

Algoritmos e Programação I

Pedro H A Konzen

21 de junho 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 Números Decimais	28
2.4.3 Números Complexos	30
2.4.4 Exercícios	31

2.5	Dados Booleanos	33
2.5.1	Operadores de Comparação	34
2.5.2	Operadores Lógicos	36
2.5.3	Exercícios	39
2.6	Sequência de Caracteres	40
2.6.1	Formatação de <i>strings</i>	43
2.6.2	Operações com <i>strings</i>	43
2.6.3	Entrada de dados	44
2.6.4	Exercícios	45
2.7	Coleção de Dados	46
2.7.1	Conjuntos: <code>set</code>	46
2.7.2	<i>N</i> -uplas: <code>tuple</code>	50
2.7.3	Listas: <code>list</code>	52
2.7.4	Dicionários: <code>dict</code>	55
2.7.5	Exercícios	57
3	Programação Estruturada	60
3.1	Estruturas de um Programa	61
3.1.1	Sequência	61
3.1.2	Ramificação	63
3.1.3	Repetição	64
3.1.4	Exercícios	67
3.2	Instruções de Ramificação	68
3.2.1	Instrução <code>if</code>	69
3.2.2	Instrução <code>if-else</code>	70
3.2.3	Instrução <code>if-elif</code>	72
3.2.4	Instrução <code>if-elif-else</code>	74
3.2.5	Múltiplos Casos	76
3.2.6	Exercícios	77
3.3	Instruções de Repetição	78
3.3.1	Instrução <code>while</code>	78
3.3.2	Instrução <code>for</code>	78
3.3.3	Exercícios	79
	Respostas dos Exercícios	80
	Referências Bibliográficas	94

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.

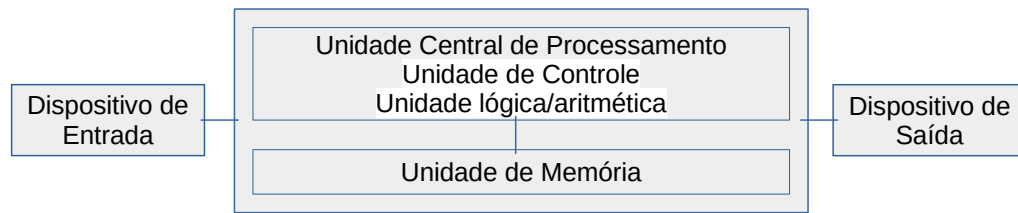


Figura 2.1: Arquitetura de computador de von Neumann.

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 aberto⁶ 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:

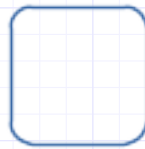
- Cada passo deve estar bem definido, i.e. não pode conter ambiguidades.
- Cada passo deve contribuir de forma efetiva na solução do problema.
- Deve ter número finito de passos que podem ser computados em um tempo finito.

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 Bernoulli⁹.

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:

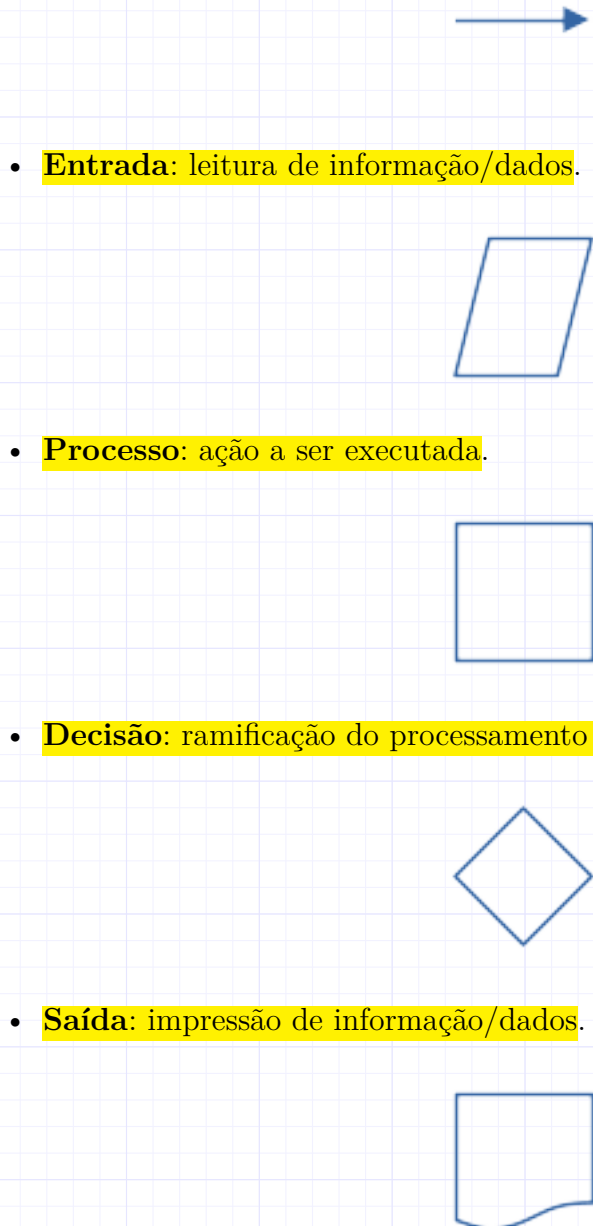
- **Terminal:** início ou final do algoritmo.



- **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 aproxi-

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

mado da raiz quadrada de um dado número x , i.e. \sqrt{x} . Consiste na iteração

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

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

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:

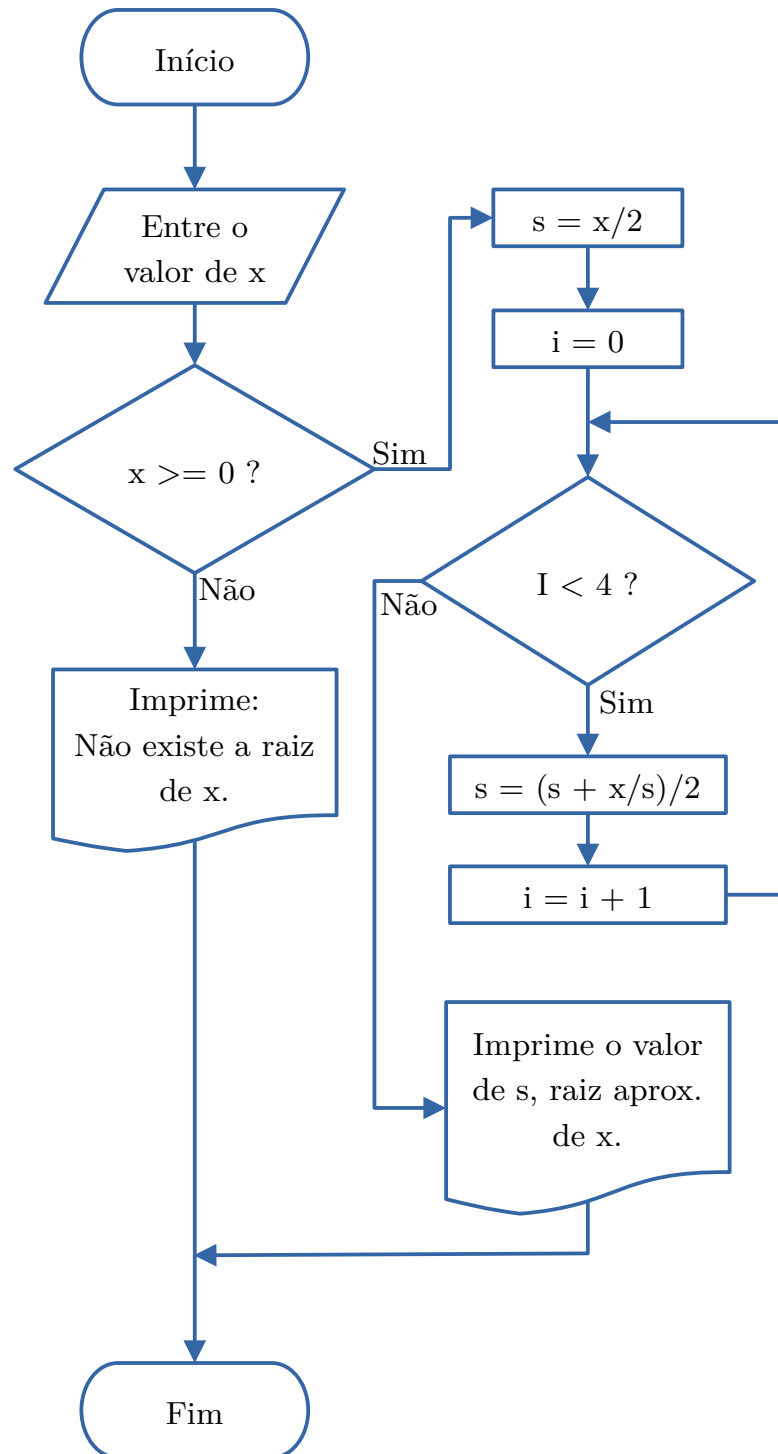
- i. $s \leftarrow (s + x/s)/2$.

- (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).

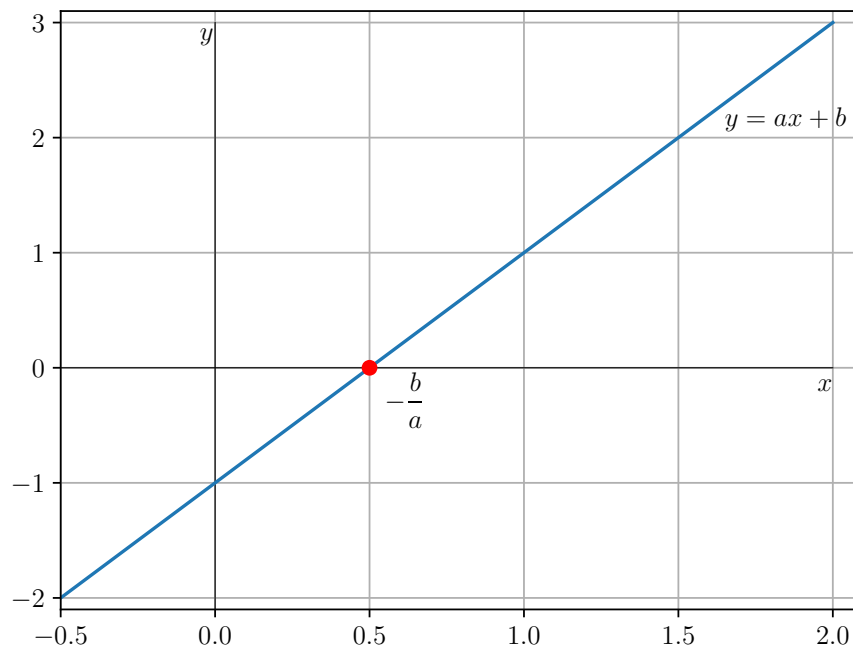


Figura 2.2: Esboço da reta de equação $y = ax + b$, com $a = 2$ e $b = -1$.

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, NoMe 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
```

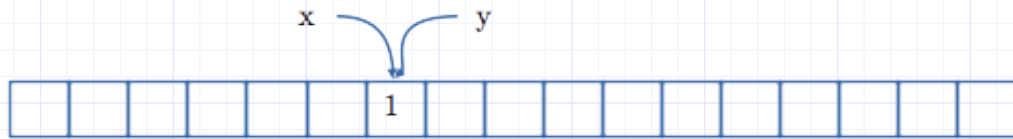
estamos criando um **objeto** na memória com valor 1 e **x** é uma referência para este dado alocado na memória. Pode-se imaginar a memória computacional como um sequência de caixinhas, de forma que **x** será a identificação da caixinha onde o valor 1 foi alocado.



Agora, quando executamos a instrução

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

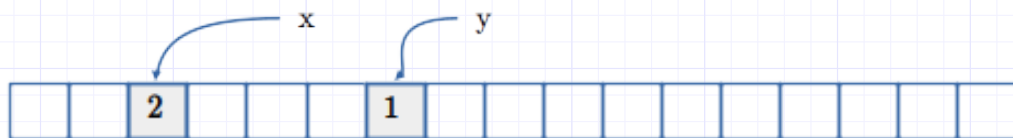
o identificador `y` passa a referenciar o mesmo local de memória de `x`.



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

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

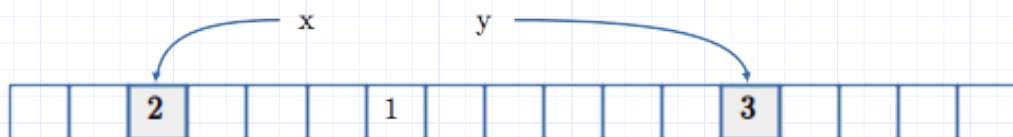
este será alocado em um novo local na memória e `x` passa a referenciar este novo local.



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 **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 precedência¹²:

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

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

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:

```
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**10)
11 32
12 >>> sys.getsizeof(10**100) #googol
13 72
```


O número [googol](#) 10^{100} é um número grande¹³, mas 72 *bytes* não necessariamente. 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 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

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

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 52 *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)
```

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

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

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

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

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

Notação Científica

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

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

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 31,415 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 de especificação de formatação¹⁶ 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)
```

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

¹⁶Consulte Subseção 2.6.1 para mais informações.

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

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

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

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

com $a = 2$ e $b = 8$. 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. $3,\bar{3} \times 10^{-1}$

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.

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 Boole¹⁷, estes dados são chamados de **booleanos**.

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

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

```
1 >>> 2/3 < 3/4
2 True
3 >>> 7/5 > 13/9
4 False
```

2.5.1 Operadores de Comparação

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

- **< : menor que**

```
1 >>> 2 < 3
2 True
```

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

```
1 >>> 4 <= 2**2
2 True
```

- **> : maior que**

```
1 >>> 5 > 7
2 False
```

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

```
1 >>> 2*5 >= 10
2 True
```

- **== : igual a**

```
1 >>> 9**2 == 81
2 True
```

- **!= : diferente de**

```
1 >>> 81 != 9**2
2 False
```

Observação 2.5.1. 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

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

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

$$(x - a)^2 + (y - b)^2 \leq r^2 \quad (2.42)$$

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 $r = 1$.

```
1 # ponto
2 x = 1
3 y = 1
4
5 # centro circunferência
6 a = 0
7 b = 0
8 # raio circunferência
9 raio = 1
10
11 # verifica se está no disco
12 v = (x-a)**2 + (y-b)**2 <= raio**2
13
14 # imprime resposta
15 print('O ponto está no disco?', v)
```

Comparação entre pontos flutuantes

Números decimais são arredondados para o número `float` (ponto flutuante) mais próximo na máquina¹⁸. Com isso, a comparação direta entre pontos flutuantes não é recomendada, em geral. Por exemplo,

```
1 >>> 0.1 + 0.2 == 0.3
2 False
```

¹⁸Consulte a Subseção 2.4.2.

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

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 aproximadamente¹⁹ 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 arredondamento de máquina) é verificando se a distância entre eles é menor que uma precisão desejada, por exemplo, 10^{-15} . No caso acima, podemos usar²⁰:

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

2.5.2 Operadores Lógicos

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

- **and : e lógico**

```
1 >>> 3 > 4 and 3 <= 4
2 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**

¹⁹Épsilon 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](#).

```

1 >>> 3 > 4 or 3 <= 4
2 True

```

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**

```

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

```

Tabela 2.3: Tabela verdade do `not`.

A	not A
True	False
False	True

Observação 2.5.2. (Ordem de 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

$$c_1 : (x - a_1)^2 + (y + b_1)^2 = r_1^2, \quad (2.43)$$

$$c_2 : (x - a_2)^2 + (y + b_2)^2 = r_2^2, \quad (2.44)$$

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

$$c_1 : (a_1, b_1) = (0, 0), r_1 = 1 \quad (2.45)$$

e a circunferência c_2 tem

$$c_2 : (a_2, b_2) = (1, 1), r_2 = 1, \quad (2.46)$$

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

```
1  # circunferência c1
2  a1 = 0
3  b1 = 0
4  r1 = 1
5
6  # circunferência c2
7  a2 = 1
8  b2 = 1
9  r2 = 1
10
11 # ponto obj
12 x = 0.5
13 y = 0.5
14
15 # está em c1?
16 em_c1 = (x-a1)**2 + (y-b1)**2 <= r1**2
17
18 # está em c2?
19 em_c2 = (x-a2)**2 + (y-b2)**2 <= r2**2
20
21 # está em c1 e c2?
22 resp = em_c1 and em_c2
23 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

Exercício 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) $2,7128 \geq 2 + \frac{2}{3}$

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

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

Exercício 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 .

Exercício 2.5.4. Considere o plano cartesiano $x - y$. Desenvolva um código que verifique se um ponto (x, y) dado está entre as curvas $y = (x - 1)^3$ e o eixo das abscissas²¹. Verifique seu código para diferentes pontos (x, y) .

²¹Eixo x .

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

- a) $A \text{ or } A == A$
- b) $A \text{ and not}(A) == \text{True}$
- c) $A \text{ or } (A \text{ and } B) == A$
- d) $\text{not}(A \text{ and } B) == \text{not}(A) \text{ or } \text{not}(B)$
- e) $\text{not}(A \text{ or } B) == \text{not}(A) \text{ and } \text{not}(B)$

Exercício 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.

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,

```
1 >>> s = 'Olá, mundo!'
2 >>> print(s)
3 Olá, mundo!
4 >>> type(s)
5 <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,

O	l	á	,		m	u	n	d	o	!
0	1	2	3	4	5	6	7	8	9	10

(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

```
1 >>> len(s)
2 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,

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

Podemos, ainda, acessar fatias²² da sequência usando o operador :,²³ por exemplo,

```
1 >>> s[:3]
2 '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,

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

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

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

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

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

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

- **aspas simples**

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

- **aspas duplas**

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

²²Em inglês, *slice*.

²³`x[start:stop:step]`, padrão `start=0`, `stop=len(x)`, `step=1`.

- **aspas triplas**²⁴

```
1 >>> '''
2 ... peritem
3 ... "diversas"
4 ... linhas
5 ... '''
6 '\nperitem\n  "diversas"\nlinhas\n'
7 >>> """
8 ... peritem
9 ... 'diversas '
10 ... linhas
11 ... """
12 "\nperitem\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**

```
1 >>> print('Uma nova\nlinha')
2 Uma nova
3 linha
```

- **'t' : tabulação**

```
1 >>> print('Uma nova\n\t linha com tabulação')
2 Uma nova
3         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,

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

²⁴'n' é o caractere que indica uma nova linha (em inglês, *newline*).

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,

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

Há várias especificações de formatação disponíveis²⁵:

- **'d' : número inteiro**

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

- **'f' : número decimal**

```
1 >>> print(f'13/7 é aproximadamente {13/7:.3f}')
2 13/7 é aproximadamente 1.857
```

- **'e' : notação científica normalizada**

```
1 >>> print(f'103/7 é aproximadamente {103/7:.3e}')
2 103/7 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**

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

- **`*` : repetição**

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


```
1 >>> 'ha'*3
2 'hahaha'
```

- **in : pertence**

```
1 >>> 'mar' in 'amarelo'
2 True
```

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,

```
1 >>> # int -> str
2 >>> str(101)
3 '101'
4 >>> # str -> int
5 >>> int('23')
6 23
7 >>> # int -> float
8 >>> float(1)
9 1.0
10 >>> # float -> int
11 >>> int(-2.9)
```

12 -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 usuário(o).

```
1 # entrada de dados
2 base = float(input('Entre com o valor da base:\n\t'))
3 altura = float(input('Entre com o valor da altura:\n\t'))
4
5 # cálculo da área
6 area = base*altura/2
7
8 # imprime a área
9 print(f'Área do triangulo de ')
10 print(f'\t base = {base:e}')
11 print(f'\t altura = {altura:e}')
12 print(f'é igual a {area:e}')
```

2.6.4 Exercícios

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

- Extrair a quarta letra da palavra.
- Extrair a *substring*²⁶ formada pelas quatro primeiras letras da palavra.
- Extrair a *string* formadas pelas segunda, quarta e sexta letras (nesta ordem) da palavra.
- Extrair a *string* formadas pelas penúltima e quarta letras (nesta ordem) da palavra.

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

Exercício 2.6.2. Considere o seguinte código

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

Sem implementá-lo, o que é impresso?

Exercício 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).

Exercício 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.

Exercício 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 .

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 uma 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:

```
1 >>> a = {1, -3.7, 'amarelo'}
2 >>> type(a)
```

```
3 <class 'set'>
4 >>> a
5 {'amarelo', 1, -3.7}
```

Observamos que a **ordem dos elementos é arbitrária**, uma vez que `set` é uma coleção 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:

```
1 >>> b = set()
2 >>> type(b)
3 <class 'set'>
4 >>> b
5 set()
```

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

```
1 >>> len(a)
2 3
3 >>> len(b)
4 0
```

Operadores de comparação

Os seguintes operadores de comparação estão disponíveis para `sets`:

- **`x in a` : pertence**

Verifica se $x \in a$.

```
1 >>> a = {1, -3.7, 'amarelo'}
2 >>> 1 in a
3 True
4 >>> 'mar' in a
5 False
```

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

Verifica se $a = b$.

```
1 >>> a == a
2 True
```

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

Verifica se $a \neq b$.

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

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

Verifica se $a \subseteq b$.

```
1 >>> b <= a
2 True
```

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

Verifica se $a \subsetneq b$.

```
1 >>> a < a
2 False
3 >>> b < a
4 True
```

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

Verifica se $a \supseteq b$.

```
1 >>> a >= b
2 True
```

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

Verifica se $a \supsetneq b$.

```
1 >>> a > b
2 True
3 >>> b > b
4 False
```

Operações com conjuntos

Em [Python](#), as seguintes operações com conjuntos estão disponíveis:

Notas de Aula - Pedro Konzen */* Licença CC-BY-SA 4.0

- **a | b : união**

Retorna o **set** equivalente a

$$a \cup b := \{x : x \in a \vee x \in b\} \quad (2.48)$$

```
1 >>> a = {1, -3.7, 'amarelo'}
2 >>> b = {'mar', -5}
3 >>> a | b
4 {1, 'amarelo', 'mar', -5, -3.7}
```

- **a & b : interseção**

Retorna o **set** equivalente a

$$a \cap b := \{x : x \in a \wedge x \in b\} \quad (2.49)$$

```
1 >>> a = {1, -3.7, 'amarelo'}
2 >>> b = {'mar', 1, -3.7, -5}
3 >>> a & b
4 {1, -3.7}
```

- **- : diferença**

Retorna o **set** equivalente a

$$a \setminus b := \{x : x \in a \wedge x \notin b\} \quad (2.50)$$

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

- **^ : diferença simétrica**

Retorna o **set** equivalente a

$$a \Delta b := (a \setminus b) \cup (b \setminus a) \quad (2.51)$$

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

2.7.2 *N*-uplas: `tuple`

Em `Python`, `tuple` é uma sequência de objetos, indexada e imutável. São similares as *n*-uplas²⁷ em matemática. A alocação é feita com uso de parênteses e os elementos separados por vírgula, por exemplo,

```
1 >>> a = (1, -3.7, 'amarelo', -5)
2 >>> type(a)
3 <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,

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

Os itens são indexados como segue

$$\begin{pmatrix} 1, -3.7, 'amarelo', -5, \{-3, 1\} \\ \begin{smallmatrix} 0 & 1 & 2 & 3 & 4 \\ -5 & -4 & -3 & -2 & -1 \end{smallmatrix} \end{pmatrix} \quad (2.52)$$

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

```
1 >>> a[2]
2 'amarelo'
3 >>> a[-1]
4 {1, -3}
```

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

```
1 >>> a[:2]
2 (1, -3.7)
3 >>> a[1:5:2]
4 (-3.7, -5)
5 >>> a[::-1]
6 ({1, -3}, -5, 'amarelo', -3.7, 1)
```

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

Operações com `tuples`

Os mesmos operadores de comparação para `sets` estão disponíveis para `tuples` (consulte a Subseção 2.7.1). Por exemplo,

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

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

Há, também, operadores para a concatenação e repetição:

- **`+` : concatenação**

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

- **`*` : repetição**

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

```
1 >>> x = 1
2 >>> y = 2
3 >>> z = x
4 >>> x = y
5 >>> y = z
6 >>> x, y
7 (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:

```
1 >>> x, y = y, x
2 >>> x, y
3 (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 `[]`:

```
1 >>> a = [1, -3.7, 'amarelo', -5, (-3,1)]
2 >>> type(a)
3 <class 'list'>
4 >>> a[2]
5 'amarelo'
6 >>> a[1:2]
7 [-3.7, -5]
```

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

$$v = (v_1, v_2, v_3), \quad (2.53)$$

$$w = (w_1, w_2, w_3). \quad (2.54)$$

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

$$v \cdot w := v_1 w_1 + v_2 w_2 + v_3 w_3. \quad (2.55)$$

O seguinte código, aloca os vetores

$$v = (-1, 2, 1), \quad (2.56)$$

$$w = (3, -1, 4) \quad (2.57)$$

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

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

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

$$A = \begin{bmatrix} -1 & 1 \\ 1 & 3 \end{bmatrix} \quad (2.58)$$

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

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

Com isso, podemos obter a segunda linha da matriz com

```
1 >>> A[1]
2 [1, 3]
```

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

```
1 >>> A[1][0]
2 1
```

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

```
1 >>> a = [1, 2]
2 >>> b = [3, 4]
3 >>> a + b
4 [1, 2, 3, 4]
5 >>> 2*a
6 [1, 2, 1, 2]
7 >>> a <= a
8 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

```
1 >>> a = [1, -3.7, 'amarelo', -5, (-3,1)]
2 >>> a[1] = 7.5
3 >>> a
4 [1, 7.5, 'amarelo', -5, (-3, 1)]
5 >>> a[1:3] = ['mar', -2.47]
6 >>> a
7 [1, 'mar', -2.47, -5, (-3, 1)]
8 >>> a[:2] = 7
```

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

- `del : deleta elemento(s)`

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

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

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

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

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

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

```
1 >>> del a[-1]
2 >>> a.extend([2,1])
3 >>> a
4 [-2.47, 'azul', -5, (-3, 1), 2, 1]
5 >>> a += [3]
6 >>> a
7 [-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

```
1 >>> a = [1, 2, 3]
2 >>> b = a
3 >>> b[1] = 4
4 >>> a
5 [1, 4, 3]
```

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

```
1 >>> a = [1, 2, 3]
2 >>> b = a.copy()
3 >>> b[1] = 4
4 >>> a
5 [1, 2, 3]
6 >>> b
7 [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:

```
1 >>> x = {'nome': 'Fulane', 'idade': 19}
2 >>> x
3 {'nome': 'Fulane', 'idade': 19}
4 >>> y = {3: 'número inteiro', 3.14: 'pi', 2.71: 2}
5 >>> y
6 {3: 'número inteiro', 3.14: 'pi', 2.71: 2}
7 >>> d = {}
8 >>> type(d)
9 <class 'dict'>
```

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

```
1 >>> x['idade']
```

```
2 19
3 >>> y[3]
4 'número inteiro'
```

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

```
1 >>> list(x)
2 ['nome', 'idade']
3 >>> list(y)
4 [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 triângulo

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

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

```
1 >>> tria['B']
2 (1, 0)
```

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

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

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

```
1 >>> x['altura'] = 171
2 >>> x
3 {'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. Agora, 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

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

Segue nosso código:

```

1  # vértices do triangulo
2  tria = {'A': (0,0), 'B': (1,0), 'C': (0,1)}
3  # aresta AB
4  tria['AB'] = ((tria['B'][0] - tria['A'][0])**2 \
5              + (tria['B'][1] - tria['A'][1])**2)**0.5
6  # aresta BC
7  tria['BC'] = ((tria['C'][0] - tria['B'][0])**2 \
8              + (tria['C'][1] - tria['B'][1])**2)**0.5
9  # aresta AC
10 tria['AC'] = ((tria['C'][0] - tria['A'][0])**2 \
11              + (tria['C'][1] - tria['A'][1])**2)**0.5
12 # novo dicionário
13 print(tria)

```

2.7.5 Exercícios

Exercício 2.7.1. Crie um código que aloque os seguintes conjuntos

$$A = \{1,4,7\} \quad (2.60)$$

$$B = \{1,3,4,5,7,8\} \quad (2.61)$$

e verifique as seguintes afirmações:

a) $A \supset B$

b) $A \subset B$

c) $B \not\supset A$

d) $A \subsetneq B$

Exercício 2.7.2. Crie um código que aloque os seguintes conjuntos

$$A = \{-3, -1, 0, 1, 6, 7\} \quad (2.62)$$

$$B = \{-4, 1, 3, 5, 6, 7\} \quad (2.63)$$

$$C = \{-5, -3, 1, 2, 3, 5\} \quad (2.64)$$

e, então, compute as seguintes operações:

a) $A \cap B$

b) $C \cup B$ c) $C \setminus A$ d) $B \cap (A \cup C)$

Exercício 2.7.3. O produto cartesiano²⁸ de um conjunto X com um conjunto Y é o seguinte conjunto de pares ordenados

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

Crie um código que aloque os conjuntos

$$X = \{-2, 1, 3\}, Y = \{5, -1, 2\} \quad (2.66)$$

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

Exercício 2.7.4. A sequência de Fibonacci²⁹ $(f_n)_{n \in \mathcal{N}}$ é definida por

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

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

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

$$\mathbf{v} = (-1, 0, 2), \quad (2.68)$$

$$\mathbf{w} = (3, 1, 2) \quad (2.69)$$

e computar:

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

²⁸René Descartes, 1596 - 1650, matemático e filósofo francês. Fonte: [Wikipédia](#).

²⁹Leonardo Fibonacci, 1170 - 1250, matemático italiano. Fonte: [Wikipédia](#).

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

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

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

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

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

$$A = \begin{bmatrix} 1 & -1 \\ 2 & 3 \end{bmatrix} \quad (2.70)$$

e imprima o determinante de A , i.e.

$$|A| := a_{1,1}a_{2,2} - a_{1,2}a_{2,1}. \quad (2.71)$$

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

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

e o vetor

$$\mathbf{x} = (-1, 2, 1). \quad (2.73)$$

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

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.

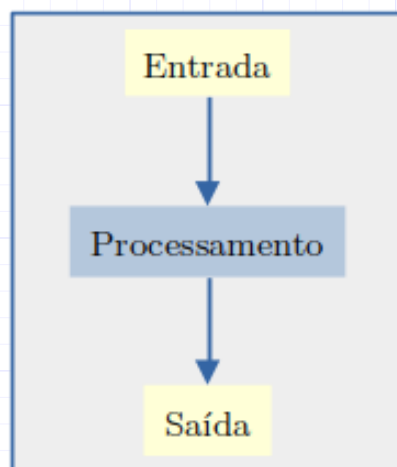


Figura 3.1: Bloco de processamento.

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.**

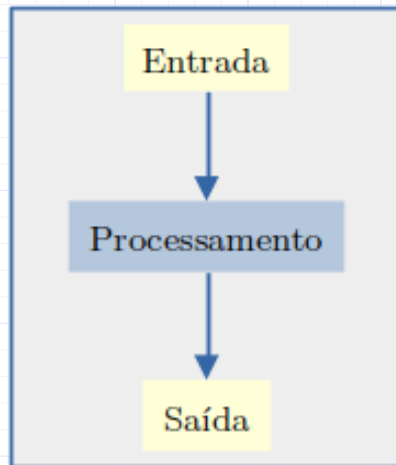


Figura 3.2: Bloco de processamento.

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.

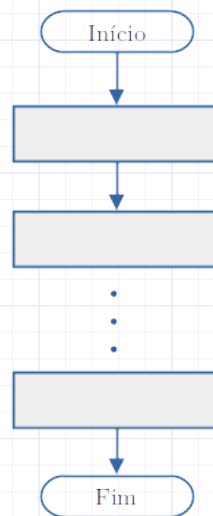


Figura 3.3: Estrutura de sequência de blocos.

Exemplo 3.1.1. O seguinte código computa 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 a 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.

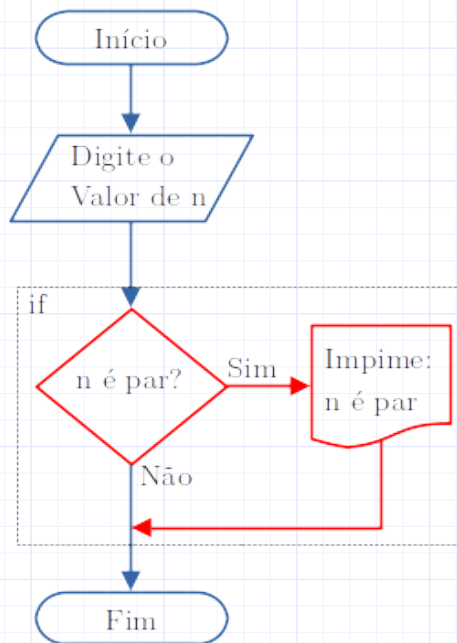


Figura 3.4: Fluxograma de uma estrutura de ramificação.

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.

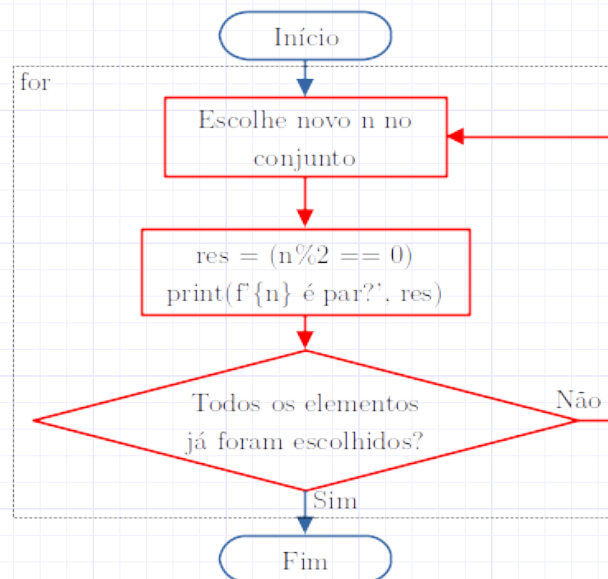


Figura 3.5: Fluxograma de uma estrutura de repetição do tipo `for`.

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.

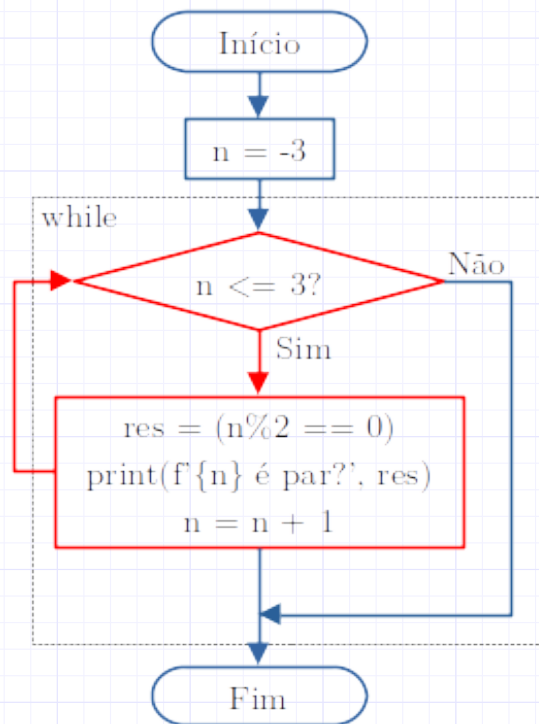


Figura 3.6: Fluxograma de uma estrutura de repetição do tipo `while`.

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

3.1.4 Exercícios

Exercício 3.1.1. Seja a reta de equação

$$y = ax + b. \quad (3.1)$$

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.

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


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

Exercício 3.1.2. Seja a reta de equação

$$y = ax + b. \quad (3.2)$$

Faça um fluxograma de um programa em que a(o) usuá(ri)a 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.

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

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

$$A = \{-4, -3, -2, -1, 0, 1, 2, 3, 4\}. \quad (3.3)$$

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

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

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

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

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.

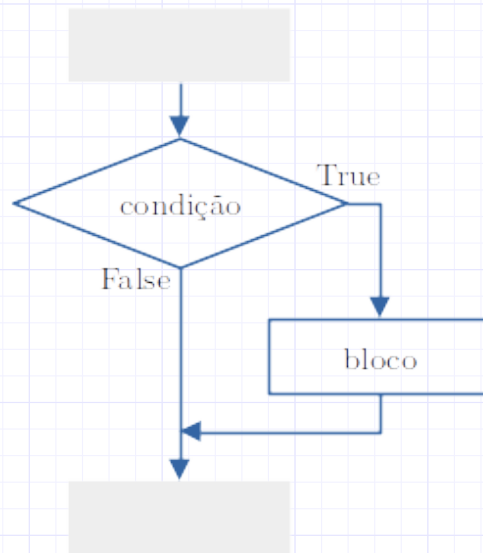


Figura 3.7: Fluxograma de uma ramificação `if`.

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

$$p(x) = ax^2 + bx + c. \quad (3.4)$$

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

```
1  # entrada de dados
2  a = float(input('Digite o valor de a:\n'))
3  b = float(input('Digite o valor de b:\n'))
4  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}')
```

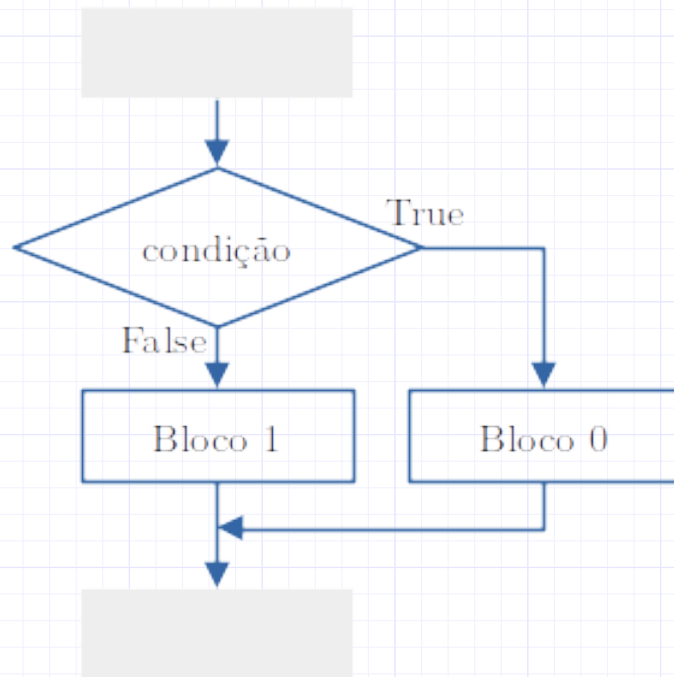
Escopo de variáveis

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:

```
1 bloco_anterior
2 if (condição):
3     bloco_0
4 else:
5     bloco_1
6 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

$$p(x) = ax^2 + bx + c. \quad (3.5)$$

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

```

1  # entrada de dados
2  a = float(input('Digite o valor de a:\n'))
3  b = float(input('Digite o valor de b:\n'))
4  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     x1 = (-b - delta**0.5)/(2*a)
12     x2 = (-b + delta**0.5)/(2*a)
13     print(f'x_1 = {x1}')
14     print(f'x_2 = {x2}')
15 else:
16     print('Não tem raízes reais.')
```

Instrução **if-else** em linha

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

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

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

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

O seguinte código, computa o valor absoluto¹ de um número dado pela(o) usuária(o).

```

1 x = float(input('Digite o valor de x:\n'))
2 abs_x = x if (x>=0) else -x
3 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.

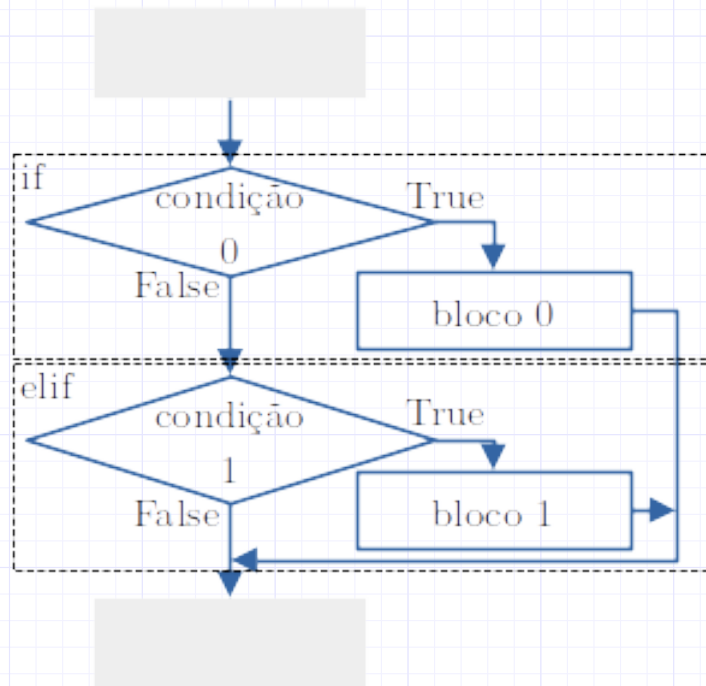


Figura 3.9: Fluxograma de uma ramificação `if-elif`.

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 (l`istinline`+`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

$$p(x) = ax^2 + bx + c. \quad (3.7)$$

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

```
1  # entrada de dados
2  a = float(input('Digite o valor de a:\n'))
3  b = float(input('Digite o valor de b:\n'))
4  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     x1 = (-b - delta**0.5)/(2*a)
12     x2 = (-b + delta**0.5)/(2*a)
13     print('Raízes reais distintas:')
14     print(f'x_1 = {x1}')
15     print(f'x_2 = {x2}')
16 elif (delta == 0):
17     print('Raiz dupla:')
18     x = -b/(2*a)
19     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:

```
1  bloco_anterior
2  if (condição_0):
3      bloco_0
4  elif (condição_1):
5      bloco_1
6  else:
7      bloco_2
8  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

$$p(x) = ax^2 + bx + c. \quad (3.8)$$

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

```
1  # entrada de dados
2  a = float(input('Digite o valor de a:\n'))
3  b = float(input('Digite o valor de b:\n'))
4  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('Raízes reais distintas:')
15     print(f'x_1 = {x1}')
16     print(f'x_2 = {x2}')
17 elif (delta == 0):
18     # raiz dupla
19     x = -b/(2*a)
20     print('Raiz dupla:')
21     print(f'x_1 = x_2 = {x}')
22 else:
23     # raízes complexas
24     # parte real
25     rea = -b/(2*a)
26     # parte imaginária
27     img = (-delta)**0.5/(2*a)
28     x1 = rea - img*1j
29     x2 = rea + img*1j
30     print('Raízes complexas:')
31     print(f'x_1 = {x1}')
32     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:

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

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

Conforme entradas dadas por usuá(ri)a(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.

```

1  # entrada de dados
2  print('c1: (x-a1)**2 + (y-b1)**2 = r1')
3  a1 = float(input('Digite o valor de a1:\n'))
4  b1 = float(input('Digite o valor de b1:\n'))
5  r1 = float(input('Digite o valor de r1:\n'))
6  print('c2: (x-a2)**2 + (y-b2)**2 = r1')
7  a2 = float(input('Digite o valor de a2:\n'))
8  b2 = float(input('Digite o valor de b2:\n'))
9  r2 = float(input('Digite o valor de r2:\n'))
10 print('Ponto de interesse (x,y).')
11 x = float(input('Digite o valor de x:\n'))
12 y = float(input('Digite o valor de y:\n'))
13
14 # pertence ao disco c1?
15 c1 = (x-a1)**2 + (y-b1)**2 <= r1
16 # pertence ao disco c2?
17 c2 = (x-a2)**2 + (y-b2)**2 <= r2
18
19 # imprime resultado
20 if (c1 and c2):
21     print(f'({x}, {y}) pertence à interseção dos discos.')
22 elif (c1):
23     print(f'({x},{y}) pertence ao disco c1.')
24 elif (c2):
25     print(f'({x},{y}) pertence ao disco c2.')
26 else:
27     print(f'({x},{y}) não pertence aos discos.')
```

3.2.6 Exercícios

Exercício 3.2.1. Seja a equação de reta

$$ax + b = 0. \quad (3.11)$$

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 $a = 0$.

Exercício 3.2.2. Considere o seguinte código.

```
1 n = int(input('Digite um número inteiro:\n'))
2 if (n % 2 == 0):
3     m = 1
4 n = n + m
5 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.

Exercício 3.2.3. Considere a equação da circunferência

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

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 .

Exercício 3.2.4. Sejam informadas por `usuária(o)` os coeficientes das retas

$$r_1 : a_1x + b_1 = 0, \quad (3.13)$$

$$r_2 : a_2x + b_2 = 0. \quad (3.14)$$

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

Exercício 3.2.5. Refaça o código do Exercício 3.2.4 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.

Exercício 3.2.6. Sejam a parábola de equação

$$a_1x^2 + a_2x + a_3 = 0 \quad (3.15)$$

e a reta

$$b_1x + b_2 = 0. \quad (3.16)$$

Conforme os coeficientes dados por usuário(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.

Exercício 3.2.7. Com dados informados por usuário(o), sejam as circunferências de equações

$$c_1 : (x - a_1)^2 + (y - b_1)^2 = r_1^2, \quad (3.17)$$

$$c_2 : (x - a_2)^2 + (y - b_2)^2 = r_2^2. \quad (3.18)$$

Desenvolva um código que informe a(o) usuário(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, $c_1 \cap c_2 = \emptyset$.

Exercício 3.2.8. Crie uma calculadora simples. A(o) usuário(o) entra com dois números decimais x e y e uma das seguintes operações: $+$, $-$, $*$ ou $/$. Então, o código imprime o resultado da operação.

3.3 Instruções de Repetição

[[tag:construcao]]

3.3.1 Instrução **while**

[[tag:construcao]]

3.3.2 Instrução **for**

[[tag:construcao]]

3.3.3 Exercícios

[[tag:construcao]]

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.

```
1 x = 3
2 y = 9
3 media = (x + y)/2
4 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) $0,\bar{3}$

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}')
```

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

Exercício 2.5.2.

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

Exercício 2.5.3.

```
1  # quadrado
2  ladoQuad = 1
3  areaQuad = ladoQuad**2
4
5  # aprox pi
6  pi = 3.14159
7
8  # circunferência
9  raioCirc = 1
10 areaCirc = pi * raioCirc**2
11
12 # verifica
13 resp = areaQuad < areaCirc
14 print('Área do quadrado é menor que da circunferência?')
```

```
15 print(resp)
```

Exercício 2.5.4.

```
1 # ponto
2 x = 2
3 y = 0.5
4
5 # y >= 0 e y <= f(x) ?
6 resp1 = y >= 0 and y <= (x-1)**3
7 # y >= f(x) e y <= 0 ?
8 resp2 = y >= (x-1)**3 and y <= 0
9
10 # conclusão
11 print("0 ponto está entre as curvas?")
12 print(resp1 or resp2)
```

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

Exercício 2.5.6. (A or B) and not(A and B)

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

Exercício 2.6.2. trator

Exercício 2.6.3.

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

Exercício 2.6.4.

```
1 lado = float(input('Digite o lado (em cm) do quadrado:\n\t'))
2 area = lado**2/100**2
3 print(f'0 quadrado de lado {lado:e} cm tem área {area:.2f} m.')
```

Exercício 2.6.5.

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

Exercício 2.7.1.

```
1 A = {1,4,7}
2 B = {1,3,4,5,7,8}
3 # a)
4 a = A >= B
5 print(f"a) A>=B: {a}")
6 # b)
7 b = A <= B
8 print(f"b) A<=B: {b}")
9 # c)
10 c = not(B >= A)
11 print(f"c) not(A>=B): {c}")
12 # d)
13 d = A < B
14 print(f"d) A<B: {d}")
```

Exercício 2.7.2.

```
1 A = {-3,-1,0,1,6,7}
2 B = {-4,1,3,5,6,7}
3 C = {-5,-3,1,2,3,5}
4 # a)
5 a = A & B
6 print(f"a)\n A&B = {a}")
7 # b)
8 b = C | B
9 print(f"b)\n A|B = {b}")
10 # c)
11 c = C - A
12 print(f"c)\n C-A = {c}")
13 # d)
14 d = B & (A | C)
15 print(f"d)\n B&(A|C) = {d}")
```

Exercício 2.7.3.

```
1 X = {-2,1,3}
2 Y = {5,-1,2}
3 XxY = {(-2,5), (-2,-1), (-2,2), \
4       (1,5), (1,-1), (1,2), \
5       (3,5), (3,-1), (3,2)}
6 print(f'#(X x Y) = {len(XxY)}')
```

Exercício 2.7.4.

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

Exercício 2.7.5.

```
1 v = [-1, 0, 2]
2 w = [3, 1, 2]
3 # a)
4 vpw = [v[0] + w[0],
5        v[1] + w[1],
6        v[2] + w[2]]
7 print(f'a) v+w = {vpw}')
8 # b)
9 vmw = [v[0] - w[0],
10        v[1] - w[1],
11        v[2] - w[2]]
12 print(f'b) v-w = {vmw}')
13 # c)
14 vdw = v[0]*w[0] + \
15        v[1]*w[1] + \
16        v[2]*w[2]
17 print(f'c) v.w = {vdw}')
18 # d)
19 norm_v = (v[0]**2 + \
20           v[1]**2 + \
```

```
21         v[2]**2)**0.5
22 print(f'd) ||v|| = {norm_v:.2f}')
23 # e)
24 norm_vmw = (vmw[0]**2 + \
25             vmw[1]**2 + \
26             vmw[2]**2)**0.5
27 print(f'e) ||v-w|| = {norm_vmw:.2f}')
```

Exercício 2.7.6.

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

Exercício 2.7.7.

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

Exercício 3.1.1.

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

Exercício 3.1.2. Dica: consulte o Exemplo 3.1.2.**Exercício 3.1.3.**

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

Exercício 3.1.4. Dica: consulte o Exemplo 3.1.3.

Exercício 3.1.5.

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

Exercício 3.1.6. Dica: consulte o Exemplo 3.1.4.

Exercício 3.1.7.

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

Exercício 3.2.1.

```
1 # entrada de dados
2 a = float(input('Digite o valor de a:\n'))
3 b = float(input('Digite o valor de b:\n'))
4
5 # computação
6 if (a != 0):
7     x = -b/a
8     y = a*x + b
9     print(f'Intercepta eixo-x em: ({x}, {y}).')
```

Exercício 3.2.2.

```
1 n = int(input('Digite um número inteiro:\n'))
2 m = 0
3 if (n % 2 == 0):
4     m = 1
5 n = n + m
6 print(n)
```

Exercício 3.2.3.

```
1 # entrada de dados
2 print('Circunferência c:')
3 a = float(input('Digite o valor de a:\n'))
4 b = float(input('Digite o valor de b:\n'))
5 r = float(input('Digite o valor de r:\n'))
6 print('Ponto (x, y):')
7 x = float(input('Digite o valor de x:\n'))
8 y = float(input('Digite o valor de y:\n'))
9
10 # resultado
11 if ((x-a)**2 + (y-b)**2 <= r**2):
12     print(f'({x}, {y}) pertence ao disco.')
13 else:
14     print(f'({x}, {y}) não pertence ao disco.')
```

Exercício 3.2.4.

```
1 # entrada de dados
2 print('r1: a1*x + b1 = 0')
3 a1 = float(input('Digite o valor de a1:\n'))
4 b1 = float(input('Digite o valor de b1:\n'))
5 print('r2: a2*x + b2 = 0')
6 a2 = float(input('Digite o valor de a2:\n'))
7 b2 = float(input('Digite o valor de b2:\n'))
8
9 # resultado
10 if (a1 == a2):
11     print('r1 // r2')
12 else:
```

```
13     x = (b1-b2)/(a2-a1)
14     y = a1*x + b1
15     print('Ponto de interseção: ({x}, {y}).')
```

Exercício 3.2.5.

```
1  # entrada de dados
2  print('r1: a1*x + b1 = 0')
3  a1 = float(input('Digite o valor de a1:\n'))
4  b1 = float(input('Digite o valor de b1:\n'))
5  print('r2: a2*x + b2 = 0')
6  a2 = float(input('Digite o valor de a2:\n'))
7  b2 = float(input('Digite o valor de b2:\n'))
8
9  # resultado
10 if (a1 == a2):
11     if (b1 == b2):
12         print('r1 = r2')
13     else:
14         print('r1 // r2 e r1 != r2')
15 else:
16     x = (b1-b2)/(a2-a1)
17     y = a1*x + b1
18     print('Ponto de interseção: ({x}, {y}).')
```

Exercício 3.2.6.

```
1  # entrada de dados
2  print('Coeficientes da parábola')
3  print('a1*x**2 + a2*x + a3 = 0')
4  a1 = float(input('Digite o valor de a1:\n'))
5  a2 = float(input('Digite o valor de a2:\n'))
6  a3 = float(input('Digite o valor de a3:\n'))
7
8  print('Coeficientes da reta')
9  print('b1*x + b2 = 0')
10 b1 = float(input('Digite o valor de b1:\n'))
11 b2 = float(input('Digite o valor de b2:\n'))
12
13 # discriminante da equação
```



```
14 #  $a_1x^2 + (a_2-b_1)x + (a_3-b_2) = 0$ 
15 delta = (a2-b1)**2 - 4*a1*(a3-b2)
16
17 # ponto(s) de interseção
18 if (delta == 0):
19     x = (b1-a2)/(2*a1)
20     y = b1*x + b2
21     print('Ponto de interseção:')
22     print(f'({x}, {y})')
23 elif (delta > 0):
24     x1 = ((b1-a2) - delta**2)/(2*a1)
25     y1 = b1*x1 + b2
26     x2 = ((b1-a2) + delta**2)/(2*a1)
27     y2 = b1*x2 + b2
28     print('Pontos de interseção:')
29     print(f'({x1}, {y1}), ({x2}, {y2})')
30 else:
31     print('Não há ponto de interseção.')
```

Exercício 3.2.7.

```
1 # entrada de dados
2 print('c1:  $(x-a_1)^2 + (y-b_1)^2 = r_1^2$ ')
3 a1 = float(input('Digite o valor de a1:\n'))
4 b1 = float(input('Digite o valor de b1:\n'))
5 r1 = float(input('Digite o valor de r1:\n'))
6 print('c2:  $(x-a_2)^2 + (y-b_2)^2 = r_2^2$ ')
7 a2 = float(input('Digite o valor de a2:\n'))
8 b2 = float(input('Digite o valor de b2:\n'))
9 r2 = float(input('Digite o valor de r2:\n'))
10
11 # verificações
12 if ((a1==a2) and (b1==b2)) and (r1==r2)):
13     print('c1 = c2')
14 else:
15     # distância entre os centros
16     dist = ((a2-a1)**2 + (b2-b1)**2)**0.5
17     if (abs(dist - (r1+r2)) < 1e-15):
18         print('c1 & c2 têm um único ponto de interseção.')
19     elif (dist < r1+r2):
```

```
20     print('c1 & c2 têm dois pontos de interseção.')
650 21     else:
22     print('c1 & c2 não tem ponto de interseção.')
```

Exercício 3.2.8.

```
1  # entrada de dados
2  x = float(input('Digite o valor de x:\n'))
550 3  op = input('Digite uma das operações +, -, * ou /:\n')
4  y = float(input('Digite o valor de y:\n'))
5
6  # calcula
500 7  if (op == '+'):
8      print(f'{x} ' + op + f' {y} = {x+y}')
9  elif (op == '-'):
450 10     print(f'{x} ' + op + f' {y} = {x-y}')
11  elif (op == '*'):
12     print(f'{x} ' + op + f' {y} = {x*y}')
13  elif (op == '/'):
400 14     print(f'{x} ' + op + f' {y} = {x/y}')
15  else:
16     print('Desculpa, não entendi!')
```

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.