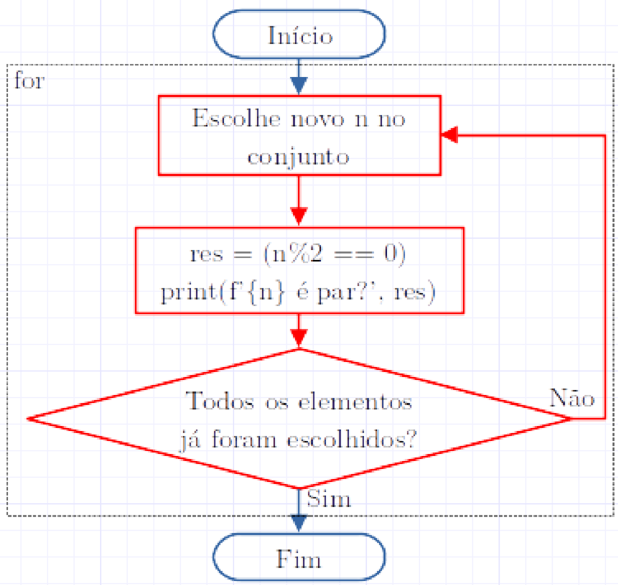


Figura 3.5: Fluxograma de uma estrutura de repetição do tipo `for`.

Assim como no caso de uma instrução de ramificação, o escopo do `for` é definido pela indentação do código. Neste exemplo, o escopo são as linhas 5 e 6.

while

A instrução `while`, permite a repetição de um bloco enquanto uma dada condição lógica é satisfeita.

Exemplo 3.1.4. O seguinte código testa a paridade dos números inteiros compreendidos de -3 a 3.

```
1 #início
2
3 n = -3
4
5 # repetição: while
6 while (n <= 3):
7     res = (n%2 == 0)
8     print(f'{n} é par?', res)
9     n += 1
10
11 #término
```

A instrução de repetição `while` faz com que o bloco de processamento definido pelas linhas 7-9 seja executado de forma sequencial enquanto o valor de `n` for menor ou igual a 3. No caso dessa condição ser verdadeira, o bloco (linhas 7-9) é executado e, então a condição é novamente verificada. No caso da condição ser falsa, esse bloco não é executado e o código segue para a linha 10. Não havendo mais nenhuma instrução, o programa é encerrado.

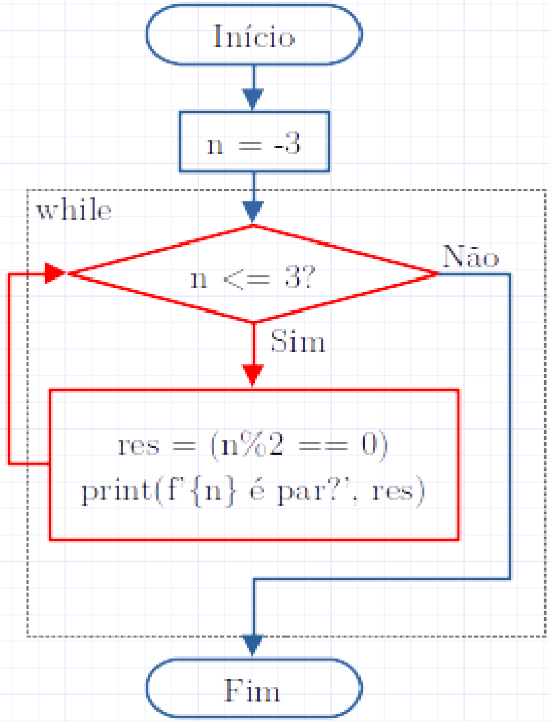


Figura 3.6: Fluxograma da estrutura de repetição do tipo `while` para o Exemplo 3.1.4.

Observamos que, neste exemplo, o escopo da instrução `while` são as linhas 7-9, determinado indentação do código.

3.1.4 Exercícios

E.3.1.1. Seja a reta de equação

$$y = ax + b. \tag{3.1}$$

Assumindo $a = 2$ e $b = -3$, o seguinte código foi desenvolvido para computar o ponto x de interseção da desta reta com o eixo das abscissas.

```
x = -b/2*a
a = 2
b = -3
```

```
print(x)
```

Identifique e explique os erros desse código. Então, apresente uma versão corrigida.

E.3.1.2. Seja a reta de equação

$$y = ax + b. \tag{3.2}$$

Faça um fluxograma de um programa em que a(o) usuária(o) entra com os valores de a e b. No caso de $a \neq 0$, o programa computa e imprime o ponto x da interseção dessa reta com o eixo das abscissas.

E.3.1.3. Implemente o código referente ao fluxograma criado no Exercício 3.1.2.

E.3.1.4. Faça o fluxograma de um programa que usa de um bloco de repetição `for` para percorrer o conjunto

$$A = \{-4, -3, -2, -1, 0, 1, 2, 3, 4\}. \tag{3.3}$$

A cada iteração, o programa imprime `True` ou `False` conforme o elemento seja ímpar ou não.

E.3.1.5. Implemente o código referente ao fluxograma criado no Exercício 3.1.4.

E.3.1.6. Faça um fluxograma análogo ao do Exercício 3.1.4 que use a instrução de repetição `while` no lugar de `for`.

E.3.1.7. Implemente um código referente ao fluxograma criado no Exercício 3.1.6.

Respostas

E.3.1.1.

```
a = 2
b = -3
x = -b/a
print(x)
```

E.3.1.2. Dica: consulte o Exemplo 3.1.2.

E.3.1.3.

```
a = float(input('Digite o valor de a:\n'))
b = float(input('Digite o valor de b:\n'))
if (a != 0):
    x = -b/(2*a)
    print(f'Ponto de interseção com o eixo x = {x}')
```

E.3.1.4. Dica: consulte o Exemplo 3.1.3.

E.3.1.5.

```
A = {-4, -3, -2, -1, \
     0, 1, 2, 3, 4}
for x in A:
    res = (x % 2 != 0)
    print(f'{x} é ímpar? {res}')
```

E.3.1.6. Dica: consulte o Exemplo 3.1.4.

E.3.1.7.

```
A = {-4, -3, -2, -1, \
     0, 1, 2, 3, 4}
n = -4
while (n <= 4):
    res = (n % 2 != 0)
    print(f'{n} é ímpar? {res}')
    n += 1
```

3.2 Instruções de Ramificação

Instruções de ramificação permitem a seleção de blocos de processamento com base em condições lógicas.

3.2.1 Instrução `if`

A instrução de ramificação `if` permite a seleção de um bloco de processamento com base em uma condição lógica.

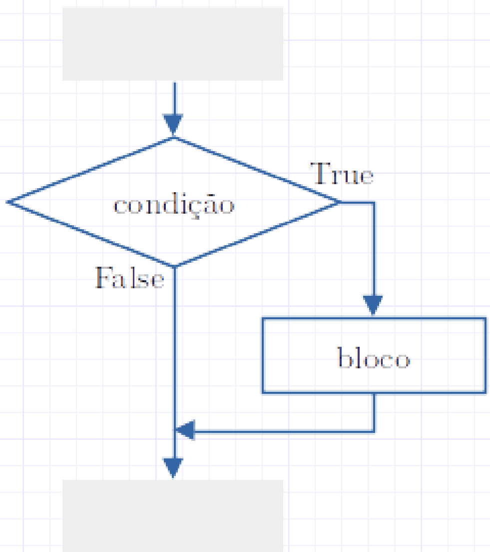


Figura 3.7: Fluxograma de uma ramificação `if`.

Em Python, a instrução `if` tem a seguinte sintaxe:

```
1 bloco_anterior
2 if (condição):
3     bloco_0
4 bloco_posterior
```

Se a `condição` é verdadeira (`True`), o bloco (linha 3) é executado. Caso contrário, este bloco não é executado e o fluxo de processamento salta da linha 2 para a linha 6. O **escopo** do bloco `if` é determinado pela indentação do código.

Exemplo 3.2.1. Seja o polinômio de segundo grau

$$p(x) = ax^2 + bx + c. \tag{3.4}$$

No caso de existirem, o seguinte código computa as raízes distintas de $p(x)$ para os coeficientes informados pela(o) usuária(o).

```
# entrada de dados
a = float(input('Digite o valor de a:\n'))
b = float(input('Digite o valor de b:\n'))
c = float(input('Digite o valor de c:\n'))
```

```
5
6 # discriminante
7 delta = b**2 - 4*a*c
8
9 # raízes
10 if (delta > 0):
11     # raízes distintas
12     x1 = (-b - delta**0.5)/(2*a)
13     x2 = (-b + delta**0.5)/(2*a)
14     print(f'x_1 = {x1}')
15     print(f'x_2 = {x2}')
```

Escopo de variáveis

O **escopo** de uma variável é a região em que ela permanece alocada. O escopo de variáveis alocadas fora do bloco `if` inclui este bloco, mas variáveis alocadas no bloco `if` não permanecem alocadas fora deste.

Exemplo 3.2.2. No Exemplo 3.2.1, o escopo da variável `delta` inicia-se na linha 7 e permanece válido ao longo do resto do programa. Já, o escopo da variável `x1` compreende somente as linhas 12-15 e, análogo para a variável `x2`.

3.2.2 Instrução `if-else`

A instrução `if-else` permite a escolha de um bloco ou outro, exclusivamente, com base em uma condição lógica.

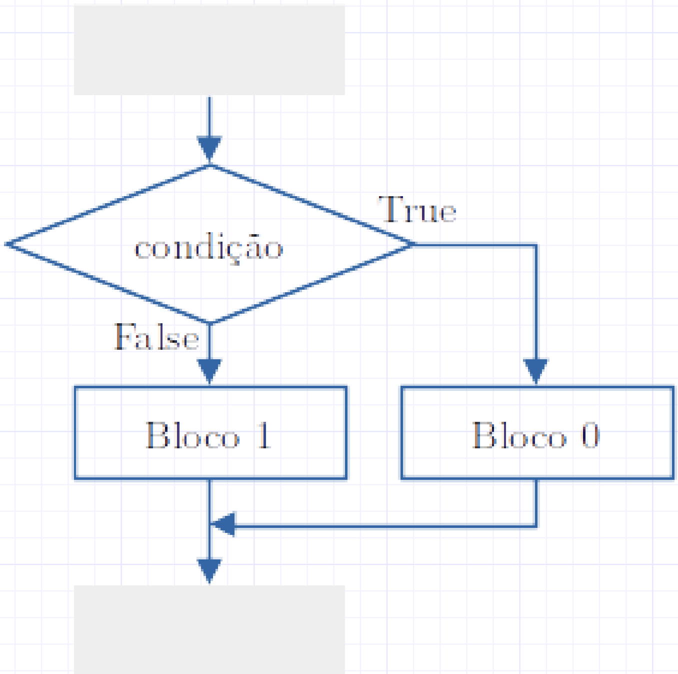


Figura 3.8: Fluxograma de uma ramificação `if-else`.

Em Python, a instrução `if-else` tem a seguinte sintaxe:

```
bloco_anterior
if (condição):
    bloco_0
else:
    bloco_1
bloco_posterior
```

Se a `condição` for verdadeira (`True`) o `bloco 0` é executado, senão o `bloco 1` é executado.

Exemplo 3.2.3. Seja o polinômio de segundo grau

$$p(x) = ax^2 + bx + c. \tag{3.5}$$

Se existirem, o seguinte código computa as raízes reais do polinômio, senão imprime mensagem informado que elas não são reais.

```python
# entrada de dados
a = float(input('Digite o valor de a:\n'))
b = float(input('Digite o valor de b:\n'))
c = float(input('Digite o valor de c:\n'))

# discriminante
delta = b**2 - 4*a*c

# raízes
if (delta >= 0):
    x1 = (-b - delta**0.5)/(2*a)
    x2 = (-b + delta**0.5)/(2*a)
    print(f'x_1 = {x1}')
    print(f'x_2 = {x2}')
else:
    print('Não tem raízes reais.')
```

Instrução `if-else` em linha

Por praticidade, Python também tem a sintaxe `if-else` em linha:

```python
x = valor if True else outro_valor
```

Exemplo 3.2.4. O valor absoluto de um número real x é

$$|x| := \begin{cases} x & , x \geq 0, \\ -x & , x < 0 \end{cases} \tag{3.6}$$

O seguinte código, computa o valor absoluto[1] de um número dado pela(o) usuária(o).

```python
x = float(input('Digite o valor de x:\n'))
abs_x = x if (x>=0) else -x
print(f'|x| = {abs_x}')
```

3.2.3 Instrução `if-elif`

A instrução `if-elif` permite a seleção condicional de blocos, sem impor a necessidade da execução de um deles.

[1] Python tem a função `abs()` que computa o valor absoluto de um número.

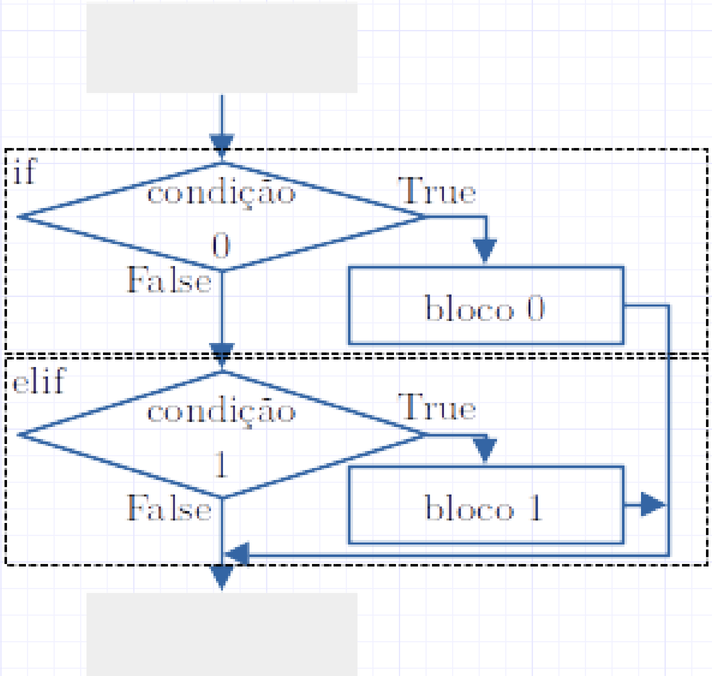


Figura 3.9: Fluxograma de uma ramificação `if-elif`.

Em Python, a instrução `if-elif` tem a seguinte sintaxe:

```
1 bloco_anterior
2 if (condição_0):
3     bloco_0
4 elif (condição 1):
5     bloco_1
6 bloco posterior
```

Se a `condição_0` for verdadeira (lstinline+True+), o `bloco_0` é executado. Senão, se a `condição_1` for verdadeira (`True`) o `bloco_1` é executado. No caso de ambas as condições serem falsas (`False`), os blocos `bloco_0` e `bloco_1` não são executados e o fluxo de processamento segue a partir da linha 6.

Exemplo 3.2.5. Seja o polinômio de segundo grau

$$p(x) = ax^2 + bx + c. \tag{3.7}$$

Conforme o caso, o seguinte código computa a raiz dupla do polinômio ou suas raízes reais distintas, a partir dos coeficientes informados pela(o) usuária(o).

```
# entrada de dados
a = float(input('Digite o valor de a:\n'))
b = float(input('Digite o valor de b:\n'))
c = float(input('Digite o valor de c:\n'))

# discriminante
delta = b**2 - 4*a*c

# raízes
if (delta > 0):
    x1 = (-b - delta**0.5)/(2*a)
    x2 = (-b + delta**0.5)/(2*a)
    print('Raízes reais distintas:')
    print(f'x_1 = {x1}')
    print(f'x_2 = {x2}')
elif (delta == 0):
    print('Raiz dupla:')
    x = -b/(2*a)
    print(f'x_1 = x_2 = {x}')
```

3.2.4 Instrução `if-elif-else`

A instrução `if-elif-else` permite a seleção condicional de blocos, sendo que ao menos um bloco será executado. Em Python, sua sintaxe é:

```
bloco_anterior
if (condição_0):
    bloco_0
elif (condição_1):
    bloco_1
else:
    bloco_2
bloco_posterior
```

Se a `condição_0` for verdadeira (`True`), então o `bloco_0` é executado. Senão, se a `condição_1` for verdadeira (`True`), então o `bloco_1` é executado. Senão, o `bloco_2` é executado.

Exemplo 3.2.6. Seja o polinômio de segundo grau

$$p(x) = ax^2 + bx + c. \tag{3.8}$$

Conforme o caso (raízes reais distintas, raiz dupla ou raízes complexas), o seguinte código computa as raízes desse polinômio, a partir dos coeficientes informados pela(o) usuária(o).

```
# entrada de dados
a = float(input('Digite o valor de a:\n'))
b = float(input('Digite o valor de b:\n'))
c = float(input('Digite o valor de c:\n'))

# discriminante
delta = b**2 - 4*a*c

# raízes
if (delta > 0):
    # raízes distintas
    x1 = (-b - delta**0.5)/(2*a)
    x2 = (-b + delta**0.5)/(2*a)
    print('Raízes reais distintas:')
    print(f'x_1 = {x1}')
    print(f'x_2 = {x2}')
elif (delta == 0):
    # raiz dupla
    x = -b/(2*a)
    print('Raiz dupla:')
    print(f'x_1 = x_2 = {x}')
else:
    # raízes complexas
    # parte real
    rea = -b/(2*a)
    # parte imaginária
    img = (-delta)**0.5/(2*a)
    x1 = rea - img*1j
    x2 = rea + img*1j
    print('Raízes complexas:')
    print(f'x_1 = {x1}')
    print(f'x_2 = {x2}')
```

3.2.5 Múltiplos Casos

Pode-se encadear instruções `if-elif-elif-...-elif[-else]` para a seleção condicional entre múltiplos blocos.

Exemplo 3.2.7. Sejam as circunferências de equações:

$$c_1 : (x - a_1)^2 + (y - b_1)^2 = r_1, \tag{3.9}$$

$$c_2 : (x - a_1)^2 + (y - b_1)^2 = r_2. \tag{3.10}$$

Conforme entradas dadas por usuária(o), o seguinte código informa se um dado ponto (x, y) pertence: à interseção dos discos determinados por c_1 e c_2, apenas ao disco determinado por c_1, apenas ao disco determinado por c_2 ou a nenhum desses discos.

```python
# entrada de dados
print('c1: (x-a1)**2 + (y-b1)**2 = r1')
a1 = float(input('Digite o valor de a1:\n'))
b1 = float(input('Digite o valor de b1:\n'))
r1 = float(input('Digite o valor de r1:\n'))
print('c2: (x-a2)**2 + (y-b2)**2 = r1')
a2 = float(input('Digite o valor de a2:\n'))
b2 = float(input('Digite o valor de b2:\n'))
r2 = float(input('Digite o valor de r2:\n'))
print('Ponto de interesse (x,y).')
x = float(input('Digite o valor de x:\n'))
y = float(input('Digite o valor de y:\n'))

# pertence ao disco c1?
c1 = (x-a1)**2 + (y-b1)**2 <= r1
# pertence ao disco c2?
c2 = (x-a2)**2 + (y-b2)**2 <= r2

# imprime resultado
if (c1 and c2):
    print(f'({x}, {y}) pertence à interseção dos discos.')
elif (c1):
    print(f'({x},{y}) pertence ao disco c1.')
elif (c2):
    print(f'({x},{y}) pertence ao disco c2.')
else:
    print(f'({x},{y}) não pertence aos discos.')
```

3.2.6 Exercícios

E.3.2.1. Seja a equação de reta

$$ax + b = 0. \tag{3.11}$$

Dados coeficientes $a \neq 0$ e b informados por usuária(o), crie um código que imprime o ponto de interseção dessa reta com o eixo das abscissas. O código não deve tentar computar o ponto no caso de $a = 0$.

E.3.2.2. Considere o seguinte código.

```
n = int(intput('Digite um número inteiro:\n')
if (n % 2 == 0):
    m = 1
n = n + m
print(n)
```

A ideia é que, se n for ímpar, o código imprime n, caso contrário, imprime $n + 1$. Este código contém erro. Identifique e explique-o, então proponha uma versão funcional.

E.3.2.3. Considere o seguinte algoritmo/pseudocódigo para verificar se um dado número inteiro n é par ou ímpar.

0. Início.
1. Usuária(o) informa o valor inteiro n.
2. Se o resto da divisão de n por 2 for igual a zero, então faça:
   2.1. Imprime a mensagem: "n é par.".
3. Senão, faça:
   3.1 Imprime a mensagem: "n é ímpar".
4. Fim.

Faça um fluxograma para esse algoritmo e implemente-o.

E.3.2.4. Considere a equação da circunferência

$$c : (x - a)^2 + (y - b)^2 = r^2. \tag{3.12}$$

Com dados informados por usuária(o), desenvolva um código que informe se um dado ponto (x, y) pertence ou não ao disco determinado por c.

E.3.2.5. Sejam informadas por usuária(o) os coeficientes das retas

$$r_1 : a_1x + b_1 = 0, \tag{3.13}$$
$$r_2 : a_2x + b_2 = 0. \tag{3.14}$$

Crie um código que informe se as retas são paralelas. Caso contrário, o código imprime o ponto de interseção delas.

E.3.2.6. Refaça o código do Exercício 3.2.5 de forma a incluir o caso em que as retas sejam coincidentes. Ou seja, o código deve informar os seguintes casos: retas paralelas não coincidentes, retas coincidentes ou, caso contrário, ponto de interseção das retas.

E.3.2.7. Sejam a parábola de equação

$$a_1x^2 + a_2x + a_3 = 0 \tag{3.15}$$

e a reta

$$b_1x + b_2 = 0. \tag{3.16}$$

Conforme os coeficientes dados por usuária(o), desenvolva um código que imprime o(s) ponto(s) de interseção da reta com a parábola. O código deve avisar os casos em que: há apenas um ponto, há dois pontos ou não há ponto de interseção.

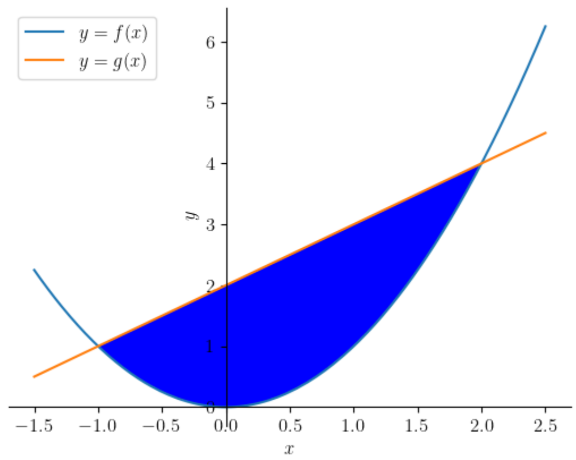
E.3.2.8. Com dados informados por usuária(o), sejam as circunferências de equações

$$c_1 : (x - a_1)^2 + (y - b_1)^2 = r_1^2, \tag{3.17}$$
$$c_2 : (x - a_2)^2 + (y - b_2)^2 = r_2^2. \tag{3.18}$$

Desenvolva um código que informe a(o) usuária(o) dos seguintes casos: c_1 e c_2 são coincidentes, $c_1 \cap c_2$ tem dois pontos, $c_1 \cap c_2$ tem somente um ponto, $c_1 \cap c_2 = \emptyset$.

E.3.2.9. Crie uma calculadora simples. A(o) usuária(o) entra com dois números decimais `x` e `y` e uma das seguintes operações: `+`, `-`, `*` ou `/`. Então, o código imprime o resultado da operação.

E.3.2.10. Informado um ponto $P = (x, y)$ por usuária(o), desenvolva um código que verifique se P está entre as curvas $x = -1$, $x = 2$, $y = x^2$ e $y = x + 2$. Consulte a figura abaixo.



E.3.2.11. Considere um polinômio da forma

$$p(x) = (x - a)(bx^2 + cx + d). \tag{3.19}$$

Desenvolva um código para a computação das raízes de $p(x)$, sendo os coeficientes a, b, c e d (números decimais) informados por usuária(o).

Respostas

E.3.2.1.

```
# entrada de dados
a = float(input('Digite o valor de a:\n'))
b = float(input('Digite o valor de b:\n'))

# computação
if (a != 0):
    x = -b/a
    y = a*x + b
    print(f'Intercepta eixo-x em: ({x}, {y}).')
```

E.3.2.2.

```
n = int(intput('Digite um número inteiro:\n')
m = 0
if (n % 2 == 0):
    m = 1
n = n + m
print(n)
```

E.3.2.4.

```
# entrada de dados
print('Circunferência c:')
a = float(input('Digite o valor de a:\n'))
b = float(input('Digite o valor de b:\n'))
r = float(input('Digite o valor de r:\n'))
print('Ponto (x, y):')
x = float(input('Digite o valor de x:\n'))
y = float(input('Digite o valor de y:\n'))

# resultado
if ((x-a)**2 + (y-b)**2 <= r**2):
    print(f'({x}, {y}) pertence ao disco.')
else:
    print(f'({x}, {y}) não pertence ao disco.')
```

E.3.2.5.

```
# entrada de dados
print('r1: a1*x + b1 = 0')
a1 = float(input('Digite o valor de a1:\n'))
b1 = float(input('Digite o valor de b1:\n'))
print('r2: a2*x + b2 = 0')
a2 = float(input('Digite o valor de a2:\n'))
b2 = float(input('Digite o valor de b2:\n'))

# resultado
if (a1 == a2):
    print('r1 // r2')
else:
    x = (b1-b2)/(a2-a1)
```

```
    y = a1*x + b1
    print('Ponto de interseção: ({x}, {y}).')
```

E.3.2.6.

```
# entrada de dados
print('r1: a1*x + b1 = 0')
a1 = float(input('Digite o valor de a1:\n'))
b1 = float(input('Digite o valor de b1:\n'))
print('r2: a2*x + b2 = 0')
a2 = float(input('Digite o valor de a2:\n'))
b2 = float(input('Digite o valor de b2:\n'))

# resultado
if (a1 == a2):
    if (b1 == b2):
        print('r1 = r2')
    else:
        print('r1 // r2 e r1 != r2')
else:
    x = (b1-b2)/(a2-a1)
    y = a1*x + b1
    print('Ponto de interseção: ({x}, {y}).')
```

E.3.2.7.

```
# entrada de dados
print('Coeficientes da parábola')
print('a1*x**2 + a2*x + a3 = 0')
a1 = float(input('Digite o valor de a1:\n'))
a2 = float(input('Digite o valor de a2:\n'))
a2 = float(input('Digite o valor de a3:\n'))

print('Coeficientes da reta')
print('b1*x + b2 = 0')
b1 = float(input('Digite o valor de b1:\n'))
b2 = float(input('Digite o valor de b2:\n'))

# discriminante da equação
# a1*x**2 + (a2-b1)*x + (a3-b2) = 0
delta = (a2-b1)**2 - 4*a1*(a3-b2)
```

```python

# ponto(s) de interseção
if (delta == 0):
    x = (b1-a2)/(2*a1)
    y = b1*x + b2
    print('Ponto de interseção:')
    print(f'({x}, {y})')
elif (delta > 0):
    x1 = ((b1-a2) - delta**2)/(2*a1)
    y1 = b1*x1 + b2
    x2 = ((b1-a2) + delta**2)/(2*a1)
    y2 = b1*x2 + b2
    print('Pontos de interseção:')
    print(f'({x1}, {y1}), ({x2}, {y2})')
else:
    print('Não há ponto de interseção.')
```

E.3.2.8.

```python
# entrada de dados
print('c1: (x-a1)**2 + (y-b1)**2 = r1**2')
a1 = float(input('Digite o valor de a1:\n'))
b1 = float(input('Digite o valor de b1:\n'))
r1 = float(input('Digite o valor de r1:\n'))
print('c2: (x-a2)**2 + (y-b2)**2 = r2**2')
a2 = float(input('Digite o valor de a2:\n'))
b2 = float(input('Digite o valor de b2:\n'))
r2 = float(input('Digite o valor de r2:\n'))

# verificações
if (((a1==a2) and (b1==b2)) and (r1==r2)):
    print('c1 = c2')
else:
    # distância entre os centros
    dist = ((a2-a1)**2 + (b2-b1)**2)**0.5
    if (abs(dist - (r1+r2)) < 1e-15):
        print('c1 & c2 têm um único ponto de interseção.')
    elif (dist < r1+r2):
        print('c1 & c2 têm dois pontos de interseção.')
    else:
```

```
        print('c1 & c2 não tem ponto de interseção.')
```

E.3.2.9.

```
# entrada de dados
x = float(input('Digite o valor de x:\n'))
op = input('Digite uma das operações +, -, * ou /:\n')
y = float(input('Digite o valor de y:\n'))

# calcula
if (op == '+'):
    print(f'{x} ' + op + f' {y} = {x+y}')
elif (op == '-'):
    print(f'{x} ' + op + f' {y} = {x-y}')
elif (op == '*'):
    print(f'{x} ' + op + f' {y} = {x*y}')
elif (op == '/'):
    print(f'{x} ' + op + f' {y} = {x/y}')
else:
    print('Desculpa, não entendi!')
```

E.3.2.10.

```
print('P = (x,y)')
x = float(input('Digite o valor de x: '))
y = float(input('Digite o valor de y: '))

if (((x >= -1.) and (x <= 2)) and
    (y >= x**2) and (y <= x+2)):
    print(f'P = ({x}, {y}) está entre as curvas.')
else:
    print(f'P = ({x}, {y}) não está entre as curvas.')
```

E.3.2.11.

```
print('p(x) = (x-a)(bx^2 + cx + d)')
# entrada de dados
a = float(input('Digite o valor de a: '))
b = float(input('Digite o valor de b: '))
c = float(input('Digite o valor de c: '))
d = float(input('Digite o valor de d: '))
```

```

# cálculo das raízes
x1 = a
print(f'x1 = {x1}')
if (b != 0.):
    delta = c**2 - 4*b*d
    x2 = (-c - delta**0.5)/(2*b)
    x3 = (-c + delta**0.5)/(2*b)
    print(f'x2 = {x2}')
    print(f'x3 = {x3}')
elif (c != 0.):
    x2 = -d/c
    print(f'x2 = {x2}')
```

3.3 Instruções de Repetição

Estruturas de repetição permitem a execução de um bloco de código várias vezes. O número de vezes que o bloco é repetido pode depender de uma condição lógica (instrução `while`) ou do número de itens de um objeto iterável (instrução `for`).

3.3.1 Instrução `while`

A instrução `while` permite a repetição condicional de um bloco de código. Em Python, sua sintaxe é

```
bloco_anterior
while condição:
    bloco
bloco_posterior
```

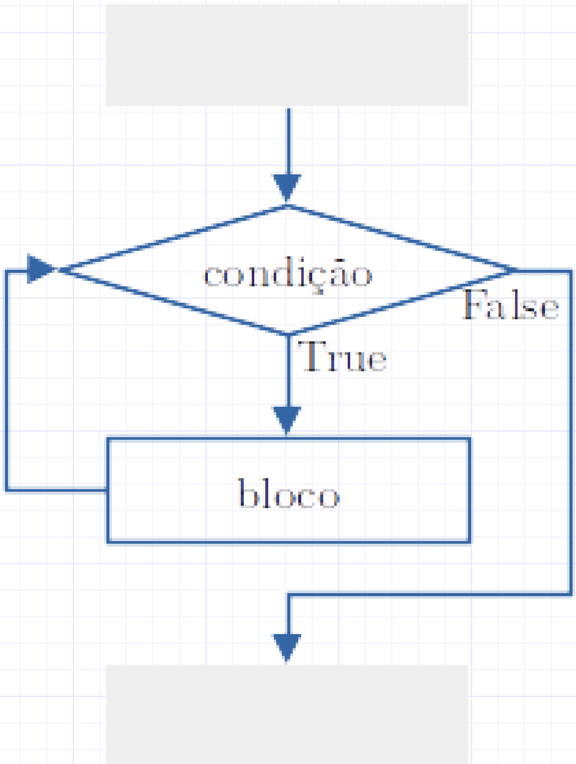


Figura 3.10: Fluxograma da estrutura de repetição `while`.

Exemplo 3.3.1. (Somatório com `while`.) O seguinte código, computa o somatório

$$s = \sum_{i=1}^{n} i \tag{3.20}$$

$$= 1 + 2 + 3 + \cdots + n. \tag{3.21}$$

```
n = int(input('Digite um número natural n:\n'))

soma = 0
i = 1
while (i <= n):
    soma = soma + i
    i = i + 1
```

```

print(f'1 + ... + {n} = {soma}')
```

Exemplo 3.3.2. (Aproximando a $\sqrt{x}$.) O método de Heron[2] é um algoritmo para o cálculo aproximado da raiz quadrada de um dado número x, i.e. $\sqrt{x}$. Consiste na iteração[3]

$$r^{(0)} = 1, \tag{3.22}$$

$$r^{(k+1)} = \frac{1}{2}\left(r^{(k)} + \frac{x}{r^{(k)}}\right), \tag{3.23}$$

para $k = 0, 1, 2, \dots, n-1$, onde n é o número de iterações calculadas. Para $x \geq 0$ fornecido por usuária(o), o seguinte código computa a aproximação $r^{(5)} \approx \sqrt{x}$.

```
x = float(input('Digite um número não negativo para x:\n'))
r = 1
k = 0
print(f'{k}: {r}')
while (k < 5):
    r = 0.5*(r + x/r)
    k = k + 1
    print(f'{k}: {r}')
print(f'sqrt({x}) = {r}')
```

`break`

A instrução `break` permite interromper um bloco de repetição e sair dele no momento em que ela é alcançada.

Exemplo 3.3.3. No Exemplo 3.3.2, podemos observar que as aproximações $s^{(k)} \approx \sqrt{x}$ vão se tornando muito próximas umas das outras conforme as iterações convergem. Dessa observação, faz sentido que interrompamos as computações no momento em que a $k+1$-ésima iterada satisfaça

$$\left|r^{(k+1)} - r^{(k)}\right| < \texttt{tol} \tag{3.24}$$

para alguma tolerância `tol` desejada.

[2]Heron de Alexandria, 10 - 80, matemático e inventor grego. Fonte: Wikipédia.

[3]Aqui, assumimos a aproximação inicial $s^{(0)} = 1$, mas qualquer outro número não negativo pode ser usado.

```
max_iter = 50
tol = 1e-15

x = float(input('Digite um número não negativo para x:\n'))

r0 = 1
k = 0
print(f'{k}: {r}')
while (k < max_iter):
    k = k + 1
    r = 0.5*(r0 + x/r0)
    print(f'{k}: {r}')
    if (abs(r-r0) < tol):
        break
    r0 = r
print(f'sqrt({x}) = {r}')
```

3.3.2 Instrução `for`

A instrução `for` iteração de um bloco de código para todos os itens de um dado objeto. Em Python, sua sintaxe é

```
bloco_anterior
for x in Iterável:
    bloco
bloco_posterior
```

Pode-se percorrer qualquer objeto iterável (`set`, `tuple`, `list`, `dict`, etc.). Em cada iteração, o índice `x` toma um novo item do objeto. A repetição termina quando todos os itens do objeto tiverem sido escolhidos. No caso de iteráveis ordenados (`tuple`, `list`, `dict`, etc.), os itens são iterados na mesma ordem em que estão alocados no objeto.

Exemplo 3.3.4. O seguinte código, computa a média aritmética do conjunto de números

$$A = \{1, 3, 5, 7, 9\}. \tag{3.25}$$

```
soma = 0
for x in {1,3,5,7,9}:
```

```
    soma = soma + x
media = soma/5
print(f'média = {media}')
```

range

A função `range([start], stop, [step])`, retorna uma sequência iterável de números inteiros, com início em `start` (padrão `start=0`), passo `step` (padão `step=1`) e limite em `stop`.

Exemplo 3.3.5. Estudamos os seguinte casos:

a) Imprime, em ordem crescente, os primeiros 11 números naturais.

```
for i in range(11):
    print(i)
```

b) Imprime, em ordem crescente, os números naturais contidos de 3 a 13, inclusive.

```
for i in range(3,14):
    print(i)
```

c) Imprime, em ordem crescente, os números naturais ímpares contidos de 3 a 13, inclusive.

```
for i in range(3,14,2):
    print(i)
```

d) Imprime, em ordem decrescente, os números naturais contidos de 3 a 13, inclusive.

```
for i in range(13,2,-1):
    print(i)
```

Exemplo 3.3.6. (Somatório com `for`.) No Exemplo 3.3.1, computados

$$s = \sum_{i=1}^{n} i \tag{3.26}$$

usando um laço `while`. Aqui, apresentamos uma nova versão do código com a instrução `for`.

```python
n = int(input('Digite um número natural n:\n'))
soma = 0
for i in range(1,n+1):
    soma = soma + i
print(f'1 + ... + {n} = {soma}')
```

Exemplo 3.3.7. No Exemplo 3.3.3, apresentamos um código para o cálculo aproximado de $\sqrt{x}$ pelo Método de Heron. Aqui, temos uma nova versão com a instrução `for` no lugar do laço `while`.

```python
max_iter = 50
tol = 1e-15

x = float(input('Digite um número não negativo para x:\n'))

r0 = 1
k = 0
print(f'{k}: {r}')
for k in range(max_iter):
    r = 0.5*(r0 + x/r0)
    print(f'{k+1}: {r}')
    if (abs(r-r0) < tol):
        break
    r0 = r
print(f'sqrt({x}) = {r}')
```

3.3.3 Exercícios

E.3.3.1. Faça o fluxograma do código apresentado no Exemplo 3.3.1. Também, desenvolva uma versão melhorada do código, que verifica se o valor de n digitado pela(o) usuária(o) é não negativa. Caso afirmativo, computa o somatório, noutro caso apenas imprime mensagem de que o n deve ser não negativo.

E.3.3.2. Faça um fluxograma para o código apresentado no Exemplo 3.3.4.

E.3.3.3. Crie um objeto do tipo `range` para cada uma das seguintes sequências:

1. Sequência crescente de todos os números inteiros de 0 até 99, inclusive.

2. Sequência crescente de todos os números pares de -5 até 15.
3. Sequência decrescente de todos os números de 100 a 0, inclusive.
4. Sequência decrescente de todos os números múltiplos de 3 entre 17 e -3.

E.3.3.4. Considere o somatório entre dois números inteiros $n \leq m$

$$\begin{aligned} s &= \sum_{i=n}^{m} i && (3.27)\\ &= n + (n+1) + (n+2) + \cdots + m && (3.28) \end{aligned}$$

Com números informados pela(o) usuária(o), escreva duas versões de códigos para a computação desse somatório:

a) Usando a instrução `while`.

b) Usando a instrução `for`.

E.3.3.5. A série harmônica é

$$\sum_{k=1}^{\infty} \frac{1}{k} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} + \cdots \tag{3.29}$$

Com n fornecido por usuária(o), crie códigos que computem o valor da soma harmônica

$$s = \sum_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}. \tag{3.30}$$

a) Use uma estrutura de repetição `while`.

b) Use uma estrutura de repetição `for`.

E.3.3.6. O cálculo do logaritmo natural pode ser feito pela seguinte série de potências

$$\ln(1+x) = \sum_{k=1}^{\infty} (-1)^{k+1} \frac{x^k}{k} \tag{3.31}$$

Desenvolva um código que compute a aproximação do $ln(2)$ dada por

$$\ln(2) = \sum_{k=1}^{n} \frac{(-1)^{k+1}}{k} \tag{3.32}$$

com $n >= 1$ número inteiro fornecido por usuária(o).

a) Use uma estrutura de repetição `while`.

b) Use uma estrutura de repetição `for`.

E.3.3.7. O fatorial de um número natural é definido pelo produtório

$$n! := \prod_{k=1}^{n} k \tag{3.33}$$

$$= 1 \cdot 2 \cdot 3 \cdot (n-1) \cdot n \tag{3.34}$$

e $0! := 1$. Com n informado por usuária(o), crie códigos para computar $n!$ usando:

a) uma estrutura de repetição `while`.

b) uma estrutura de repetição `for`.

E.3.3.8. O número de Euler[4] é tal que

$$e := \sum_{k=0}^{\infty} \frac{1}{n!} \tag{3.35}$$

$$= \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \cdots + \frac{1}{n!} + \cdots \tag{3.36}$$

Com n fornecido por usuária(o), desenvolva um código que computa a aproximação

$$e \approx e^{(n)} = \sum_{k=0}^{n} \frac{1}{n!}. \tag{3.37}$$

Qual o número n tal que $\left|e^{(n)} - e^{(n-1)}\right| < 10^{-15}$?

E.3.3.9. Com $n \geq 1$ número natural fornecido por usuária(o), crie um código que verifique se n é um número primo.

[4]Leonhard Paul Euler, 1707-1783, matemático e físico suíço. Fonte: Wikipédia.

Respostas

E.3.3.1.

```python
n = int(input('Digite um número natural n:\n'))
if (n >= 0):
    soma = 0
    i = 1
    while (i <= n):
        soma = soma + i
        i = i + 1

    print(f'1 + ... + {n} = {soma}')
else:
    print('ERRO: n deve ser não negativo.')
```

E.3.3.2. Dica: consulte o fluxograma apresentado no Exemplo 3.1.3.

E.3.3.3.

a) `range(100)`

b) `range(-4,15,2)`

c) `range(100,-1,-1)`

d) `range(15,-4,-3)`

E.3.3.4. a)

```python
n = int(input('Digite um número inteiro n:\n'))
m = int(input('Digite um número inteiro m>n:\n'))
soma = 0
i = n
while (i<=m):
    soma = soma + i
    i = i + 1
print(f'n+...+m = {soma}')
```

b)

```
n = int(input('Digite um número inteiro n:\n'))
m = int(input('Digite um número inteiro m>n:\n'))
soma = 0
for i in range(n,m+1):
    soma = soma + i
print(f'n+...+m = {soma}')
```

E.3.3.5. a)

```
n = int(input('Digite um número natural n:\n'))
s = 0
k = 1
while (k <= n):
    s = s + 1/k
    k = k + 1
print(s)
```

b)

```
n = int(input('Digite um número natural n:\n'))
s = 0
for k in range(n):
    s = s + 1/(k+1)
print(s)
```

E.3.3.6. a)

```
n = int(input('Digite um número natural n >= 1: '))

s = 0
k = 1
while (k <= n):
    s += (-1)**(k+1)/k
    k += 1
print(f'ln(2) aprrox. {s}')
```

b)

```
n = int(input('Digite um número natural n >= 1: '))

s = 0
for k in range(1, n+1):
    s += (-1)**(k+1)/k
```

```
    k += 1
print(f'ln(2) aprrox. {s}')
```

E.3.3.7. a)

```
n = int(input('Digite um número natural n:\n'))
fat = 1
k = 1
while (k < n):
    k = k + 1
    fat = fat * k
print(f'{n}! = {fat}')
```

b)

```
n = int(input('Digite um número natural n:\n'))
fat = 1
for k in range(1,n+1):
    fat = fat * k
print(f'{n}! = {fat}')
```

E.3.3.8.

```
n = int(input('Digite um número natural n:\n'))
fat = 1
e = 1
for k in range(1,n+1):
    fat = fat * k
    e = e + 1/fat
print(f'e = {e}')
```

E.3.3.9.

```
n = int(input('Digite um número natural n>=1:\n'))
primo = True
for i in range(2, n//2+1):
    if (n % i == 0):
        primo = False
        break
print(f'{n} é primo? {primo}')
```

Capítulo 4

Funções

Uma **função** (ou método) é um **subprograma** (ou subalgoritmo), um bloco de programação para o processamento de uma tarefa e que pode ser chamado à execução, sempre que necessário, pelo programa a que pertence.

4.1 Funções Predefinidas e Módulos

4.1.1 Funções Predefinidas

Como o nome indica, **funções predefinidas** são aquelas disponíveis por padrão na linguagem de programação, i.e. sem a necessidade de serem explicitamente definidas no código. As funções predefinidas do Python podem ser consultadas em

`https://docs.python.org/3/library/functions.html`

Nós já vinhamos utilizando várias dessas funções.

Entrada e Saída de Dados

Na entrada e saída de dados, utilizamos

- `input()` **entrada**

  Essa função lê uma linha digitada no *prompt*, converte-a em uma *string* e a retorna. Admite como entrada uma *string* que é impressa no *prompt* antes da leitura.

- `print()` **saída**

  Essa função recebe um objeto e o imprime em formato texto, por padrão, no *prompt* de saída.

```
>>> s = input('Olá, qual o seu nome? ')
Olá, qual o seu nome? Fulane
>>> print(f'Bem vinde, {s}!')
Bem vinde, Fulane!
```

Construtores de Dados

Temos as funções que constroem objetos de classes de números:

- `bool()` **booleano**

  Recebe um objeto e retorna outro da classe `bool`.

```
>>> bool(0)
False
>>> bool(1)
True
>>> bool('')
False
>>> bool('0')
True
```

- `int()` **inteiro**

  Recebe um número ou *string* `x` e retorna um objeto da classe `int`.

```
>>> int(-2.1)
-2
>>> int(3.9)
3
>>> int(5.5)
5
>>> int('51')
51
```

- `float()` **decimal**

  Recebe um número ou *string* `x` e retorna um objeto da classe `float`.

```
>>> float(1)
1.0
>>> float('-2.7')
-2.7
```

- `complex()` **complexo**

  Recebe as partes real e imaginária de um número complexo ou uma *string* e retorna um objeto da classe `complex`.

```
>>> complex(2,-3)
(2-3j)
>>> complex('-7+5j')
(-7+5j)
```

Para a construção de objetos de classes de coleção de dados, temos:

- `dict()` **dicionário**

  Recebe um mapeamento ou um iterável e retorna um objeto da classe `dict`.

- `list()` **lista**

  Recebe um iterável e retorna um objeto da classe `list`.

- `set()` **conjunto**

  Recebe um iterável e retorna um objeto da classe `set`.

- `str()` ***string***

  Recebe um objeto e retorna um outro da classe `str`

- `tuple` **n-upla**

  Recebe um iterável e retorna um objeto da classe `tuple`.

Alguns construtores de iteráveis especiais são:

- `range()` **sequência de números**

  Recebe até três inteiros `start`, `stop`, `step` e retorna um objeto iterável com início em `start` (incluído) e término em `stop` (excluído).

```
>>> list(range(5))
[0, 1, 2, 3, 4]
>>> tuple(range(-10,1,2))
(-10, -8, -6, -4, -2, 0)
```

- `enumerator()` **enumeração**

  Recebe um iterável e retorna um objeto `enumerate`, um iterável de tuples que enumera os objetos do iterável de entrada.

```
>>> cores = ['amarelo', 'azul', 'vermelho', ]
>>> list(enumerate(cores))
[(0, 'amarelo'), (1, 'azul'), (2, 'vermelho')]
```

4.1.2 Módulos

Módulos são bibliotecas computacionais, i.e. um arquivo contendo funções (e/ou constantes) que podem ser incorporadas e usadas em outros programas. Existem vários módulos disponíveis na linguagem Python, para citar alguns:

- `math` **módulo de matemática elementar**
- `random` **módulo de números randômicos**
- `numpy` **módulo de computação matricial**
- `matplotlib` **módulo de vizualização gráfica**
- `sympy` **módulo de matemática simbólica**
- `torch` **módulo de aprendizagem de máquina**

Nesta seção vamos apenas introduzir o módulo `math`. Mais a frente, também fazemos uma introdução aos módulos `numpy` e `matplotlib`.

Módulo `math`

O módulo `math` fornece acesso a constantes e funções matemáticas elementares para números reais. Para **importar o módulo** em nosso código, podemos usar a instrução `import`. Por exemplo,

```
>>> import math
>>> help(math)
```

Então, para usar algum recurso do módulo usamos `math.` seguido do nome do recurso que queremos. Por exemplo,

```
>>> math.e
2.718281828459045
```

retorna o número de Euler[1] em ponto flutuante.

Alternativamente, podemos importar o módulo com o nome que quisermos. Por padrão, usa-se

```
>>> import math as m
>>> m.pi
3.141592653589793
```

Ainda, pode-se importar apenas um ou mais recursos específicos, por exemplo

```
>>> from math import pi, sin, cos
>>> sin(pi)**2 + cos(pi) == 1
False
```

Exemplo 4.1.1. Considere um polinômio de segundo grau da forma

$$p(x) = ax^2 + bx + c. \tag{4.1}$$

O seguinte código, computa as raízes de p para valores dos coeficientes fornecidos por usuária(o).

```
import math as m

# entrada de dados
a = float(input('Digite o valor de a:\n'))
b = float(input('Digite o valor de b:\n'))
c = float(input('Digite o valor de c:\n'))

# discriminante
delta = b**2 - 4*a*c

# raízes
# raízes distintas
```

[1]Leonhard Paul Euler, 1707-1783, matemático e físico suíço. Fonte: Wikipédia.

```python
if (delta > 0):
    x1 = (-b + m.sqrt(delta))/(2*a)
    x2 = (-b - m.sqrt(delta))/(2*a)
# raiz dupla
elif (delta == 0):
    x1 = -b/(2*a)
    x2 = x1
# raízes complexas
else:
    real = -b/(2*a)
    img = m.sqrt(-delta)/(2*a)
    x1 = complex(real, img)
    x2 = x1.conjugate()

print(f'x1 = {x1}')
print(f'x2 = {x2}')
```

4.1.3 Exercícios

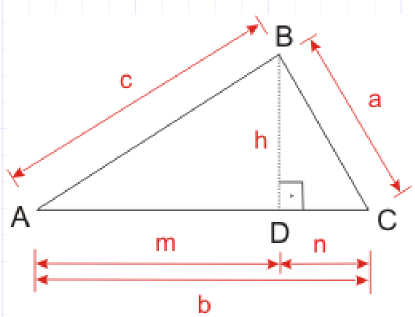
E.4.1.1. Desenvolva um código que computa e imprime a hipotenusa h de um triângulo retângulo com catetos a e b fornecidos por usuária(o).

E.4.1.2. Um triângulo de lados a, b e c, existe se

$$|b - c| < a < b + c. \tag{4.2}$$

Desenvolva um código que verifica e informa a existência de um triângulo de lados fornecidos por usuária(o).

E.4.1.3. Considere um triangulo com as seguintes medidas



Desenvolva um código que computa e imprime o valor da área de um triangulo de lados a, b e c fornecidos por usuária(o).

E.4.1.4. Desenvolva um código em que a(o) usuária forneça um ângulo θ em graus e seja computado e impresso os $\text{sen}(\theta)$ e $\cos(\theta)$.

E.4.1.5. Desenvolva um jogo em que a(o) usuária(o) tenha três tentativas para adivinhar um número inteiro entre 0 a 51 (incluídos).

Respostas

E.4.1.1. Dica: use `h = math.sqrt(a**2` b**2)+.

E.4.1.2. Dica: verifique a condição `(m.fabs(b-c) < a) and (a < b` c)+

E.4.1.3. Dica: use a lei dos cossenos e relações fundamentais de triangulo retângulo para obter o valor da altura h.

E.4.1.4. Dica: consulte as funções `math.sin`, `math.cos`.

E.4.1.5. Dica: O módulo `random` fornece a função `random.randint(a, b)` que retorna um inteiro $a \leq x \leq b$.

4.2 Definindo Funções

Em Python, criamos ou definimos uma função com a instrução `def`, com a seguinte sintaxe

```
def foo(x):
    bloco
```

Aqui, `foo` é o nome da função, `x` é o parâmetro (variável) de entrada e `bloco` é o bloco de programação que a função executa ao ser chamada. Uma função pode ter mais parâmetros ou não ter parâmetro de entrada.

Exemplo 4.2.1. O seguinte código, define a função `areaCirc` que computa e imprime a área de uma circunferência de raio r.

```
import math as m

def areaCirc(r):
```

```
    area = m.pi * r**2
    print(f'área = {area}')
```

Uma vez definida, a função pode ser chamada em qualquer parte do código. Por exemplo, vamos continuar o código de forma que a(o) usuária(o) informe os raios de duas circunferências e o código compute e imprima o valor das áreas de cada circunferência.

```
import math as m

# def fun
def areaCirc(r):
    area = m.pi * r**2
    print(f'área = {area}')

# entrada de dados
raio1 = float(input('Digite o raio da 1. circ.:\n'))
raio2 = float(input('Digite o raio da 2. circ.:\n'))

print(f'Circunferência de raio = {raio1}')
areaCirc(raio1)

print(f'Circunferência de raio = {raio2}')
areaCirc(raio2)
```

Observação 4.2.1. (`docstring`.) Python recomenda a utilização do sistema de documentação `docstring`. Na definição de funções, um pequeno comentário sobre sua funcionalidade, seguido da descrição sobre seus parâmetros podem ser feito usando `'''`, logo abaixo da instrução `def`. Por exemplo,

```
import math as m

# def fun
def areaCirc(r):
    '''
    Computa e imprime a área de uma
    circunferência.

    Entrada
    -------
    r : float
```

```
    Raio da circunferência.
    '''
    area = m.pi * r**2
    print(f'área = {area}')
```

Com isso, podemos usar a função `help` para obter a documentação da função `areaCirc`.

```
>>> help(areaCirc)
```

Verifique!

Uma função pode ser definida sem parâmetro de entrada.

Exemplo 4.2.2. O seguinte código, implementa uma função que imprime um número randômico par entre 0 e 100 (incluídos).

```
import random

def randPar100():
    '''
    Imprime um número randômico
    par entre 0 e 100 (incluídos).
    '''
    n = random.randint(0, 99)
    if (n % 2 == 0):
        print(n)
    else:
        print(n+1)
```

Para chamá-la, usamos

```
>>> randPar100()
```

Verifique!

4.2.1 Funções com Saída de Dados

Além de parâmetros de entrada, uma função pode ter saída de dados, i.e. pode retornar dados para o programa. Para isso, usamos a instrução `return` que interrompe a execução da função e retorna ao programa principal. Quando o `return` é seguido de um objeto, a função tem como saída o valor desse objeto.

Exemplo 4.2.3. Vamos atualizar a versão de nosso código do Exemplo 4.2.1. Aqui, em vez de imprimir, a função `areaCirc(r)` tem como saída o valor computado da área da circunferência de raio `r`

```python
import math as m

# def fun
def areaCirc(r):
    area = m.pi * r**2
    return area

# entrada de dados
raio1 = float(input('Digite o raio da 1. circ.:\n'))
raio2 = float(input('Digite o raio da 2. circ.:\n'))

print(f'Circunferência de raio = {raio1}')
area1 = areaCirc(raio1)
print(f'\tárea = {area1}')

print(f'Circunferência de raio = {raio2}')
area2 = areaCirc(raio2)
print(f'\tárea = {area2}')
```

Funções podem retornar objetos de qualquer classe de dados. Quando queremos retornar mais de um objeto por vez, usualmente usamos um `tuple` como variável de saída.

Exemplo 4.2.4. O seguinte código, cria uma função para a computação das raízes de um polinômio de grau 2

$$p(x) = ax^2 + bx + c. \tag{4.3}$$

Código 4.1: raizes_v1.py

```python
import math as m

def raizes(a, b, c):
    '''
    Computa as raízes de
    p(x) = ax^2 + bx + c

```

```
    Entrada
    -------
    a : float
    Coeficiente do termo quadrático.
    Atenção! Deve ser diferente de zero.

    b : float
    Coeficiente do termo linear.

    c: float
    Coeficiente do termo constante.

    Saída
    -----
    x1 : float
    Uma raíz do polinômio.

    x2 : float
    Outra raíz do polinômio.
    Atenção! No caso de raiz dupla,
    x1 == x2.
    '''

    # auxiliares
    _2a = 2*a
    _b2a = -b/_2a

    # discriminante
    delta = b**2 - 4*a*c

    # raízes
    if (delta > 0):
        x1 = _b2a + m.sqrt(delta)/_2a
        x2 = _b2a - m.sqrt(delta)/_2a
        return x1, x2
    elif (delta < 0):
        img = m.sqrt(-delta)/_2a
        x1 = _b2a + img*1j
        return x1, x1.conjugate()
    else:
```

```
        return _b2a, _b2a
```

Verifique!

4.2.2 Capturando Exceções

Exceções são classes de erros encontrados durante a execução de um código. Ao encontrar uma exceção, a execução do código Python é imediatamente interrompida e uma mensagem é impressa indicando a classe do erro e a linha do código em ocorreu. Por exemplo, ao chamarmos `raizes(0, 1, 2)` definida no Código 4.1, obtemos uma exceção da classe `ZeroDivisionError`.

```
>>> raizes(0,1,2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/pkonzen/GitHub/notas/src/AlgoritmosProgramacaoI/cap_fun/
    _b2a = -b/_2a
ZeroDivisionError: division by zero
```

Podemos controlar as exceções com a instrução `try-except`. Sua sintaxe é

```
try:
    comando1
except:
    comando2
```

Ou seja, o código tenta executar o `comando1`, caso ele gere uma exceção, o `comando2` é executado. A lista de exceções predefinidas na linguagem pode ser consultada em

https://docs.python.org/3/library/exceptions.html

Exemplo 4.2.5. No Código 4.1, podemos evitar e avisar a(o) usuária(o) da divisão por zero no caso de $a = 0$.

Código 4.2: raizes_v2.py

```
import math as m

def raizes(a, b, c):
    '''
    Computa as raízes de
    p(x) = ax^2 + bx + c
```

```

    Entrada
    -------
    a : float
    Coeficiente do termo quadrático.
    Atenção! Deve ser diferente de zero.

    b : float
    Coeficiente do termo linear.

    c: float
    Coeficiente do termo constante.

    Saída
    -----
    x1 : float
    Uma raíz do polinômio.

    x2 : float
    Outra raíz do polinômio.
    Atenção! No caso de raiz dupla,
    x1 == x2.
    '''

    # auxiliares
    _2a = 2*a

    try:
        _b2a = -b/_2a
    except ZeroDivisionError:
        raise ZeroDivisionError('a deve ser != 0.')

    # discriminante
    delta = b**2 - 4*a*c

    # raízes
    if (delta > 0):
        x1 = _b2a + m.sqrt(delta)/_2a
        x2 = _b2a - m.sqrt(delta)/_2a
        return x1, x2
```

```
    elif (delta < 0):
        img = m.sqrt(-delta)/_2a
        x1 = _b2a + img*1j
        return x1, x1.conjugate()
    else:
        return _b2a, _b2a
```

Observação 4.2.2. Nos casos gerais, pode-se utilizar a seguinte sintaxe:

```
try:
    comando1
except:
    raise Exception('msg')
```

4.2.3 Criando um Módulo

Para criar um módulo em Python, basta escrever um código `foo.py` com as funções e constantes que quisermos. Depois, podemos importá-lo em outro código com a instrução `import`.

Exemplo 4.2.6. Considere um retângulo de lados a e b. Na sequência, temos um módulo com algumas funções.

Código 4.3: retangulo.py

```
'''
Módulo com funcionalidades sobre
retângulos.
'''

import math as m

def perimetro(a, b):
    '''
    Perímetro de um retângulo de
    lados a e b.

    Entrada
    -------
    a : float
    Comprimento de um dos lados.
```

```

    b : float
    Comprimento de outro dos lados.

    Saída
    -----
    p : float
    Perímetro do retângulo.
    '''

    p = 2*a + 2*b
    return p

def area(a, b):
    '''
    Área de um retângulo de
    lados a e b.

    Entrada
    -------
    a : float
    Comprimento de um dos lados.

    b : float
    Comprimento de outro dos lados.

    Saída
    -----
    area : float
    Área do retângulo.
    '''

    area = a*b
    return area

def diagonal(a, b):
    '''
    Comprimento da diagonal de
    um retângulo de lados a e b.

```

```
    Entrada
    -------
    a : float
    Comprimento de um dos lados.

    b : float
    Comprimento de outro dos lados.

    Saída
    -----
    diag : float
    Diagonal do retângulo.
    '''

    diag = m.sqrt(a**2 + b**2)
    return diag
```

Agora, usamos nosso módulo `perimetro.py` em um outro código que fornece informações sobre o retângulo de lados a e b informados por usuária(o).

```
import retangulo as rect

a = float(input('Lado a: '))
b = float(input('Lado b: '))

diag = rect.diagonal(a, b)
print(f'diagonal = {diag}')

perim = rect.perimetro(a, b)
print(f'perímetro = {perim}')

area = rect.area(a, b)
print(f'área = {area}')
```

4.2.4 Exercícios

E.4.2.1. Defina uma função que recebe os catetos a e b de um triângulo retângulo e retorne o valor de sua hipotenusa. Use-a para escrever um código em que a(o) usuária(o) informa os catetos e obtenha o valor da hipotenusa.

E.4.2.2. Defina uma função que recebe os lados a, b e c de um triângulo qualquer e retorne o valor de sua área. Use-a para escrever um código em que a(o) usuária(o) informa os lados do triângulo e obtenha o valor da área.

E.4.2.3. Defina uma função que retorna um número randômico ímpar entre 1 e 51 (incluídos). Use-a para escrever um código em que:

1. A(o) usuária(o) informa um número inteiro $n \geq 1$.

2. Cria-se uma lista de n números randômicos ímpares entre 1 e 51 (incluídos).

3. Computa-se e imprime-se a média dos n números.

E.4.2.4. Desenvolva um código para computar a raiz de uma função afim

$$f(x) = ax + b, \tag{4.4}$$

com coeficientes a e b informados por usuária(o). Use instruções `try-except` para monitorar as exceções em que a usuária informe números inválidos ou $a = 0$.

E.4.2.5. Considere polinômios de segundo grau

$$p(x) = ax^2 + bx + c. \tag{4.5}$$

Desenvolva um módulo com as seguintes funções:

a) `intercepta_y()`: função que retorna o ponto de interseção do gráfico de $y = p(x)$ com o eixo das ordenadas[2].

b) `raizes()`: função que retorna as raízes de p.

c) `vertice()`: função que retorna o vértice do gráfico de $y = p(x)$.

Então, use seu módulo em um código em que a(o) usuária(o) informa os coeficientes a, b e c e obtém informações sobre as raízes, o ponto de interseção com o eixo y e o vértice de p.

[2] Eixo y.

Respostas

E.4.2.2. Dica: Use o https://pt.wikipedia.org/wiki/Teorema_de_Her%C3%A3o.

E.4.2.3.

```
import random

def randImpar(m=51):
    '''
    Retorna um número randômico
    ímpar entre 1 e m (incluídos).

    Entrada
    -------
    m : int
    Maior inteiro ímpar que pode ser
    gerado. Padrão: m = 51.

    Saída
    -----
    n : int
    Número randômico ímpar.
    '''
    n = random.randint(0, m-1)
    if (n % 2 != 0):
        return n
    else:
        return n+1

# entrada de dados
n = int(input('Digite o tamanho da lista:\n'))

# gera a lista
lista = [0]*n
for i in range(n):
    lista[i] = randImpar()

# calcula a média
soma = sum(lista)
```

```
media = soma/len(lista)

# imprime o resultados
print(f'média = {media}')
```

E.4.2.4.

```
import math as m

def raizFunAfim(a, b):
    '''
    Computa a raiz de
    f(x) = ax + b

    Entrada
    -------
    a : float
    Coeficiente angular.

    b : float
    Coeficiente linear.

    Saída
    -----
    x : float
    Raiz de f(x).
    '''

    try:
        x = -b/a
    except ZeroDivisionError:
        raise ZeroDivisionError('coef. angular deve ser != 0.')

    return x

# entrada de dados
try:
    a = float(input('Coef. angular: '))
except ValueError:
    raise ValueError('Número inválido.')
```

```python

try:
    b = float(input('Coef. linear: '))
except ValueError:
    raise ValueError('Número inválido.')

# raiz
raiz = raizFunAfim(a, b)

# imprime
print(f'raiz = {raiz}')
```

4.3 Passagem de Parâmetros

Uma função pode ter parâmetros de entada, são as varáveis de entrada que são usadas para que ela receba dados no momento em que é chamada. Esta estrutura de passar dados para uma função é chamado de passagem de parâmetros. Os parâmetros de entrada são alocados como novas variáveis no chamamento da função e ficam livres ao término de sua execução.

Exemplo 4.3.1. Consideramos o seguinte código:

```python
1 def fun(n):
2     print('Na função:')
3     print(f'\tn = {n}, id = {id(n)}')
4     n = n + 1
5     print(f'\tn = {n}, id = {id(n)}')
6     return n
7 
8 n = 1
9 print(f'n = {n}, id = {id(n)}')
10 
11 m = fun(n)
12 print(f'n = {n}, id = {id(n)}')
13 print(f'm = {m}, id = {id(m)}')
```

Na linha 10, o identificador `n` é criado com valor 1. Na linha 13, a função `fun` é chamada, um novo identificador `n` é criado apontando para o mesmo valor. No escopo da função (linhas 4-8), apenas este novo `n` é afetado. Ao término da função, este é liberado e o programa principal segue com o identificador `n` original.

4.3.1 Variáveis Globais e Locais

Variáveis globais são aquelas que podem ser acessadas por subprogramas (como funções) e locais são aquelas que existem somente dentro do escopo de um subprograma.

Variáveis Locais

Variáveis criadas dentro do escopo de uma função (incluindo-se os parâmetros de entrada) são locais, i.e. só existem durante a execução da função.

Exemplo 4.3.2. Consideramos o seguinte código:

```
def fun(x):
    y = 2*x - 1
    return y

z = fun(2)

try:
    print(f'id(y) = {id(y)}')
except:
    print(f'y não está definida.')
```

Ao executarmos, imprime-se a mensagem “y não está definida”. Isto ocorre, pois `y` é variável local na função `fun`, é criada e liberada durante sua execução.

Variáveis Globais

Variáveis definidas no programa principal são globais, i.e. podem ser acessadas[3] no escopo de funções, mesmo que não sejam passadas por parâmetros.

Exemplo 4.3.3. Consideramos o seguinte código:

```
x = 3

def fun():
    y = 2*x - 1
    return y

y = fun()
print(f'x = {x}, y = {y}')
```

[3] Em modo somente leitura.

A variável `x` é global, i.e. é acessível na função `fun`. Execute o código e verifique o valor impresso.

A instrução `global` permite que variáveis globais possam ser modificadas dentro do escopo de funções.

Exemplo 4.3.4. Consideramos o seguinte código:

```
def fun():
    global x
    x = x - 1
    y = 2*x - 1
    return y

x = 3
y = fun()
print(f'x = {x}, y = {y}')
```

4.3.2 Parâmetros com Valor Padrão

Funções podem ter parâmetros com valor padrão, i.e. no caso que a função ser chamada sem esses parâmetros, eles assumem o valor predefinido na declaração da função.

Exemplo 4.3.5. O seguinte código, imprime uma lista com a Sequência de Fibonacci[4]. Por padrão, apenas os cinco primeiros números da sequência são retornados pela função declarada.

```
def bigollo(n=5):
    fibo = [1]*n
    for i in range(2,n):
        fibo[i] = sum(fibo[i-2:i])
    return fibo

print(bigollo())
```

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

4.3.3 Vários Parâmteros

Uma função pode ter vários parâmetros de entrada. A ordem em que os parâmetros são definidos na função devem ser seguidos na passagem de valores. Por exemplo, consideramos a função

```python
def fun(x, y):
    print(f'x = {x}')
    print(f'y = {y}')
```

Ao chamá-la, devemos passar os valores dos parâmetros `x` e `y` na mesma ordem em que aparecem na definição da função. Por exemplo,

```python
>>> fun(1,2)
x = 1
y = 2
```

Podemos superar esta restrição, passando os parâmetros de forma explícita. Por exemplo,

```python
>>> fun(y=2, x=1)
x = 1
y = 2
```

4.3.4 Parâmetros Arbitrários

hlUma função pode ter uma quantidade arbitrária de parâmetros.

`Tuple` como Parâmetro Arbitrário

Usa-se a seguinte sintaxe para passar parâmetros arbitrários com `tuples`:

```python
def fun(*args):
    pass
```

Exemplo 4.3.6. Os seguinte código implementa funções para a computação de raízes (reais) de polinômios de até grau 1 e de grau 2.

```python
import math as m

def raizPoli1(a, b):
    '''
    ax + b = 0
```

```python
    '''
    return {-b/a}

def raizPoli2(a, b, c):
    '''
    ax^2 + bx + c = 0
    '''
    delta = b**2 - 4*a*c
    x1 = (-b - m.sqrt(delta))/(2*a)
    x2 = (-b + m.sqrt(delta))/(2*a)
    return {x1, x2}

def raizPoli12(*coefs):
    if (len(coefs) == 2):
        return raizPoli1(coefs[0], coefs[1])
    elif (len(coefs) == 3):
        return raizPoli2(coefs[0], coefs[1], coefs[2])
    else:
        raise Exception('Polinômio inválido.')

print('x - 2 = 0')
print(f'x = {raizPoli12(1,-2)}')

print('2x^2 - 3x + 1 = 0')
print(f'x = {raizPoli12(2, -3, 1)}')
```

Dicionários como Parâmetros Arbitrários

Usa-se a seguinte sintaxe para passar parâmetros arbitrários com `dicts`:

```python
def fun(**kwargs):
    pass
```

Exemplo 4.3.7. Os seguinte código implementa funções para a computação de raízes (reais) de polinômios de até grau 1 e de grau 2.

```python
import math as m

def raizPoli1(a, b):
    '''
    ax + b = 0
```

```
    '''
    return {-b/a}

def raizPoli2(a, b, c):
    '''
    ax^2 + bx + c = 0
    '''
    delta = b**2 - 4*a*c
    x1 = (-b - m.sqrt(delta))/(2*a)
    x2 = (-b + m.sqrt(delta))/(2*a)
    return {x1, x2}

def raizPoli12(**coefs):
    if (len(coefs) == 2):
        return raizPoli1(coefs['a'], coefs['b'])
    elif (len(coefs) == 3):
        return raizPoli2(coefs['a'], coefs['b'], coefs['c'])
    else:
        raise Exception('Polinômio inválido.')

print('x - 2 = 0')
print(f'x = {raizPoli12(a=1, b=-2)}')

print('2x^2 - 3x + 1 = 0')
print(f'x = {raizPoli12(a=2, b=-3, c=1)}')
```

4.3.5 Exercícios

E.4.3.1. Considere o seguinte código:

```
x = 1
def fun(x):
    print(x)
fun(2)
```

Sem executá-lo, diga qual seria o valor impresso no caso do código ser rodado. Justifique sua resposta.

E.4.3.2. Considere o seguinte código:

```
def fun(x):
```

```
    global x
    x = x - 1
```

Ao executá-lo, Python gera um erro de sintaxe. Qual é esse erro e por quê ele ocorre?

E.4.3.3. Considere o seguinte código:

```
y = 1
def fun(x=y):
    y = 2
    print(x)
fun()
```

Sem executá-lo, diga qual seria o valor impresso no caso de o código ser rodado. Justifique sua resposta.

E.4.3.4. Defina uma função Python que retorna uma lista com os termos da Progressão Aritmética (P.A.) $a_i = a_{i-1} + r$, $i = 0, 1, 2, \ldots, n$. Como parâmetros de entrada, tenha a_0 (termo inicial), r (razão da P.A.) e, por padrão, $n = 5$ (número de termos a serem computados).

E.4.3.5. Desenvolva uma função que retorna a lista de números primos entre n e m, $m \geq n$. Caso n ou m não sejam fornecidos, a função deve usar $n = 1$ e $m = 29$ como padrão.

E.4.3.6. Desenvolva uma função que verifica se um ponto pertence a um dado disco

$$(x - a)^2 + (y - b)^2 \leq r^2. \tag{4.6}$$

Crie-a de forma que ela possa receber uma quantidade arbitrária de pontos para serem verificados. Os parâmetros do disco não sejam informados, ela deve usar, como padrão, o disco unitário com centro na origem.

Respostas

E.4.3.1. 2

E.4.3.2. Como parâmetro, `x` é variável local, mas está definida como global dentro do escopo da função. Isto causa uma ambiguidade que não é permitida em programas de computadores.

E.4.3.3. 1

E.4.3.4.

```
def progAritm(a0, r, n=5):
    return [a0 + i*r for i in range(n+1)]
```

E.4.3.5.

```
def EhPrimo(n):
    info = True
    for i in range(2,n//2+1):
        if (n % i == 0):
            info = False
            break
    return info

def primos(n=1, m=29):
    lista = []
    for x in range(n, m+1):
        if EhPrimo(x):
            lista.append(x)
    return lista
```

E.4.3.6. def inDisk(*pts, a=0, b=0, r=1): for pt in pts: if ((pt[0]-a)**2 + (pt[1]-b)**2 <= r**2): print(f'(pt[0], pt[1]) pertence ao disco.') else: print(f'(pt[0], pt[1]) não pertence ao disco.')

Capítulo 5

Arranjos e Matrizes

Um arranjo é uma coleção de objetos (todos de um mesmo tipo) em que os elementos são organizados por eixos. É a estrutura de dados mais utilizada para a alocação de vetores e matrizes, fundamentais na computação matricial.

5.1 Arranjos

Um arranjo (em inglês, *array*) é uma coleção de objetos (todos do mesmo tipo) em que os elementos são organizados por eixos. Nesta seção, vamos nos restringir a **arranjos unidimensionais** (de apenas um eixo). Esta é a estrutura computacionais usualmente utilizada para a alocação de vetores.

NumPy é uma biblioteca Python que fornece suporte para a alocação e manipulação de arranjos. Usualmente, a biblioteca é importada como segue

```
import numpy as np
```

Na sequência, vamos assumir que o NumPy já está importado como acima.

5.1.1 Alocação de Arranjos

Na linguagem, a alocação de um arranjo pode ser feita com o método `np.array(list)`. Como parâmetro de entrada, recebe uma `list` contendo os elementos do arranjo. Por exemplo,

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

```
>>> type(v)
<class 'numpy.ndarray'>
```

aloca o arranjo de números inteiros `v`. Embora arranjos não sejam vetores, a modelagem computacional de vetores usualmente é feita utilizando-se `arrays`. Por exemplo, em um código Python, o vetor

$$\boldsymbol{v} = (-2, 1, 3) \tag{5.1}$$

pode ser alocado usando-se o `array v` acima.

O tipo dos dados de um `array` é definido na sua criação. Pode ser feita de forma automática ou explícita pela propriedade `dtype`. Por exemplo,

```
>>> v = np.array([-2, 1, 3])
>>> v.dtype
dtype('int64')
>>> v = np.array([-2., 1, 3])
>>> v.dtype
dtype('float64')
>>> v = np.array([-2, 1, 3], dtype='float')
>>> v.dtype
dtype('float64')
```

Exemplo 5.1.1. Aloque o vetor

$$\boldsymbol{v} = (\pi, 1, e) \tag{5.2}$$

como um `array` do NumPy.

```
>>> import numpy as np
>>> v = np.array([np.pi, 1, np.e])
>>> v
array([3.14159265, 1.        , 2.71828183])
```

O NumPy conta com métodos úteis para a **inicialização** de `arrays`:

- `np.zeros()` **arranjo de elementos nulos**

```
>>> np.zeros(3)
array([0., 0., 0.])
```

- `np.ones()` **arranjo de elementos iguais a um**

```
>>> np.ones(2, dtype='int')
array([1, 1])
```

- `np.empty()` **arranjo de elementos não predefinidos**

```
>>> np.empty(3)
array([4.64404327e-310, 0.00000000e+000, 6.93315702e-310])
```

- `np.linspace(start, stop, num=50)` **arranjo de elementos uniformemente espaçados**

```
>>> np.linspace(0, 1, 5)
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

5.1.2 Indexação e Fatiamento

Um `array` é uma coleção de objetos mutável, ordenada e indexada. Indexação e fatiamento podem ser feitos da mesma forma que para `tuples` e `lists`. Por exemplo,

```
>>> v = np.array([-1, 1, 2, 0, 3])
>>> v[0]
-1
>>> v[-1]
3
>>> v[1:4]
array([1, 2, 0])
>>> v[::-1]
array([ 3,  0,  2,  1, -1])
>>> v[3] = 4
>>> v
array([-1,  1,  2,  4,  3])
```

5.1.3 Reordenamento dos Elementos

Em programação, o reordenamento (em inglês, *sorting*) de elementos de uma sequência ordenada de números (`array`, `tuple`, `list`, etc.) consiste em alterar a sequência de forma que os elementos sejam organizados do menor para o mair valor. Na sequência, vamos estudar alguns métodos para isso.

Método Bolha

Dado um `array`[1], o método bolha consiste em percorrer o arranjo e permutar dois elementos consecutivos de forma que o segundo seja sempre maior que o primeiro. Uma vez que percorrermos o arranjo, teremos garantido que o maior valor estará na última posição do arranjo e os demais elementos ainda poderão estar desordenados. Então, percorremos o arranjo novamente, permutando elementos dois-a-dois conforme a ordem desejada, o que trará o segundo maior elemento para a penúltima posição. Ou seja, para um arranjo com n elementos, temos garantido o reordenamento de todos os elementos após $n-1$ repetições desse algoritmo.

Exemplo 5.1.2. Na sequência, implementamos o Método Bolha para o reordenamento de arranjos e aplicamos para

$$\boldsymbol{v} = (-1, 1, 0, 4, 3). \tag{5.3}$$

Código 5.1: bubbleSort_v1.py

```python
import numpy as np

def bubbleSort(arr):
    arr = arr.copy()
    n = len(arr)
    for k in range(n-1):
        for i in range(n-k-1):
            if (arr[i] > arr[i+1]):
                arr[i], arr[i+1] = arr[i+1], arr[i]
    return arr

v = np.array([-1,1,0,4,3])
w = bubbleSort(v)
print(w)
```

Observação 5.1.1. Em geral, para um arranjo de n elementos, o Método Bolha requer $n-1$ repetições para completar o ordenamento. Entretanto, dependendo do caso, o ordenamento dos elementos pode terminar em menos passos.

[1]Ou, um `tuple`, `list`, etc..

Exemplo 5.1.3. Na sequência, implementamos uma nova versão do Método Bolha para o reordenamento de arranjos. Esta versão verifica se há elementos fora de ordem e, caso não haja, interrompe o algoritmo. Como exemplo, aplicamos para

$$\boldsymbol{v} = (-1, 1, 0, 4, 3). \tag{5.4}$$

Código 5.2: bubbleSort_v2.py

```
import numpy as np

def bubbleSort(arr):
    arr = arr.copy()
    n = len(arr)
    for k in range(n-1):
        noUpdated = True
        for i in range(n-k-1):
            if (arr[i] > arr[i+1]):
                arr[i], arr[i+1] = arr[i+1], arr[i]
                noUpdated = False
        if (noUpdated):
            break
    return arr

v = np.array([-1,1,0,4,3])
w = bubbleSort(v)
```

Observação 5.1.2. (Métodos de Ordenamento.) Existem vários métodos para o ordenamento de uma sequência. O Método Bolha é um dos mais simples, mas também, em geral, menos eficiente. O NumPy tem disponível a função `np.sort(arr)` para o reordenamento de elementos. Também bastante útil, é a função `np.argsort(arr)`, que retorna os índices que reordenam os elementos.

5.1.4 Operações Elemento-a-Elemento

No NumPy, temos os **operadores aritméticos elemento-a-elemento** (em ordem de precedência)

- `**`

```
>>> v = np.array([-2., 1, 3])
>>> w = np.array([1., -1, 2])
>>> v ** w
array([-2.,  1.,  9.])
```

- `*`, `/`, `//`, `%`

```
>>> v * w
array([-2., -1.,  6.])
>>> v / w
array([-2. , -1. ,  1.5])
>>> v // w
array([-2., -1.,  1.])
>>> v % w
array([ 0., -0.,  1.])
```

- `+`, `-`

```
>>> v + w
array([-1.,  0.,  5.])
>>> v - w
array([-3.,  2.,  1.])
```

Exemplo 5.1.4. Vamos usar `arrays` para alocar os vetores

$$\boldsymbol{v} = (1., 0, -2), \tag{5.5}$$
$$\boldsymbol{w} = (2., -1, 3). \tag{5.6}$$

Então, computamos o produto interno

$$\boldsymbol{v} \cdot \boldsymbol{w} := v_1 w_1 + v_2 w_2 + v_3 w_3 \tag{5.7a}$$
$$= 1 \cdot 2 + 0 \cdot (-1) + (-2) \cdot 3 \tag{5.7b}$$
$$= -4. \tag{5.7c}$$

```
import numpy as np
# vetores
v = np.array([1., 0, -2])
w = np.array([2., -1, 3])
# produto interno
vdw = np.sum(v*w)
```

Observação 5.1.3. (Concatenação de Arranjos.) No NumPy, a concatenação de arranjos pode ser feita com a função np.concatenate(). Por exemplo,

```
>>> v = np.array([1,2])
>>> w = np.array([3,4])
>>> np.concatenate((v,w))
array([1, 2, 3, 4])
```

5.1.5 Exercícios

E.5.1.1. Aloque os seguintes vetores como `array` do NumPy:

a) $\boldsymbol{a} = (0, -2, 4)$

b) $\boldsymbol{b} = (0.1, -2.7, 4.5)$

c) $\boldsymbol{c} = (e, \ln(2), \pi)$

E.5.1.2. Considere o seguinte `array`

```
>>> v = np.array([4, -1, 1, -2, 3]).
```

Sem implementar, escreva os arranjos derivados:

a) `v[1]`

b) `v[1:4]`

c) `v[:3]`

d) `v[1:]`

e) `v[1:4:2]`

f) `v[-2:-5:-1]`

g) `v[::-2]`

Então, verifique seus resultados implementando-os.

E.5.1.3. Desenvolva uma função `argBubbleSort(arr)`, i.e. uma função que retorna os índices que reordenam os elementos do arranjo `arr` em ordem crescente. Teste seu código para o ordenamento de diversos arranjos e compare os resultados com a aplicação da função `np.argsort(arr)`.

E.5.1.4. Desenvolva um Método Bolha para o reordenamento dos elementos de um dado arranjo em ordem decrescente. Teste seu código para o reordenamento de diversos arranjos. Como pode-se usar a função `np.sort(arr)` para obter os mesmos resultados?

E.5.1.5. Desenvolva uma função `argBubbleSort(arr, emOrdem)`, i.e. uma função que retorna os índices que reordenam os elementos do arranjo `arr` na ordem definida pela função `emOrdem`. Teste seu código para o ordenamento de diversos arranjos, tanto em ordem crescente como em ordem decrescente. Como pode-se obter os mesmos resultados usando-se a função `np.sort(arr)`?

E.5.1.6. Crie uma função `media(arr)` que returna o valor médio do arranjo de números `arr`. Teste seu código para diferentes arranjos e compare os resultados com o da função `np.mean(arr)`.

E.5.1.7. Desenvolva uma função que retorna o ângulo entre dois vetores $\boldsymbol{v}$ e $\boldsymbol{w}$ dados.

Respostas

E.5.1.1.

```
>>> import numpy as np
>>> a = np.array([0, -2, 4])
>>> b = np.array([0.1, -2.7, 4.5])
>>> c = np.array([np.e, np.log(2), np.pi])
```

E.5.1.2. Dica: consulte a Subseção 5.1.2.

E.5.1.3.

```
import numpy as np

def argBubbleSort(arr):
    n = len(arr)
    ind = np.arange(n)
    for k in range(n-1):
        noUpdated = True
        for i in range(n-k-1):
            if (arr[ind[i]] > arr[ind[i+1]]):
```

```
                ind[i], ind[i+1] = ind[i+1], ind[i]
                noUpdated = False
        if (noUpdated):
            break
    return ind
```

E.5.1.4.

```
import numpy as np

def emOrdem(x, y):
    return x < y

def bubbleSort(arr, emOrdem=emOrdem):
    arr = arr.copy()
    n = len(arr)
    for k in range(n-1):
        noUpdated = True
        for i in range(n-k-1):
            if not(emOrdem(arr[i], arr[i+1])):
                arr[i], arr[i+1] = arr[i+1], arr[i]
                noUpdated = False
        if (noUpdated):
            break
    return arr
```

E.5.1.5.

```
import numpy as np

def argBubbleSort(arr, emOrdem=emOrdem):
    n = len(arr)
    ind = np.arange(n)
    for k in range(n-1):
        noUpdated = True
        for i in range(n-k-1):
            if not(emOrdem(arr[ind[i]], arr[ind[i+1]])):
                ind[i], ind[i+1] = ind[i+1], ind[i]
                noUpdated = False
        if (noUpdated):
            break
```

```
    return ind
```

E.5.1.6.

```
import numpy as np

def media(arr):
    return np.sum(arr)/len(arr)
```

E.5.1.7.

```
import numpy as np

def dot(v, w):
    return np.sum(v*w)

def angulo(v, w):
    # norma de v
    norm_v = np.sqrt(dot(v,v))
    # norma de w
    norm_w = np.sqrt(dot(w,w))
    # cos(theta)
    cosTheta = dot(v,w)/(norm_v*norm_w)
    # theta
    theta = np.acos(cosTheta)
    return theta
```

5.2 Vetores

O uso de `arrays` é uma das formas mais adequadas para fazermos a modelagem computacional de **vetores**. Entretanto, devemos ficar atentos que vetores e arranjos não são equivalentes. Embora, a soma/subtração e multiplicação por escalar são similares, a multiplicação e potenciação envolvendo vetores não estão definidas, mas para arranjos são operações elemento-a-elemento.

No que segue, vamos assumir que a biblioteca NumPy está importada, i.e.

```
>>> import numpy as np
```

Exemplo 5.2.1. Podemos alocar os vetores

$$\boldsymbol{v} = (1, 0, -2), \tag{5.8}$$
$$\boldsymbol{w} = (2, -1, 3), \tag{5.9}$$

como `arrays` do NumPy

```
>>> v = np.array([1, 0, -2])
>>> w = np.array([2, -1, 3])
```

A soma dos vetores é uma operação elemento-a-elemento

$$\boldsymbol{v} + \boldsymbol{w} = (1, 0, -2) + (2, -1, 3) \tag{5.10a}$$
$$= (1 + 2, 0 + (-1), -2 + 3) \tag{5.10b}$$
$$= (3, -1, 1) \tag{5.10c}$$

e a dos `arrays` é equivalente

```
>>> v+w
array([ 3, -1,  1])
```

A subtração dos vetores também é uma operação elemento-a-elemento

$$\boldsymbol{v} - \boldsymbol{w} = (1, 0, -2) - (2, -1, 3) \tag{5.11a}$$
$$= (1 - 2, 0 - (-1), -2 - 3) \tag{5.11b}$$
$$= (-1, 1, -5) \tag{5.11c}$$

e a dos `arrays` é equivalente

```
>>> v-w
array([ -1, 1,  -5])
```

Ainda, a multiplicação por escalar

$$2\boldsymbol{v} = 2(1, 0, -2) \tag{5.12a}$$
$$= (2 \cdot 1, 2 \cdot 0, 2 \cdot (-2)) \tag{5.12b}$$
$$= (2, 0, -4) \tag{5.12c}$$

também é feita elemento-a-elemento, assim como com `arrays`

```
>>> 2*v
array([ 2,  0, -4])
```

Agora, para vetores, a multiplicação $\boldsymbol{v}\boldsymbol{w}$, divisão $\boldsymbol{v}/\boldsymbol{w}$, potenciação $\boldsymbol{v}^2$ não são operações definidas. Diferentemente, para arranjos são operações elemento-a-elemento

```
>>> v*w
array([ 2,  0, -6])
>>> v/w
array([ 0.5        , -0.         , -0.66666667])
>>> v**2
array([1, 0, 4])
```

5.2.1 Funções Vetoriais

Funções vetoriais $f : \mathbb{R}^n \to \mathbb{R}^n$ e funcionais $f : \mathbb{R}^n \to \mathbb{R}$ também podem ser adequadamente modeladas com o emprego de `arrays` do NumPy. A biblioteca também conta com várias funções matemáticas predefinidas, consulte

https://numpy.org/doc/stable/reference/routines.math.html

Exemplo 5.2.2. (Função Vetorial.) A implementação da função vetorial $\boldsymbol{f} : \mathbb{R}^3 \to \mathbb{R}^3$

$$\boldsymbol{f}(\boldsymbol{x}) = (x_1^2 + \sin(x_1), x_2^2 + \sin(x_2), x_3^2 + \sin(x_3)) \tag{5.13}$$

para $\boldsymbol{x} = (x_1, x_2, x_3) \in \mathbb{R}^3$, pode ser feita da seguinte forma

```
import numpy as np

def f(x):
    return x**2 + np.sin(x)

# exemplo
x = np.array([0, np.pi, np.pi/2])
print(f'y = {f(x)}')
```

Verifique!

5.2.2 Produto Interno

O **produto interno** (ou, produto escalar) é a operação entre vetores $\boldsymbol{v}, \boldsymbol{w} \in \mathbb{R}^n$ definida por

$$\boldsymbol{v} \cdot \boldsymbol{v} := v_1 w_1 + v_2 w_2 + \cdots + v_n w_n. \tag{5.14}$$

A função numpy.dot(v,w) computa o produto interno dos `arrays`.

Exemplo 5.2.3. Consideramos os vetores

$$\boldsymbol{v} = (1, 0, -2), \tag{5.15}$$

$$\boldsymbol{w} = (2, -1, 3). \tag{5.16}$$

O produto interno desses vetores é

$$\boldsymbol{v} \cdot \boldsymbol{w} = v_1 w_1 + v_2 w_2 + v_3 w_3 \tag{5.17a}$$

$$= 1 \cdot 2 + 0 \cdot (-1) + (-2) \cdot 3 \tag{5.17b}$$

$$= 2 + 0 - 6 = -4 \tag{5.17c}$$

Usando `arrays`, temos

```
>>> v = np.array([1, 0, -2])
>>> w = np.array([2, -1, 3])
>>> np.sum(v*w)
-4
>>> np.dot(v,w)
-4
```

5.2.3 Norma de Vetores

A **norma** L^2 de um vetor $\boldsymbol{v} \in \mathbb{R}^n$ é definida por

$$\|\boldsymbol{v}\| := \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2} \tag{5.18}$$

O módulo `numpy.linalg` de Álgebra Linear contém a função `numpy.linalg.norm(v)` para a computação da norma de `arrays`.

Exemplo 5.2.4. A norma do vetor $\boldsymbol{v} = (3, 0, -4)$ é

$$\|\boldsymbol{v}\| = \sqrt{v_1^2 + v_2^2 + v_3^2} \tag{5.19a}$$

$$= \sqrt{3^2 + 0^2 + (-4)^2} \tag{5.19b}$$

$$= \sqrt{9 + 0 + 16} \tag{5.19c}$$

$$= \sqrt{25} = 5. \tag{5.19d}$$

Usando o módulo `numpy.linalg`, obtemos

```
>>> import numpy as np
>>> import numpy.linalg as npla
>>> v = np.array([3, 0, -4])
>>> np.sqrt(np.dot(v,v))
5.0
>>> npla.norm(v)
5.0
```

5.2.4 Produto Vetorial

O **produto vetorial** entre dois vetores $\boldsymbol{v}, \boldsymbol{w} \in \mathbb{R}^3$ é definido por

$$\begin{align}
\boldsymbol{v} \times \boldsymbol{w} &:= \begin{vmatrix} \boldsymbol{i} & \boldsymbol{j} & \boldsymbol{k} \\ v_1 & v_2 & v_3 \\ w_1 & w_2 & w_3 \end{vmatrix} \tag{5.20a}\\
&= \begin{vmatrix} v_2 & v_3 \\ w_2 & w_3 \end{vmatrix} \boldsymbol{i} \tag{5.20b}\\
&- \begin{vmatrix} v_1 & v_3 \\ w_1 & w_3 \end{vmatrix} \boldsymbol{j} \tag{5.20c}\\
&+ \begin{vmatrix} v_1 & v_2 \\ w_1 & w_2 \end{vmatrix} \boldsymbol{k} \tag{5.20d}\\
& \tag{5.20e}
\end{align}$$

A função `numpy.cross(v,w)` computa o produto vetorial entre `arrays` (unidimensionais de 3 elementos).

Exemplo 5.2.5. O produto vetorial entre os vetores $\boldsymbol{v} = (1, -2, 1)$ e $\boldsymbol{w} = (0, 2, -1)$ é

$$\begin{align}
\boldsymbol{v} \times \boldsymbol{w} &= \begin{vmatrix} \boldsymbol{i} & \boldsymbol{j} & \boldsymbol{k} \\ 1 & -2 & 1 \\ 0 & 2 & -1 \end{vmatrix} \tag{5.21a}\\
&= 0\boldsymbol{i} + \boldsymbol{j} + 2\boldsymbol{k} \tag{5.21b}\\
&= (0, 1, 2). \tag{5.21c}
\end{align}$$

Com o NumPy, temos

```
>>> v = np.array([1, -2, 1])
>>> w = np.array([0, 2, -1])
>>> np.cross(v,w)
array([0, 1, 2])
```

5.2.5 Exercícios

E.5.2.1. Considere os seguintes vetores

$$\boldsymbol{u} = (2, -1, 1) \tag{5.22}$$
$$\boldsymbol{v} = (1, -3, 2) \tag{5.23}$$
$$\boldsymbol{w} = (-2, -1, -3) \tag{5.24}$$

Usando `arrays` do NumPy, compute:

a) $\boldsymbol{u} \cdot \boldsymbol{v}$

b) $\boldsymbol{u} \cdot (2\boldsymbol{v})$

c) $\boldsymbol{u} \cdot (\boldsymbol{w} + \boldsymbol{v})$

d) $\boldsymbol{v} \cdot (\boldsymbol{v} - 2\boldsymbol{u})$

E.5.2.2. Considere os seguintes vetores

$$\boldsymbol{u} = (2, -1, 1) \tag{5.25}$$
$$\boldsymbol{v} = (1, -3, 2) \tag{5.26}$$
$$\boldsymbol{w} = (-2, -1, -3) \tag{5.27}$$

Usando `arrays` do NumPy, compute:

a) $\|\boldsymbol{u}\|$

b) $\|\boldsymbol{u} + \boldsymbol{v}\|$

c) $|\boldsymbol{u} \cdot \boldsymbol{w}|$

E.5.2.3. Dados vetores $\boldsymbol{u}$ e $\boldsymbol{v}$, temos que

$$\boldsymbol{u} \cdot \boldsymbol{v} = \|\boldsymbol{u}\|\|\boldsymbol{v}\| \cos\theta, \tag{5.28}$$

onde θ é o ângulo entre esses vetores. Implemente uma função que recebe dois vetores e retorna o ângulo entre eles. Teste seu código para diferentes vetores.

E.5.2.4. A projeção ortogonal do vetor $\boldsymbol{u}$ na direção do vetor $\boldsymbol{v}$ é definida por

$$\operatorname{proj}_{\boldsymbol{v}} \boldsymbol{u} := \frac{\boldsymbol{u} \cdot \boldsymbol{v}}{\|\boldsymbol{v}\|^2}\boldsymbol{v}. \tag{5.29}$$

Implemente uma função que recebe dois vetores $\boldsymbol{u}$, $\boldsymbol{v}$ e retorne a projeção de $\boldsymbol{u}$ na direção de $\boldsymbol{v}$. Teste seu código para diferentes vetores.

E.5.2.5. Considere os vetores

$$\boldsymbol{u} = (2, -3, 1), \tag{5.30}$$
$$\boldsymbol{v} = (1, -2, -1). \tag{5.31}$$

Usando `arrays` do NumPy, compute os seguintes produtos vetoriais:

a) $\boldsymbol{u} \times \boldsymbol{v}$

b) $\boldsymbol{v} \times (2\boldsymbol{v})$

Respostas

E.5.2.1. Dica: use a função *np.dot*() e verifique as computações calculando os resultados esperados.

E.5.2.2. Dica: do módulo `numpy.linalg` use a função `npla.norm` e verifique as computações calculando os resultados esperados.

E.5.2.3.

```python
import numpy as np
import numpy.linalg as npla

```

```python
def angulo(v, w):
    # \|v\|
    norm_v = npla.norm(v)
    # \|w\|
    norm_w = npla.norm(w)
    # u.v
    vdw = np.dot(v, w)
    # cos \theta
    ct = norm_v*norm_w/udw
    return np.acos(ct)
```

E.5.2.4.

```python
import numpy as np
import numpy.linalg as npla

def proj(u, v):
    # \|v\|
    norm_v = npla.norm(v)
    # u.v
    udv = np.dot(u, v)
    # proj_v u
    proj_vu = udv/norm_v * v
    return proj_vu
```

E.5.2.5. Dica: use a função `numpy.cross()` e verifique as computações calculando os resultados esperados.

5.3 Arranjos Multidimensionais

Um arranjo no NumPy (`numpy.array`) é um tabelamento de elementos de um mesmo tipo. Os elementos são organizados por eixos indexados (em inglês, *axes*). Enquanto que nas seções anteriores nos restringimos a `arrays` unidimensionais (de apenas um eixo), aqui, vamos estudar a alocação e manipulação de arranjos de vários eixos.

5.3.1 Alocação, Indexação e Fatiamento

A **alocação** de um `numpy.array` com mais de um eixo pode ser feita usando-se de **listas encadeadas**. Por exemplo,

```
>>> import numpy as np
>>> a = np.array([[1,2,3],[4,5,6]])
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
```

cria o arranjo `a` de dois eixos, enquanto

```
>>> b = np.array([[[1,2],[3,4]],[[-1,-2],[-3,-4]]])
>>> b
array([[[ 1,  2],
        [ 3,  4]],

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

cria o arranjo `b` de três eixos. Para fazer um paralelo com a matemática, o arranjo `a` é similar (mas, não equivalente) a matriz $A = [a_{i,j}]_{i,j=1}^{2,3}$

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, \tag{5.32}$$

e o arranjo `b` é similar (mas, não equivalente) ao tensor $B = [b_{i,j,k}]_{i,j,k=1}^{2,2,2}$.

A propriedade `.shape` é um *tuple* contendo o tamanho de cada eixo. Por exemplo,

```
>>> a.shape
(2, 3)
>>> a.shape[0]
2
>>> a.shape[1]
3
```

informa que `a` tem dois eixos, o primeiro com tamanho 2 e o segundo com tamanho 3. Um paralelo com matrizes, dizemos que `a` tem duas linhas e três colunas. No caso do arranjo `b`, temos

```
>>> b.shape
(2, 2, 2)
>>> b.shape[2]
2
```

o que nos informa tratar-se de um `array` de três eixos, cada um com tamanho 2.

Os elementos em um arranjo são **indexados por eixos** e o **fatiamento** também pode ser feito por eixos. Por exemplo,

```
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
>>> a[1,0]
4
>>> a[0]
array([1, 2, 3])
>>> a[:,1]
array([2, 5])
>>> a[:,2:]
array([[3],
       [6]])
>>> a[1,::-1]
array([6, 5, 4])
```

No caso do arranjo `b` de três eixos, temos

```
>>> b
array([[[ 1,  2],
        [ 3,  4]],

       [[-1, -2],
        [-3, -4]]])
>>> b[1,1,0]
-3
>>> b[0,1]
array([3, 4])
>>> b[1,0,::-1]
array([-2, -1])
```

5.3.2 Inicialização

O NumPy conta com várias funções para a inicialização de `arrays`, algumas das mais usadas são:

- `np.zeros()` **inicialização com zeros**

```
>>> np.zeros((2,3))
array([[0., 0., 0.],
       [0., 0., 0.]])
```

- `np.ones()` **inicialização com uns**

```
>>> np.ones((2,3,2))
array([[[1., 1.],
        [1., 1.],
        [1., 1.]],

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

- `np.empty()` **inicialização com valor da memória**

```
>>> np.empty((2,1))
array([[5.73021895e-300],
       [6.95260453e-310]])
```

Observamos que o tamanho de cada eixo é passado por um `tuple`.

5.3.3 Manipulação

O NumPy contém várias funções para a manipulação de `arrays`. Algumas das mais usadas são:

- `arr.reshape()` : reformatação de um arranjo.

```
>>> a = np.array([[1,2,3],[4,5,6]])
>>> a.reshape(3,2)
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> a.reshape(-1)
array([1, 2, 3, 4, 5, 6])
```

- `arr.concatenate()` : concatena um `tuple` de arranjos.

```
>>> a = np.array([1,2,3])
>>> b = np.array([4,5,6])
>>> np.concatenate((a,b))
array([1, 2, 3, 4, 5, 6])
>>> a = a.reshape(1,-1)
>>> b = b.reshape(1,-1)
```

```
>>> np.concatenate((a,b))
array([[1, 2, 3],
       [4, 5, 6]])
>>> np.concatenate((a,b), axis=1)
array([[1, 2, 3, 4, 5, 6]])
```

5.3.4 Operações e Funções Elementares

De forma análoga a arranjos unidimensionais, as operações aritméticas e funções elementares são aplicadas elemento-a-elementos em um arranjo. Por exemplo,

```
>>> a = np.array([[0.,1/6],[1/4,1/3]])
>>> b = np.array([[0.,6],[4,3]])
>>> np.sin(np.pi*a*b)
array([[0.0000000e+00, 1.2246468e-16],
       [1.2246468e-16, 1.2246468e-16]])
```

Multiplicação Matriz-Vetor

Dada uma matriz $A = [a_{i,j}]_{i,j=1}^{n,m}$ e um vetor $\boldsymbol{x} = (x_i)_{i=1}^{m}$, a **multiplicação matriz-vetor** $A\boldsymbol{x}$ é definida por

$$A\boldsymbol{x} = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,m} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,m} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,m} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} \tag{5.33a}$$

$$= \begin{bmatrix} a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,m}x_m \\ a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,m}x_m \\ \cdots \\ a_{n,1}x_1 + a_{n,2}x_2 + \cdots + a_{n,m}x_m \end{bmatrix} \tag{5.33b}$$

Exemplo 5.3.1. Considere a matriz

$$A = \begin{bmatrix} 1 & -1 & 2 \\ 2 & 1 & 3 \\ 0 & 2 & 1 \end{bmatrix} \tag{5.34}$$

e o vetor $\boldsymbol{x} = (-1, 2, 1)$. A multiplicação matriz-vetor é

$$A\boldsymbol{x} = \begin{bmatrix} 1 & -1 & 2 \\ 2 & 1 & 3 \\ 0 & 2 & 1 \end{bmatrix} \begin{bmatrix} -1 \\ 2 \\ 1 \end{bmatrix} \tag{5.35a}$$

$$= (-1, 3, 5) \tag{5.35b}$$

```python
import numpy as np

def MatrizVetor(A, x):
    n,m = A.shape
    y = np.empty(n)
    for i in range(n):
        y[i] = 0.
        for j in range(m):
            y[i] += A[i,j]*x[j]
    return y

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

Multiplicação Matriz-Matriz

Dadas matrizes $A = [a_{i,j}]_{i,j=1}^{n,p}$ e $B = [b_{i,j}]_{i,j=1}^{p,m}$, a **multiplicação matriz-matriz** AB é a matriz $AB = C = [c_{i,j}]_{i,j=1}^{n,m}$ de elementos

$$c_{i,j} = \sum_{k=1}^{p} a_{i,k} \cdot b_{k,j}. \tag{5.36}$$

Exemplo 5.3.2. Consideramos as matrizes

$$A = \begin{bmatrix} 1 & -1 & 2 \\ 2 & 1 & 3 \\ 0 & 2 & 1 \end{bmatrix} \tag{5.37}$$

$$B = \begin{bmatrix} 2 & 0 \\ 1 & 2 \\ 0 & 2 \end{bmatrix} \tag{5.38}$$

A multiplicação matriz-matriz AB é

$$AB = \begin{bmatrix} 1 & -1 & 2 \\ 2 & 1 & 3 \\ 0 & 2 & 1 \end{bmatrix} \begin{bmatrix} 2 & 0 \\ 1 & 2 \\ 0 & 2 \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 5 & 8 \\ 2 & 6 \end{bmatrix} \tag{5.39a}$$

```
def MatrizMatriz(A, B):
    n,p = A.shape
    m = B.shape[1]
    C = np.empty((n,m))
    for i in range(n):
        for j in range(m):
            C[i,j] = 0.
            for k in range(p):
                C[i,j] += A[i,k]*B[k,j]
    return C

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

5.3.5 Exercícios

E.5.3.1. Aloque o arranjo que corresponde a matriz

$$A = \begin{bmatrix} -2 & 1 & -4 \\ 0 & -3 & 2 \\ -1 & 5 & -7 \\ 2 & 3 & 6 \end{bmatrix} \tag{5.40}$$

Sem implementar, forneça a saída das seguintes instruções:

a) `A[2,1]`

b) `A[0,2]`

c) `A[-2,-2]`

d) `A[3]`

e) `A[3:,:]`

f) `A[:,2]`

g) `A[:,1:2]`

E.5.3.2. Considere o arranjo

```
A = np.array([[[-1,2,0],
               [3,-2,1],
               [1,-4,2]],
              [[2,-1,0],
               [5,-2,0],
               [2,6,3]],
              [[1,-1,0],
               [7,-2,4],
               [2,-2,1]]])
```

Sem implementar, forneça a saída das seguintes instruções:

a) `A[2,0,1]`

b) `A[1,1,0]`

c) `A[2]`

d) `A[1,2]`

e) `A[1:,:2,2]`

E.5.3.3. Considere o arranjo

```
>>> a = np.array([[1,2],[5,8],[2,6]])
>>> a
array([[1, 2],
       [5, 8],
       [2, 6]])
```

Sem implementar, escreva os seguintes arranjos derivados:

a) `a.reshape(6)`

b) `a.reshape(2,3)`

c) `a.reshape(-1)`

d) `a.reshape(-1,3)`

e) `a.reshape(3,-1)`

f) `a.reshape(4,-1)`

E.5.3.4. Considere os arranjos

```
>>> a = np.array([1,2,3]).reshape(1,-1)
>>> b = np.array([4,5,6]).reshape(-1,1)
```

Sem implementar, escreva os seguintes arranjos derivados:

a) `np.concatenate((a,b.reshape(1,-1)))`

b) `np.concatenate((a.reshape(-1,1),b))`

c) `np.concatenate((a,b.reshape(1,-1)), axis=1)`

d) `np.concatenate((a.reshape(-1,1),b), axis=1)`

E.5.3.5. Implemente uma função que recebe uma matriz (representada por um `array`) e retorna a sua transposta. Teste seu código para diversas matrizes com diversos formatos.

E.5.3.6. Implemente uma função que compute a multiplicação vetor-matriz

$$\boldsymbol{y} = \boldsymbol{x}A \tag{5.41a}$$

$$:= \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix} \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,m} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,m} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,m} \end{bmatrix} \tag{5.41b}$$

onde, por definição, $\boldsymbol{y} = [y_j]_{j=1}^m$, com elementos

$$y_j = \sum_{k=1}^{n} x_k \cdot a_{k,j}. \tag{5.42}$$

Respostas

E.5.3.1. Dica: aloque o arranjo e implemente as instruções.

E.5.3.2. Dica: aloque o arranjo e implemente as instruções.

E.5.3.3. Dica: aloque o arranjo e implemente as instruções.

E.5.3.4. Dica: aloque o arranjo e implemente as instruções.

E.5.3.5.

```
import numpy as np

def Transposta(A):
    n,m = A.shape
    B = np.empty((m,n))
    for i in range(n):
        for j in range(m):
            B[j,i] = A[i,j]
    return B
```

E.5.3.6. Dica: a função `np.dot()` também computa a multiplicação vetor-matriz. Teste sua implementação para diferentes matriz e vetores de diferentes tamanhos.

5.4 Matrizes

Arranjos multidimensionais[2] fornecem uma boa estrutura para a representação de matrizes em computador. Uma matriz A, assim como um arranjo bidimensional, é uma coleção de valores organizados de forma retangular, por exemplo, a matriz $A = [a_{i,j}]_{i,j=1}^{n,m}$ tem a forma

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,m} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,m} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,m} \end{bmatrix} \tag{5.43}$$

Seus elementos $a_{i,j}$ são organizados por eixos, o eixo das linhas (`axis=0`) e o eixo das colunas (`axis=1`).

[2] Consulte a Seção 5.3

Exemplo 5.4.1. O sistema linear

$$2x_1 - x_2 + x_3 = -3 \quad (5.44a)$$
$$-x_1 + x_2 + 3x_3 = 6 \quad (5.44b)$$
$$x_1 + 3x_2 - 3x_3 = 2 \quad (5.44c)$$

pode ser escrito na seguinte forma matricial

$$A\boldsymbol{x} = \boldsymbol{b}, \quad (5.45)$$

onde $A = [a_{i,j}]_{i,j=1}^{3,3}$ é a matriz de coeficientes

$$A = \begin{bmatrix} 2 & -1 & 1 \\ -1 & 1 & 3 \\ 1 & 3 & -3 \end{bmatrix}, \quad (5.46)$$

o vetor dos termos constantes $\boldsymbol{b} = (b_1, b_2, b_3)$ é

$$\boldsymbol{b} = (-3, 6, 2), \quad (5.47)$$

enquanto que o vetor das incógnitas é $\boldsymbol{x} = (x_1, x_2, x_3)$. No seguinte código, usamos de `numpy.arrays` para alocamos a matriz dos coeficientes A e o vetor dos termos constantes $\boldsymbol{b}$.

```
import numpy as np
# matriz dos coefs
A = np.array([[2, -1, 1],
              [-1, 1, 3],
              [1, 3, -3]])
# vet termos consts
b = np.array([-3, 6, 2])
```

5.4.1 Operações Matriciais

Embora úteis para a representação de matrizes, arranjos bidimensionais não são equivalentes a matrizes. Em arranjos, as operações aritméticas elementares (`+`, `-`, `*`, `/`, etc.) são operações elemento-a-elemento, para matrizes a multiplicação não é assim calculada e a divisão não é definida.

Multiplicação Matricial

No NumPy, a multiplicação matricial está disponível com as funções `numpy.dot()`, `numpy.matmul()` e com o operador `@`[3].

Exemplo 5.4.2. Considerando as matrizes

$$A = \begin{bmatrix} 2 & -1 & 1 \\ -1 & 1 & 3 \\ 1 & 3 & -3 \end{bmatrix}, \tag{5.48}$$

$$B = \begin{bmatrix} 1 & -1 \\ 2 & 1 \\ 1 & 0 \end{bmatrix} \tag{5.49}$$

temos

$$AB = \begin{bmatrix} 1 & -3 \\ 4 & 2 \\ 4 & 2 \end{bmatrix} \tag{5.50}$$

Usando o NumPy, temos

```
import numpy as np

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

B = np.array([[1, -1],
              [2, 1],
              [1, 0]])

AB = A@B
print(f'AB =\n {AB}')

AB = np.matmul(A, B)
print(f'AB =\n {AB}')

AB = np.dot(A, B)
print(f'AB =\n {AB}')
```

[3] Oi, sumido! ;-)

Exemplo 5.4.3. Consideramos o sistema linear introduzido no Exemplo 5.4.1. Vamos verificar que sua solução é $x_1 = -1$, $x_2 = 2$ e $x_3 = 1$. Equivalentemente, temos que

$$A\boldsymbol{x} = \boldsymbol{b}, \tag{5.51}$$

com $\boldsymbol{x} = (-1, 2, 1)$. Isto é, se $\boldsymbol{x}$ é solução do sistema, então é nulo o resíduo $\boldsymbol{b} - A\boldsymbol{x}$, i.e.

$$\boldsymbol{b} - A\boldsymbol{x} = \boldsymbol{0}. \tag{5.52}$$

Ou equivalentemente, $\|\boldsymbol{b} - A\boldsymbol{x}\| = 0$.

```python
import numpy as np
import numpy.linalg as npla

# matriz dos coefs
A = np.array([[2, -1, 1],
              [-1, 1, 3],
              [1, 3, -3]])
# vet termos consts
b = np.array([-3, 6, 2])
# solucao
x = np.array([-1, 2, 1])

# verificacao
res = b - A@x
norm_res = npla.norm(res)
print(f'É solução? {np.isclose(norm_res, 0.)}')
```

Matriz Transposta

Por definição, a transposta de uma matriz $A = [a_{i,j}]_{i,j=1}^{n,m}$ é a matriz $A^T = [a_{j,i}]_{j,i=1}^{m,n}$, i.e. a matriz B obtida de A pela permutação de suas linhas com suas colunas. No NumPy, a transposta de um arranjo bidimensional pode ser calculado com a função `numpy.transpose()`, com o método `numpy.ndarray.transpose()` ou com o atributo `numpy.ndarray.T`.

Exemplo 5.4.4. Uma matriz é dita ser simétrica, quando $A = A^T$. Observamos que é simétrica a matriz

$$A = \begin{bmatrix} 2 & -1 & 1 \\ -1 & 1 & 3 \\ 1 & 3 & -3 \end{bmatrix}, \tag{5.53}$$

```python
import numpy as np
A = np.array([[2, -1, 1],
              [-1, 1, 3],
              [1, 3, -3]])

trans_A = np.transpose(A)
print(f'A^T =\n {trans_A}')

trans_A = A.transpose()
print(f'A^T =\n {trans_A}')

trans_A = A.T
print(f'A^T =\n {trans_A}')

print(f'É simétrica? {np.allclose(A, A.T)}')
```

Agora, não é simétrica a matriz

$$B = \begin{bmatrix} 1 & -1 \\ 2 & 1 \\ 1 & 0 \end{bmatrix}. \tag{5.54}$$

```python
import numpy as np
B = np.array([[1, -1],
              [2, 1],
              [1, 0]])
print(B.T)
```

Determinante

Por definição, o determinante de uma matriz $A = [a_{i,j}]_{i,j=1}^{n,n}$ é o escalar

$$\det(A) = \begin{vmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{vmatrix} \tag{5.55a}$$

$$:= \sum_{\sigma \in S_n} \operatorname{sign}(\sigma) a_{1,\sigma_1} a_{2,\sigma_2} \cdots a_{n,\sigma_n} \tag{5.55b}$$

onde S_n é o conjunto de todas as permutações de $1, 2, \ldots, n$ e $\operatorname{sign}(\sigma)$ é o sinal (ou assinatura) da permutação $\sigma \in S_n$. Para matrizes 2×2, temos

$$\det(A) = \begin{vmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{vmatrix} \tag{5.56a}$$
$$= a_{1,1}a_{2,2} - a_{1,2}a_{2,1}. \tag{5.56b}$$

Enquanto que no caso de matriz 3×3, temos[4]

$$\det(A) = \begin{vmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{vmatrix} \tag{5.57a}$$
$$= a_{1,1}a_{2,2}a_{3,3} \tag{5.57b}$$
$$+ a_{1,2}a_{2,3}a_{3,1} \tag{5.57c}$$
$$+ a_{1,3}a_{2,1}a_{3,2} \tag{5.57d}$$
$$- a_{1,3}a_{2,2}a_{3,1} \tag{5.57e}$$
$$- a_{1,1}a_{2,3}a_{3,2} \tag{5.57f}$$
$$- a_{1,2}a_{2,1}a_{3,3}. \tag{5.57g}$$

No NumPy, o determinante de um arranjo pode ser computado com a função `numpy.linalg.det()`.

Exemplo 5.4.5. O determinante

$$\det(A) = \begin{vmatrix} 2 & -1 & 1 \\ -1 & 1 & 3 \\ 1 & 3 & -3 \end{vmatrix} \tag{5.58a}$$
$$= -28 \tag{5.58b}$$

```python
import numpy as np
import numpy.linalg as npla
A = np.array([[2, -1, 1],
              [-1, 1, 3],
              [1, 3, -3]])

detA = npla.det(A)
print(f'det(A) = {detA}')
```

[4] Pela regra de Sarrus.

5.4.2 Aplicação: Método de Cramer

O **Método de Cramer**[5] usa de determinantes para o cálculo da solução de sistemas lineares. Dado um sistema linear $n \times n$

$$A\boldsymbol{x} = \boldsymbol{b} \tag{5.59}$$

denotamos a matriz dos coeficientes por

$$A = \begin{bmatrix} \boldsymbol{a}_1 & \boldsymbol{a}_2 & \cdots & \boldsymbol{a}_n \end{bmatrix}, \tag{5.60}$$

onde $\boldsymbol{a}_i$ denota a i-ésima coluna de A. Vamos denotar por A_i a matriz obtida de A substituindo $\boldsymbol{a}_i$ pelo vetor dos termos constantes $\boldsymbol{b}$, i.e.

$$A_i := \begin{bmatrix} \boldsymbol{a}_1 & \cdots & \boldsymbol{a}_{i-1} & \boldsymbol{b} & \boldsymbol{a}_{i+1} & \cdots & \boldsymbol{a}_n \end{bmatrix} \tag{5.61}$$

O método consiste em computar a solução $\boldsymbol{x} = (x_1, x_2, \dots, x_n)$ com

$$x_i = \frac{\det(A_i)}{\det(A)}, \tag{5.62}$$

para cada $i = 1, 2, \dots, n$.

Exemplo 5.4.6. Vamos resolver o sistema linear dado no Exercício 5.4.1. Sua forma matricial é

$$A\boldsymbol{x} = \boldsymbol{b} \tag{5.63}$$

com matriz dos coeficientes

$$A = \begin{bmatrix} 2 & -1 & 1 \\ -1 & 1 & 3 \\ 1 & 3 & -3 \end{bmatrix}, \tag{5.64}$$

e vetor dos termos constantes

$$\boldsymbol{b} = (-3,\ 6,\ 2). \tag{5.65}$$

Para aplicação do Método de Cramer, calculamos

$$\det(A) := \begin{vmatrix} 2 & -1 & 1 \\ -1 & 1 & 3 \\ 1 & 3 & -3 \end{vmatrix} \tag{5.66a}$$

[5] Gabriel Cramer, 1704 - 1752, matemático suíço. Fonte: Wikipédia.

$$= -28 \tag{5.66b}$$

e das matrizes auxiliares

$$\det(A_1) := \begin{vmatrix} -3 & -1 & 1 \\ 6 & 1 & 3 \\ 2 & 3 & -3 \end{vmatrix} \tag{5.67a}$$

$$= 28 \tag{5.67b}$$

$$\det(A_2) := \begin{vmatrix} 2 & -3 & 1 \\ -1 & 6 & 3 \\ 1 & 2 & -3 \end{vmatrix} \tag{5.68a}$$

$$= -56 \tag{5.68b}$$

$$\det(A_3) := \begin{vmatrix} 2 & -1 & -3 \\ -1 & 1 & 6 \\ 1 & 3 & 2 \end{vmatrix} \tag{5.69a}$$

$$= -28 \tag{5.69b}$$

Com isso, obtemos a solução

$$x_1 = \frac{\det(A_1)}{\det(A)} = -1, \tag{5.70a}$$

$$x_2 = \frac{\det(A_2)}{\det(A)} = 2, \tag{5.70b}$$

$$x_3 = \frac{\det(A_3)}{\det(A)} = 1. \tag{5.70c}$$

```python
import numpy as np
import numpy.linalg as npla
# matriz dos coefs
A = np.array([[2, -1, 1],
              [-1, 1, 3],
              [1, 3, -3]])
# vet termos consts
```

```
b = np.array([-3, 6, 2])

# det(A)
detA = npla.det(A)
print(f'det(A) = {detA}')

# matrizes auxiliares
## A1
A1 = np.copy(A)
A1[:,0] = b
detA1 = npla.det(A1)
print(f'det(A1) = {detA1}')

## A2
A2 = np.copy(A)
A2[:,1] = b
detA2 = npla.det(A2)
print(f'det(A2) = {detA2}')

## A3
A3 = np.copy(A)
A3[:,2] = b
detA3 = npla.det(A3)
print(f'det(A3) = {detA3}')

# solucao
x = np.array([detA1/detA, detA2/detA, detA3/detA])
print(f'x = {x}')
```

5.4.3 Exercícios

E.5.4.1. Aloque e imprima as seguintes matrizes como `arrays`:

a)

$$A = \begin{bmatrix} -1 & 2 \\ 7 & -3 \end{bmatrix} \tag{5.71}$$

b)

$$B = \begin{bmatrix} -1 & 2 & 4 \\ 7 & -3 & -5 \end{bmatrix} \tag{5.72}$$

c)

$$C = \begin{bmatrix} -1 & 2 & 4 \\ 7 & -3 & -5 \\ 2 & 9 & 6 \end{bmatrix} \tag{5.73}$$

d)

$$D = \begin{bmatrix} -1 & 2 & 4 \\ 7 & -3 & -5 \\ 2 & 9 & 6 \\ 1 & -1 & 1 \end{bmatrix} \tag{5.74}$$

E.5.4.2. Aloque as matrizes como `array` do NumPy

$$A = \begin{bmatrix} -1 & 2 & 4 \\ 7 & -3 & -5 \end{bmatrix} \tag{5.75}$$

e

$$B = \begin{bmatrix} 1 & -1 & 2 \\ 3 & 0 & 5 \end{bmatrix}. \tag{5.76}$$

Então, compute e imprima o resultado das seguintes operações matriciais

a) $A + B$

b) $A - B$

c) $2A$

E.5.4.3. Aloque as matrizes como `array` do NumPy

$$A = \begin{bmatrix} -1 & 2 & 4 \\ 7 & -3 & -5 \end{bmatrix} \tag{5.77}$$

e

$$B = \begin{bmatrix} 1 & -1 \\ 3 & 0 \\ 2 & 5 \end{bmatrix}. \tag{5.78}$$

Então, compute e imprima o resultado das seguintes operações matriciais

a) AB

b) BA

c) B^T

d) $A^T B^T$

E.5.4.4. Escreva a forma matricial $A\boldsymbol{x} = \boldsymbol{b}$ do seguinte sistema linear

$$-x_1 + 2x_2 - 2x_3 = 6 \tag{5.79a}$$
$$3x_1 - 4x_2 + x_3 = -11 \tag{5.79b}$$
$$x_1 - 5x_2 + 3x_3 = -10 \tag{5.79c}$$

Aloque a matriz dos coeficientes A e o vetor dos termos constantes $\boldsymbol{b}$ como `array` do NumPy. Então, verifique quais dos seguintes vetores é solução do sistema

a) $\boldsymbol{x} = (1, -1, -2)$

b) $\boldsymbol{x} = (-1, -2, 1)$

c) $\boldsymbol{x} = (-2, 1, -1)$

E.5.4.5. Calcule e compute o determinante das seguintes matrizes

a)

$$A = \begin{bmatrix} -1 & 2 \\ 7 & -3 \end{bmatrix} \tag{5.80}$$

b)

$$B = \begin{bmatrix} -1 & 2 & 4 \\ 1 & -3 & -5 \\ 2 & 0 & 6 \end{bmatrix} \tag{5.81}$$

c)

$$C = \begin{bmatrix} -1 & 2 & 4 & 1 \\ 7 & -3 & -5 & -1 \\ 2 & 0 & 1 & 0 \\ 1 & -1 & 1 & -2 \end{bmatrix} \tag{5.82}$$

E.5.4.6. Use o Método de Cramer para computar a solução do sistema dado no Exercício 5.4.4. Verifique sua solução com a computada pelo método `numpy.linalg.solve()`.

E.5.4.7. Desenvolva sua própria função Python para a computação do determinante de uma matriz A $n \times n$.

Respostas

E.5.4.1.

```python
import numpy as np
# a)
A = np.array([[-1, 2],
              [7, -3]])
print(f'A =\n {A}')
# b)
B = np.array([[-1, 2, 4],
              [7, -3, -5]])
print(f'B =\n {B}')
# c)
C = np.array([[-1, 2, 4],
              [7, -3, -5],
              [2, 9, 6]])
print(f'C =\n {C}')
# d)
```

```
D = np.array([[-1, 2, 4],
              [7, -3, -5],
              [2, 9, 6],
              [1, -1, 1]])
print(f'D =\n {D}')
```

E.5.4.2.

```
import numpy as np
A = np.array([[-1, 2, 4],
              [7, -3, -5]])
B = np.array([[1, -1, 2],
              [3, 0, 5]])
# a)
ApB = A + B
print(f'A+B =\n {ApB}')
# b)
AmB = A - B
print(f'A-B =\n {AmB}')
# c)
_2A = 2*A
print(f'2A =\n {_2A}')
```

E.5.4.3.

```
import numpy as np
A = np.array([[-1, 2, 4],
              [7, -3, -5]])
B = np.array([[1, -1],
              [3, 0],
              [2, 5]])
# a)
AB = A @ B
print(f'AB =\n {AB}')
# b)
BA = B @ A
print(f'BA =\n {BA}')
# c)
Bt = B.T
print(f'B^T =\n {Bt}')
# d)
```

```
AtBt = A.T @ B.T
print(f'A^T.B^T =\n {AtBt}')
```

E.5.4.4.

```
import numpy as np
A = np.array([[-1, 2, -2],
              [3, -4, 1],
              [1, -5, 3]])
b = np.array([6, -11, -10])
# c)
x = np.array([-2, 1, -1])
print(f'É solução? {np.allclose(A@x, b)}')
```

E.5.4.5.

```
import numpy as np
import numpy.linalg as npla
# a)
A = np.array([[-1, 2],
              [7, -3]])
detA = npla.det(A)
print(f'det(A) =\n {detA}')
# b)
B = np.array([[-1, 2, 4],
              [1, -3, -5],
              [2, 0, 6]])
detB = npla.det(B)
print(f'det(B) =\n {detB}')
# c)
C = np.array([[-1, 2, 4, 1],
              [-1, -3, -5, -1],
              [2, 0, 1, 0],
              [1, -1, 1, -2]])
detC = npla.det(C)
print(f'det(C) =\n {detC}')
```

E.5.4.6. Dica: $\boldsymbol{x} = (-2, 1, -1)$.

E.5.4.7. Dica: use a função `itertools.permutations()` para obter um iterador sobre as permutações.

Capítulo 6

Arquivos e Gráficos

6.1 Arquivos

6.1.1 Arquivo Texto

Um **arquivo texto** usualmente é identificado com a extensão `.txt` e contém uma `string`, i.e. uma coleção de caracteres.

Vamos considerar que o seguinte arquivo

Código 6.1: foo.txt

```
'''
Tabela de valores de
y = log(x).
'''

n, x, y
1, 1.0, 0.0000
2, 1.5, 0.4055
3, 2.0, 0.6931
4, 2.5, 0.9163
```

O nome deste aquivo é `foo.txt`. Baixe-o e salve-o com o mesmo nome em uma pasta de sua área de usuário no sistema de seu computador.

Leitura

Em programação, a **leitura de um arquivo** consiste em importar dados/informação de um arquivo para um código/programa. Para tanto, precisamos **abrir o arquivo**, i.e. criar um objeto da classe `file` associado ao arquivo. Em Python, abrimos um arquivo com a função `open(file, mode)`. Nela, `file` consiste em uma `string` com o **caminho para o aquivo** no sistema de arquivo do sistema operacional e, `mode` é uma string que especifica o modo de abertura. Para a abertura em modo leitura de um arquivo texto, usa-se `mode='r'` (`r`, do inglês, *read*).

Um vez aberto a **leitura do arquivo** pode ser feita com métodos específicos do objeto `file`, por exemplo, com o método `f.read()`. Para uma lista de métodos disponíveis em Python, consulte

https://docs.python.org/3/tutorial/inputoutput.html#methods-of-file-objects

Por fim, precisamos **fechar o arquivo**, o que pode ser feito com o método `f.close()`.

Por exemplo, consideramos o seguinte código

```
fl = open('foo.txt', 'r')
texto = fl.read()
fl.close()
print(texto)
```

Na primeira linha, o código: 1. abre o arquivo `foo.txt`[1], 2. lê o aquivo inteiro, 3. fecha-o e, 4. imprime o conteúdo lido. No código, `texto` é uma `string` que pode ser manipulada com os métodos e técnicas na Seção 2.6.

Alternativamente, pode-se fazer a **leitura linha-por-linha** do arquivo, como segue

```
fl = open('foo.txt', 'r')
for linha in fl:
    print(linha)
fl.close()
```

Escrita

A **escrita de um arquivo** consiste em exportar dados/informações de um código/programa para um arquivo de dados. Para tanto: 1. abrimos o arquivo no código com o comando `open(file, mode='w')` (`'w'`, do inglês,

[1]Aqui, é considerado que o arquivo está na mesma pasta em que o código está sendo executado.

write); 2. usamos um método de escrita, por exemplo, `f.write()` para escrever no arquivo; 3. fechamos o arquivo com `f.close()`.

Por exemplo, o seguinte código escreve o arquivo `foo.txt` (consulte o Código 6.1).

Código 6.2: foo.py

```
import numpy as np
# abre o arq
fl = open('foo.txt', mode='w')
# cabeçalho
fl.write("""'''
Tabela de valores de
y = log(x)
'''\n""")
# linha em branco
fl.write('\n')
# id das entradas
fl.write('n, x, y\n')
# entradas da tabela
xx = np.arange(1., 3., 0.5)
for i,x in enumerate(xx):
    fl.write(f'{i+1}, {x:.1f}, {np.log(x):.4f}\n')
# fecha o arq
fl.close()
```

Observamos que abertura de arquivo no modo `mode='w'` sobrescreve o arquivo caso ele já exista. Para escrever em um arquivo já existente, sem perdê-lo, podemos usar o modo `mode='a'` (`'a'`, do inglês, *append*).

Exemplo 6.1.1. Vamos fazer um código que adiciona uma nova entrada na tabela de valores do arquivo `foot.txt`, disponível no Código 6.1. A nova entrada, corresponde ao valor de $y = \ln(3.0)$.

```
import numpy as np
# abre o arq
fl = open('foo.txt', mode='a')
x = 3.
y = np.log(x)
fl.write(f'5, {x:.1f}, {y:.4f}\n')
# fecha o arq
fl.close()
```

6.1.2 Arquivo Binário

Um arquivo binário permite a escrita e leitura de dados binários de qualquer tipo (`int`, `float`, `string`, `tuple`, `list`, etc.). A módulo `pickle` contém funções para a escrita e leitura de dados em aquivos binários.

Escrita

Em um arquivo binário, os dados são escritos como registros binários, i.e. precisam ser convertidos para binário (serializados) e escritos no arquivo. A função `pickle.dump(obj)` faz isso para qualquer objeto Python.

Exemplo 6.1.2. Vamos escrever uma versão binária `'foo.pk'` do arquivo texto `foo.txt` trabalho acima. Para tanto, precisamos organizar os dados em um único objeto Python. Aqui, usamos um `dict` para organizar a informação e, então, salvar em arquivo binário.

Código 6.3: foo.bin

```
import numpy as np
import pickle
# dados
data = {}
## cabeçalho
data['info'] = 'Tabela de valores de y = log(x)'
## entradas
data['x'] = np.arange(1., 3., 0.5)
data['y'] = np.log(data['x'])
# abre arquivo
fl = open('foo.bin', 'wb')
# escreve no arquivo
pickle.dump(data, fl)
# fecha arquivo
fl.close()
```

Leitura

A leitura de um arquivo binário requer conhecer a estrutura dos dados alocados. No caso de um arquivo `pickle`, a leitura pode ser feita com a função `pickle.load()`. Por exemplo, o arquivo `foo.bin` (criado no Código 6.3) pode ser lido como segue

```
fl = open('foo.bin', 'rb')
data = pickle.load(fl)
fl.close()
print(data)
```

Observação 6.1.1. (Atenção.) Não abra e leia arquivos `pickle` que você não tenha certeza do conteúdo. Aquivos deste formato podem conter qualquer objeto Python, inclusive funções maliciosas.

6.1.3 Escrita e Leitura com NumPy

O NumPy contém a função `np.save(fn, arr)` para escrita no arquivo binário `fn` um arranjo `arr`. Por padrão, a extensão `.npy` é usada. Por exemplo,

```
import numpy as np
xx = np.arange(1., 3., 0.5)
yy = np.log(xx)
data = np.vstack((xx, yy))
np.save('foo.npy', data)
```

A leitura de um arquivo `.npy` pode ser feita com a função `np.load(fn)`, que retorna o arranjo lido a partir do arquivo binário `fn`. Por exemplo,

```
import numpy as np
data = np.load('foo.npy')
print(data)
```

6.1.4 Exercícios

E.6.1.1. Baixe o arquivo `foo.txt` disponível no Código 6.1. Então, desenvolva um código que:

a) leia o aquivo `foo.txt` e,

b) salve um novo arquivo `novo.txt` que não contenha a terceira entrada da tabela contida no arquivo original.

E.6.1.2. Desenvolva um código que escreve a seguinte tabela de ângulos fundamentais em um arquivo texto.

θ	$\sin(\theta)$	$\cos(\theta)$
0	0	1
$\pi/6$	$\sqrt{3}/2$	$1/2$
$\pi/4$	$\sqrt{2}/2$	$\sqrt{2}/2$
$\pi/3$	$1/2$	$\sqrt{3}/2$
$\pi/2$	1	0

E.6.1.3. Desenvolva um código que escreve a tabela dada no Exercício 6.1.2 como um dicionário em um arquivo binário. Então, leia o arquivo gerado e verifique os dados salvos.

E.6.1.4. Baixe para seu computador o seguinte arquivo texto.

Código 6.4: mat.txt

```
'''
Matriz A
'''
1, -1, 2
2, 1, 3,
-3, -1, -2
```

Este arquivo contém os elementos da matriz $A = [a_{i,j}]_{i,j=1}^{3,3}$. Desenvolva um código que leia este arquivo, aloque a matriz A associada e, então, calcule e imprima o valor de seu determinante, i.e. $\det(A)$.

E.6.1.5. Faça um código que salve a matriz

$$A = \begin{bmatrix} 1 & -1 & 2 \\ 2 & 1 & 3 \\ -3 & -1 & -2 \end{bmatrix} \tag{6.1}$$

como um arquivo binário `.npy`. Em um outro código, leia o arquivo gerado, compute e imprima o traço de A, i.e.

$$\operatorname{tr}(A) = \sum_{i=1}^{3} a_{i,i}. \tag{6.2}$$

Respostas

E.6.1.1.

```
fin = open('foo.txt')
fout = open('novo.txt', 'w')
for i, linha in enumerate(fin):
    if (i != 8):
        fout.write(linha)
fin.close()
fout.close()
```

E.6.1.2. Dica: consulte o Código 6.2.

E.6.1.3. Dica: consulte a Subseção 6.1.2.

E.6.1.4. Dica: use o método `str.split()`.

E.6.1.5. Dica: $\operatorname{tr}(A) = 0$.

6.2 Gráficos

Vamos usar o pacote computacional Matplotlib para a elaboração de gráficos de funções. Usualmente, vamos utilizar os seguintes módulos Python

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
```

6.2.1 Área Gráfica

No Matplotlib, os gráficos são colocados em um *container* `Figure` (uma janela gráfica ou um arquivo de imagem). O *container* pode ter um ou mais Axes, uma área gráfica contendo todos os elementos de um gráfico (eixos, pontos, linhas, anotações, legendas, etc.). Podemos usar Axes.plot para plotar dados.

Exemplo 6.2.1. Consideramos a função

$$f(x) = |x|, \; -\frac{1}{2} \leq x < 1. \tag{6.3}$$

A Figura 6.1, mostra o gráfico de f plotado com o código abaixo.

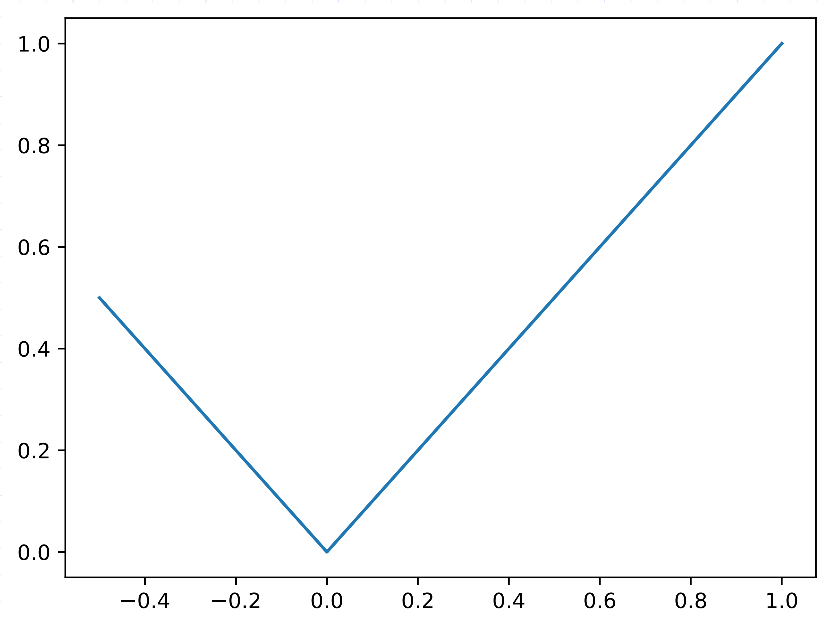


Figura 6.1: Gráfico referente ao Exemplo **??**.

```
import matplotlib.pyplot as plt

# figure
fig = plt.figure()
# axes
ax = fig.add_subplot()
# plot
ax.plot([-0.5, 0, 1],
        [0.5, 0, 1])
# display
plt.show()
```

No caso de curvas, podemos usamos um número adequado de pontos de forma que os segmentos de linhas ficam imperceptíveis a olho nu.

Exemplo 6.2.2. Consideramos a função

$$f(x)=\begin{cases}-(x+1)^2-2 & , -2\leq x<-\frac{1}{2},\\ |x| & , -\frac{1}{2}\leq x<1,\\ (x-2)^3+2, & , 1\leq x<3.\end{cases}\tag{6.4}$$

A Figura 6.2, mostra o gráfico de f plotado com o código abaixo.

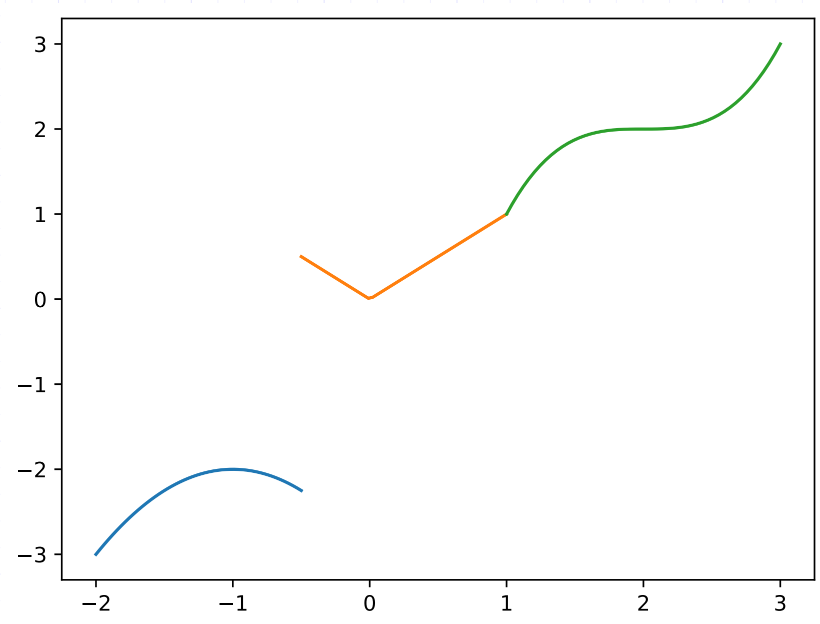


Figura 6.2: Gráfico referente ao Exemplo 6.2.2.

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np

# figure
fig = plt.figure()
# axis
ax = fig.add_subplot()
# -2 <= x < -0.5
x = np.linspace(-2, -0.5)
ax.plot(x, -(x+1)**2-2)
# -0.5 <= x < 1
```

```
x = np.linspace(-0.5, 1)
ax.plot(x, np.fabs(x))
# 1 <= x < 3
x = np.linspace(1, 3)
ax.plot(x, (x-2)**3+2)
# display
plt.show()
```

6.2.2 Eixos

No Matplotlib, os eixos de um gráfico são objetos da classe `Axis`[2].

Exemplo 6.2.3. Com o código abaixo, produzimos a Figura 6.3, a qual contém o gráfico da função do Exemplo 6.2 com os eixos editados.

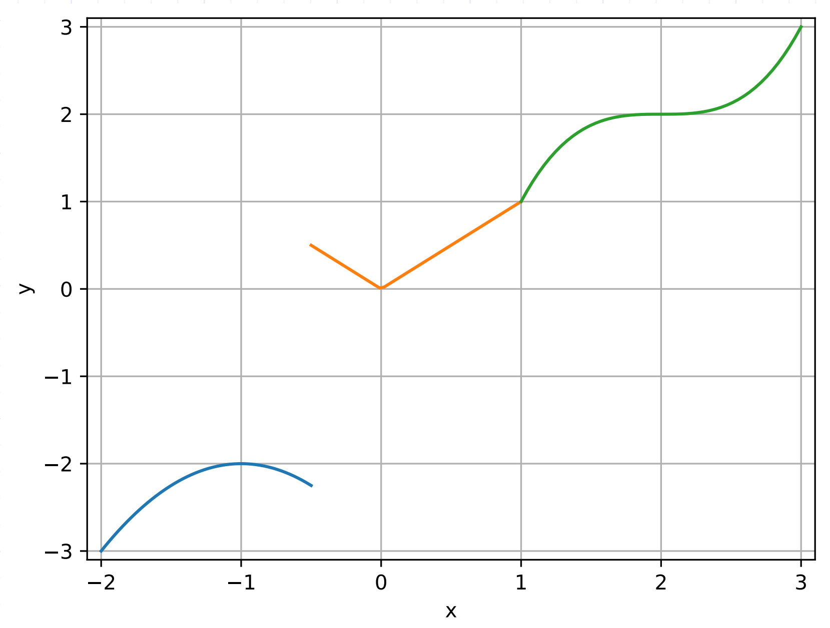


Figura 6.3: Gráfico referente ao Exemplo **??**.

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

[2]Não confundir com Axes, um objeto que contém todos os elementos de um gráfico.

```python
import numpy as np

# figure
fig = plt.figure()
# axis
ax = fig.add_subplot()
# -2 <= x < -0.5
x = np.linspace(-2, -0.5)
ax.plot(x, -(x+1)**2-2)
# -0.5 <= x < 1
x = np.linspace(-0.5, 1)
ax.plot(x, np.fabs(x))
# 1 <= x < 3
x = np.linspace(1, 3)
ax.plot(x, (x-2)**3+2)
# eixo-x
ax.set_xlim((-2.1, 3.1))
ax.set_xticks([-2, -1, 0, 1, 2, 3])
ax.set_xlabel('x')
#eixo-y
ax.set_ylim((-3.1, 3.1))
ax.set_yticks([-3, -2, -1, 0, 1, 2, 3])
ax.set_ylabel('y')
# grid
ax.grid()
# display
plt.savefig('fig.png', bbox_inches='tight')
plt.savefig('fig.pdf', bbox_inches='tight')
plt.show()
```

6.2.3 Elementos Gráficos

No Matplotlib, os elementos gráficos (basicamente tudo o que é visível, pontos, linhas, eixos, etc.) são objetos da classe Artist.

Exemplo 6.2.4. Com o código abaixo, produzimos a Figura 6.4, a qual contém o gráfico da função do Exemplo 6.4 com os eixos editados.

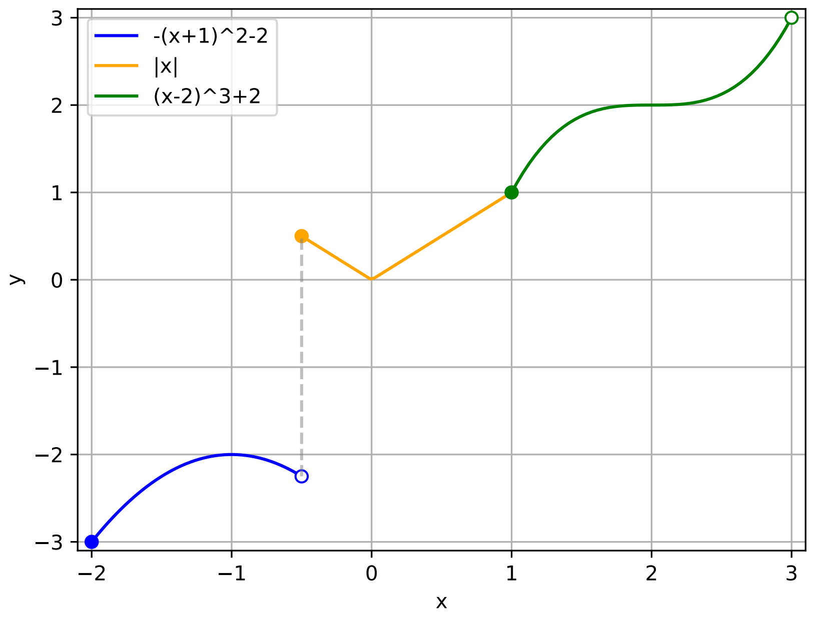


Figura 6.4: Gráfico referente ao Exemplo 6.2.4.

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np

# figure
fig = plt.figure()
# axis
ax = fig.add_subplot()

# -2 <= x < -0.5
x = np.linspace(-2, -0.5)
f1 = lambda x: -(x+1)**2-2
ax.plot(x, f1(x), color='blue',
        label='-(x+1)^2-2')
ax.plot([-2.], f1(-2.), linestyle='', marker='o',
        color='blue')
ax.plot([-0.5], f1(-0.5), ls='', marker='o',
        markerfacecolor='white', markeredgecolor='blue')

```

```
# -0.5 <= x < 1
x = np.linspace(-0.5, 1)
f2 = lambda x: np.fabs(x)
ax.plot(x, f2(x), color='orange', label='|x|')
ax.plot([-0.5], [f2(-0.5)], ls='', marker='o',
        color='orange')

ax.plot([-0.5, -0.5], [f1(-0.5), f2(-0.5)],
        ls = '--', color='gray', alpha=0.5)

# 1 <= x < 3
x = np.linspace(1, 3)
f3 = lambda x: (x-2)**3+2
ax.plot(x, f3(x), color='green',
        label='(x-2)^3+2')
ax.plot([1.], [f3(1.)], ls='', marker='o',
        color='green')
ax.plot([3.], [f3(3.)], ls='', marker='o',
        mfc='white', mec='green')

# eixo-x
ax.set_xlim((-2.1, 3.1))
ax.set_xticks([-2, -1, 0, 1, 2, 3])
ax.set_xlabel('x')
# eixo-y
ax.set_ylim((-3.1, 3.1))
ax.set_yticks([-3, -2, -1, 0, 1, 2, 3])
ax.set_ylabel('y')
# grid
ax.grid()
ax.legend()
# display
plt.savefig('fig.png', bbox_inches='tight')
plt.savefig('fig.pdf', bbox_inches='tight')
plt.show()
```

6.2.4 Textos e Anotações

Elementos texto podem ser adicionados a um `Axes` com o comando `axes.text()`. Anotações, consistem em um apontamento, e podem ser adicionadas com o comando `axes.annotate()`. Elementos texto suportam LaTeXusando-se o marcador de texto `$`.

Exemplo 6.2.5. Com o código abaixo, produzimos a Figura 6.5, a qual contém o gráfico da função do Exemplo 6.5 com os eixos editados.

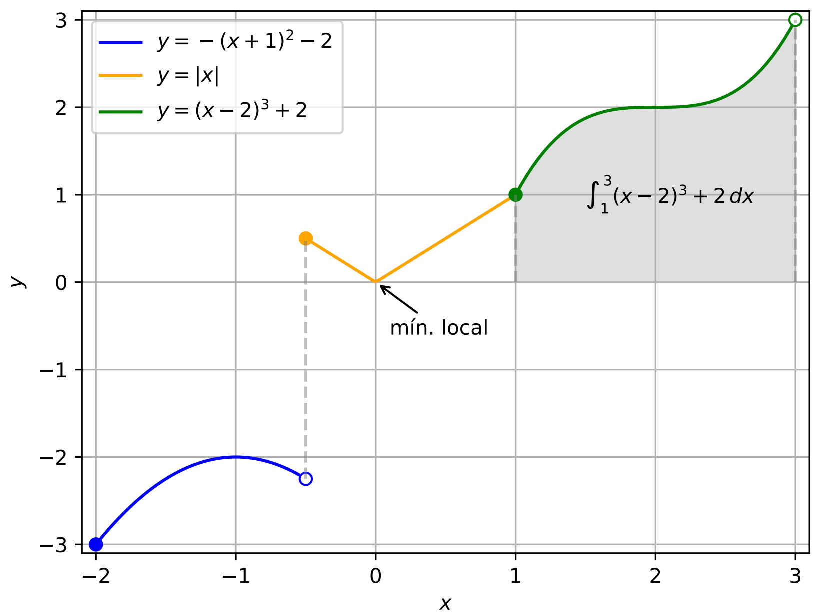


Figura 6.5: Gráfico referente ao Exemplo 6.2.5.

```python
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np

# figure
fig = plt.figure()
# axis
ax = fig.add_subplot()

# -2 <= x < -0.5
```

```
x = np.linspace(-2, -0.5)
f1 = lambda x: -(x+1)**2-2
ax.plot(x, f1(x), color='blue',
        label='$y=-(x+1)^2-2$')
ax.plot([-2.], f1(-2.), linestyle='', marker='o',
        color='blue')
ax.plot([-0.5], f1(-0.5), ls='', marker='o',
        markerfacecolor='white', markeredgecolor='blue')

# -0.5 <= x < 1
x = np.linspace(-0.5, 1)
f2 = lambda x: np.fabs(x)
ax.plot(x, f2(x), color='orange', label='$y=|x|$')
ax.plot([-0.5], [f2(-0.5)], ls='', marker='o',
        color='orange')

ax.plot([-0.5, -0.5], [f1(-0.5), f2(-0.5)],
        ls = '--', color='gray', alpha=0.5)
# anotação
ax.annotate('mín. local', xy=(0,0), xytext=(0.1,-0.6),
            arrowprops={'arrowstyle':'->'})

# 1 <= x < 3
x = np.linspace(1, 3)
f3 = lambda x: (x-2)**3+2
ax.plot(x, f3(x), color='green',
        label='$y=(x-2)^3+2$')
ax.plot([1.], [f3(1.)], ls='', marker='o',
        color='green')
ax.plot([3.], [f3(3.)], ls='', marker='o',
        mfc='white', mec='green')

# hachurando
ax.fill_between(x, f3(x), color='gray', alpha=0.25)
ax.plot([1., 1.], [0., f3(1.)],
        ls='--', color='gray', alpha=0.5)
ax.plot([3., 3.], [0., f3(3.)],
        ls='--', color='gray', alpha=0.5)
# texto
ax.text(1.5, 0.9, '$\\int_{1}^3 (x-2)^3+2\,dx$')
```

```

# eixo-x
ax.set_xlim((-2.1, 3.1))
ax.set_xticks([-2, -1, 0, 1, 2, 3])
ax.set_xlabel('$x$')
# eixo-y
ax.set_ylim((-3.1, 3.1))
ax.set_yticks([-3, -2, -1, 0, 1, 2, 3])
ax.set_ylabel('$y$')
# grid
ax.grid()
ax.legend()
# display
plt.savefig('fig.png', bbox_inches='tight')
plt.savefig('fig.pdf', bbox_inches='tight')
plt.show()
```

6.2.5 Exercícios

E.6.2.1. Use o Matplotlib para produzir um gráfico para as seguintes funções:

a) $f(x) = x^2$, $-2 \leq x \leq 2$.

b) $g(x) = 2x^3 + 2$, $-3 \leq x \leq 0$.

c) $h(x) = \operatorname{sen}(x)$, $-\pi \leq x \leq \pi$.

E.6.2.2. Use o Matplotlib para plotar o gráfico da função sigmoid

$$f(x) = \frac{1}{1 + e^{-x}}. \tag{6.5}$$

Na mesma área gráfica, plote retas tracejadas identificando suas assíntotas horizontais.

E.6.2.3. Use o Matplotlib para plotar o gráfico de $f(x) = 1/x$, $-2 \leq x \leq 2$. Na mesma área gráfica, plote uma reta tracejada identificando a assíntota vertical de f.

E.6.2.4. Use o Matplotlib para produzir um gráfico para a seguinte função definida por partes

$$f(x) = \begin{cases} \cos(x) & , -\pi < x \leq 0, \\ 1 - x^2 & , 0 < x \leq 2. \end{cases} \tag{6.6}$$

Use de marcadores para identificar os pontos extremos de cada parte da função. Também, adicione o *label* de cada eixo e uma legenda para identificar cada parte da função.

E.6.2.5. Em uma mesma área gráfica, plote as curvas $y = x + 1$ e $y = x^2$, e marque seus pontos de interseção. Para cada um destes pontos, inclua a anotação “pto. de interseção”.

E.6.2.6. No gráfico da função sigmoid

$$f(x) = \frac{1}{1 + e^{-x}} \tag{6.7}$$

hachure (pinte) a região que corresponde a área associada a integral definida

$$\int_1^3 f(x)\,dx. \tag{6.8}$$

Exemplo 6.2.6. Em uma mesma área gráfica, plote a área entre as curvas $y = x + 1$ e $y = x^2$, $x = -1$ e $x = 2$.

Respostas

E.6.2.2. Dica: $y = 0$ e $y = 1$ são assíntotas horizontais da função sigmoid.

E.6.2.3. Dica: $x = 0$ é assíntota vertical de f.

E.6.2.6. Dica: use a função `Axes.fill_between()`.

Capítulo 7

Orientação a Objetos

Programação Orientação a Objetos (POO) é um paradigma de programação baseado no conceito de classes de objetos. A classe define seus atributos (propriedades e métodos) de seus objetos. Todos os objetos de uma classe têm os mesmos atributos, mas são independentes um dos outros, sendo que cada um é uma instância própria da classe contendo seus próprios valores de seus atributos.

7.1 Classe e Objeto

Uma **classe** é uma forma de estrutura que permite a alocação conjunta de dados e funções. Em Python, a sintaxe de definição de uma classe é

```
class NomeDaClasse:
    <bloco-0>
    <bloco-1>
    ...
    <bloco-2>
```

Usualmente, os blocos de programação consistem de definições de funções (métodos). Por exemplo,

```
class MinhaClasse:
    def digaOla(self):
        print('Olá, Mundo!')

obj = MinhaClasse()
obj.digaOla()
```

Neste código, temos a definição da classe `MinhaClasse` (linhas 1-3). Esta classe contém o método `MinhaClasse.digaOla()` (linhas 2-3). Obrigatoriamente, na definição de um método de uma classe deve conter o primeiro parâmetro `self`. Um objeto desta classe[1] e identificado por `obj` é alocado na linha 5. Na linha 6, este objeto chama seu método `obj.digaOla()`.

O método especial `__init__()` é executado na construção de cada nova instância da classe (objeto da classe). Por exemplo,

```
1  class Brasileira:
2      pais = 'Brasil'
3      def __init__(self, nome):
4          self.nome = nome
5  
6      def digaOla(self):
7          print('\nOlá!')
8          print(f'Eu me chamo {self.nome}.')
9          print(f'Sou do {self.pais}. :)')
10 
11 x = Brasileira('Fulane')
12 x.digaOla()
13 y = Brasileira('Beltrane')
14 y.digaOla()
```

Aqui, o atributo `Brasileira.pais` é compartilhada entre todas as instâncias da classe (objetos), enquanto que `Brasileira.nome` é um atributo de cada objeto. O método `__init()__` (linhas 3-4) é executada no momento da criação de cada nova instância (linhas 11 e 13).

Exemplo 7.1.1. No seguinte código, começamos a definição de uma classe para a manipulação de triângulos.

Código 7.1: classTriangulo.py

```
import matplotlib.pyplot as plt

class Triangulo:
    '''
    Classe Triangulo ABC.
    '''
    num_lados = 3
```

[1]Uma nova instância da classe.

```
    def __init__(self, A, B, C):
        # vértices
        self.A = A
        self.B = B
        self.C = C

    def plot(self):
        fig = plt.figure()
        ax = fig.add_subplot()
        # lados
        ax.plot([self.A[0], self.B[0]],
                [self.A[1], self.B[1]], marker='o', color='blue')
        ax.text((self.A[0]+self.B[0])/2,
                (self.A[1]+self.B[1])/2, 'c')
        ax.plot([self.B[0], self.C[0]],
                [self.B[1], self.C[1]], marker='o', color='blue')
        ax.text((self.B[0]+self.C[0])/2,
                (self.B[1]+self.C[1])/2, 'a')
        ax.plot([self.C[0], self.A[0]],
                [self.C[1], self.A[1]], marker='o', color='blue')
        ax.text((self.A[0]+self.C[0])/2,
                (self.A[1]+self.C[1])/2, 'b')
        # vertices
        ax.text(self.A[0], self.A[1], 'A')
        ax.text(self.B[0], self.B[1], 'B')
        ax.text(self.C[0], self.C[1], 'C')
        ax.grid()
        plt.show()

tria = Triangulo((0., 0.),
                 (2., 0.),
                 (1., 1.))
tria.plot()
```

7.1.1 Exercícios

E.7.1.1. Considere o Código 7.1. Adicione o método `calcLados()`, que computa e aloca o comprimento de cada lado do triângulo.

E.7.1.2. Considere o Código 7.1. Adicione o método `calcPerimetro()`, que computa e retorna o valor do perímetro do triângulo.

E.7.1.3. Considere o Código 7.1. Adicione o método `calcAngulos()`, que computa e aloca os ângulos do triângulo.

E.7.1.4. Considere o Código 7.1. Adicione o método `area()`, que computa a área do triângulo.

E.7.1.5. Similar a classe `Triangulo` (Código 7.1), implemente uma nova classe `Quadrilateros` com as seguintes propriedades e métodos de quadriláteros $ABCD$:

a) vértices (`tuples`).

b) lados (`floats`).

c) cálculo do perímetro (método).

d) cálculo da área (método).

e) visualização gráfica (método +plot+).

E.7.1.6. Implemente uma classe para a manipulação de polinômios de segundo grau. A classe deve conter as seguintes propriedades e métodos:

a) coeficientes (`floats`).

b) cálculo do ponto de interseção com o eixo y (método).

c) cálculo do vértice da parábola associada ao polinômio (método).

d) cálculo das raízes do polinômio (método).

e) plotagem do gráfico do polinômio (método).

Respostas

E.7.1.1.

```
import numpy as np
import matplotlib.pyplot as plt

class Triangulo:
    '''
    Classe Triangulo ABC.
    '''
    num_lados = 3
    def __init__(self, A, B, C):
        # vértices
        self.A = A
        self.B = B
        self.C = C
        # lados
        self.a = 0.
        self.b = 0.
        self.c = 0.

    def calcLados(self):
        self.a = np.sqrt((self.B[0]-self.C[0])**2\
                         + (self.B[1]-self.C[1])**2)
        self.b = np.sqrt((self.A[0]-self.C[0])**2\
                         + (self.A[1]-self.C[1])**2)
        self.c = np.sqrt((self.A[0]-self.B[0])**2\
                         + (self.A[1]-self.B[1])**2)
```

E.7.1.2.

```
import numpy as np
import matplotlib.pyplot as plt

class Triangulo:

    ...

    def perimetro(self):
        return self.a + self.b + self.c
```

```

    ...
```

E.7.1.3. Dica: use a Lei dos Cossenos.

E.7.1.4. Dica: use o Teorema de Herão.

E.7.1.6. Dica: utilize a notação $p(x) = ax^2 + bx + c$.

7.2 Herança

Na programação orientada-a-objetos, **herança** consiste na definição de uma classe derivada a partir de uma dada classe base. A sintaxe de definição de uma classe derivada é

```
class ClasseDerivada(ClasseBase):
    bloco-0
    bloco-1
    ...
    bloco-n
```

A classe derivada herda todos os atributos da classe base. Por exemplo, consideramos o seguinte código

```
1 class ClasseBase:
2     def __init__(self, nome):
3         self.nome = nome
4 
5     def digaOi(self):
6         print(f'{self.nome}: Oi!')
7 
8 class ClasseDerivada(ClasseBase):
9     def digaTchau(self):
10        print(f'{self.nome}: Tchau!')
11 
12 obj = ClasseDerivada('Fulane')
13 obj.digaOi()
14 obj.digaTchau()
```

Nas linhas 1-6, a classe base é definida contendo dois métodos: `self.__init__()` chamado na criação de um objeto da classe (uma instância) e, `self.digaOi()` que imprime uma saudação. A classe derivada é definida nas linhas 8-10, ela herda os atributos da classe base e contém um novo método `self.digaTchau()`, que imprime uma mensagem de despedida.

A criação de uma instância (objeto) de uma classe derivada é feita da mesma forma que de uma classe base. A referência a um atributo do objeto é, primeiramente, buscada na classe derivada e, se não encontrada, é buscada na classe base. Este regra aplica-se recursivamente se a classe base também é derivada de outra classe. Isso permite que uma classe derivada sobreponha atributos de sua classe base.

Observação 7.2.1. (`super()`.) O método `super()` retorno um objeto *proxy* da classe base, que acessa os atributos desta classe.

Exemplo 7.2.1. Vamos criar uma classe para manipular triângulo isósceles. Para tanto, vamos derivá-la a partir da classe `Triangulo` definida no Exemplo 7.1.1. Vamos assumir que os triângulos isósceles têm vértices ΔABC com lados $b = AC$ e $a = BC$ de mesmo tamanho.

Código 7.2: classTrianguloIsosceles.py

```
import numpy as np
import matplotlib.pyplot as plt

class Triangulo:
    '''
    Classe Triangulo ABC.
    '''
    num_lados = 3
    def __init__(self, A, B, C):
        # vértices
        self.A = A
        self.B = B
        self.C = C

    def plot(self):
        fig = plt.figure()
        ax = fig.add_subplot()
        # lados
        ax.plot([self.A[0], self.B[0]],
                [self.A[1], self.B[1]], marker='o', color='blue')
        ax.text((self.A[0]+self.B[0])/2,
                (self.A[1]+self.B[1])/2, 'c')
```

```
        ax.plot([self.B[0], self.C[0]],
                [self.B[1], self.C[1]], marker='o', color='blue')
        ax.text((self.B[0]+self.C[0])/2,
                (self.B[1]+self.C[1])/2, 'a')
        ax.plot([self.C[0], self.A[0]],
                [self.C[1], self.A[1]], marker='o', color='blue')
        ax.text((self.A[0]+self.C[0])/2,
                (self.A[1]+self.C[1])/2, 'b')
        # vertices
        ax.text(self.A[0], self.A[1], 'A')
        ax.text(self.B[0], self.B[1], 'B')
        ax.text(self.C[0], self.C[1], 'C')
        ax.grid()
        plt.show()


class TrianguloIsosceles(Triangulo):
    def __init__(self,A,B,C):
        # vertices
        super().__init__(A,B,C)
        # lados
        self.a = self.b = self.c = 0.

    def calcLados(self):
        self.a = np.sqrt((self.B[0] - self.C[0])**2\
                         + (self.B[1] - self.C[1])**2)
        self.b = np.sqrt((self.A[0] - self.C[0])**2\
                         + (self.A[1] - self.C[1])**2)
        self.c = np.sqrt((self.B[0] - self.A[0])**2\
                         + (self.B[1] - self.A[1])**2)
        assert(self.a == self.b)

tria = TrianguloIsosceles((1,0),
                          (3,0),
                          (2,1))
tria.plot()
tria.calcLados()
```

Observação 7.2.2. (Herança Múltipla.) Python suporta a herança múltipla de classes. A sintaxe é

```python
class ClasseDerivada(Base1, Base2, ..., BraseN):
    bloco-0
    bloco-1
    ...
    bloco-m
```

Quando um objeto da classe derivada faz uma referência a um atributo, este é procurado de forma sequencial (e recursiva, caso uma das classe bases seja também uma classe derivada) começando por essa e, caso não encontrado, buscando-se nas classes `Base1`, `Base2`, ..., `BaseN`.

7.2.1 Exercícios

E.7.2.1. No Código 7.2, adicione à classe `Triangulo` o método `Triangulo.perimetro()` que computa, aloca e retorna o valor do perímetro do triângulo. Então, sobreponha o método à classe `TrianguloIsosceles`. Teste seu código para diferentes triângulos.

E.7.2.2. Implemente uma classe `Retangulo(largura, altura)` para a manipulação de retângulos de `largura` e `altura` dadas. Equipe sua classe com métodos para o cálculo do perímetro, da diagonal e da área de retângulo. Então, implemente a classe derivada `Quadrado(lado)` para a manipulação de quadrados de `lado` dado. Teste sua implementação para diferentes retângulos e quadrados.

E.7.2.3. Refaça o Exercício 7.2.2 sobrepondo os métodos do cálculo do perímetro, da diagonal e da área para quadrados.

E.7.2.4. Implemente uma classe `TrianguloEquilatero`, derivada da classe `TrianguloIsosceles` definida no Código 7.2. Adicione métodos para o cálculo do perímetro e da altura de triângulo equiláteros. Teste seu código para diferentes triângulos.

E.7.2.5. Implemente:

a) Uma classe `Quadrilatero` para a manipulação de quadriláteros de lados $abcd$. Equipe sua classe com um método `self.perimetro()` para o cálculo do perímetro.

b) Uma classe `Retangulo`, derivada da classe `Quadrilatero`, para a manipulação de retângulos de lado dado e altura dada. Na classe derivada, sobreponha o método `self.perimetro()` para o cálculo do perímetro e implemente novos métodos para o cálculo da diagonal e da área de retângulos.

c) Uma classe `Quadrado`, derivada da classe `Retangulo`, para a manipulação de quadrados de lado dado. Na classe derivada, sobreponha os métodos para os cálculos do perímetro, da diagonal e da área.

Respostas

E.7.2.1.

```python
class Triangulo:
    def __init(self,A,B,C)__:
        ...
        self.p = 0.
        ...
    ...
    def perimetro(self):
        self.p = self.a\
               + self.b\
               + self.c
        return self.p
    ...

class TrianguloIsosceles(Triangulo):
    ...
    def perimetro(self):
        self.p = 2*self.a + self.c
        return self.p
    ...
```

E.7.2.2.

```python
import math as m

class Retangulo:
```

```
    def __init__(self, largura, altura):
        self.largura = largura
        self.altura = altura

    def perimetro(self):
        return self.largura\
               + self.altura

    def diagonal(self):
        return m.sqrt(self.largura**2\
                      + self.altura**2)

    def area(self):
        return self.largura\
               * self.altura

class Quadrado(Retangulo):
    def __init__(self,lado):
        super().__init__(lado,lado)
```

E.7.2.3. Dica: para um quadrado de lado l, o perímetro é $p = 2l$, por exemplo.

E.7.2.4. Dica:

```
...
class Triangulo:
   def __init__(self,A,B,C):
      ...
   ...

class TrianguloIsosceles(Triangulo):
   ...

class TrianguloEquilatero(TrianguloIsosceles):
   def __init__(self,A,B,C):
      super().__init__(A,B,C)

   def perimetro(self):
      ...
```

```

    def altura(self):
        ...

    def area(self):
        ...
```

Bibliografia

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

[2] Cormen, T.. Desmitificando Algoritmos, Grupo GEN: São Paulo, 2021. ISBN: 978-8595153929.

[3] Cormen, T.. Algoritmos - Teoria e Prática, Grupo GEN: São Paulo, 2012. ISBN: 978-8595158092.

[4] Grus, J.. Data Science do Zero, Alta Books: Rio de Janeiro, 2021. ISBN: 978-8550816463.

[5] Ribeiro, J.A.. Introdução à Programação e aos Algoritmos, LTC: São Paulo, 2021. ISBN: 978-8521636410. Acesso pelo SABi+UFRGS: https://bit.ly/42Z4VFC

[6] Wazlawick, R.. Introdução a Algoritmos e Programação com Python - Uma Abordagem Dirigida por Testes, Grupo GEN: São Paulo, 2021. ISBN 978-8595156968.