

Algoritmos e Programação I

Pedro H A Konzen

23 de maio de 2023

Licença

Este trabalho está licenciado sob a Licença Atribuição-CompartilhaIgual 4.0 Internacional Creative Commons. Para visualizar uma cópia desta licença, visite http://creativecommons.org/licenses/by-sa/4.0/deed.pt_BR ou mande uma carta para Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Prefácio

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	v
1 Introdução	1
2 Linguagem de Programação	3
2.1 Computador	3
2.1.1 Linguagem de programação	6
2.1.2 Instalação e execução	8
2.1.3 Exercícios	10
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	24
2.4.1 Números Inteiros	27
2.4.2 Ponto Flutuantes	28
2.4.3 Números Complexos	30
2.4.4 Exercícios	31

Respostas dos Exercícios	34
---------------------------------	-----------

Referências Bibliográficas	39
-----------------------------------	-----------

Capítulo 1

Introdução

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

```
1 >>> 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

```
1 >>> print('Olá, mundo!')
2 Olá, mundo!
3 >>>
```

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

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

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

$$(1, 3, 5, \dots, 99) \quad (1.1)$$

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

$$\sum_{i=0}^{49} a_i = a_0 + a_1 + \dots + a_{49} \quad (1.2)$$

$$= 1 + 3 + \dots + 99 \quad (1.3)$$

$$= \frac{50(1 + 99)}{2} \quad (1.4)$$

$$= 2500 \quad (1.5)$$

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

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

Capítulo 2

Linguagem de Programação

2.1 Computador

[YouTube] | [Vídeo] | [Áudio] | [Contatar]

Um computador¹ 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.

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

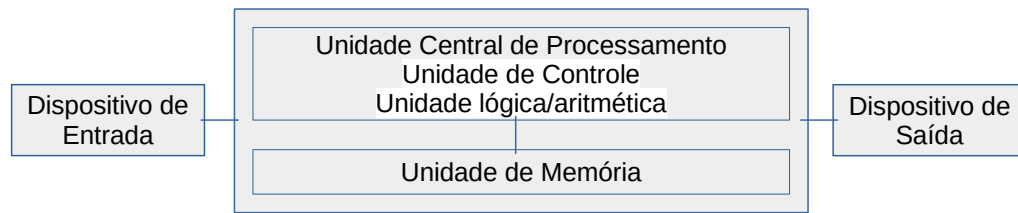


Figura 2.1: Arquitetura de computador de von Neumann.

Os computadores que comumente utilizamos seguem a arquitetura de John von Neumann², 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**

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

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.

- **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 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*

³Usualmente de tamanho 64-*bits*.

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

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 Rossum⁵ 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](#):

```
1 >>> 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 completamente 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 nessa 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

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

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

⁶Consulte a licença de uso em <https://docs.python.org/3/license.html>.

<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 seu sistema operacional. Ainda, há plataformas e IDEs⁷ [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:

```
1      >>> print('Olá, mundo!')
2      Olá, mundo!
3      >>>
```

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 arquivo 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

```
1      $ 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](#):

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

<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

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

Exercício 2.1.2. Verifique os seguintes elementos de seu computador:

- a) CPUs
- b) Placa(s) gráfica(s)
- c) Memória RAM
- d) Armazenamento HDD/SSD.

Exercício 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.

Exercício 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!”.

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

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

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

Exercício 2.1.6. Use um *notebook* [Python](#) para escrever e executar o código do exercício anterior.

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 sequência 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. $a \leftarrow \frac{b \cdot h}{2}$.
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:

```
1 b = float(input('Informe o valor da base.\n'))
2 h = float(input('Informe o valor da altura.\n'))
3 # cálculo da área
4 a = b*h/2
5 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 King⁸. 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.

⁸Augusta Ada King, 1815 - 1852, matemática e escritora inglesa. Fonte: [Wikipédia](#).

⁹Jacob Bernoulli, 1655-1705, matemático suíço. Fonte: [Wikipédia](#).



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



- **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

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

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

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

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:

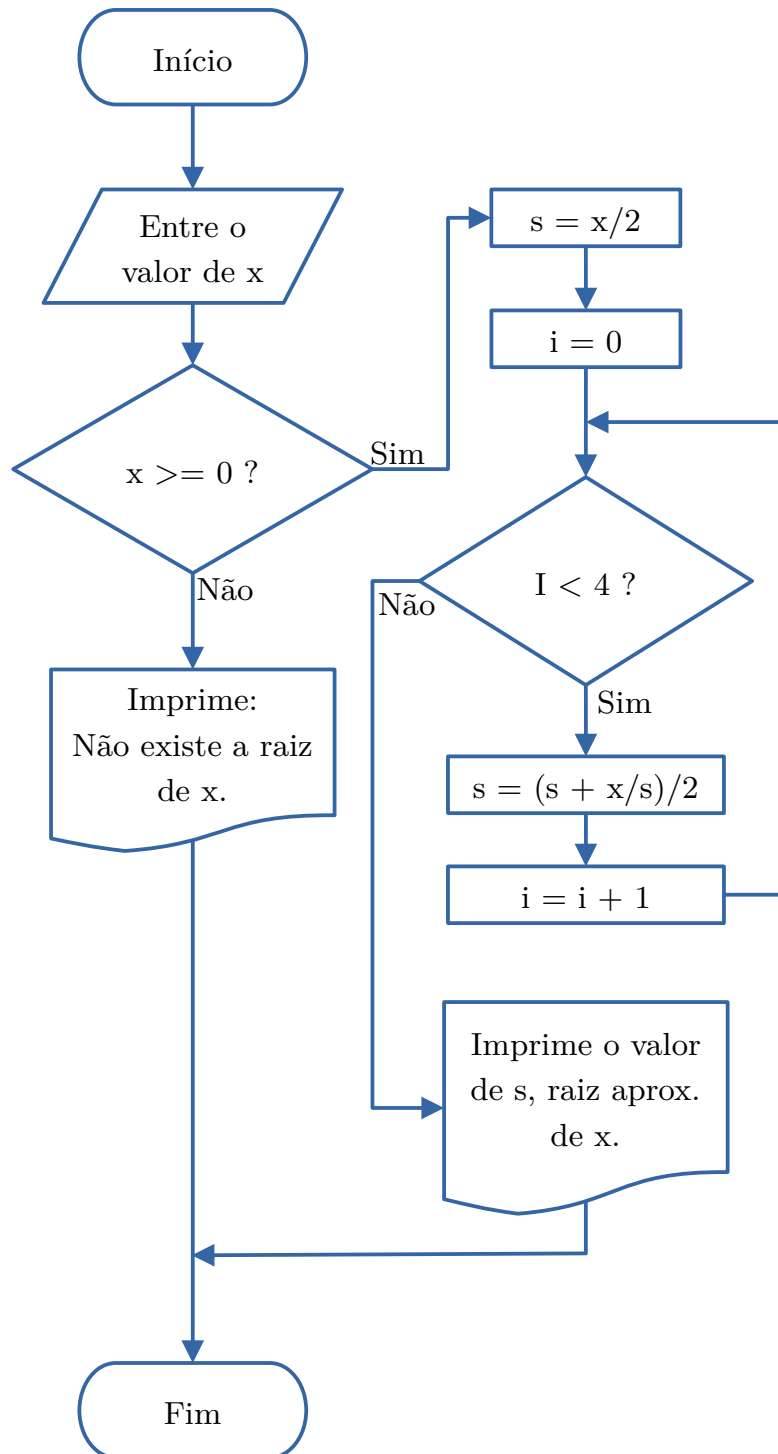
- i. $s \leftarrow (s + x/s)/2$.

- (c) Imprime o valor de s .

3. Senão, faça:

- (a) Imprime mensagem “Não existe!”.

- **Fluxograma**



- Código Python

Código 2.1: metHeron.py

```
1 x = float(input('Entre com o valor de x: '))
2 if (x >= 0.):
3     s = x/2
4     for i in range(4):
5         s = (s + x/s)/2
6     print(f'Raiz aprox. de x = {s}')
7 else:
8     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

Exercício 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.

Exercício 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.

Exercício 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.

Exercício 2.2.4. Escreva um algoritmo/pseudocódigo e um fluxograma correspondente para o calcular triângulo retângulo de hipotenusa h e um dos

dados l dados. Como desafio, tente escrever um código [Python](#) baseado em seu algoritmo.

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

$$f(x) = ax + b \quad (2.3)$$

dados, os coeficientes a e b . Como desafio, tente escrever um código [Python](#) baseado em seu algoritmo.

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

$$p(x) = ax^2 + bx + c \quad (2.4)$$

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

Exercício 2.2.7. A [Série Harmônica](#) é definida por

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

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

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

Exercício 2.2.8. O [número de Euler](#)¹¹ pode ser definido pela série

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

$$= \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \cdots \quad (2.8)$$

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

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

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

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

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

Exercício 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](#).

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

$$y = ax + b \quad (2.12)$$

com o eixo x (consulte a Figura 2.2).

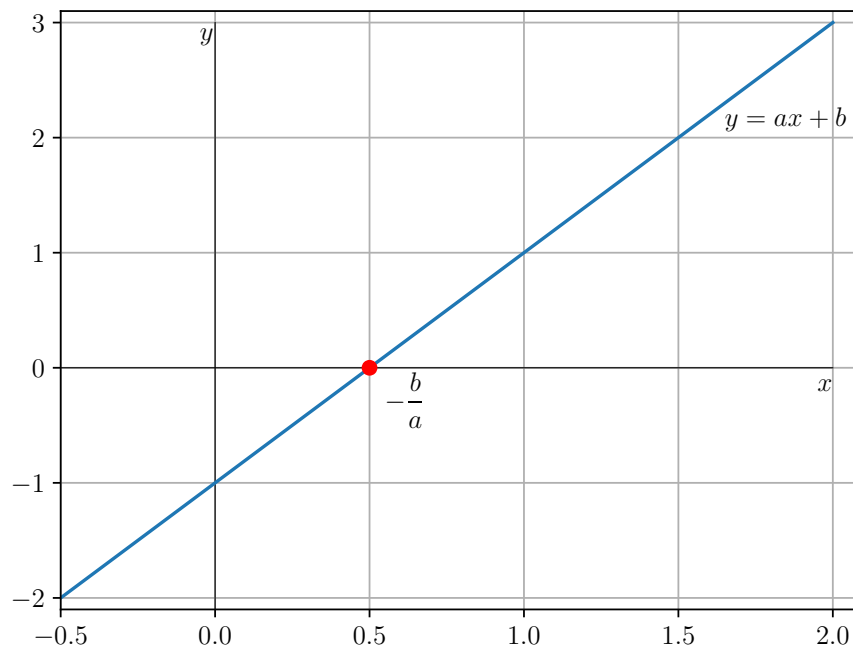


Figura 2.2: Esboço da reta de equação $y = ax + b$, com $a = 2$ e $b = -1$.

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

$$x = -\frac{b}{a} \quad (2.13)$$

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

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

$$a \leftarrow 2. \quad (2.14)$$

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

$$b \leftarrow -1. \quad (2.15)$$

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

$$x \leftarrow -\frac{b}{a}. \quad (2.16)$$

4. Imprime o valor de x .

No algoritmo acima, os identificadores 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`, `NaMe` e `NOME`. Por exemplo:

```
1 >>> a = 7
2 >>> print(A)
3 Traceback (most recent call last):
4   File "<stdin>", line 1, in <module>
5 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:

```
1 coefAngular = 2
2 coefLinear = -1
3 intercepEixoX = -coefLinear/coefAngular
4 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

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

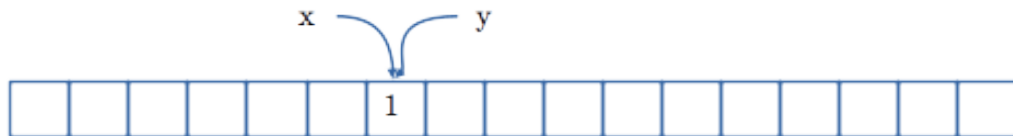
estamos criando um **objeto** na memória com valor 1 e `x` é uma referência para este dado alocado na memória. Pode-se imaginar a memória computacional como um sequência de caixinhas, de forma que `x` será a identificação da caixinha onde o valor 1 foi alocado.



Agora, quando executamos a instrução

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

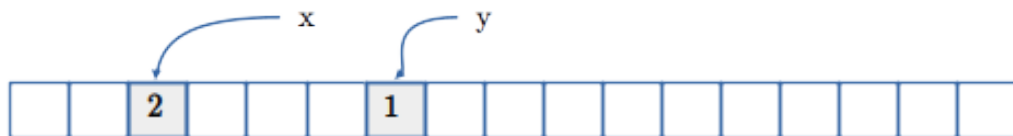
o identificador `y` passa a referenciar o mesmo local de memória de `x`.



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

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

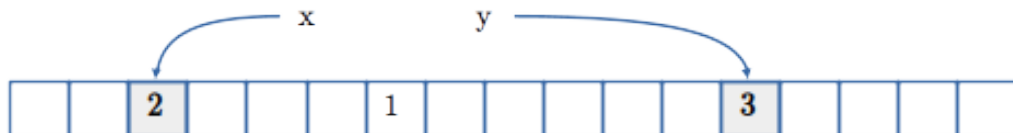
este será alocado em um novo local na memória e `x` passa a referenciar este novo local.



Ainda, se atribuirmos um novo valor para `y`

```
1 >>> 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.

```
1 >>> x = 1
2 >>> id(x)
3 139779845161200
4 >>> y = x
5 >>> id(y)
6 139779845161200
```

```
7 >>> x = 2
8 >>> id(x)
9 139779845161232
10 >>> id(y)
11 139779845161200
12 >>> y = 3
13 >>> id(y)
14 139779845161264
```

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

```
1 x = 1
2 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*).

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

Verifique!

2.3.3 Exercícios

Exercício 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) $13x$

Exercício 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.

Exercício 2.3.3. O seguinte código [Python](#) tem um erro:

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

Identifique-o e apresente uma nova versão código corrigido.

Exercício 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:

```
1 x = 1
2 y = 2
3 z = x
4 x = y
5 y = z
```

Exercício 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?

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 >>> 1 + 2
2 3
```

- **-: subtração**

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

- ***: multiplicação**

```
1 >>> 2*3
2 6
```

- **/: divisão**

```
1 >>> 5/2
2 2.5
```

- **//: divisão inteira**

```
1 >>> 5//2
2 2
```

- **?: resto da divisão**

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

A precedência das operações deve ser observada em [Python](#). Uma expressão é executada da esquerda para a direita, mas os operadores tem a seguinte precedência¹²:

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

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

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:

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

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

$$2 + 8 = 10, \quad (2.17)$$

$$10 \cdot 3 = 30, \quad (2.18)$$

$$30 \div 2 = 15, \quad (2.19)$$

$$15^2 = 225, \quad (2.20)$$

$$225 - 1 = 224. \quad (2.21)$$

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 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:

$$2^2 = 4, \quad (2.22)$$

$$8 \cdot 3 = 24, \quad (2.23)$$

$$24 \div 4 = 6, \quad (2.24)$$

$$2 + 6 = 8, \quad (2.25)$$

$$8 - 1 = 7. \quad (2.26)$$

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

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

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

```
1 >>> 2 * - 3 + 2
2 -4
```

Essa expressão é computada na seguinte ordem:

$$- 3 = -3 \quad (2.27)$$

$$2 \cdot (-3) = -6 \quad (2.28)$$

$$-6 + 2 = -4 \quad (2.29)$$

Observamos que ela seria melhor escrita da seguinte forma:

```
1 >>> 2 * -3 + 2
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* ($1 \text{ byte} = 8 \text{ bits}$).

```
1 >>> import sys
2 >>> sys.getsizeof(0)
3 24
4 >>> sys.getsizeof(1)
5 28
6 >>> sys.getsizeof(100)
7 28
8 >>> sys.getsizeof(10**9)
9 28
10 >>> sys.getsizeof(10**100)
11 32
12 >>> sys.getsizeof(10**100) #googol
13 72
```

O número *googol* 10^{100} é um número grande¹³, mas 72 *bytes* não neces-

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

sariamente. Um computador com 4 Gbytes¹⁴ 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`.

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

2.4.2 Ponto Flutuantes

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

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

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 53 bits , a característica 11 bits e 1 bit é usado para o sinal do número.

```
1 >>> import sys
2 >>> sys.float_info
3 sys.float_info(max=1.7976931348623157e+308,
4                 max_exp=1024,
5                 max_10_exp=308,
6                 min=2.2250738585072014e-308,
7                 min_exp=-1021,
8                 min_10_exp=-307,
9                 dig=15,
10                mant_dig=53,
11                epsilon=2.220446049250313e-16,
12                radix=2,
13                rounds=1)
```

Vamos denotar $\text{fl}(x)$ o número em ponto flutuante mais próximo do número decimal x dado. Quando digitamos

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

¹⁴1 Gbytes = 1024 Mbytes, 1 Mbytes = 1024 Kbytes, 1 Kbytes = 1024 bytes.

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

Notação Científica

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

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

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 31,515 pode ser representado em notação científica das seguintes formas

$$31,415 \times 10^0 = 3,1415 \times 10^1 \quad (2.32)$$

$$= 314,15 \times 10^{-1} \quad (2.33)$$

$$= 0,031415 \times 10^3, \quad (2.34)$$

entre outras tantas possibilidades.

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

```

1      >>> # 31.415 X 10^0
2      >>> 31.415e0
3      31.515
4      >>> # 3.1415 X 10^1
5      >>> 3.1415e1
6      31.515
7      >>> # 314.15 X 10^-1
8      >>> 314.15e-1
9      31.515
10     >>> # 0.031415 X 10^3
11     >>> 0.031415e3
12     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

$$d_0, d_{-1} d_{-2} d_{-3} \dots \times 10^E, \quad (2.35)$$

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 o método **format** para imprimir um número em notação científica normalizada. Por exemplo, temos

```
1 >>> x = 31.415
2 >>> print(f"{x:e}")
3 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

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

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

Exemplo 2.4.6. Estudamos os seguintes casos:

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

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

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

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

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

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

¹⁵No caso do número zero, temos $d_0 = 0$.

2.4.4 Exercícios

Exercício 2.4.1. Desenvolva um código [Python](#) para computar a interseção com o eixo das abscissas da reta de equação

$$y = 2ax - b. \quad (2.36)$$

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

Exercício 2.4.2. Assuma que o seguinte código [Python](#)

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

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

$$y = 2ax - b \quad (2.37)$$

O código acima contém um erro, qual é? Identifique-o, corrija-o e justifique sua resposta.

Exercício 2.4.3. Desenvolva um código [Python](#) para computar a média aritmética entre dois números x e y dados.

Exercício 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.

Exercício 2.4.5. Desenvolva um código [Python](#) para computar as raízes reais de uma equação quadrática

$$ax^2 + bx + c = 0. \quad (2.38)$$

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

Exercício 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?

Exercício 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

Exercício 2.4.8. Escreva os seguintes números em notação decimal:

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

Exercício 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. $(7 \times 10^{-4}) \div (2 \times 10^2)$

Exercício 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. $(1 - i)^3$

Exercício 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.

Exercício 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.

Exercício 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

$$y = a_1x + b_1 \tag{2.39}$$

$$y = a_2x + b_2, \tag{2.40}$$

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.

Resposta dos Exercícios

Exercício 2.1.1. Dica: Em [Linux](#), `$ uname --all` ou `$ cat /etc/version`.

Exercício 2.1.2. Dica: Em [Linux](#): `$ lshw`

Exercício 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.

Exercício 2.1.4.

```
1 >>> print('Olá, meu Python!')
2 Olá, meu Python!
3 >>>
```

Exercício 2.1.6. Dica: use um notebook online [Google Colab](#), [Kaggle](#) ou [Jupyter](#).

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

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

Exercício 2.3.2.

```
1 base = float(input('Informe o valor da base.\n'))
2 altura = float(input('Informe o valor da altura.\n'))
```

```
3 # cálculo da área
4 area = base * altura /2
5 print(f'Área = {area}')
```

Exercício 2.3.3. Erro: variável X não foi definida.

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

Exercício 2.3.5.

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

Exercício 2.4.1.

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

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

Exercício 2.4.3. $x = 3$ $y = 9$ $media = (x + y)/2$ `print('média = ', media)`

Exercício 2.4.4.

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


Exercício 2.4.5.

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

Exercício 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**.

Exercício 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

Exercício 2.4.8. a) 0.0028; b) 87120; c) 0,3

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

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

b) 8×10^{-2}

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

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

```
1 >>> x = 7e4 * 2e-2
2 >>> print(f'{x:e}')
3 1.400000e+03
```

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

```
1 >>> x = 7e-4 / 2e2
2 >>> print(f'{x:e}')
3 3.500000e-06
```

Exercício 2.4.10. a) $3 + 7i$

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

b) $5i$

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

c) $-2 + 16i$

```
1 >>> (2-3j) * (-4+2j)
2 (-2+16j)
```

d) $-2 - 2i$

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

Exercício 2.4.11.

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

Exercício 2.4.12.

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

Exercício 2.4.13.

```
1 # parametros
2 a1 = 1
```

```
3 a2 = -1
4 b1 = 1
5 b2 = -1
6 # ponto x de interseção
7 x_intercep = (b2-b1)/(a1-a2)
8 # ponto y de interseção
9 y_intercep = a1*x_intercep + b1
10 # imprime o resultado
11 print(f'x_i = {x_intercep:e}')
12 print(f'y_i = {y_intercep:e}')
```

Bibliografia

- [1] S. L. Banin. *Python 3 - Conceitos e Aplicações - Uma Abordagem Didática*. Saraiva, São Paulo, 2021.
- [2] T. Cormen. *Algoritmos - Teoria e Prática*. Grupo GEN, São Paulo, 2012.
- [3] T. Cormen. *Desmitificando Algoritmos*. Grupo GEN, São Paulo, 2021.
- [4] J. Grus. *Data Science do Zero*. Alta Books, Rio de Janeiro, 2021.
- [5] J. A. Ribeiro. *Introdução à Programação e aos Algoritmos*. LTC, São Paulo, 2021. Acesso pelo SABi+/UFRGS: <https://bit.ly/42Z4VFC>.
- [6] R. Wazlawick. *Introdução a Algoritmos e Programação com Python - Uma Abordagem Dirigida por Testes*. Grupo GEN, São Paulo, 2021.