programe_ce

Fundamentos de Programação Convencional

Fundamentos de Programação Visual
Programação Convencional

MÓDULO 2

CIENTISTA CHEFE
EDUCAÇÃO

FUNCAP

CEARÁ
GOVERNO DO ESTADO
SECRETARIA DA EDUCAÇÃO

EXPEDIENTE

Autores:

Mara Franklin Bonates
Tibérius de Oliveira e Bonates
José Maria da Silva Monteiro Filho
Eder Jacques Porfírio Farias
Joaquim Bento Cavalcante Neto
Jorge Herbert Soares de Lira
Ana Gardennya Linard Sírio Oliveira

Dados Internacionais de Catalogação na Publicação (CIP)

I61 Introdução à Lógica e à Programação Visual [recurso eletrônico] / Jorge Herbert Soares de Lira, Joaquim Bento Cavalcante Neto, Ana Gardennya Linard Sírio Oliveira, (orgs.).- Fortaleza: Ed. Jorge Herbert Soares de Lira, 2021.

Livro eletrônico v.2
ISBN 978-65-00-36596-2 (E-book)

1. Lógica. 2. Matemática. 3. Programação visual. I. Lira, Jorge Herbert Soares de, org. II. Cavalcante Neto, Joaquim Bento, org. III. Oliveira, Ana Gardennya Linard Sírio, org. IV. Título.

CDD: 511.3

Sumário

1 Fundamentos em Programação Visual

Os conteúdos teóricos abordados no Módulo 1 deste curso constituem a base necessária para desenvolver programas e implementar aplicações interativas para os mais variados fins. Neste Módulo, teremos a oportunidade de aprofundar tais conteúdos por meio de atividades práticas, que consistirão na construção de aplicações.

1.1 – Revisão

Para começarmos, é interessante relembrar o **Ambiente de Programação** que iremos utilizar: o Scratch[1]. Além disso, como vamos dedicar este Módulo à construção de programas, vale a pena revisarmos a definição do que é um **Programa**.

1.1.1 – Ambiente de Programação Visual

A Figura 1.1 indica onde encontrar, na interface do Scratch, cada uma das seções a seguir:

(a) Categorias de instruções;
(b) Blocos de instruções;
(c) Área de trabalho;
(d) Palco;
(e) Painel dos atores;
(f) Painel do palco.

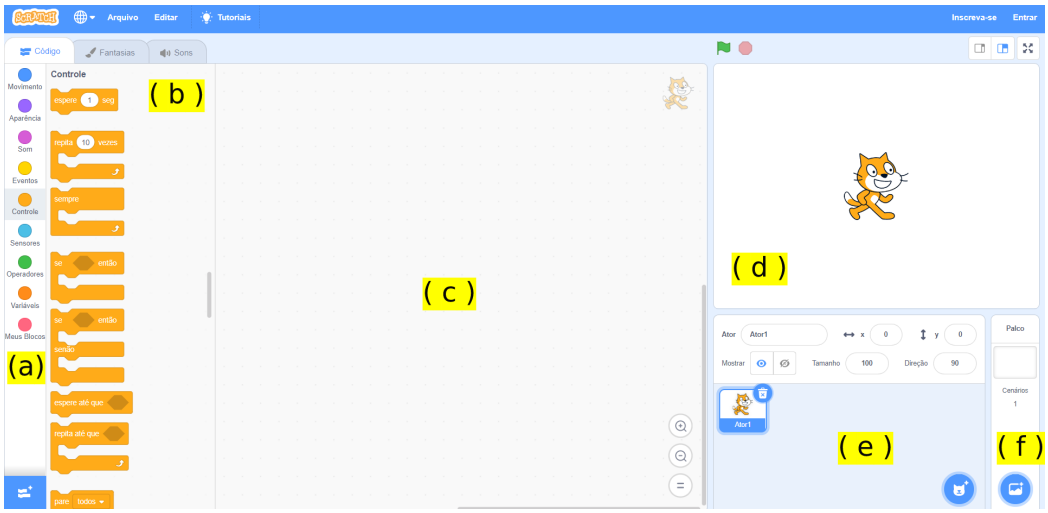

Figura 1.1: Interface do Scratch, com indicações dos painéis a serem abordados nesta seção.

[1] Acesso disponível em https://scratch.mit.edu/

Categorias de instruções

Em programação visual, a construção de um programa consiste em escolher blocos de instruções prontos, já oferecidos pela linguagem que escolhemos para trabalhar, e combiná-los arrastando-os para alguma área especificada pelo ambiente de programação. No Scratch, podemos encontrar blocos de instruções para realizar diversas tarefas, agrupadas em **categorias**, para facilitar a busca por um comando específico.

Obs As categorias de blocos podem ser encontradas na aba "Código", situada no canto superior esquerdo da tela.

Blocos de instruções

Todos os comandos e estruturas que utilizaremos para construir um programa na linguagem Scratch estão disponíveis na forma de **blocos de instruções**. Assim, vamos encontrar blocos para as estruturas de controle – sequenciação, decisão e repetição –, além de blocos de instruções para outros comandos.

Obs Conheceremos mais instruções e comandos à medida que formos realizando nossas atividades práticas!

A construção de um programa nesta linguagem consistirá, portanto, da escolha dos blocos referentes às instruções que desejamos utilizar e, com o mouse, clicar e arrastar esses blocos para a área de trabalho, realizando as devidas combinações entre eles, encaixando os blocos uns nos outros, conforme indicado nos formatos de cada um deles.

Obs Convém ressaltar que as instruções e comandos levados para a área de trabalho podem ser atribuídos a um ator escolhido, ou a todos os atores inseridos na sua aplicação. O Exemplo 1.1 ilustra uma situação em que temos conjuntos de instruções diferentes para dois diferentes atores.

Área de trabalho

Indicada na Figura 1.1 com a letra (c), a **área de trabalho** é a área para onde vamos arrastar os blocos de instruções escolhidos e combiná-los.

A Figura 1.2 apresenta um exemplo de blocos de instruções combinados na área de trabalho. O código em questão pode ser visualizado em detalhe na Figura 1.3.

Obs Observe como as **cores** dos blocos de instruções mudam conforme sua categoria.

Obs Para remover um bloco da área de trabalho, basta arrastá-lo de volta para o painel de bloco de instruções.

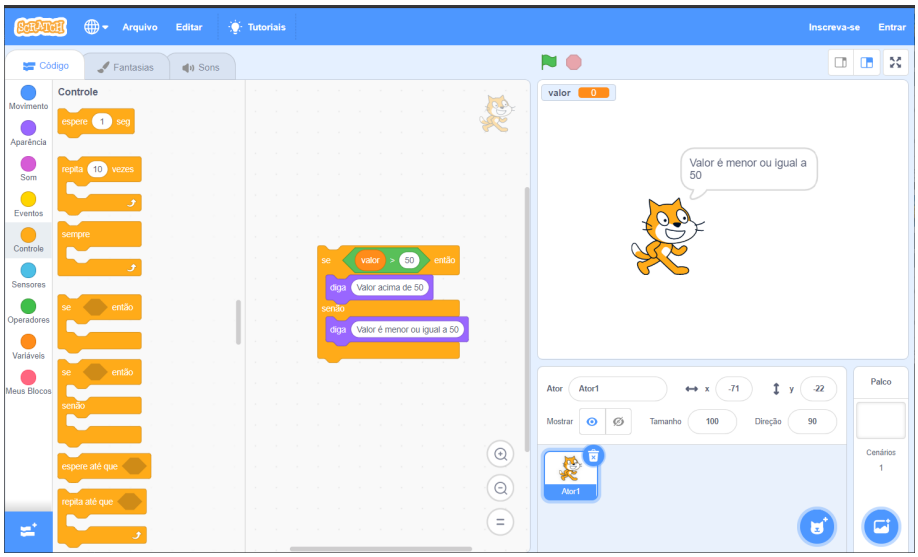

Figura 1.2: Interface do Scratch com blocos de instruções inseridos na área de trabalho.

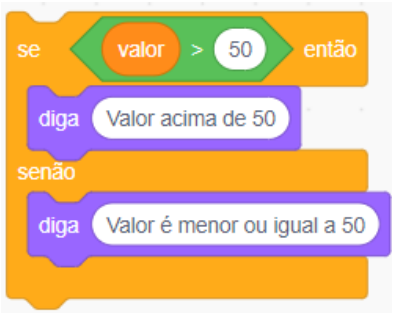

Figura 1.3: Detalhamento do programa escrito na área de trabalho da Figura 1.2.

Palco

O **palco**, indicado na Figura 1.1 com a letra (d), é a área onde as aplicações que desenvolvemos funcionam e são testadas. É nele que visualizaremos e testaremos os programas, animações e jogos que podemos desenvolver utilizando esta linguagem.

O espaço do palco é organizado em coordenadas x e y, com a origem posicionada no centro do palco. A Figura 1.4 mostra o espaço de coordenadas do palco.

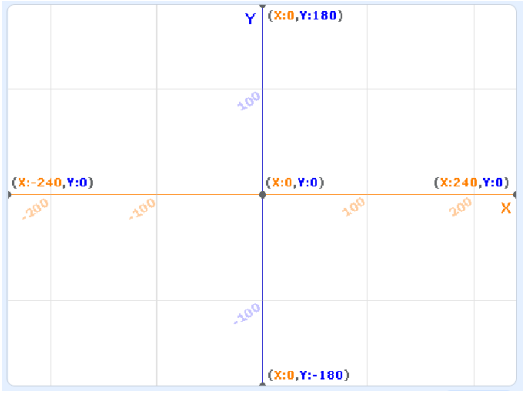

Figura 1.4: Espaço de coordenadas do palco.

Na Figura 1.5, vemos um ator no palco, localizado na posição com coordenadas (0, 0). Os valores das coordenadas x e y podem ser vistas em destaque na figura, circuladas em amarelo.

Figura 1.5: Ator posicionado na origem do palco.

Painel dos atores

Um ator é um elemento ou personagem que podemos inserir em um programa. Toda vez que formos iniciar um novo programa no Scratch, teremos, por padrão, um ator já inserido no projeto. O ator que aparecerá é o gatinho que podemos ver Figura 1.1.

Podemos inserir um ou mais atores em um mesmo programa. A inserção de novos atores é uma das funcionalidades oferecidas pelo **painel de atores**. Além disso, neste painel é possível ajustar várias propriedades de um ator, como o nome, posição no espaço do palco (em coordenadas x e y), tamanho, direção e se deve ficar visível ou não. A Figura 1.6 mostra as propriedades de Ator1.

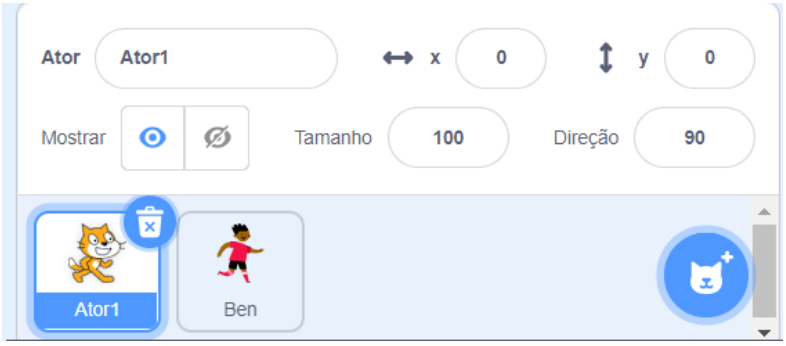

Figura 1.6: Painel dos atores, mostrando as propriedades do ator selecionado, o Ator1.

Obs

Perceba que há dois atores na aplicação, porém podemos afirmar que as propriedades que vemos na Figura 1.6 referem-se ao Ator1, pois este é o ator selecionado. Ele aparece em destaque, com um contorno azulado em torno de seu ícone.

Obs Podemos construir programas separados para cada ator. A área de trabalho alterna a exibição do programa atribuído a cada ator, ao se clicar sobre um ator escolhido. O exemplo 1.1 ilustra essa situação.

■ **Exemplo 1.1** Considere uma aplicação que contenha dois atores, que realizam um diálogo simples. A programação de cada um deles consiste na exibição de um balãozinho de diálogo, cada um com uma mensagem diferente. Assim, os comandos para cada um deles será levemente diferente. As Figuras 1.7 e 1.8 mostram a programação separada de cada um dos atores. A execução do programa completo, executando as programações de todos os atores, pode ser vista na Figura 1.9.

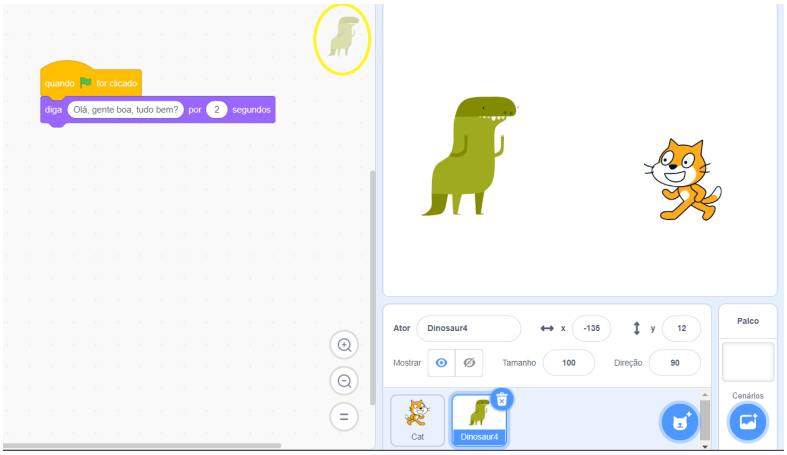

Figura 1.7: Programação atribuída ao ator "Dinossaur4". Observe a indicação de que ator está sendo programado com os comandos visíveis na área de trabalho, em destaque com um círculo amarelo.

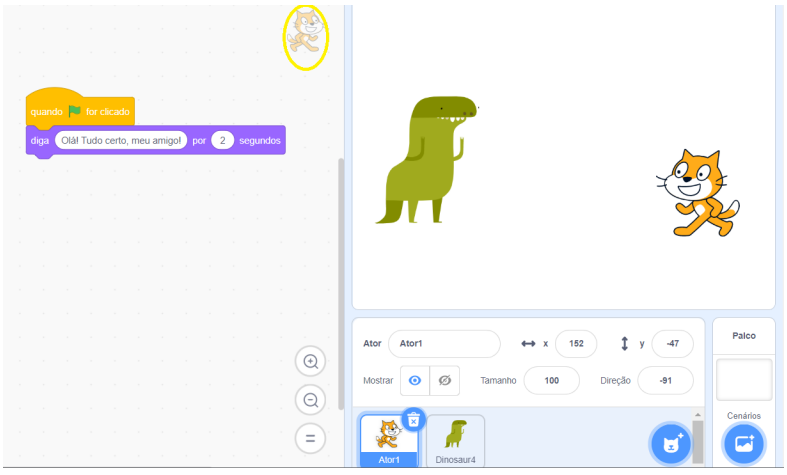

Figura 1.8: Programação atribuída ao ator "Ator1". Observe a indicação de que ator está sendo programado com os comandos visíveis na área de trabalho, em destaque com um círculo amarelo.

■

Figura 1.9: Execução do programa completo, exibido no palco do Scratch.

Painel do palco

O Scratch permite que um mesmo programa possua mais de um cenário, o que possibilita a criação de aplicações interessantes, como jogos com diferentes fases. O **painel do palco** oferece funções para inserir novos cenários, que podem ser escolhidos de um conjunto de cenários prontos e disponíveis no Scratch, ou um arquivo de imagem salvo em seu computador, ou ainda um desenho a ser criado por você, utilizando as ferramentas de desenho do Scratch. A lista de todos os cenários inseridos em seu programa pode ser vista na aba "Cenários", no canto superior esquerdo da tela.

Obs As instruções em um programa no Scratch podem manipular tanto os atores quanto os cenários. Isso nos dá uma vasta gama de possibilidades para explorar a criatividade!

■ **Exemplo 1.2** Considere uma aplicação que começa exibindo um cenário e, depois de um segundo, passa a exibir outro cenário. A Figura 1.10 mostra os comandos necessários para realizar esta tarefa.

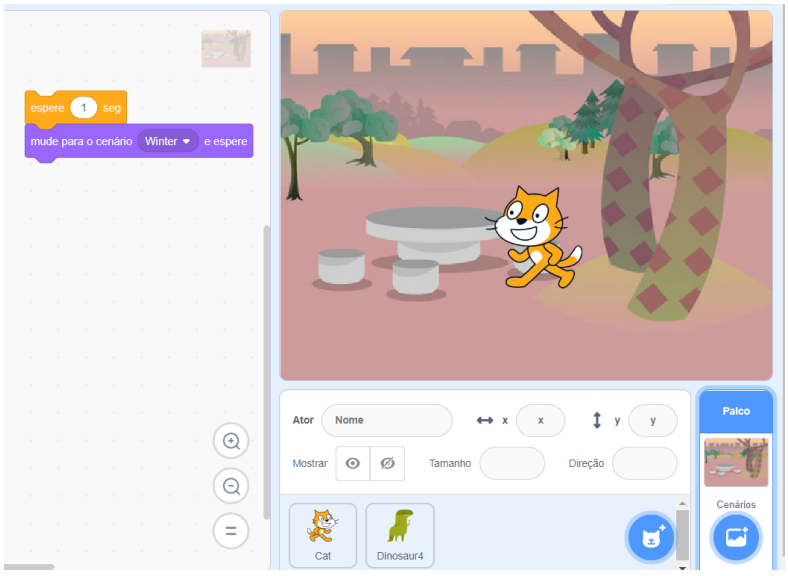

Figura 1.10: Programação atribuída a um cenário específico.

A Figura 1.11 mostra a mudança de cenário após um segundo de execução do programa.

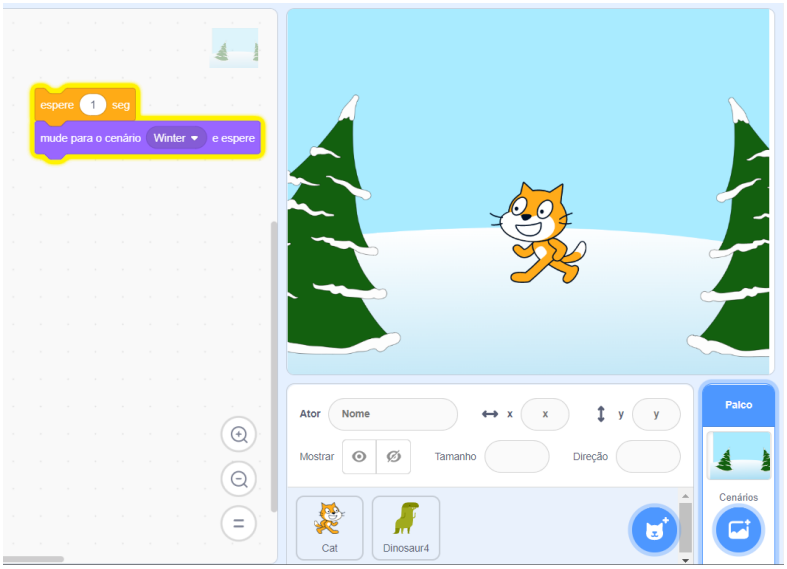

Figura 1.11: Mudança de cenário, conforme programação.

■

Obs Para visualizar todos os cenários inseridos em seu programa, clique na Aba "Cenários", visível no canto superior esquerdo da tela quando um cenário está selecionado com um clique de mouse.

O vídeo a seguir apresenta, de maneira bem detalhada, cada elemento da tela do Scratch.

Aula 1 - Scratch - Interface do Scratch. Duração 13min38s.

Exercícios

Vamos praticar, por meio dos exercícios propostos a seguir. Como sugestão, que tal tentar resolvê-los sem ver as respostas?

Exercício 1.1 Crie um programa que faça o gato se deslocar de um lado a outro da tela.

Solução. Há diferentes formas de se resolver esse exercício. Uma das possíveis soluções está apresentada no conjunto de passos a seguir:

- No palco, clique no gato e o arraste com o mouse para o lado esquerdo da janela. Este será o ponto de partida da caminhada do gato.
  Observe, no painel dos atores, as coordenadas x e y do gato na posição para onde você o arrastou. Você pode ajustar as coordenadas manualmente. Para este exemplo, vamos, manualmente, ajustar a coordenada y para zero. Para alterar manualmente as coordenadas de um objeto no palco, basta clicar no valor a ser modificado e editar o valor desejado.
- Na categoria "Movimento", você encontrará comandos que permitirão a programação do deslocamento pedido. Para este exercício, foi escolhido o comando "deslize por 1 segs até x: y:". A Figura 1.12 ilustra esse bloco.

- Escolha o bloco indicado na Figura 1.12 e, com o mouse, arraste-o para a área de trabalho.
- Agora, será necessário descobrir o ponto de chegada do gato. Uma maneira de se conseguir isso é arrastar o gato no palco até a posição desejada e observar suas coordenadas x e y no painel dos atores. Para este exemplo, foi observado que a posição x = 220 e y = 0 produzirá um resultado satisfatório.
- Preencha os valores numéricos escolhidos para x e y nos devidos espaços do bloco de instrução da Figura 1.12.
- Para executar o programa, basta clicar no bloco de instrução. O gato realizará o deslocamento no tempo total de 1 segundo.

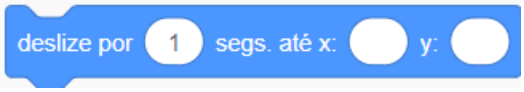

Figura 1.12: Bloco de instrução escolhido para a movimentação do gato no Exercício 1.1.

■

Exercício 1.2 Feito o Exercício 1.1, fica a pergunta: o que acontece se aumentarmos o tempo para a movimentação do gato?

Solução. Aumentando o tempo de movimentação, e mantendo a mesma distância a ser percorrida, podemos afirmar que a velocidade do gato vai diminuir. Para verificar esse comportamento, basta modificar o tempo no bloco de instrução da Figura 1.12 e executar o programa novamente. ■

Obs Observe que o programa do Exercício 1.2 é uma demonstração prática de que a velocidade e o tempo são grandezas inversamente proporcionais.

Exercício 1.3 Vamos incrementar o programa do Exercício 1.1, fazendo o gato mudar poses, melhorando a impressão de que ele está realmente caminhando.

Solução. O que está sendo pedido no Exercício 1.3 é basicamente que o personagem seja animado. Assim, ele será exibido, alternando poses a cada instante. Para o gato, temos duas poses pré-definidas, que podem ser vistas na aba "Fantasias". Esta aba aparece na parte superior da interface quando o ator é selecionado com o mouse. A Figura 1.13 mostra a aba "Fantasias" em destaque com um círculo amarelo, e as duas poses disponibilizadas para o gato.

Assim, nosso programa precisará de comandos que alternam a exibição dessas duas poses. Os blocos que nos permitem fazer isso podem ser encontrados na categoria "Aparência". A Figura 1.14 mostra alguns dos blocos de instruções que compõem essa categoria.

Vamos escolher o bloco "próxima fantasia". Ao testá-lo, podemos perceber que a fantasia seria trocada apenas uma vez. Assim, vamos adicionar mais um bloco de instrução, permitindo que essa instrução se repita indefinidamente (ou seja, se repita "para sempre"). Há um bloco específico para possibilitar essas repetições na categoria "Controle". A Figura 1.15 mostra o comando escolhido e a Figura 1.16 mostra o programa montado.

Ao testar o programa, vemos que a animação do gato ficou muito rápida! É possível adicionar comandos para criar um tempo de espera entre ações. Vamos experimentar usar o comando "espere 1 seg", disponível na categoria Controle. A nova versão do programa pode ser vista na Figura 1.17.

■

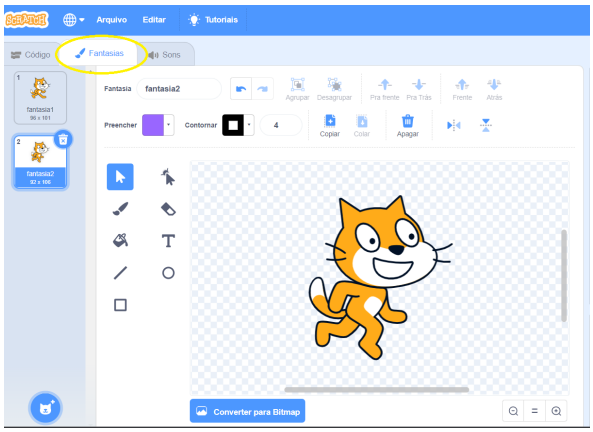

Figura 1.13: Conteúdo da aba “Fantasias” exibindo as duas poses disponíveis para o gato.

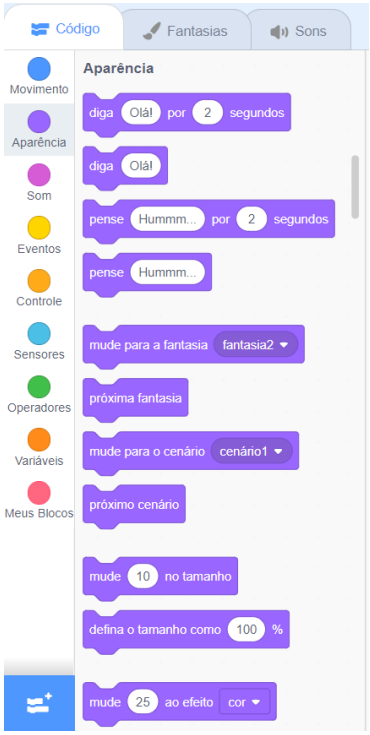

Figura 1.14: Conteúdo da categoria “Aparência” exibindo alguns dos blocos de instruções disponibilizados pela linguagem.

Obs A solução final do Exercício 1.3 pode ainda ser melhorada, ajustando possibilidades de tempos diferentes, até a animação ficar no ritmo desejado. É possível usar frações de segundo. Por exemplo, para diminuir o tempo para meio segundo, basta apagar o “1” do bloco “espere 1 seg” e digitar “0.5” no lugar. Note que deve-se utilizar ponto, e não vírgula, para separar casas decimais.

Obs Note que o programa do Exercício 1.3 fica em execução por tempo indeterminado. Isso se deve ao bloco de instrução “sempre”. Para encerrar a execução do seu programa, clique no símbolo de “Pare”, situado acima do palco.

Figura 1.15: Bloco de instrução para repetição indefinida de comandos.

Figura 1.16: Programa montado para trocar de fantasias em repetição indefinida.

Figura 1.17: Programa montado para trocar de fantasias em repetição indefinida, com um tempo de espera entre cada troca.

1.1.2 – Programação

Vamos aprofundar os fundamentos teóricos estudados no Módulo 1, construindo nossos próprios programas no Scratch. Para isso, vale a pena relembrar a definição de Programa:

Definição 1.1.1 Um **Programa** é uma sequência de instruções, escritas em uma **linguagem de programação**, que descrevem, passo a passo, como uma determinada tarefa deve ser executada por um computador.

Nossas atividades práticas neste curso serão, essencialmente, escrever programas utilizando programas! Isso pareceu confuso? Pois vamos relembrar o que já vimos sobre conceitos de software.

Os programas que vamos desenvolver serão exemplos de softwares **aplicativos**. E, como já vimos, para escrever programas, utilizamos uma IDE (ou um abiente de programação) para facilitar nosso trabalho. Dentro de uma IDE temos, além do programa que escrevemos, outros programas em execução dando o suporte necessário ao nosso trabalho. Um desses softwares é o **tradutor**, um exemplo de **software básico**. O software tradutor se encarrega de tomar o programa que nós escrevemos e traduzir para a linguagem que o computador compreende e pode executar, a chamada **linguagem de máquina**.

A criação de linguagens de programação e de softwares tradutores tornam nosso trabalho de escrita de programas muito mais fácil e mais rápido. Neste Módulo, vamos realizar atividades práticas de programação utilizando duas linguagens de programação distintas: a linguagem Scratch, um exemplo de linguagem de programação visual, e a linguagem Python, um exemplo de linguagem de programação convencional.

Exercício 1.4 Revisando a definição 1.1.1, qual a diferença entre um programa e um algoritmo?

Solução. Tanto o algoritmo como o programa são sequências de instruções para a realização de uma tarefa. A única diferença é que um programa é escrito seguindo uma determinada **Linguagem de Programação**. Neste capítulo, vamos utilizar a linguagem Scratch, e no segundo capítulo deste material, utilizaremos a linguagem Python. ■

1.2 – Estruturas Básicas

Vamos aplicar, de maneira intuitiva e por meio de projetos práticos, as estruturas básicas para a construção de programas: sequenciação, decisão e repetição.

Obs Para um revisão mais aprofundada, vale a pena consultar a Unidade 3 do material do Módulo 1.

Sequenciação

As linguagens de programação funcionam de tal forma que as instruções são executadas na ordem em que foram escritas. Nos exercícios a seguir, vamos observar a importância da ordem em que as instruções são escritas em um programa.

Exercício 1.5 Construa uma aplicação que simula o lançamento de uma bola de basquete. Seu programa deverá gerar a trajetória da bola entrando em cena como se tivesse sido lançada de fora do palco, quicando uma vez no chão e subindo até uma altura acima da rede, vindo a cair de volta no chão, passando por dentro da rede.

Solução. Para construir essa aplicação, vamos precisar de um cenário mostrando uma quadra de basquete. Nosso ator será uma bola de basquete, e a programação a ser feita possibilitará que a bola descreva a trajetória pedida. A Figura 1.18 mostra o cenário e o ator escolhidos para este exercício.

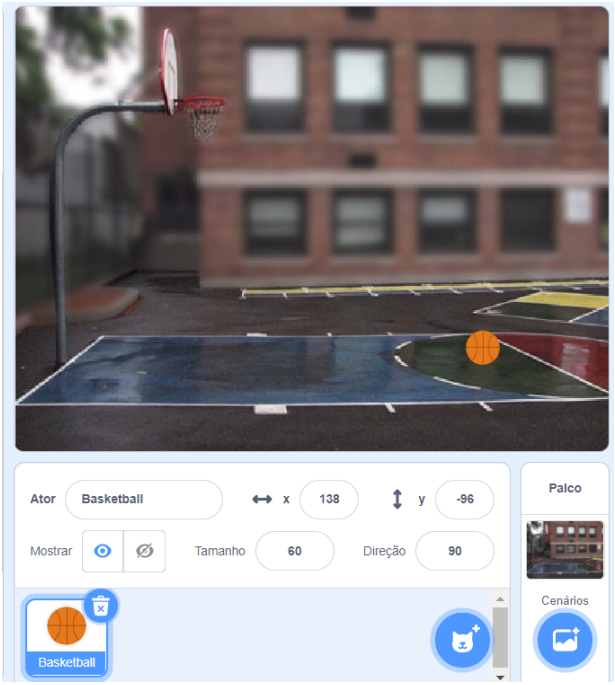

Figura 1.18: Cenário e ator escolhidos para o Exercício 1.5.

A programação da trajetória da bola foi feita com comandos que fazem a bola realizar diferentes deslocamentos, para simular o comportamento desejado. A Figura 1.19 mostra a programação feita.

Figura 1.19: Programação para a bola de basquete do Exercício 1.5.

Observe que a instrução "deslize" necessita dos valores das coordenadas x e y para determinar os deslocamentos da bola. Para conseguir esses valores, basta mover com o mouse a bola até cada ponto desejado e tomar nota das coordenadas x e y que aparecem no Painel dos Atores.

Observe, também, uma mudança no tempo, em segundos, de um dos deslocamentos, para ajustar o ritmo da trajetória da bola. ■

Exercício 1.6 Para comprovar a importância de uma boa sequenciação em seu programa, experimente trocar a ordem de alguns dos comandos e execute o programa novamente.

Solução. Você pode escolher mudar de lugar qualquer um dos comandos. Para mudar uma instrução de lugar, você pode clicar e arrastar os blocos de instrução para desfazer as conexões entre eles e rearranjá-los, criando novas conexões. Certifique-se de que todos os blocos estão conectados após ter feito o "embaralhamento" deles.

Lembre-se de comentar com sua turma que comando você mudou de lugar e o que você observou. ■

Exercício 1.7 Observe a Figura 1.20 e responda:

(a) Dados o cenário e o ator presentes na Figura, o que aconteceria se executássemos, para o gato, o programa apresentado na Figura 1.21?
(b) Escolha alguns dos comandos presentes no programa e mude a ordem deles. Execute o programa novamente e observe que mudanças ocorreram em seu programa.

Figura 1.20: Cenário a ser percorrido pelo gato na aplicação do Exercício 1.7.

Solução.

(a) A execução do programa fará com que o gato se desloque do ponto onde ele aparece na Figura 1.20 até a posição central do topo do palanque. Ao chegar lá, o gato mia.
(b) Você pode escolher quantos e quais comandos você quer mudar de lugar. Ao trocar a ordem das instruções do programa, execute-o novamente e observe as alterações no resultado.

■

Figura 1.21: Programação para o gato no Exercício 1.7.

Exercício 1.8 Considere uma aplicação com as seguintes características: um personagem está parado em um cenário contendo árvores, e uma maçã cai de cima até a boca do personagem. Quando a maçã está quase chegando no personagem, ele abre a boca e come a maçã! Como programar essa aplicação?

Solução. Vamos por partes. Primeiro, vamos inserir o cenário e o personagem. O personagem escolhido foi um dinossauro, pois ele tem fantasias com a boca fechada e com a boca aberta. O cenário escolhido foi uma fotografia mostrando um local abaixo de uma árvore. A Figura 1.22 mostra os elementos escolhidos.

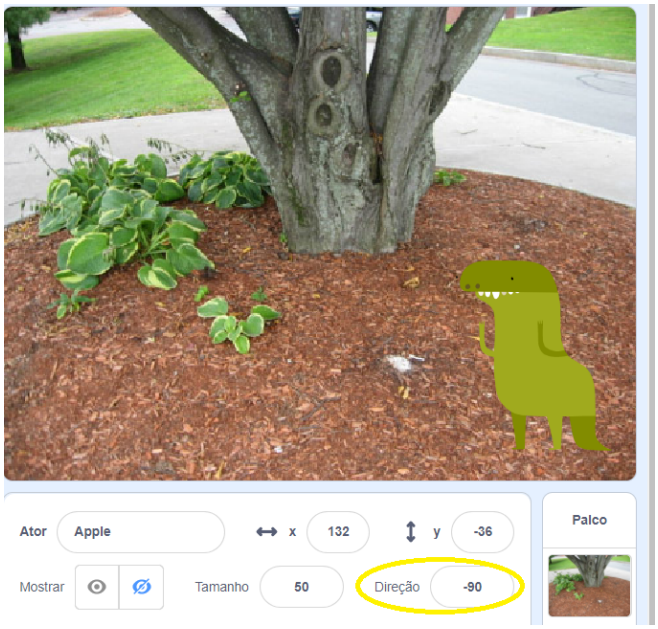

Figura 1.22: Cenário e personagem escolhidos para o Exercício 1.8.

Obs Observe que na Figura 1.22, a Direção do personagem foi modificada para o valor -90 (para que o personagem virasse para o outro lado).

Continuando, vamos inserir uma maçã. Ela vai cair de um local na parte superior do palco e chegar até a boca do personagem. Precisamos, então, determinar as coordenadas x e y das posições inicial e final da maçã. Isso é simples de se resolver: basta posicionar a maçã no local desejado e tomar nota das coordenadas x e y. A Figura 1.23 mostra em destaque as coordenadas x e y da maçã, após ter sido posicionada com o mouse em um ponto escolhido para ser o ponto de partida dela.

O mesmo processo é feito para determinar as coordenadas x e y da posição final da maçã. A Figura 1.24 mostra um exemplo.

Vamos programar as ações dos atores. Primeiro, a maçã. Ela fará uma trajetória do ponto inicial até o ponto final. A Figura 1.25 mostra as instruções necessárias para que a maçã realize a trajetória. Observe, na figura, os valores de x e y devidamente copiados para dentro dos blocos de instrução.

Agora, vamos programar o dinossauro. Vemos que a maçã está realizando sua trajetória no tempo de 1 segundo. Assim, é interessante que o dinossauro abra a boca logo após esse 1 segundo tiver passado. Em outras palavras, o dinossauro precisa mudar de fantasia para exibir sua boca aberta. A Figura 1.26 mostra as instruções necessárias.

Para deixar o comportamento do dinossauro mais realista, é interessante que ele volte à posição inicial, com a boca fechada. A Figura 1.27 mostra uma versão atualizada do programa

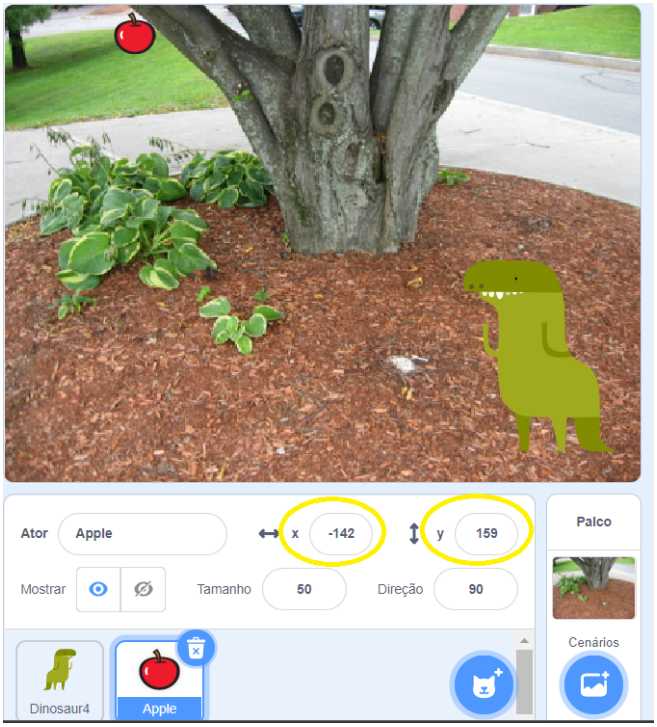

Figura 1.23: Escolha da posição inicial da maçã.

para o dinossauro. Observe que foi adicionado um tempo de meio segundo antes de o dinossauro voltar para a fantasia inicial.

Agora, vamos fazer com que os programas dos dois personagens sejam executados de uma só vez. Para isso, é necessário adicionar a cada programa um bloco de instrução "Quando a bandeirinha verde for clicada". As Figuras 1.28 e 1.29 mostram as novas versões de cada programa.

Agora, para executar o programa completo, basta clicar sobre a bandeirinha verde, situada logo acima do palco.

É possível melhorar um pouco mais o programa: tornar a maçã invisível quando ela chegar à boca do dinossauro. A categoria "Aparência" disponibiliza blocos de instrução para exibir ou esconder um ator. Assim, a versão final do programa da maçã pode ser vista na Figura 1.30.

Obs Lembre-se: para alternar entre os programas de cada ator, basta clicar sobre o ícone do ator no Painel de Atores.

■

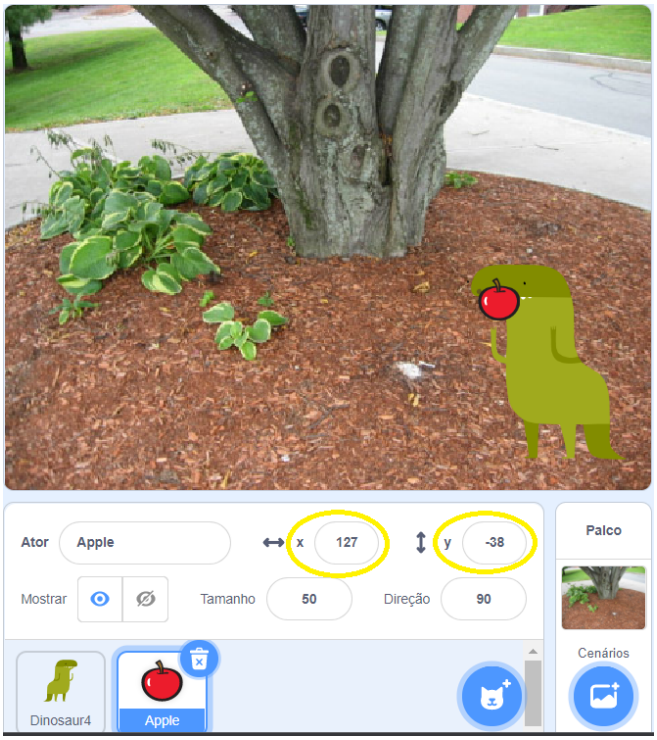

Figura 1.24: Escolha da posição final da maçã.

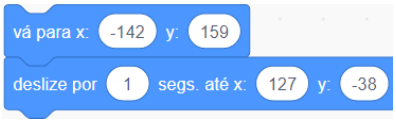

Figura 1.25: Instruções necessárias para a maçã realizar a trajetória desejada.

Figura 1.26: Instruções necessárias para o dinossauro abrir a boca no momento em que a maçã chegar perto dele.

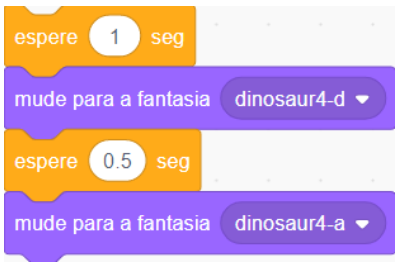

Figura 1.27: Versão melhorada do programa para o dinossauro, fazendo o personagem abrir e fechar a boca.

Figura 1.28: Versão melhorada do programa para o dinossauro, possibilitando a sua execução junto com outros programas.

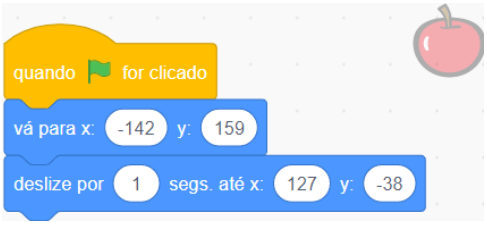

Figura 1.29: Versão melhorada do programa para a maçã, possibilitando a sua execução junto com outros programas.

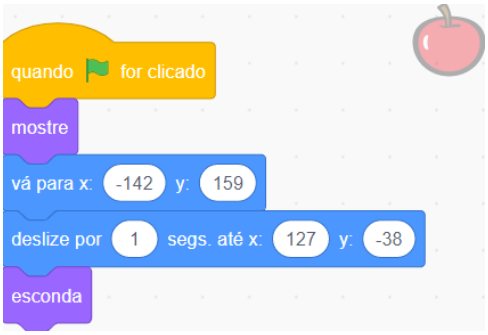

Figura 1.30: Versão final do programa para a maçã, fazendo a maçã desaparecer ao chegar à boca do dinossauro.

Exercício 1.9 No programa da maçã, troque de lugar os blocos de instrução, conforme indicado na Figura 1.31. Execute o programa por inteiro e observe o efeito.

Figura 1.31: Quebra na sequenciação das instruções do programa da maçã.

Decisão

As estruturas de decisão são comandos que nos permitem alterar o fluxo da execução de um programa, a depender de uma determinada condição. No ambiente de programação que estamos utilizando, vamos encontrar blocos para estruturas de decisão na categoria "Controle".

A Figura 1.32 apresenta os blocos de instrução referentes às estruturas de decisão.

Figura 1.32: Blocos de instruções para estruturas de decisão.

As estruturas de decisão serão estudadas de maneira mais aprofundada na Seção 1.5. Por ora, vamos apenas fazer alguns exercícios simples, para reforçarmos nossa noção intuitiva sobre seu funcionamento.

Exercício 1.10 O gato está se sentindo confiante! Ele sabe dizer se um dado número é par ou ímpar! Faça a programação para concretizar essa habilidade no gato.

Solução. A aplicação consiste em obter um número do usuário e fazer o gato dizer se o dado número é par ou ímpar. Para determinar a paridade de um número, basta dividi-lo por 2 e observar se o resto é igual ou se é diferente de zero. A Figura 1.33 apresenta a programação para esta aplicação.

■

Exercício 1.11 O gato agora sabe dizer se uma letra digitada é uma vogal ou uma consoante! Faça a programação para confirmar essa esperteza toda do gato.

Solução. A aplicação consiste em obter uma letra digitada pelo usuário e fazer o gato dizer se a letra é vogal ou consoante. Para isso, escolhemos como estratégia testar se a letra informada é igual a 'a', 'e', 'i', 'o', ou 'u'. Felizmente, podemos contar com o operador booleano "OU" para

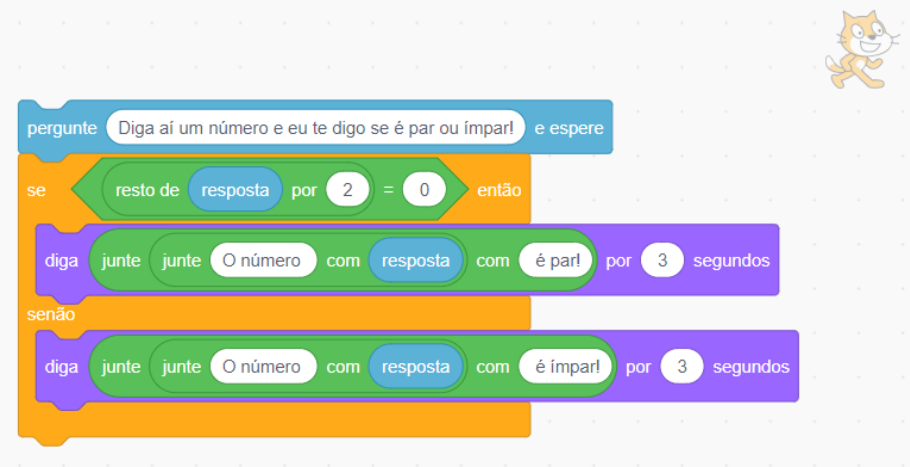

Figura 1.33: Programação para o gato determinar se um dado número é par ou ímpar.

nos ajudar a juntar todos esses testes em um só. A Figura 1.34 apresenta a programação para esta aplicação.

Figura 1.34: Programação para o gato determinar se uma letra digitada é vogal ou consoante.

■

Repetição

As estruturas de repetição nos permitem programar ações que podem necessitar ser executadas mais de uma vez. A Figura 1.35 apresenta os blocos de instruções para as estruturas de repetição disponibilizadas pela linguagem Scratch.

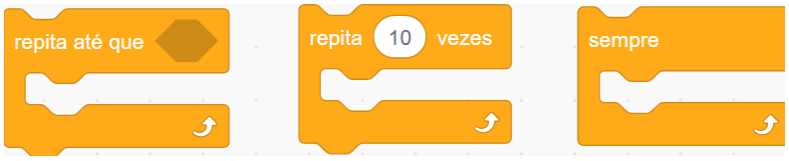

Figura 1.35: Blocos de instruções para estruturas de repetição.

As estruturas de repetição serão estudadas de maneira mais aprofundada na Seção 1.6. Por ora, vamos apenas fazer alguns exercícios simples, para reforçarmos nossa noção intuitiva sobre seu funcionamento.

Exercício 1.12 Observe a Figura 1.36 e responda:

(a) O que o programa faz?
(b) Que problema estratégico você aponta para a escrita do programa apresentado?
(c) Como você pode melhorar esse programa?

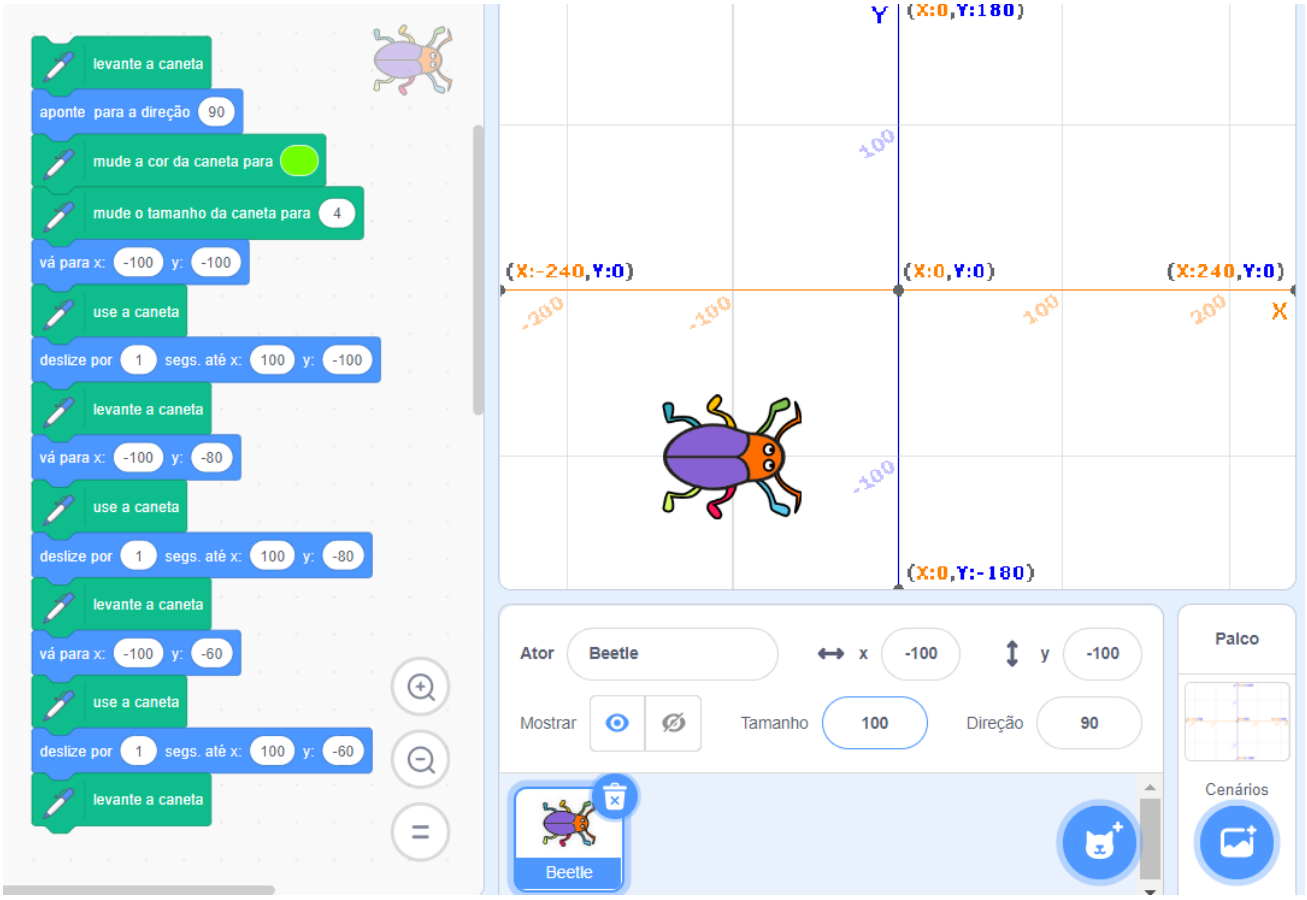

Figura 1.36: Programação abordada no Exercício 1.12.

Solução. As respostas para o Exercício 1.12 estão enumeradas abaixo:

(a) O programa faz com que o personagem descreva três trajetórias retas e horizontais, deixando um risco com 4 pixels de espessura, na cor verde. As linhas resultantes dos rastros deixados têm medida igual a 200 pixels.

(b) O problema que vemos é a repetição desnecessária de comandos. A Figura 1.37 destaca que comandos se repetem, de maneira desnecessária. É possível afirmar que as repetições são desnecessárias, porque existem uns padrões nos comandos repetidos:

- o valor de x nos comandos "vá para" se mantém sempre o mesmo: -100.
- o valor de x nos os comandos "deslize", também se mantém o mesmo: 100
- a atualização do valor de y é sempre igual, aumentando seu valor em 20 unidades em cada comando "deslize".

(c) A solução para melhorar o programa é utilizar uma estrutura de repetição, inserindo dentro dela somente os comandos que se repetem, ajustando os comandos conforme os padrões de comportamento dos valores de x e de y. A Figura 1.38 mostra uma versão melhorada do programa da Figura 1.36

A Figura 1.39 resume a simplificação feita no programa apresentado no Exercício 1.12.

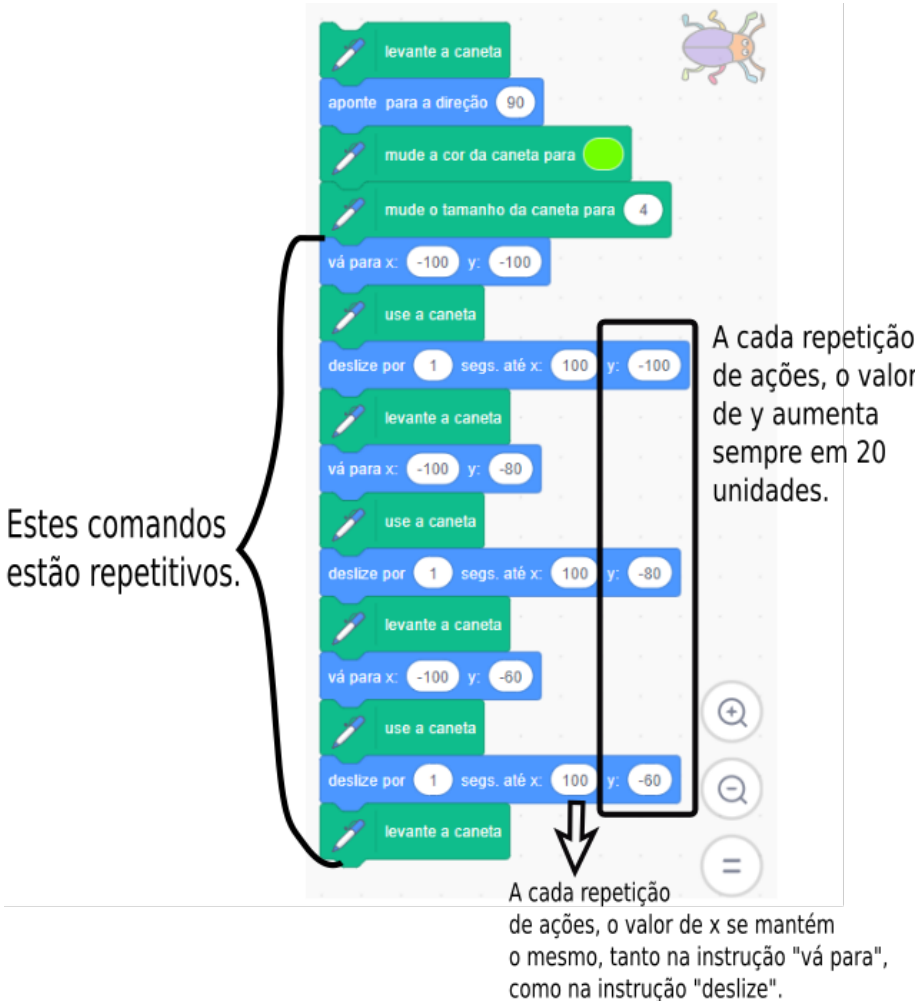

Figura 1.37: Análise da programação abordada no Exercício 1.12.

Obs

Não conhecia a caneta do Scratch, ou já usou e não se lembra mais de onde encontrá-la? Para obter os blocos de instruções referentes ao uso da caneta, clique no botão “Adicionar uma extensão”, situado no canto inferior esquerdo da janela do Scratch e ilustrado na Figura 1.40.

■

Figura 1.38: Versão melhorada da programação abordada no Exercício 1.12.

Figura 1.39: Comparação entre a versão original e a versão melhorada do Exercício 1.12.

Figura 1.40: Botão para adicionar extensão ao programa.

Exercício 1.13 O gato quer lhe desafiar a adivinhar em que letra ele está pensando (aqui entre nós: ele está pensando na letra 's'). Ele pedirá para você digitar uma letra até que você consiga finalmente acertar. Construa uma aplicação que implementa esse desafio.

Solução. Para este tipo de problema, não é possível prever de antemão quantas vezes o gato vai precisar pedir para o usuário digitar uma letra. Para esse tipo de situação, a melhor opção de estrutura de repetição é a instrução "repita ... até que ...". A Figura 1.41 ilustra um programa para resolver o problema proposto no Exercício 1.13.

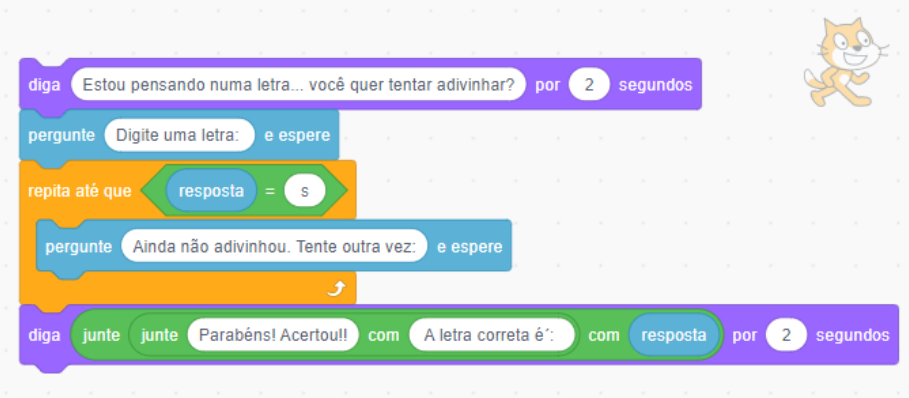

Figura 1.41: Programação para o problema proposto no Exercício 1.13.

■

1.3 – Variáveis e Constantes

As variáveis e as constantes são os dados que queremos manipular por meio de instruções de nossos programas. Nesta seção, vamos aprender como podemos trabalhar com variáveis e constantes no ambiente de programação visual que estamos utilizando neste curso.

1.3.1 – Variáveis

No Scratch, podemos encontrar algumas **variáveis próprias da linguagem**, prontas para usarmos em nossos programas. Podemos ver alguns exemplos listados abaixo:

- **posição x**, **posição y** e **direção**, na categoria "Movimento";
- **tamanho**, na categoria "Aparência";
- **resposta**, na categoria "Sensores".

1.3.2 – Operações sobre Variáveis

Em um programa no Scratch, podemos efetuar as seguintes operações sobre uma variável, seja ela criada por nós ou uma variável interna da linguagem.
Vamos abordar as operações de:

- Criação de variáveis;
- Inicialização de variáveis;
- Atribuição de valores a uma variável;
- Visualização de valores armazenados em variáveis.

Criação de Variáveis

Além das variáveis já criadas na linguagem, podemos criar nossas próprias variáveis. Para isso, basta clicar no botão "Criar uma Variável", disponível na categoria "Variáveis". Após clicar no botão, você deverá fornecer duas informações:

- um nome para a variável;
- se ela é válida somente para o ator selecionado no momento, ou se será válida para todos os atores incluídos em seu programa.

Exercício 1.14 O gato quer que você forneça para ele dois números, para que ele possa dizer para você o resultado da soma desses dois números. Crie um programa para implementar esta aplicação.

Solução. O usuário deverá fornecer dois números para o gato. Esses dois valores precisam ficar armazenados na memória, de modo que possam ser recuperados pelo programa no momento em que a operação de soma for aplicada a eles. Assim, os dois números precisam estar armazenados em variáveis. Como cada variável só armazena um valor por vez, vamos precisar, para esta aplicação, de duas variáveis, uma para cada valor fornecido.

A Figura 1.42 mostra as duas variáveis criadas para o Exercício 1.14: "valor1" e "valor2".

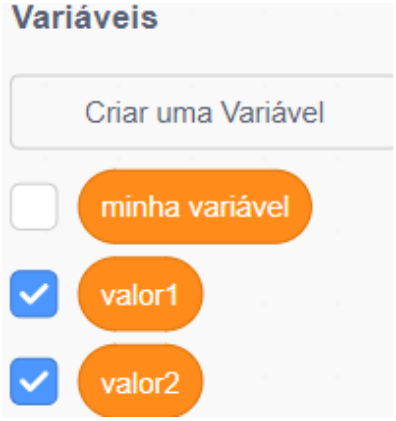

Figura 1.42: Variáveis criadas para a solução do Exercício 1.14.

Agora que temos as variáveis necessárias, podemos construir a programação, que pode ser vista na Figura 1.43.

Figura 1.43: Programa criado para a solução do Exercício 1.14.

■

Exercício 1.15 O programa do Exercício 1.14 se limita à exibição do resultado da soma de dois números fornecidos, sem armazenar esse resultado. Isso pode vir a ser um problema, caso esse resultado precise ser utilizado mais adiante na aplicação. Indique que modificação devemos fazer ao programa, para que o resultado da soma efetuada fique armazenado na memória.

Solução. Para que o resultado da soma fique armazenado na memória principal, é necessário criar uma variável para ele. A Figura 1.44 mostra uma versão melhorada do programa feito anteriormente no Exercício 1.14. Nessa nova versão, uma variável de nome "soma" foi criada.

Figura 1.44: Versão melhorada do programa criado para a solução do Exercício 1.14.

■

Inicialização de Variáveis

A inicialização de uma variável é simplesmente a atribuição de um valor a ela logo no início do programa. No Scratch, o bloco de instrução que nos permite inicializar uma variável está ilustrado na Figura 1.45.

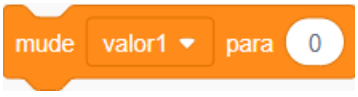

Figura 1.45: Bloco de instrução para armazenar um valor em uma variável. No caso da Figura, a variável de nome "valor1" armazenará o valor zero.

Atribuição de valores a variáveis

O bloco de instrução apresentado na Figura 1.45, além de possibilitar a inicialização de uma variável, possibilita também a atribuição de um valor a uma variável em qualquer outro momento na sequência de instruções de seu programa. Resumindo, a diferença entre inicialização e atribuição é justamente o momento em que estamos armazenando um valor em uma variável: se é no início do programa, ou se é em um momento mais à frente na sequência de instruções do programa.

Podemos atribuir a uma variável um valor conhecido, um valor armazenado em outra variável ou até mesmo uma expressão.

O Exemplo 1.3 lista alguns comandos válidos para atribuição de valores a variáveis.

Figura 1.46: Diferentes possibilidades para atribuir valores a uma variável. Nos exemplos, a variável que está sendo atualizada tem o nome "minha variável".

■ **Exemplo 1.3** Alguns comandos válidos para atribuição de valores a variáveis podem ser vistos na Figura 1.46.

■

A atribuição de um valor a uma variável provoca uma modificação, ou atualização, no valor armazenado por ela. Podemos atualizar o valor previamente armazenado em uma variável utilizando o comando apresentado na Figura 1.45, ou ainda, o bloco apresentado na Figura 1.47. Na figura, uma variável de nome "valor1" será atualizada com um novo valor, que seria seu valor anterior mais um.

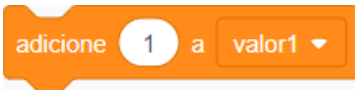

Figura 1.47: Bloco de instrução para atualizar o valor de uma variável. No caso da Figura, a variável de nome "valor1" terá seu valor aumentado em uma unidade.

Obs A operação que atualiza uma variável somando uma unidade ao valor previamente armazenado nela leva o nome especial de **Incremento da variável**.

Obs De maneira semelhante, a operação que atualiza uma variável subtraindo uma unidade ao valor previamente armazenado nela leva o nome especial de **Decremento da variável**.

Exercício 1.16 O gato está dizendo que é capaz de calcular somatórios de 1 até um valor dado. Confirme a afirmação do gato, criando um programa que o habilite a fazer isso.

Solução. O problema de calcular um somatório é um exemplo clássico para treinar as operações de inicialização e de incremento de uma variável. A Figura 1.48 mostra uma solução para o problema proposto.

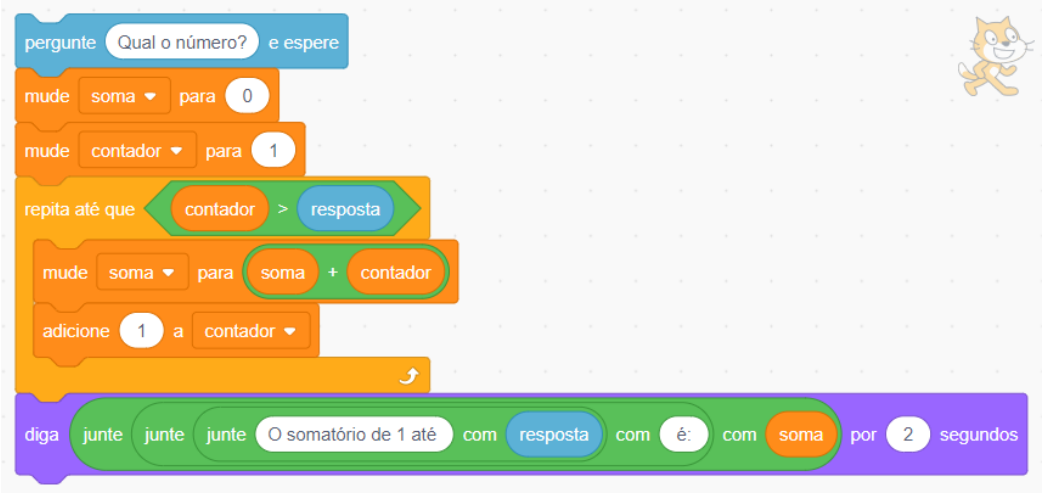

Figura 1.48: Programação para calcular o somatório de 1 até um valor fornecido pelo usuário.

■

Exercício 1.17 Observando o programa do Exercício 1.16, identifique que comandos são de inicialização, de atribuição e de incremento de variáveis.

Solução. A Figura 1.49 apresenta as respostas para o Exercício 1.17.

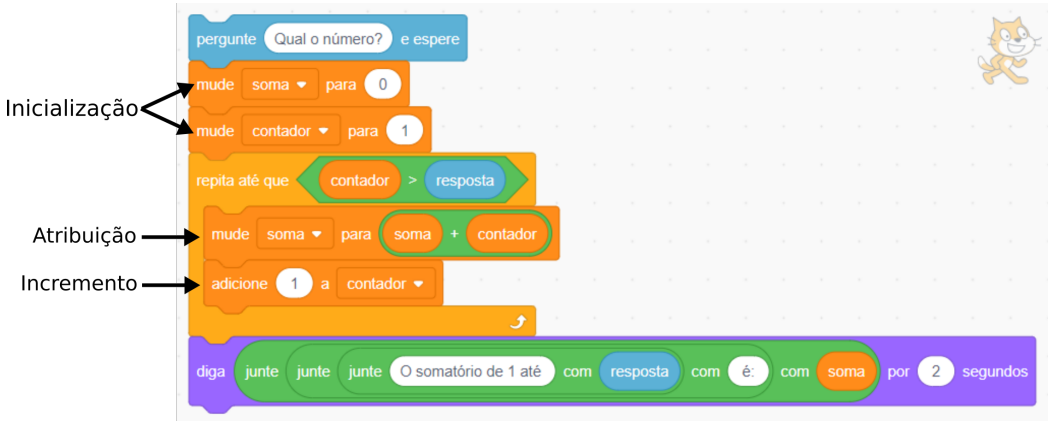

Figura 1.49: Programação para calcular o somatório de 1 até um valor fornecido pelo usuário.

■

É importante ressaltar que **uma variável só guarda um valor por vez**. Assim, quando aplicamos um comando de atribuição a uma variável, o antigo valor armazenado nela será perdido. O Exemplo 1.4 ilustra esse processo.

■ **Exemplo 1.4** Considere a sequência de instruções, ilustradas na Figura 1.50. Os dois comandos serão executados em sequência, e após a execução do segundo comando, o valor armazenado em "minha variável" será 1. Não será mais possível recuperar o valor zero, anteriormente armazenado em "minha variável".

■

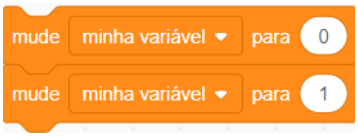

Figura 1.50: Sequência de instruções, do Exemplo 1.4, para demonstrar o efeito do comando de atribuição de valor a variáveis.

Exercício 1.18 Considere um programa que seja capaz de trocar os valores armazenados em duas variáveis, A e B. Por exemplo, suponha a situação inicial:

- A variável A armazenando o valor 5;
- A variável B armazenando o valor 7;

Ao final da execução do programa, teremos:

- A variável A armazenando o valor 7;
- A variável B armazenando o valor 5;

Em uma tentativa de resolver o problema, uma pessoa escreveu o programa apresentado na Figura 1.51. Porém, é possível ver que o programa **não resolve o problema proposto**. Ao final da execução do programa, ambas as variáveis estão armazenando o valor 7, como se pode ver na figura. Diante do exposto, responda ao que se pede:

(a) O que está errado no programa?
(b) Como corrigir o problema?

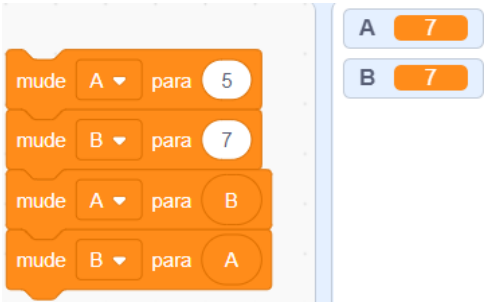

Figura 1.51: Primeira tentativa de programar a troca de valores de variáveis, como pedido no Exercício 1.18. Note que essa sequência de instruções **não soluciona o problema!**

Solução. As respostas para o Exercício 1.18 seguem abaixo:

(a) O problema no programa está no terceiro bloco de instrução da Figura 1.51. Ao ser executado, a variável A recebe o valor de B, que é 7. O valor 5, anteriormente armazenado em A, se perde. Daí, no quarto bloco de instrução, a variável B recebe o valor atual de A, que é 7.
(b) Para solucionar o problema, é necessário criar uma terceira variável, que será auxiliar no processo de troca de valores. Vamos chamá-la de "temporária". Ela receberá o valor de uma das variáveis, para servir de "memória" do antigo valor dessa vriável quando ela tiver sido atualizada. A Figura 1.52 apresenta um programa que realiza o trabalho corretamente.

■

Figura 1.52: Versão corrigida do programa do Exercício 1.18

Visualização de valores armazenados em variáveis

Independente de a variável ter sido criada por nós ou já estar disponível na linguagem, elas apresentam uma característica interessante: a possibilidade de podermos visualizar o valor armazenado em cada uma delas. Para tanto, basta clicar na caixa de verificação que aparece ao lado do bloco referente à variável, dentro da categoria que a contém. Ao clicar em alguma caixa de verificação, surge uma caixinha no palco, com o nome e o valor armazenado na variável.

Obs Podemos visualizar quantas variáveis quisermos no palco. As caixinhas referentes a cada uma delas podem ser movidas com arrastos de mouse, para melhor acomodar a disposição delas.

Alternativamente, podemos utilizar em nossos programas blocos de instrução que alternam entre a exibição ou a ocultação do valor de uma variável. Tais blocos podem ser vistos na Figura 1.53.

Figura 1.53: Blocos de instruções para alternar a visualização de variáveis.

Uma última dica para visualizarmos os valores das variáveis, de um jeito diferente, sem ocupar espaço no palco: neste caso, a visualização acontecerá na própria área de trabalho. Basta arrastar mais um bloco para a variável que se deseja visualizar e clicar sobre ela, como mostrado na Figura 1.54.

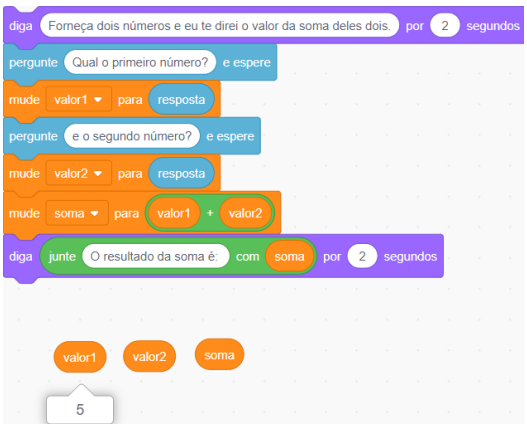

Figura 1.54: Visualização do valor armazenado na variável “valor1” na área de trabalho.

Listas

Vimos que as variáveis são um recurso para armazenarmos dados na memória do computador, para que sejam manipulados por meio de instruções de nossos programas. Vimos, também, que uma variável armazena somente um valor por vez.
Dito isto, vamos imaginar uma aplicação que precise armazenar as temperaturas medidas ao longo de um dia, com um registro de temperatura a cada hora. Um programa para armazenar e processar esses dados precisaria de pelo menos 24 variáveis, uma para cada temperatura medida a cada hora. A programação para manipular esses dados pode ficar confusa. A situação seria ainda pior se nossa aplicação precisasse registrar temperaturas a cada hora durante um longo período, como, por exemplo, por um mês. Em uma situação ideal, esses dados todos ficariam agrupados dentro de uma única **estrutura de armazenamento**, que facilitaria o acesso a todos eles.

Uma lista é uma estrutura de armazenamento que agrupa dados que compartilham de um mesmo significado lógico. A lista funciona como um conjunto de elementos e possui as seguintes características:

- Uma lista possui **itens**, que são os dados propriamente ditos;
- Cada item da lista possui uma informação sobre sua **posição** na lista.
  - O primeiro item de uma lista fica na posição 1.
- Uma lista possui a informação de seu tamanho. Tamanho é a quantidade total de itens presentes na lista.

Podemos encontrar, no Scratch, algumas operações para manipular listas. Uma vez criada, uma lista pode ser manipulada por meio de blocos de instrução que permitem:

- Inserir um novo elemento na lista;
- Encontrar um elemento da lista;
- Substituir um elemento da lista por outro elemento;
- Retirar um elemento da lista;

Os blocos de instrução para essas operações surgem na categoria “Variáveis”, logo após a criação de uma lista.

Vamos compreender melhor as características e operações sobre listas por meio de exercícios práticos.

Exercício 1.19 Crie uma lista de temperaturas, para armazenar as temperaturas registradas nas 4 primeiras horas do dia. A entrada de dados será feita por meio de digitação por teclado.

Solução. A Figura 1.55 mostra um programa para atender ao que foi pedido no Exercício 1.19.

Obs Assim como fazemos com as variáveis, podemos ativar a visualização do conteúdo de uma lista no palco.

■

Exercício 1.20 Aproveite a criação da lista no programa do Exercício 1.19 e adicione a seguinte funcionalidade: informar qual foi a hora registrada na terceira hora do dia.

Solução. A Figura 1.56 mostra uma versão atualizada do programa do Exercício 1.19, para atender ao que foi pedido no Exercício 1.20.

■

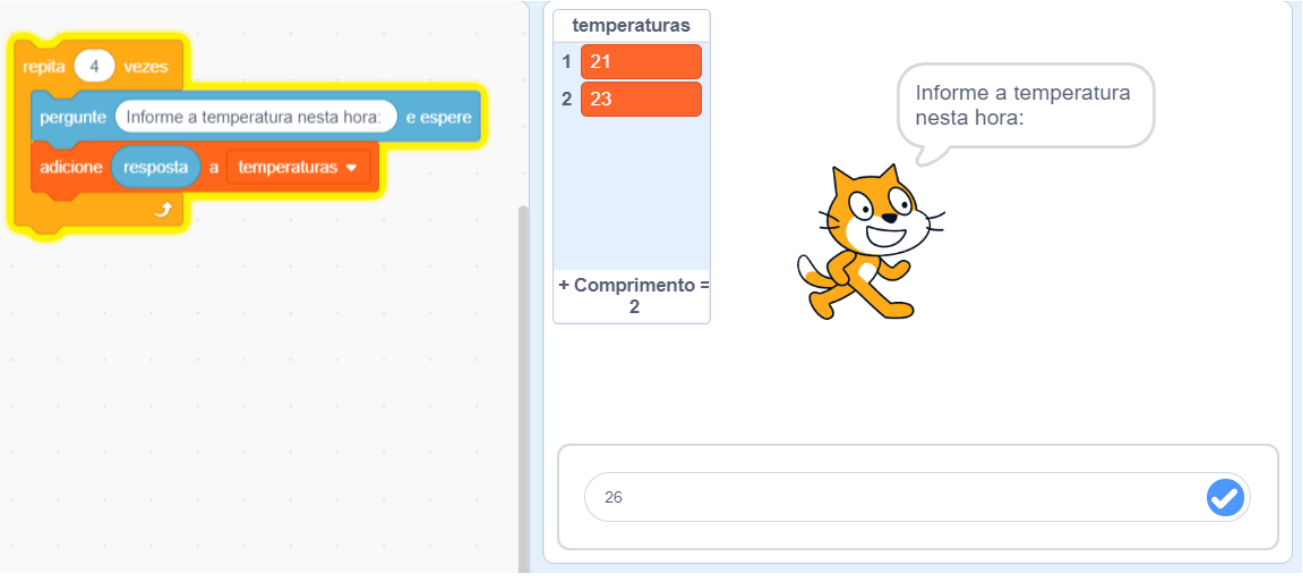

Figura 1.55: Um programa para solucionar o problema proposto no Exercício 1.19. No palco, pode-se ver a digitação do terceiro item da lista. Pode-se ver, também, os dois primeiros itens da lista.

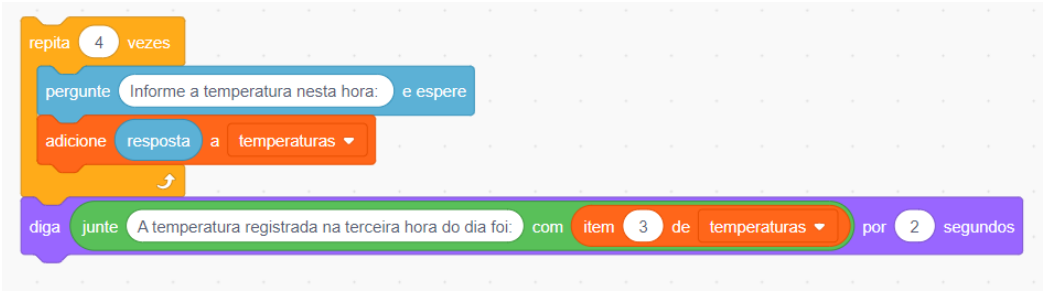

Figura 1.56: Um programa para solucionar o problema proposto no Exercício 1.20.

Constantes

Enquanto as variáveis armazenam valores que podem ser modificados, as constantes são valores fixos. No Scratch, as constantes são os valores – numéricos, letras, textos ou booleanos – que inserimos diretamente no programa. A Figura 1.57 mostra um bloco de instrução com a constante 15 inserida nele. Já na Figura 1.58, a constante "Maria" é usada para ser armazenada na variável "resposta" do programa.

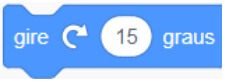

Figura 1.57: Exemplo de constante. Na figura, é o valor 15, inserido no bloco de instrução.

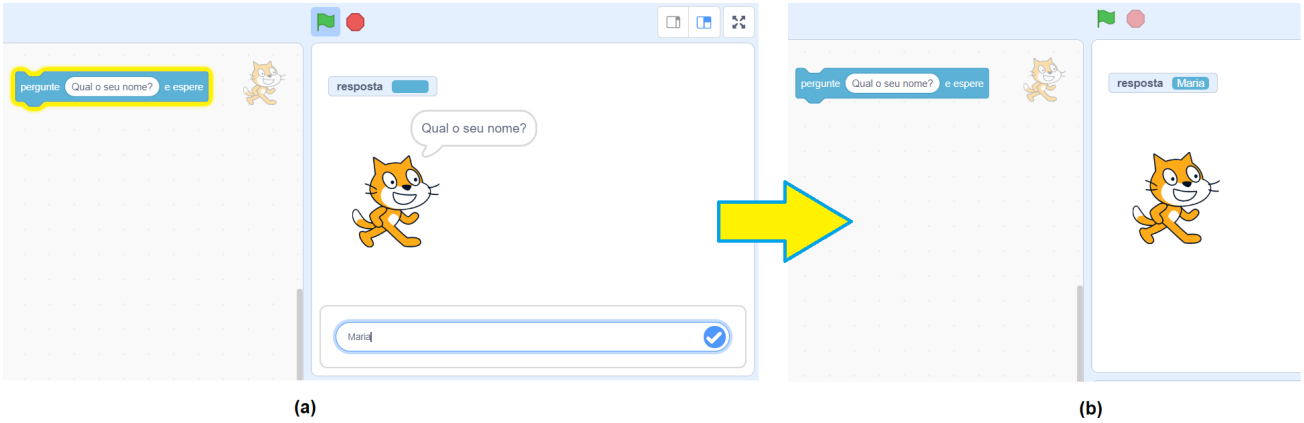

Figura 1.58: Exemplo de constante. Na figura (a), o valor “Maria” é informado pelo usuário do programa. Em (b), a constante “Maria” foi armazenada na variável “resposta” do programa.

1.3.3 – Comandos de Entrada e Saída

O fluxo de execução de um programa geralmente inclui: a **entrada de dados**, o processamento desses dados e a **saída** (disponibilização) de resultados do processamento aplicado aos dados. Nesta seção, vamos estudar com mais detalhes os comandos de entrada e saída no ambiente de programação visual que estamos utilizando neste curso.

Comando de Entrada

A entrada de dados em um programa na linguagem Scratch é feita por meio do bloco de instrução “pergunte ... e espere”, disponível na categoria “Sensores”. A Figura 1.59 apresenta o bloco de instrução e descreve o processo de utilização do mesmo.

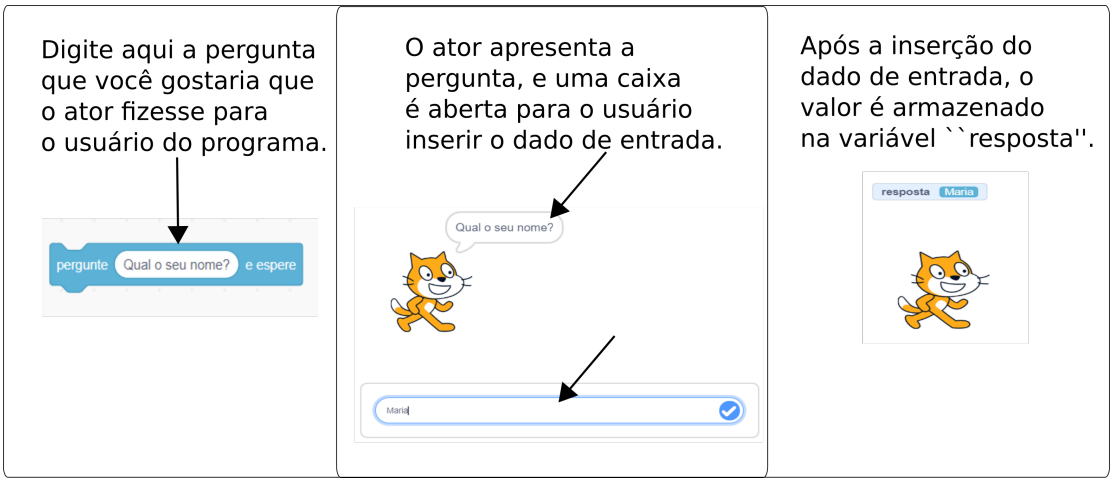

Figura 1.59: Processo de entrada de dados em um programa no Scratch.

Comandos de Saída

Em geral, o propósito de um programa é produzir resultados que são de interesse para alguém. Quando este for o caso, a construção do programa irá requerer o uso de instruções que possibilitem a exibição (saída) de tais resultados.

A saída de dados no Scratch pode ser feita por meio da utilização de alguns blocos de instruções, ilustrados na Figura 1.60.

Figura 1.60: Blocos de instrução para implementar saída de dados no Scratch.

A saída produzida pode ser na forma de texto ou de som. Para saída de som, o Scratch disponibiliza o som "Miau". Para inserir novos sons, use a opção "gravar", disponível nos blocos de instrução "toque o som", como mostra a FIgura 1.61.

Figura 1.61: Opção "gravar" nos blocos de instrução "toque o som".

Obs Importante: para gravar um som, será necessário um dispositivo com microfone e que o Scratch tenha permissão de acesso a ele.

Para as saídas em texto, temos a opção de exibi-las como balãozinho de fala ou de pensamento. O texto exibido pode conter:

- Texto simples;
- Valor armazenado em uma variável;
- Resultado do cálculo de uma expressão;
- Uma combinação de todos os elementos acima, unidos em uma única mensagem.

Obs Para ajudar a criar saídas mais elaboradas, juntando textos e valores de variáveis, devemos utilizar o comando "junte ... com". A Figura 1.62 mostra o bloco de instrução para esse comando.

Figura 1.62: Bloco de instrução “junte ... com”.

Exercício 1.21 Construa um programa que receba um número fornecido pelo usuário e exiba uma mensagem informando o valor do dobro do número fornecido. A mensagem deve conter um texto, iniciando com “O dobro de”, seguida do valor informado, seguido de “ é: ”, seguido do valor calculado. Por exemplo, se o usuário informar o valor 11, o programa vai exibir a mensagem: “O dobro de 11 é: 22”.

Solução. A Figura 1.63 apresenta uma solução para o problema proposto.

Figura 1.63: Um programa para solucionar o problema proposto no Exercício 1.21.

Obs Teste o programa e verifique se a mensagem exibida por ele apresenta os espaçamentos necessários para a correta exibição da mensagem.

■

Exercício 1.22 Faça um programa para obter dois números e exibir o resultado da média aritmética simples desses números. Informe o resultado em uma mensagem bem formatada, mostrando os dois valores de entrada e o valor do resultado, em uma frase contendo a informação completa.

Solução. A Figura 1.64 apresenta uma solução para o problema proposto.

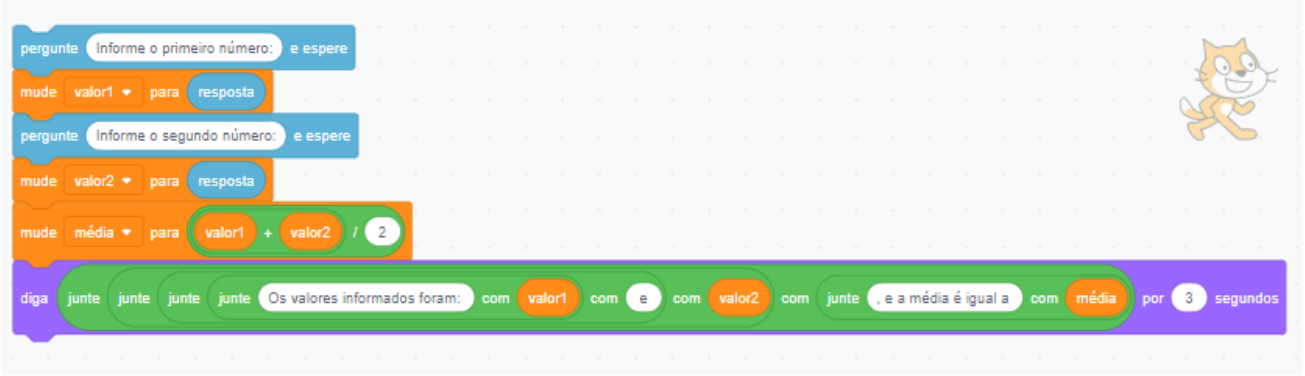

Figura 1.64: Um programa para solucionar o problema proposto no Exercício 1.22.

■

1.4 – Operadores

1.4.1 – Expressões Aritméticas

Os blocos de instrução referentes às operações aritméticas estão disponíveis na categoria "Operadores". A Figura 1.65 mostra os blocos referentes às operações de soma, subtração, multiplicação e divisão.

Figura 1.65: Blocos referentes aos operadores aritméticos.

Obs No caso do Scratch, é necessário prestar bastante atenção na montagem dos blocos para gerar a ordem correta de execução das operações!

Exercício 1.23 Monte blocos no Scratch para produzir os resultados corretos das expressões abaixo:

(a) $3 + 5{\times}6 = 33$
(b) $3{\times}3 + 2{\times}4 + 2{\times}5 = 27$
(c) $\frac{180-56}{4} = 31$
(d) $8 + 5 - 2{\times}8 = -3$
(e) $8 + (5 - 2){\times}8 = 32$

Solução. A Figura 1.66 mostra a correta montagem para a representação das expressões do Exercício 1.23.

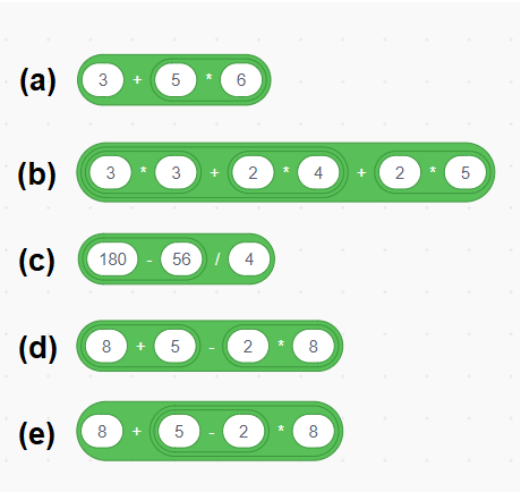

Figura 1.66: Representações em blocos das expressões aritméticas do Exercício 1.23.

■

Exercício 1.24 Joãozinho ficou feliz por ter tirado nota 10 em sua segunda prova! Como ele tinha tirado 4 na primeira, ele acredita que com essa nova nota, ele vai poder passar direto na eletiva de programação! Porém, um detalhe do qual ele não se deu conta é de que a primeira nota tem peso 2 e a segunda nota tem peso apenas 1. Desta forma, a média dele foi 6 e ele terá de fazer um exame de recuperação. Apresente a expressão aritmética referente ao cálculo dessa média usando blocos de instrução do Scratch.

Solução. Vamos aproveitar a oportunidade para exercitar outros conteúdos já estudados, como entrada e saída de dados, além das expressões aritméticas, para criar um programa completo. A Figura 1.67 mostra um programa para atender ao que foi pedido no Exercício 1.24.

Figura 1.67: Programa para a solução do Exercício 1.24.

■

Além dos operadores aritméticos, o Scratch disponibiliza algumas **funções**, apresentadas na Figura 1.68.

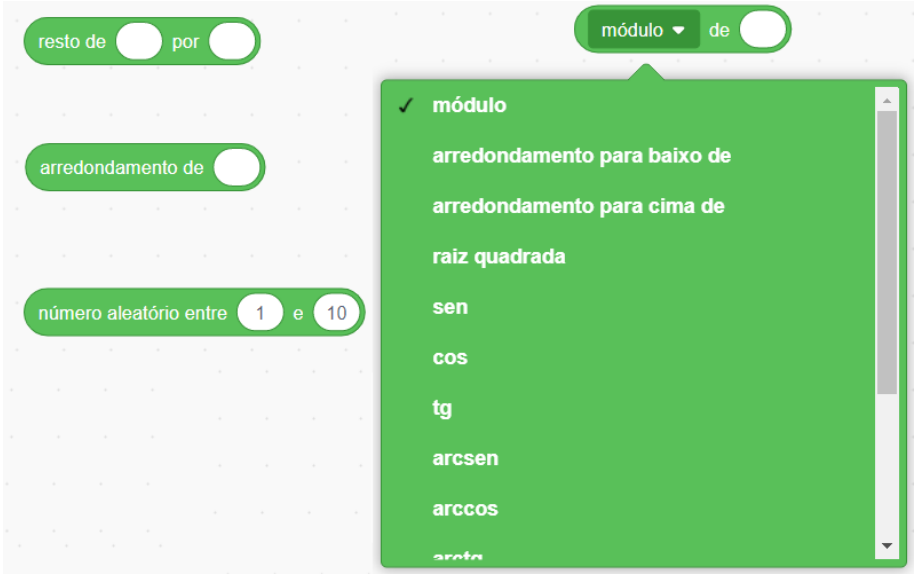

Figura 1.68: Blocos de instruções para as funções disponíveis no Scratch.

Exercício 1.25 Faça um programa para calcular a distância Euclidiana entre dois pontos no espaço bidimensional (2D).

Solução. Para dois pontos $P = (p_x, p_y)$ e $Q = (q_x, q_y)$, a distância Euclidiana entre eles é dada pela equação

$$d = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}. \tag{1.1}$$

Nosso programa precisará conhecer as coordenadas x e y de cada um dos dois pontos, para poder aplicar a equação. A Figura 1.69 apresenta um programa para solucionar o Exercício 1.25.

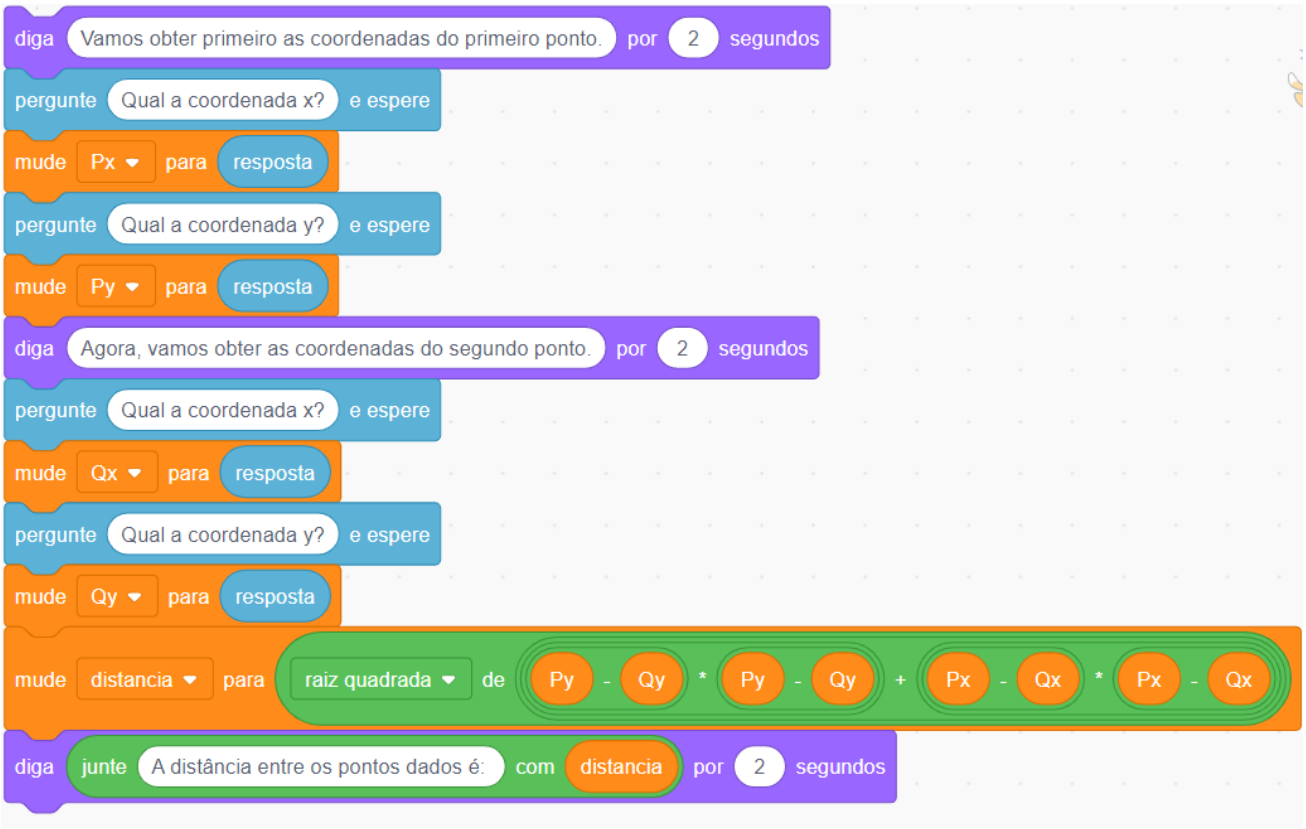

Figura 1.69: Um programa para calcular a distância Euclidiana entre dois pontos no espaço 2D.

■

Exercício 1.26 O gato está de volta dizendo que vai sortear uma passagem para a excursão para a Ilha do Sol, onde tudo começou, há um tempo atrás! Como o gato só tem uma passagem e nós somos 10 pessoas interessadas, ele vai sortear um número de 1 a 10 para definir quem vai ser o(a) felizardo(a). Faça um programa para executar o sorteio!

Solução. Para efetuar o sorteio, vamos usar uma função que é capaz de gerar um número aleatório entre dois valores escolhidos pelo(a) programador(a). O Scratch disponibiliza uma função com essa capacidade por meio do bloco de instrução "número aleatório entre ... e ...". A Figura 1.70 apresenta um programa que utiliza essa função para solucionar o problema proposto no Exercício 1.26.

■

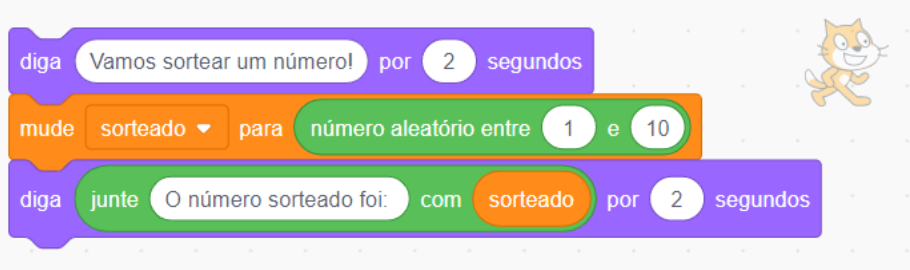

Figura 1.70: Um programa para sortear um número de 1 a 10.

1.4.2 – Operadores Relacionais

Algumas operações nas instruções que compõem um programa envolvem a comparação entre dois dados valores. Em outras situações, podemos precisar comparar valores com resultados de expressões. Para efetuar tais comparações, utilizamos os **operadores relacionais**. A Figura 1.71 apresenta os blocos de instrução referentes a operadores relacionais no Scratch.

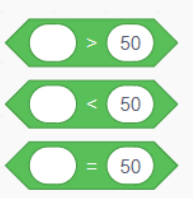

Figura 1.71: Blocos de instrução referentes a operadores relacionais no Scratch.

Obs É importante lembrar que as expressões escritas com operadores relacionais resultam em valores lógicos. Em outras palavras, uma expressão que contém operadores relacionais sempre vai resultar em VERDADEIRO ou FALSO.

Exercício 1.27 O gato quer testar seus conhecimentos sobre a tabuada de multiplicação! Para ser bonzinho, ele vai te fazer uma pergunta considerando somente os números de 1 a 9. Uma vez que o gato apresente a pergunta, você deve digitar a resposta e ele te dirá se você acertou ou não. Mas, para tudo isso funcionar, você precisar criar o programa primeiro.

Solução. Vamos ao trabalho, então! Usaremos um operador relacional para comparar a resposta digitada com a multiplicação a ser feita pelo programa, para determinar se o usuário acertou ou não a operação de multiplicação apresentada pelo gato. Como os valores envolvidos no desafio serão escolhidos pelo gato, vamos usar a função que gera números aleatórios para simular essas escolhas. A Figura 1.72 apresenta um programa para o funcionamento do jogo proposto. ■

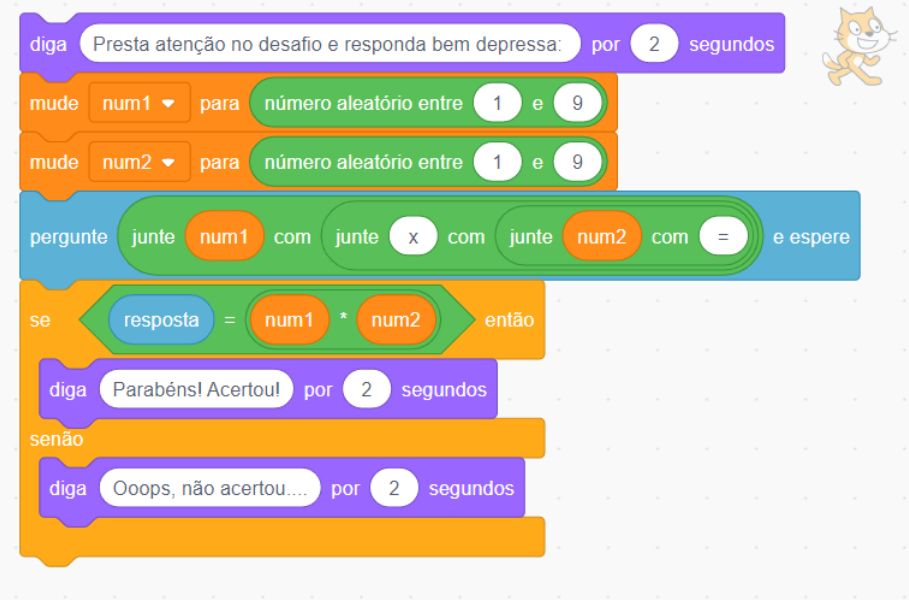

Figura 1.72: Um programa para o jogo proposto no Exercício 1.27.

1.4.3 – Operadores Lógicos

A linguagem Scratch disponibiliza, também, blocos de instrução referentes aos operadores lógicos, apresentados na Figura 1.73.

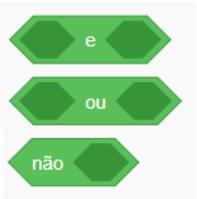

Figura 1.73: Blocos de instrução referentes aos operadores lógicos.

As **expressões lógicas** são em geral utilizadas para definir condições complexas (por exemplo, que combinam várias expressões relacionais).

Obs Assim como foi visto para as expressões aritméticas, é necessário prestar bastante atenção na montagem dos blocos para gerar a ordem correta de execução das operações lógicas!

Obs No Scratch, os valores VERDADEIRO e FALSO são identificados como true e false, respectivamente.

Exercício 1.28 Revisando os operadores relacionais do Scratch, eu vi que só tem blocos de instrução referentes aos operadores "$>$", "$<$" e "$=$". Como eu faço se eu quiser fazer um teste de uma relação "$\geq$" ou "$\leq$" no meu programa?

Solução. Podemos utilizar operadores lógicos para juntar os operadores. Por exemplo, o teste "$x \geq y$" é equivalente ao teste "$x > y$ OU $x = y$". Desta forma, a relação "$x \geq y$" pode ser montada no Scratch como mostra a Figura 1.74.
De maneira similar, um teste de "$x \leq y$" pode ser montado como apresentado na Figura 1.75. ■

Figura 1.74: Montagem para teste de "$x \geq y$".

Figura 1.75: Montagem para teste de "$x \leq y$".

Exercício 1.29 Monte blocos no Scratch para produzir os resultados corretos das expressões abaixo:

(a) **NAO** $3 \leq 3$ **E** $2 = 4 \times 0.5$
(b) **NAO** $2 + 0.5 < 3$ **OU** $2 + 1 \leq 3$ **E** $2 = 2 \times 1$
(c) **NAO**(A = resposta **OU** B < resposta) **E** (A + 1 = 2 **OU** C > 5)
(d) A = 1 **OU** B = 2 **E** A = 1 **E** C = 3
(e) (A = 1 **OU** B = 2) **E** (A = 1 **E** C = 3)

Solução. A Figura 1.76 mostra a correta montagem para a representação das expressões do Exercício 1.29.

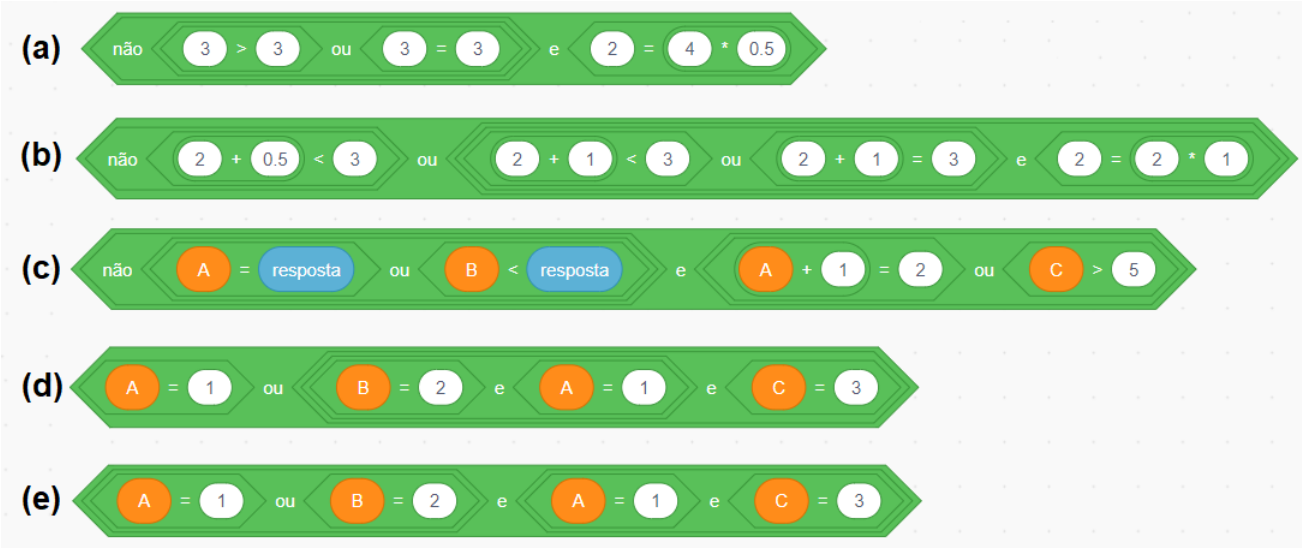

Figura 1.76: Representações em blocos das expressões lógicas do Exercício 1.29.

■

Exercício 1.30 O gato quer saber se ele poderá viajar nas próximas férias. Ele só viajará de férias se:

- Ele puder levar o cachorro dele
- Se a família dele for junto
- Os amigos do irmão dele não forem também
- As passagens de avião estiverem baratas ou se ele ganhar a passagem de alguém!

Faça um programa para ajudar o gato a obter as informações sobre cada uma das condições

e, com base nessas informações, possa decidir se vai viajar ou não.

Solução. O programa, apresentado na Figura 1.77, tem 5 variáveis, uma para cada uma das condições a serem testadas. Cada uma das condições receberá uma resposta, que poderá ser "sim" ou "não", digitadas por um(a) usuário(a) durante a execução do programa. O teste feito na estrutura de decisão é a expressão lógica que determina se as condições informadas resultarão ou não em um cenário positivo para o gato viajar.

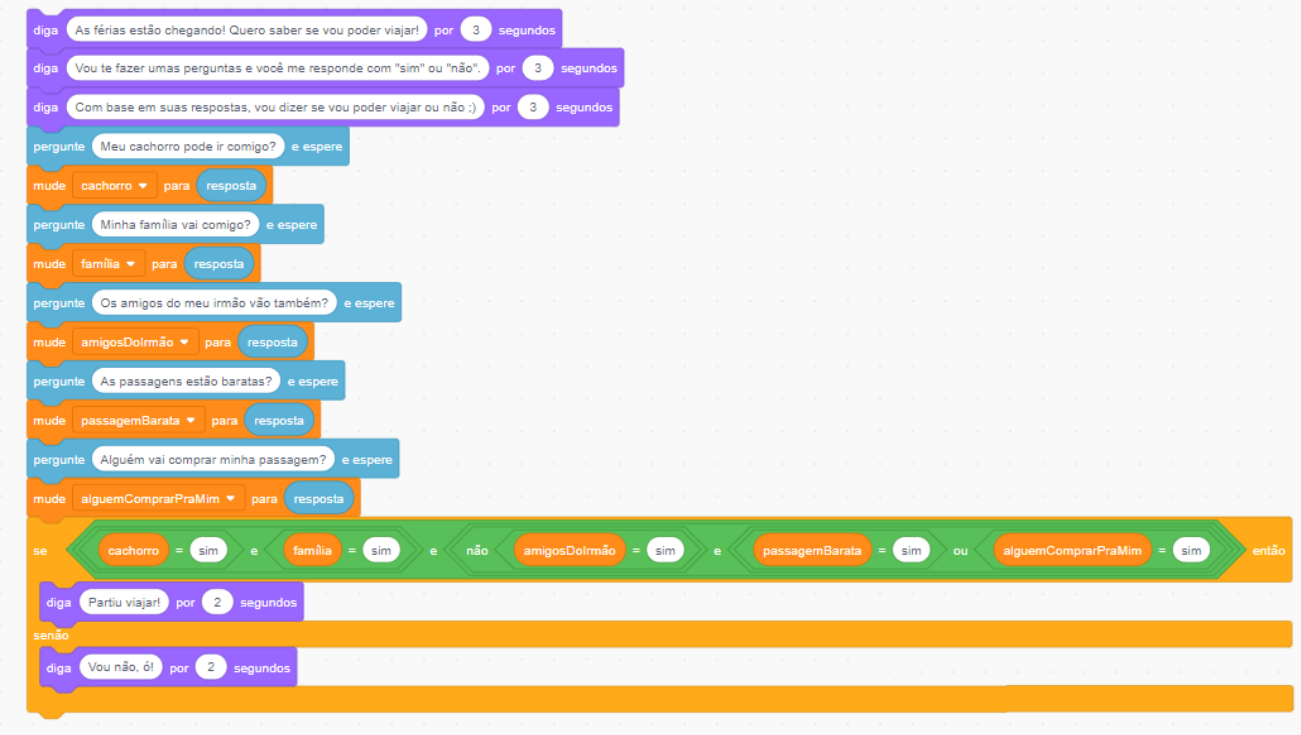

Figura 1.77: Um programa para resolver o problema proposto no Exercício 1.30.

■

1.5 – Estruturas de Decisão

Sabemos que um programa executa as instruções na ordem em que foram listadas pelo(a) programador(a). Porém, em algumas situações, precisamos listar instruções que deverão ser executadas somente se uma determinada condição se verificar. Caso contrário, tais instruções não deveriam ser executadas, e o fluxo de execução do programa seria desviado para instruções seguintes. Observe a situação descrita no Exemplo 1.5:

■ **Exemplo 1.5** O gato vai buscar pra você um, e somente um, item na geladeira. Você pode escolher entre uma maçã ou uma geléia. Uma vez que você tenha informado o que quer, ele vai buscar o item. A Figura 1.78 apresenta um programa para implementar essa tarefa.

Observando o programa da Figura 1.78, vemos dois conjuntos de comandos, um para cada ação a ser realizada pelo gato. Note que o gato realizará somente uma dessas duas ações. A primeira ação é o ato de ir buscar a maçã, detalhada na Figura 1.79.

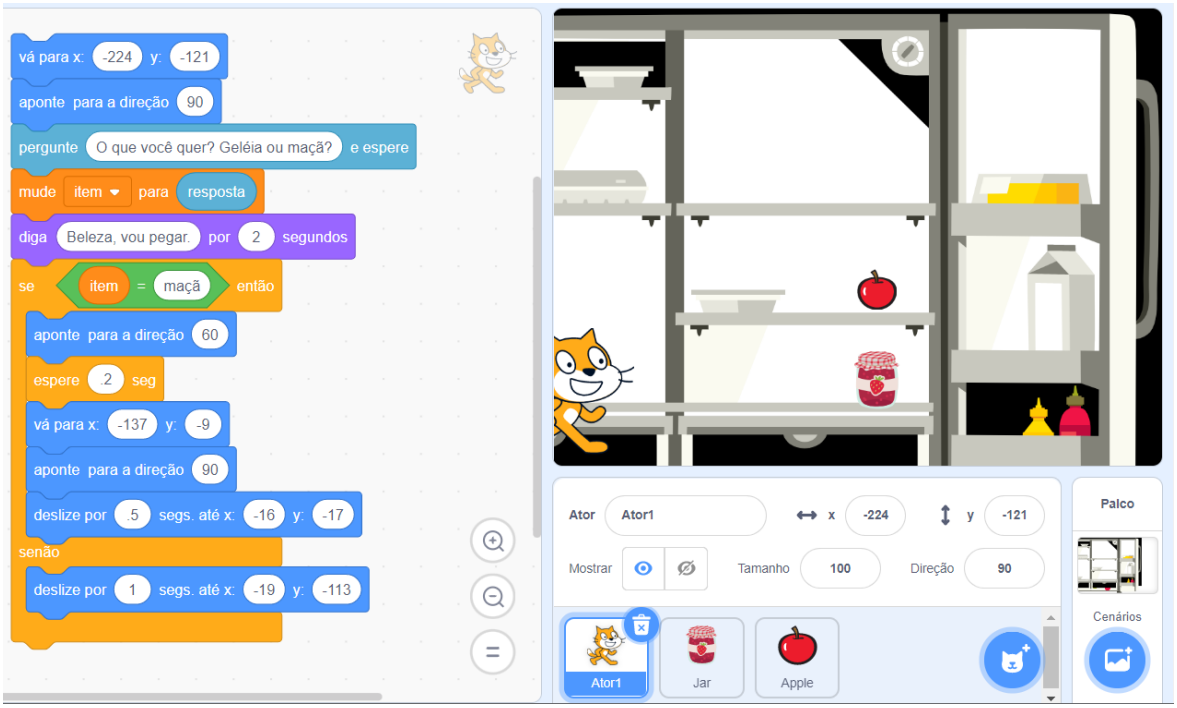

Figura 1.78: Um programa para fazer o gato buscar um item na geladeira.

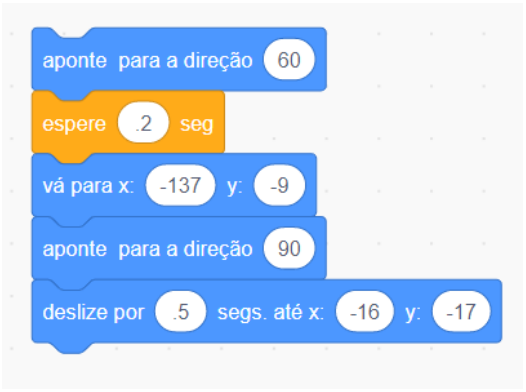

Figura 1.79: Sequência de instruções para fazer o gato buscar a maçã na geladeira.

A segunda ação, detalhada na Figura 1.80 é o ato de ir buscar a geléia.

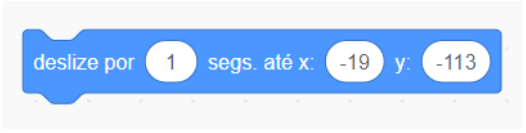

Figura 1.80: Sequência de instruções para fazer o gato buscar a geléia na geladeira.

Observe que se montássemos o programa em sequência apenas juntando as instruções das Figuras 1.79 e 1.80, o gato ia acabar indo buscar os dois itens, o que não corresponde ao comportamento esperado para o programa! Será necessário decidir, dentre essas duas ações, qual delas será executada. Assim, para possibilitar a escrita de todos os comandos que desejamos em nosso programa, e ainda assim garantindo que, a depender de uma certa condição apenas parte deles será executada, usaremos as estruturas de decisão. A estrutura de decisão "se...então...senão" foi aplicada na implementação do programa completo deste exemplo, como pode ser visto na Figura 1.78. ■

Além de determinar que comandos devem ser executados, as estruturas de decisão podem ser aplicadas para prevenir a execução de tarefas desnecessárias. Observe o Exemplo 1.6.

■ **Exemplo 1.6** O gato foi convidado para uma partida de futebol. Ele tá meio com preguiça, e precisa de um incentivo para sair do canto. Ele vai te perguntar se você acha que ele deve ir ou não. Se você responder "sim", ele vai se deslocar para o meio do campo. Se você responder qualquer outra coisa (um "não", ou um "talvez", ou qualquer outra coisa), ele vai continuar como está. A Figura 1.81 apresenta um programa que implementa essa situação.

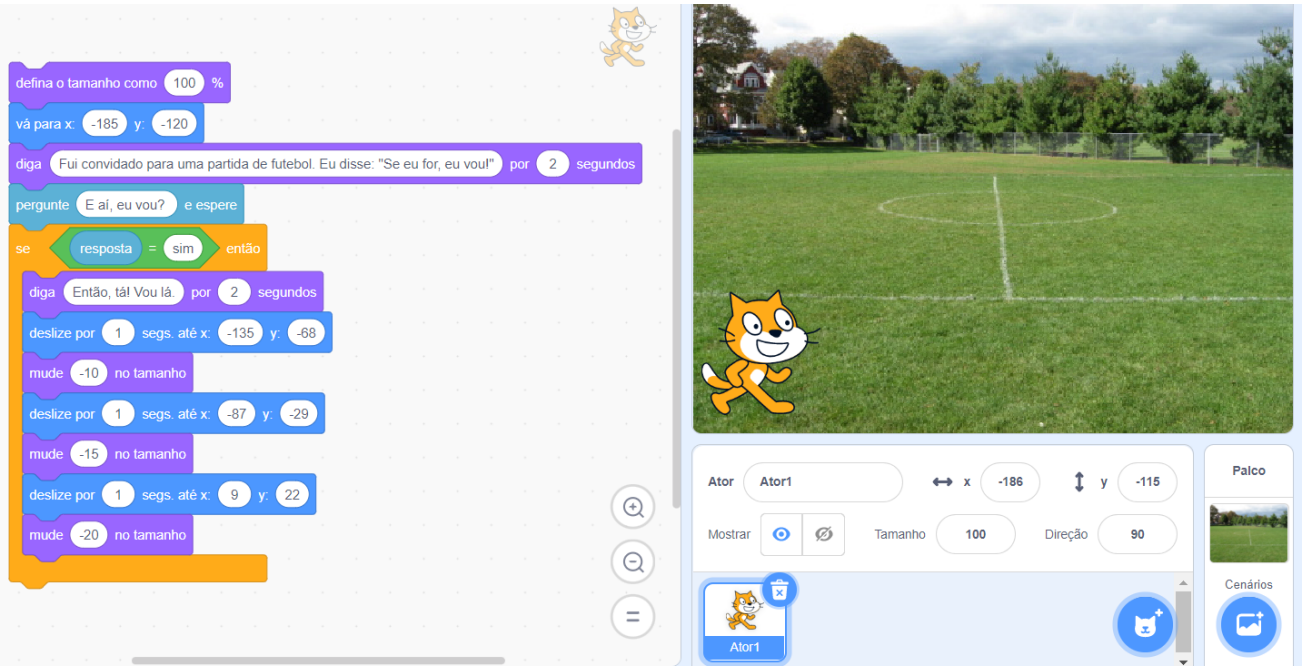

Figura 1.81: Um programa para implementar a situação apresentada no Exemplo 1.6.

Observe, neste exemplo, que a estrutura de decisão contém apenas a porção "se...então". Nenhuma ação específica será necessária caso o(a) usuário(a) do programa digite uma resposta diferente de "sim". Assim, todos os comandos referentes à caminhada do gato até o centro do campo serão executados somente se for realmente necessário. ■

Obs Resumindo, em alguns programas, iremos usar a estrutura 'se...então", e em outros, a estrutura "se...então...senão". Esta última, nós usaremos sempre que tivermos instruções a serem executadas no caso específico de a condição testada resultar em FALSO.

■ **Exemplo 1.7** O programa na Figura 1.82 implementa uma aplicação em que o gato pergunta se o(a) usuário(a) quer ler uma revelação bombástica. Observe que a estrutura utilizada foi "se...então". Assim, nenhuma ação específica será efetuada no caso de o(a) usuário(a) fornecer uma entrada difernete de "sim". O fluxo de execução continua e o comando seguinte é executado, exibindo a mensagem final "Obrigado por usar nosso sistema". Note que essa mensagem final não está condicionada à resposta do(a) usuário(a) – ela será exibida independentemente da resposta dada.

■

Exercício 1.31 Inclua no programa do Exemplo 1.7 a capacidade de fazer o gato dizer: "Tudo bem, fica para a próxima oportunidade", caso o(a) usuário(a) digite uma resposta diferente de "sim".

Solução. Perceba que a mensagem a ser incluída no programa da Figura 1.82 será exibida especificamente no caso do teste "resposta = sim" resultar em false. Desta forma, vamos

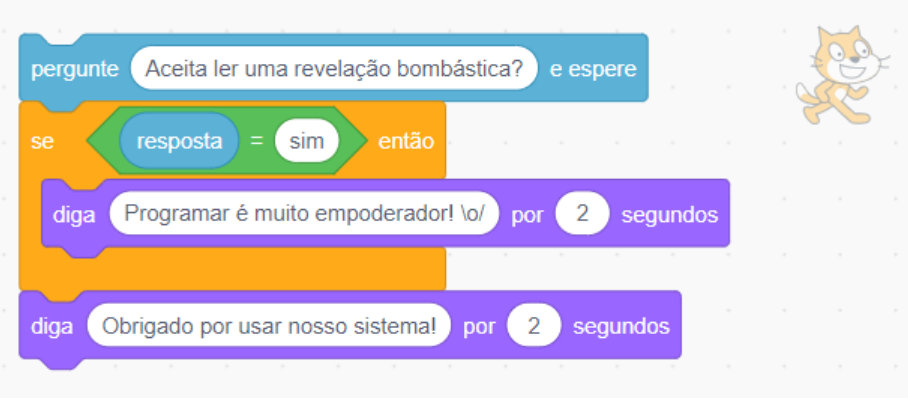

Figura 1.82: Um programa para implementar a situação apresentada no Exemplo 1.7.

precisar utilizar a estrutura "se...então...senão" para adicionar o comando para a exibição da nova mensagem em seu devido lugar. A Figura 1.83 apresenta a versão atualizada do programa.

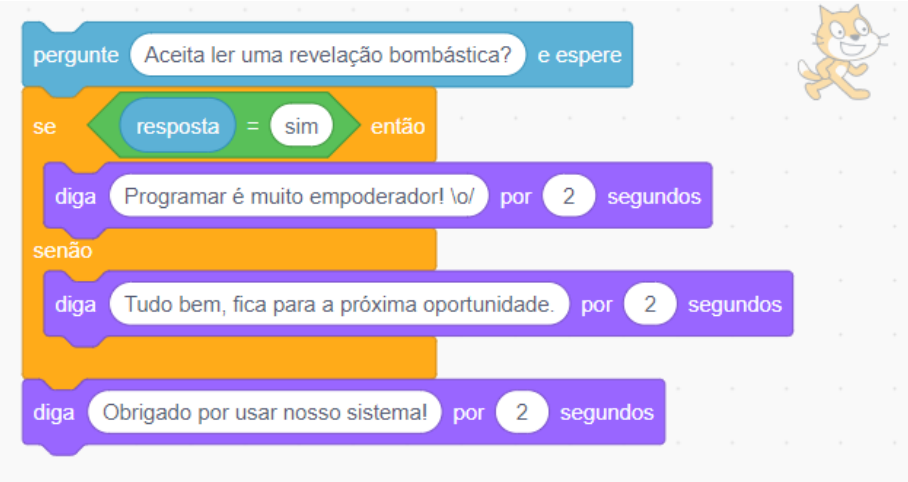

Figura 1.83: Um programa para implementar a solução do Exercício 1.31.

Obs Observe, no programa da Figura 1.83, que a exibição da mensagem final segue sendo executada, independente da resposta fornecida pelo(a) usuário(a). Esse comando está **fora da estrutura de decisão**, e é executado **seguindo a convenção de sequenciação**.

■

1.5.1 – Desvio condicional simples

Podemos pensar em uma estrutura de decisão como uma válvula, que desvia o fluxo da execução do programa, conforme for o resultado de um teste feito. Esse teste pode ser compreendido como uma "pergunta" que seu programa faz para decidir quais são os próximos passos a serem executados. Em situações em que precisamos de apenas uma "pergunta" para decidir o fluxo de execução, vamos utilizar um desvio condicional simples.

Exercício 1.32 Faça um programa que precisa informar se dois números dados são iguais ou são diferentes entre si.

Solução. O programa consiste em obter os dois valores informados pelo(a) usuário(a) e compará-los. Para solucionar o problema, é suficiente comparar se um valor é igual ao outro. Assim, precisamos fazer apenas uma "pergunta" para decidir qual será o comando a ser exibido

em seguida: se vai informar que os números são iguais ou se vai informar que os números são diferentes. A Figura 1.84 apresenta um programa para resolver o problema proposto no Exercício 1.32.

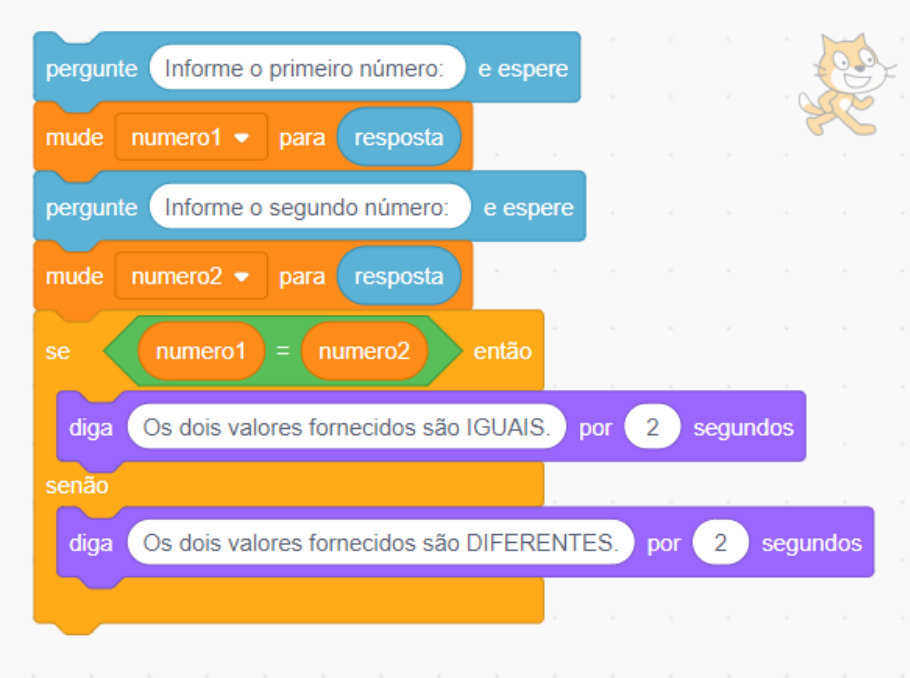

Figura 1.84: Um programa para implementar a solução do Exercício 1.32.

Observe, na figura, que apenas um teste foi necessário para implementar o programa: o teste se numero1 é igual ao numero2. Isto caracteriza um desvio condicional simples. ■

Exercício 1.33 O gato agora quer realizar operações de divisão de números dados. E ele está tão esperto, que só realiza a divisão se o denominador for diferente de zero. Afinal de contas, divisão por zero não existe! Faça a programação para dar toda essa esperteza para o gato.

Solução. A Figura 1.85 apresenta um programa para implementar a solução do problema proposto no Exercício 1.33. Observe que apenas um teste é necessário para determinar que instrução deverá ser executada.

Figura 1.85: Um programa para implementar a solução do Exercício 1.33.

■

1.5.2 – Desvio condicional encadeado

Em algumas situações, é comum depararmos com situações em que uma resposta a uma pergunta nos leva a uma segunda pergunta (que pode nos levar a uma terceira, e por aí vai).

Vamos relembrar o clássico problema de se determinar o resultado final de um(a) aluno(a), com base em sua média final.

As possibilidades seriam três, a saber:

- Se a média ≥ 7.0, então passou direto!
- Se a média está entre 6.9 e 4.0, então vai para a recuperação.
- Se a média < 4.0, então reprovou direto.

Cada resultado deve ser inferido por meio de algumas perguntas que precisamos fazer. Um conjunto possível de tais perguntas está organizado abaixo:

(a) Qual a média obtida por este(a) aluno(a)?
(b) A média informada é maior do que 7?
    - ...e se não for, ela é pelo menos maior do que 4?

Observe que se a resposta à pergunta do item (b) for positiva, podemos afirmar com toda a certeza que o(a) aluno(a) passou direto! No entanto, se a pergunta tiver sido respondida com uma negativa, ainda não teremos elementos suficientes para dizer qual o resultado final do(a) aluno(a). Uma média inferior a 7 pode levar o (a) aluno(a) a dois possíveis resultados finais: recuperação ou reprovação direta. Diante disto, é necessário se fazer uma nova pergunta, indicada no sub-item do item (b).

Esse cenário em que uma pergunta (ou um teste) leva a necessidade de um teste adicional é modelado em algoritmos e programação com o aninhamento (ou encadeamento) de estruturas de decisão.

Encadear estruturas de decisão no Scratch é bem simples: basta encaixar os blocos de estruturas de decisão de acordo com as necessidades do seu programa. A Figura 1.86 mostra um exemplo de estruturas de decisão encaixadas para implementar encadeamento de desvios condicionais.

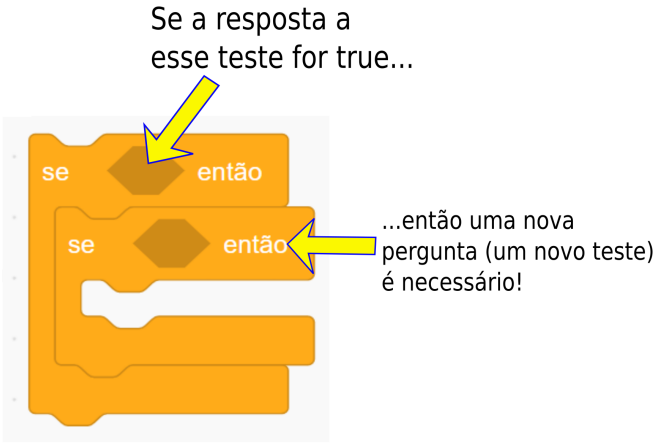

Figura 1.86: Um exemplo de encadeamento de estruturas de decisão e seu significado.

Os blocos de instrução para estruturas de decisão nos permitem criar inúmeras combinações de encadeamentos. A Figura 1.87 mostra algumas das possibilidades.

Figura 1.87: Diferentes possibilidades de encadeamento de estruturas de decisão.

Exercício 1.34 Já que mencionamos o caso clássico do algoritmo que determina o resultado final de um(a) aluno(a), dada a sua média final, faça um programa para este problema.

Solução. A Figura 1.88 apresenta um programa para solucionar o problema proposto no Exercício 1.34.

Figura 1.88: Um programa para resolver o problema proposto no Exercício 1.34.

Obs Nunca é demais relembrar que o programa apresentado é **apenas uma das soluções possíveis**. Você pode escrever um programa diferente e obter o mesmo resultado.

■

Exercício 1.35 O gato está empolgado! Ele quer que você forneça para ele três números, em qualquer ordem, e ele vai listar esses três números em ordem crescente! Por exemplo, se você digitar os números 4, 7, e 2, ele vai te mostrar a lista ordenada 2, 4, 7. Faça um programa para concretizar essa façanha do gato.

Solução. Este é um exercício clássico quando se estuda aninhamento de estruturas de decisão. Ao se tomar três valores numéricos A, B e C, temos as seguintes possibilidades de ordenação por ordem crescente:

- A > B > C
- A > C > B
- C > A > B
- B > A > C
- B > C > A
- C > B > A

Para identificar cada possível caso, utilizaremos estruturas de decisão aninhadas, para conduzir os testes entre os três valores fornecidos. A Figura 1.89 apresenta um programa para resolver este problema.

■

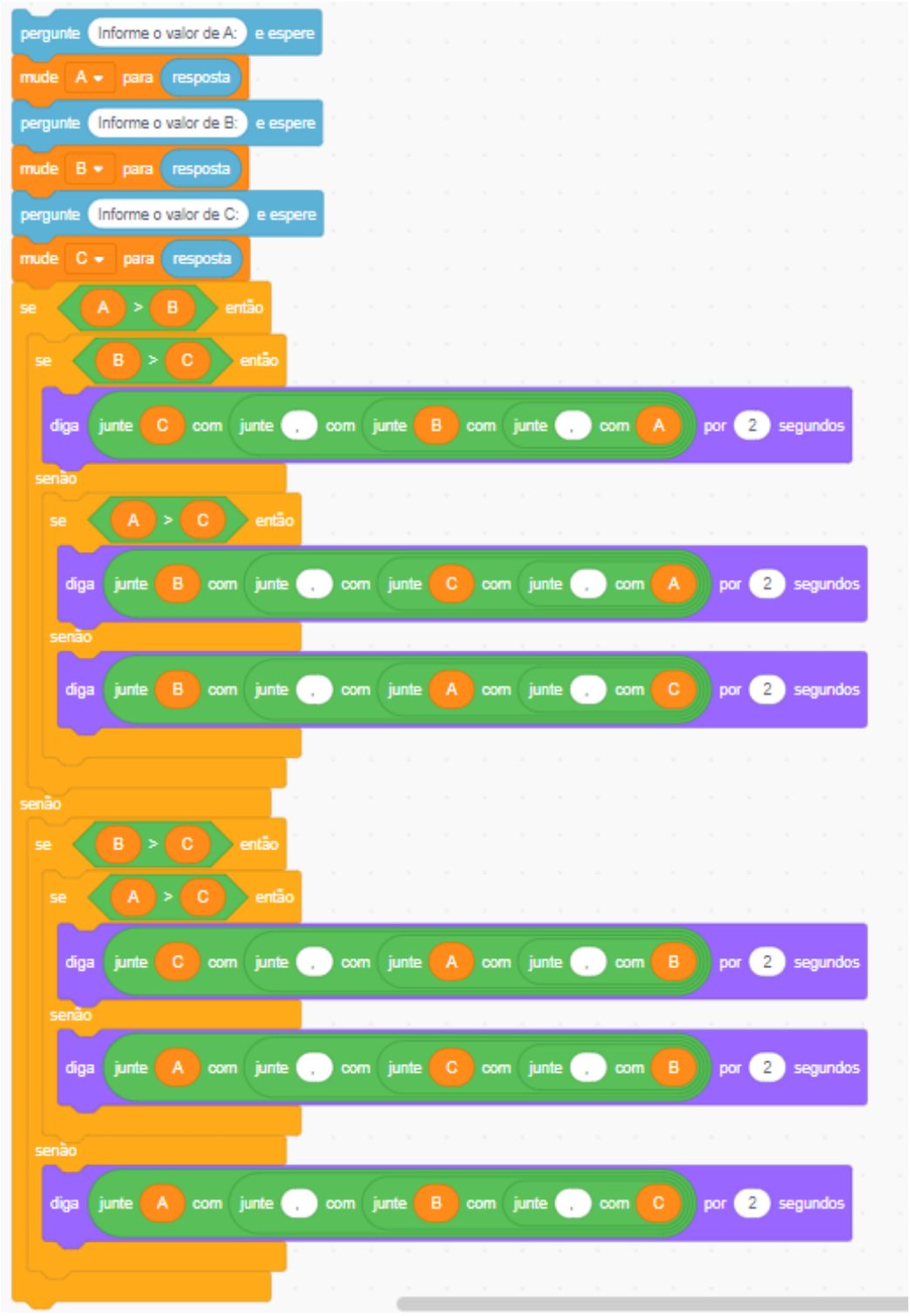

Figura 1.89: Um programa para fazer o gato exibir três números fornecidos em ordem crescente.

Exercício 1.36 Dado o programa da Figura 1.89, mostre como seria o fluxo de execução para as entradas: 7, 5 e 9.

Solução. A Figura 1.90 aponta os comandos executados no programa para a entrada de dados indicada no Exercício 1.36.

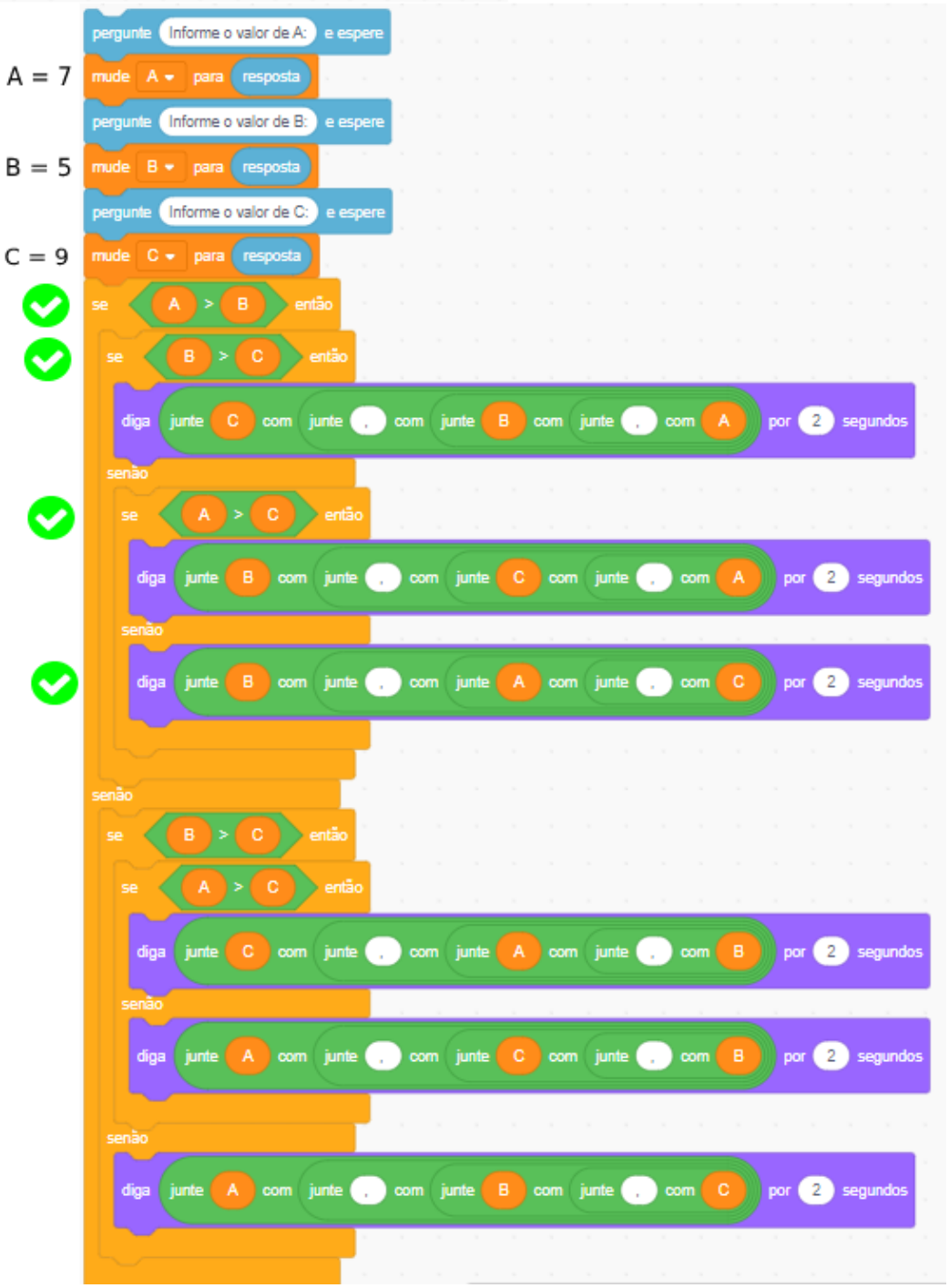

Figura 1.90: Comandos executados no programa do Exercício 1.35, com as entradas indicadas no Exercício 1.36.

■

Exercício 1.37 O gato só quer ser o educado e sabidão, dando Bom dia, Boa tarde e Boa noite, conforme a hora atual do relógio do computador. Faça um programa para concretizar essa proeza do gato.

Solução. Este exercício será uma boa oportunidade para explorar um pouco mais as opções que o Scratch disponibiliza na categoria "Sensores". Olhando os blocos de instrução dessa categoria, vemos um que captura a informação da hora atual do relógio do computador. Vai cair como uma luva para o nosso projeto! A Figura 1.91 mostra o bloco de instrução que iremos utilizar em nosso programa.

Figura 1.91: Bloco de instrução que obtém a hora atual do computador.

A lógica de programação incluirá a descoberta da hora atual e comparar com os horários que convencionamentos como manhã, tarde e noite. Assim, o gato dirá:

- "Bom dia", se hora no computador for menor que 12;
- "Boa tarde, se hora no computador for maior ou igual a 12 e menor que 18;
- "Boa noite, se hora no computador for maior ou igual a 18.

A Figura 1.92 apresenta um programa para o problema exposto no Exercício 1.37.

Figura 1.92: Um programa para fazer o gato dar uma saudação de acordo com a hora obtida do computador.

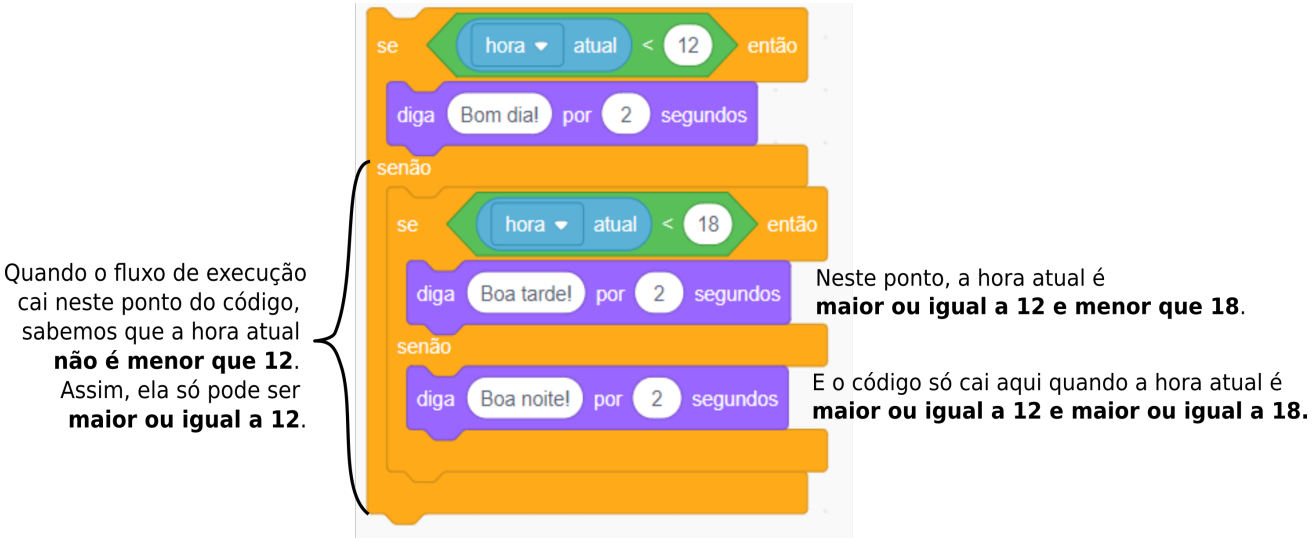

Figura 1.93: Lógica implementada no programa do Exercício 1.37.

Observe como o encadeamento de estruturas de decisão ajuda a descobrirmos o turno atual (manhã, tarde ou noite). A Figura 1.93 mostra a lógica por trás da programação apresentada neste exercício.

■

Exercício 1.38 Faça um programa para encontrar as raízes de uma equação do segundo grau.

Solução. Uma equação do segundo grau é escrita como

$$ax^2 + bx + c = 0. \tag{1.2}$$

Nosso programa precisará realizar as seguintes tarefas:

1 - Obter os valores de a, b e c.
2 - Calcular o discriminante Δ, obtido pela fórmula

$$\Delta = b^2 - 4ac. \tag{1.3}$$

3 - Aplicar a fórmula de Bhaskara:

$$x = \frac{-b \pm \sqrt{\Delta}}{2a}. \tag{1.4}$$

Com base nesses passos, vamos construir um programa para resolver o problema, como apresentado na Figura 1.94.

■

Figura 1.94: Um programa para obter as raízes de uma equação do segundo grau.

Exercício 1.39 O gato quer saber a sua idade para te informar a sua classe eleitoral. Dependendo da sua idade, ele vai te informar que você é:

- não eleitor (abaixo de 16 anos);
- eleitor obrigatório (entre 18 e 65 anos);
- eleitor facultativo (entre 16 e 18 anos ou acima dos 65 anos).

Faça um programa para que o gato realize essa tarefa.

Solução. A Figura 1.95 apresenta um programa para o problema proposto no Exercício 1.39.

Figura 1.95: Um programa para habilitar o gato a determinar a classe eleitoral de uma pessoa, com base na idade informada.

■

Exercício 1.40 O gato agora vai querer que você informe as medidas dos três lados de um triângulo. Com isso, ele vai poder classificá-lo como EQUILÁTERO, ISÓSCELES e ESCALENO. Faça um programa para que o gato ganhe essa habilidade.

Solução. A Figura 1.96 apresenta um programa para o problema proposto no Exercício 1.40.

■

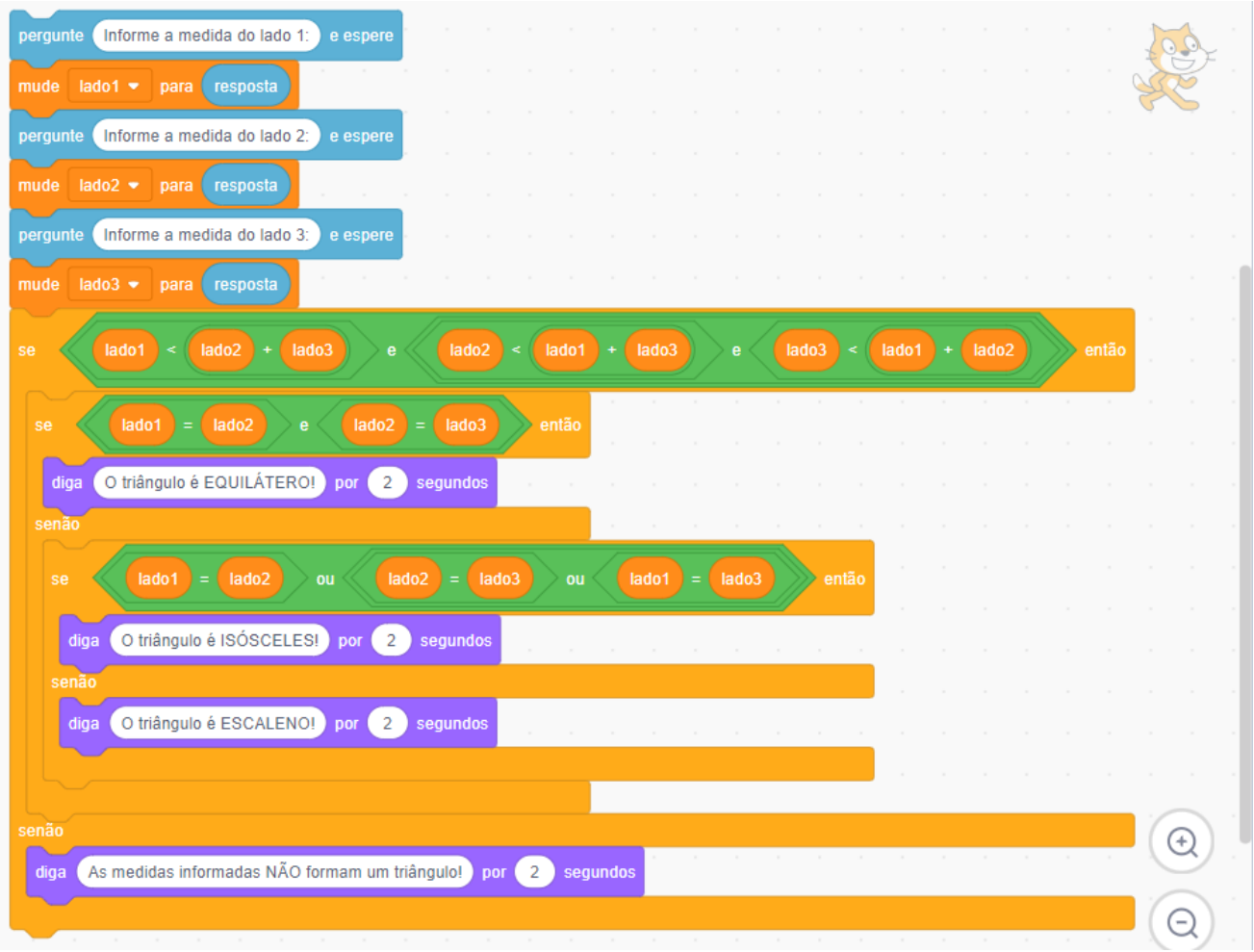

Figura 1.96: Um programa que classifica triângulos, conforme as medidas de seus lados.

1.6 – Estruturas de Repetição

Algumas ações precisam ser executadas repetidas vezes em um programa. No entanto, isso não significa que precisamos escrever as mesmas instruções repetidas vezes. Vamos aprofundar, nesta seção, alguns conceitos e fundamentos importantes para aprimorarmos nossos conhecimentos sobre as estruturas de repetição.

Observando a Figura 1.35, podemos relembrar as três estruturas de repetição disponíveis na linguagem Scratch:

- **repita até que**: realiza um teste e repete instruções dentro da instrução até que o teste resulte em true.
- **repita ... vezes**: estrutura que agrupa comandos que se repetem uma determinada quantidade de vezes. É equivalente à estrutura de repetição **para** ... **faça**, estudada no Módulo 1.
- **sempre**: repete as instruções dentro dela indefinidamente. As ações só param de executar quando o usuário clicar no botão de "Pare" do Scratch.

Obs

É comum referir-se a estruturas de repetição pelo nome de "laço de repetição".

As diferentes opções de estruturas de repetição nos permitem escolher como será feito o controle de quantas vezes os comandos serão executados dentro da estrutura. A seguir, vamos estudar um pouco melhor cada uma das opções disponíveis no Scratch.

Estrutura de Repetição repita...vezes

A instrução “repita ... vezes”, executa os comandos dentro dela em uma quantidade de vezes bem definida.

Há várias formas de se determinar a quantidade de repetições que a estrutura irá realizar:

- por digitação direta da quantidade de vezes que desejamos que as repetições aconteçam, como mostra a Figura 1.97;
- por obtenção de um valor em uma variável, como mostra a Figura 1.98;
- por meio do resultado de uma expressão aritmética. A expressão pode envolver variáveis e constantes combinadas, como se pode ver na Figura 1.99.

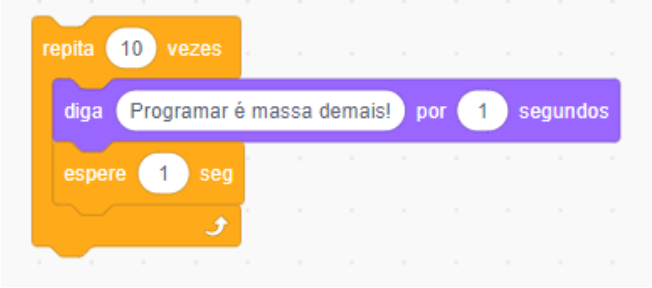

Figura 1.97: Estrutura de repetição com quantidade de repetições determinada por digitação direta.

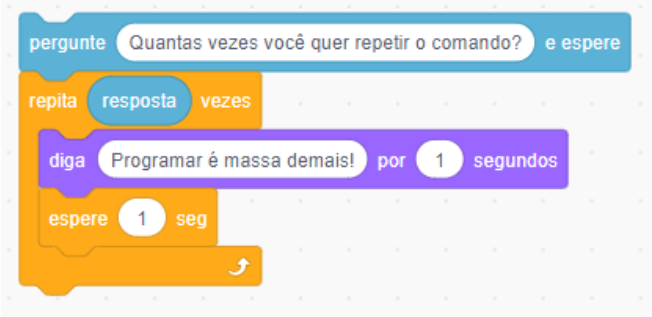

Figura 1.98: Estrutura de repetição com quantidade de repetições obtida de uma variável.

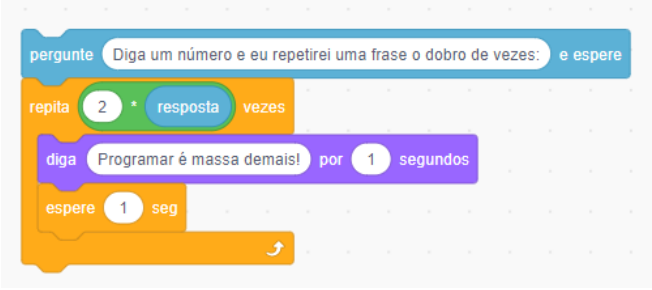

Figura 1.99: Estrutura de repetição com quantidade de repetições obtida de uma expressão.

Estrutura de Repetição repita até que ...

Em algumas ocasiões, não sabemos *a priori* quantas vezes um conjunto de instruções serão repetidas. Se quiséssemos, por exemplo, automatizar uma tarefa para que uma máquina mova caixas de um carregamento, e a cada carregamento tivermos uma quantidade diferente de caixas para mover, não temos como determinar de antemão quantas vezes as ações referentes a mover as caixas precisariam ser repetidas. Situações como essas são também comuns em programas que repetem a exibição de um menu até que o(a) usuário(a) selecione uma opção para sair do sistema.

A estrutura "repita até que ..." é uma instrução apropriada para esse tipo de situação. Nela, o controle de quantas repetições serão efetuadas é feito por meio de um **teste**, como aqueles que utilizamos em estruturas de decisão, que resultam em true ou false. Assim, a estrutura saberá quando é hora de encerrar a repetição de instruções.

Obs Ao encerrar as repetições necessárias, o fluxo de execução do programa segue para o próximo comando, encaixado logo abaixo da estrutura de repetição. O Exemplo 1.8 apresenta esse comportamento da estrutura.

■ **Exemplo 1.8** A Figura 1.100 apresenta um programa que recebe do(a) usuário(a) uma letra repetidas vezes, até que a letra 's' seja digitada. Por fim, o programa exibe uma mensagem de agradecimento por usar o sistema.

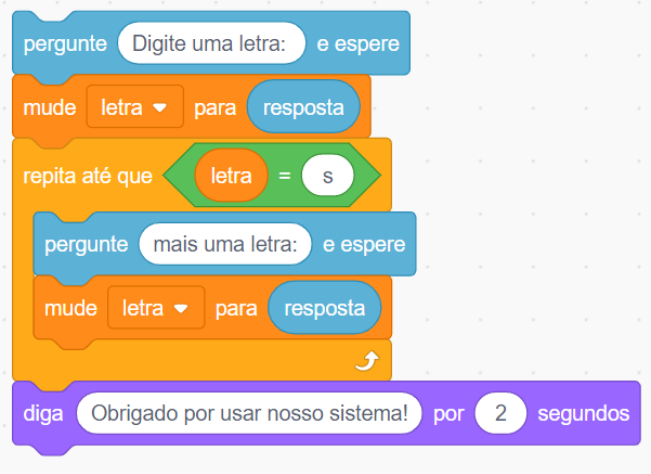

Figura 1.100: Um programa que obtém uma letra repetidas vezes, e pára quando a letra 's' é digitada.

Observe que o último comando do programa está fora da estrutura de repetição. Isso significa que, pela convenção de sequenciação, ele será executado logo após o encerramento da execução da estrutura de repetição, independente de o fluxo de execução ter entrado nela ou não. ■

Exercício 1.41 Explique o fluxo de execução do programa da Figura 1.100 para as seguintes entradas de dados:

(a) O(A) usuário(a) entrou com a letra 'a' apenas.
(b) O(A) usuário(a) entrou com as letras 'a', 'p', 's', nessa ordem.
(c) O(A) usuário(a) entrou com a letra 's' apenas.

Solução. As discussões pedidas nos itens (a) a (c) do Exercício 1.41 estão listadas abaixo:

(a) O programa não encerra a sua execução. O ator programado vai exibir a mensagem "mais uma letra:" e ficar esperando resposta do(a) usuário(a).

(b) Ao digitar a primeira letra, o(a) usuário verá uma mensagem pedindo mais uma letra; digitando a segunda letra – 'p' – o usuário verá novamente uma mensagem pedindo por mais uma letra. Ao digitar a letra 's', o usuário verá a mensagem de agradecimento por usar o sistema.

(c) Ao digitar a letra 's' logo no começo, o programa **não entra na estrutura de repetição**, e salta para o comando seguinte no programa (que é a exibição da mensagem de agradecimento).

■

Obs O teste realizado para controlar a quantidade de repetições a serem realizadas é chamado de **condição de parada**.

Exercício 1.42 O gato quer brincar com você o jogo do intervalo. Ele vai pensar em um valor numérico entre 0 e 99 e você vai tentar adivinhar qual é. A cada palpite que você der, ele vai te dar a dica se o número que você falou é maior ou menor que o número que ele está pensando. A brincadeira só pára quando você acertar o número! Faça um programa para implementar esse jogo.

Solução. O jogo consiste em fazer sucessivas comparações entre o número pensado e o número palpítado, até que eles sejam igauis. Não é possível prever quantos palpites serão necessários para que o jogo termine. Desta forma, a instrução que precisaremos utilizar na programação é a estrutura "repita até que...". A Figura 1.101 apresenta um programa para implementar esse jogo.

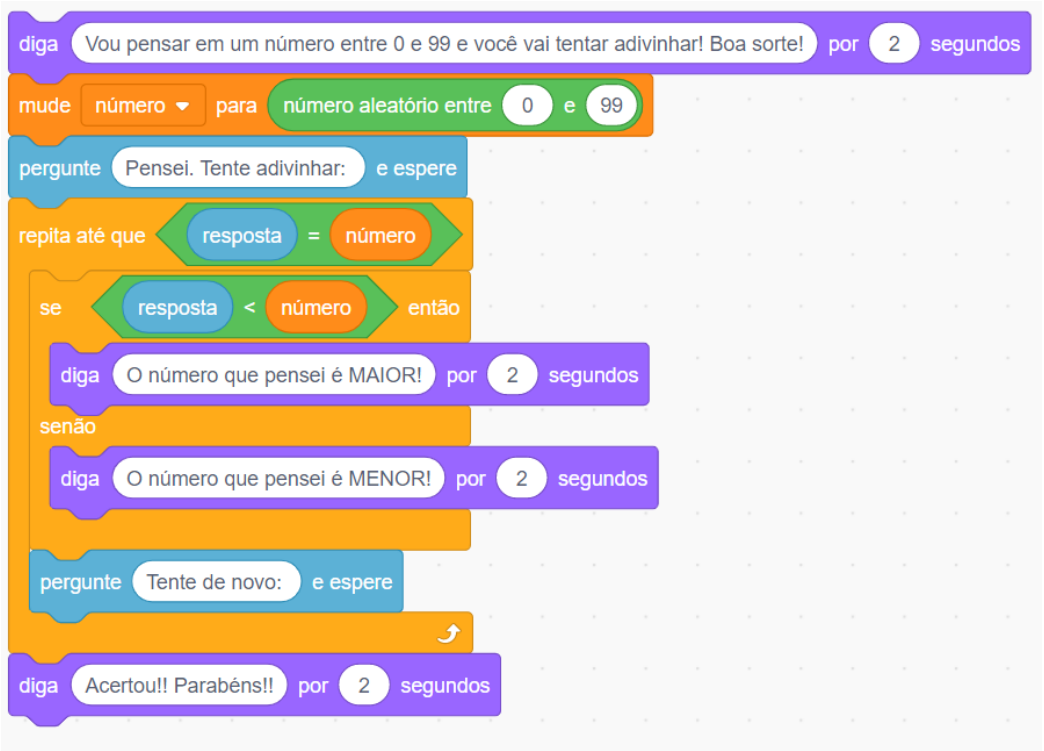

Figura 1.101: Um programa para solucionar o problema proposto no Exercício 1.42.

Observe, no programa, que a instrução que exibe uma mensagem confirmando que o(a) jogador(a) acertou vem logo após a estrutura de repetição. Note que, quando a pessoa acertar o número, o teste feito pela estrutura vai resultar em false e o fluxo de execução salta a estrutura de repetição, indo direto para o próximo comando.

■

Exercício 1.43 Olhando para o programa da Figura 1.101, como podemos saber se o código está correto?

Solução. Há várias maneiras de verificarmos se o programa está certinho. Uma delas é adicionar um comando para exibir o número pensado, como mostra a Figura 1.102. Porém, se não quiser alterar o código, é possível ativar a exibição da variável "número" no palco. Daí, depois de verificar se o programa está funcionando como deveria, é só desativar a visualização da variável, para não estragar a brincadeira. ■

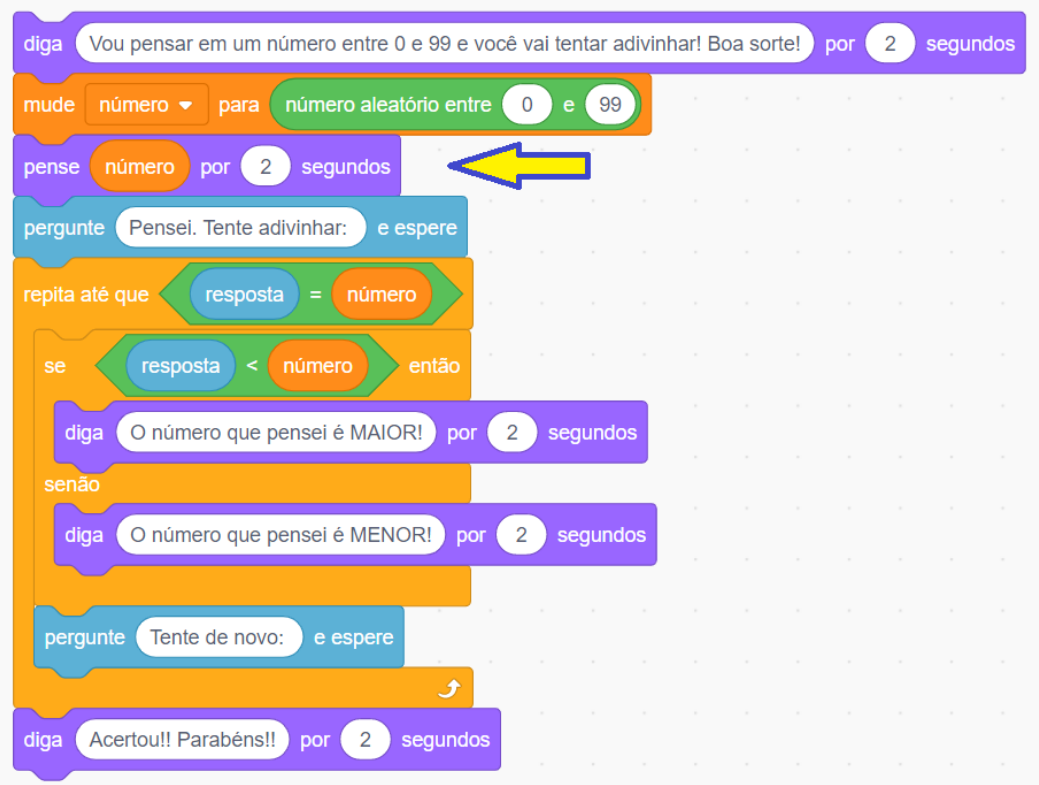

Figura 1.102: Uma alteração no programa para solucionar o problema proposto no Exercício 1.42.

Estrutura de Repetição sempre

Algumas aplicações necessitam ficar em execução por tempo indeterminado. Isso significa que os comandos que compõem o programa de tais aplicações se repetem indefinidamente, até que um processo externo, ou mesmo um(a) usuário(a) cause a parada do sistema. Um exemplo bastante popular de tais sistemas são os jogos. A programação de um jogo é composta por ações que se repetem indefinidamente, até que o jogador vença ou perca o jogo.

No Scracth, temos a estrutura de repetição "sempre". É dentro dela que podemos encaixar comandos que precisarão se repetir indefinidamente. O Exemplo 1.44 descreve um aplicação com essas características.

Exercício 1.44 Entrei no quarto e vi uma cena engraçada: vários gatinhos saindo bem apressados debaixo da cama, um gato por vez! Não dá nem pra dizer quantos gatos tinham, porque era gato saindo o tempo todo, sem parar. Olha só a situação na Figura 1.103! Será que você pode fazer um programa para ilustrar a situação que vi no quarto?

Figura 1.103: Situação inusitada a ser implementada para o Exercício 1.44.

Solução. A cena fofinha descrita no Exercício 1.44 pode ser programada com a ajuda de uma estrutura de repetição "sempre". Com ela, teremos gatinhos infinitos saindo debaixo da cama. Além disso, para complementar, vamos relembrar o que estudamos sobre aritmética modular, para usar, na verdade, um só gatinho, descrevendo uma trajetória que se reinicia debaixo da cama toda vez que ele sai de cena. Assim, criaremos a ilusão de se estar vendo inúmeros gatinhos, saindo um por vez. A determinação das posições x e y usadas no programa foi feita com a ajuda do posicionamento do gato nas posições inicial e final desejadas, arrastando o personagem com o mouse e tomando nota das coordenadas x e y para preencher os blocos de instrução. Por fim, foi adicionado o comando para mudar a fantasia do gato, para o movimento ficar animado. O programa completo pode ser visto na Figura 1.104.

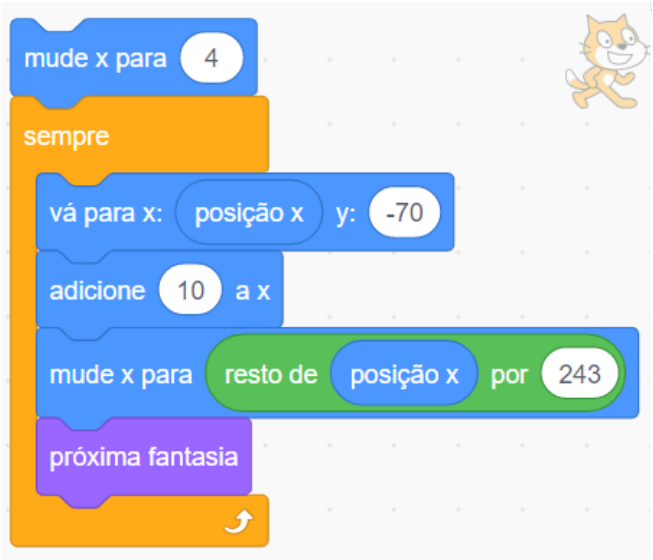

Figura 1.104: Um programa para implementar a situação inusitada descrita no Exercício 1.44.

■

Obs Observe que em programas com a instrução “sempre”, sua execução só termina quando clicamos no botão “Pare”, ao lado da bandeirinha verde.

Exercício 1.45 Vamos explorar nossas capacidades artísticas criando uma exibição animada do nosso nome. As ações que movimentarão as letras do nome deverão ser executadas até que o(a) usuário(a) pressione o botão “Pare”.

Solução. Para programar o que foi pedido no Exercício 1.45, vamos criar códigos separados para cada uma das letras. Em cada código, vamos escolher alguma propriedade da letra para ser alterada: tamanho, posição x ou y e rotação. Além disso, para todas as letras, vamos incluir a possibilidade de alterar as cores de cada uma das letras. A Figura 1.105 mostra o palco e a programação específica para a letra ’N’.

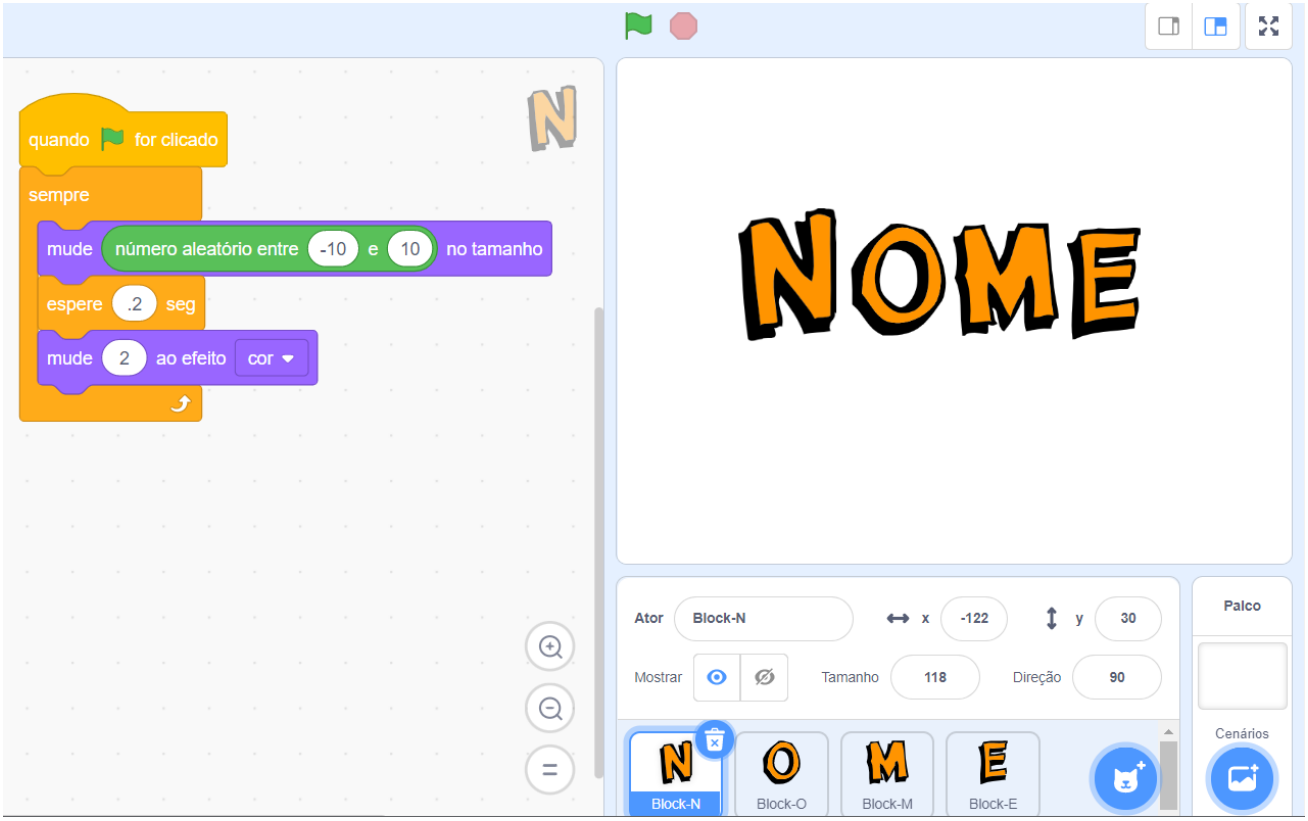

Figura 1.105: Um programa para implementar a exibição de um nome animado.

Observe que cada letra é um ator. Assim, cada letra terá seu próprio código, possibilitando diferentes alterações a cada uma delas. Como temos mais de um ator programado na aplicação, é necessário incluir a instrução “Quando a bandeirinha verde for clicada” em todos os códigos, que podem ser vistos na Figura 1.106.

Figura 1.106: Programas para implementar a animação de cada letra de um nome.

Obs Ao fazer esse programa, experimente criar outros efeitos, melhorando o resultado obtido pelo programa aqui apresentado.

■

Encadeamento de estruturas de repetição

Vamos considerar a seguinte situação, descrita no Exemplo 1.9:

■ **Exemplo 1.9** Um Elfo quer aproveitar um cantinho no quintal dele para plantar umas flores. Ele tem 20 sementes e deseja colocá-las enfileiradas no quintal. Para implementar esta tarefa, foi feito um programa. O código e o resultado de sua execução podem ser vistos na Figura 1.107.

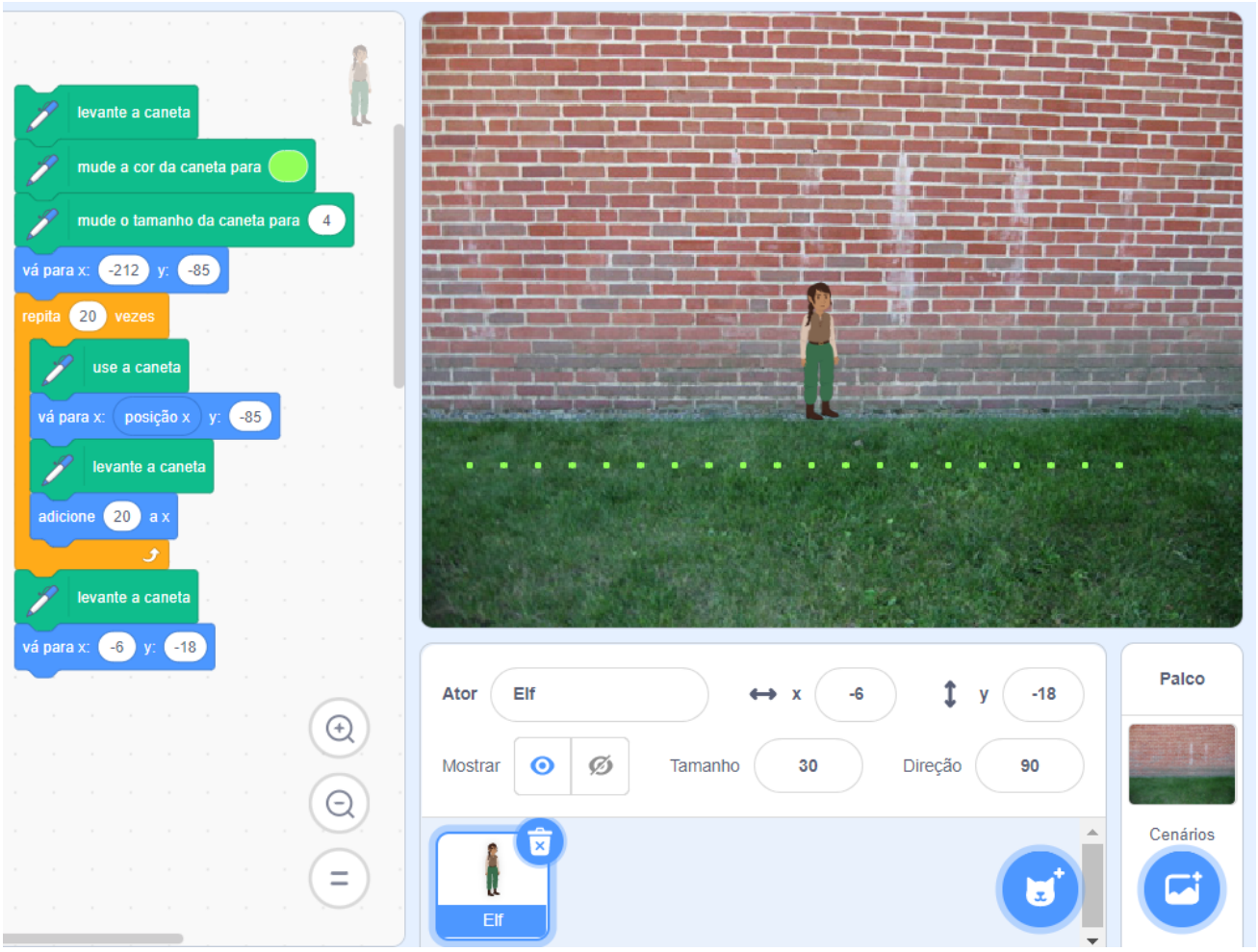

Figura 1.107: Um programa para implementar a distribuição de sementes pelo Elfo, com o resultado de sua execução no palco.

■

Observando o Exemplo 1.9, percebemos que, dentro da estrutura de repetição, a coordenada y se mantém constante. O laço varia apenas o valor da coordenada x. O resultado é a distribuição das sementes em uma fileira, como se vê na Figura 1.107. Após o laço de repetição, o Elfo é levado a uma posição próxima ao muro, para facilitar a visualização das 20 sementes.

Exercício 1.46 Imagine, agora, que o Elfo tem 100 sementes, e deseja distribuí-las em 5 fileiras de 20 sementes. Que alterações seriam necessárias para o programa da Figura 1.107?

Solução. A aplicação agora precisará dispor as sementes não só na horizontal, como também na vertical, para criar as 5 fileiras de 20 sementes.

O programa anterior utilizava um laço para repetir o posicionamento de sementes 20 vezes ao longo do eixo horizontal do palco. Agora, vamos precisar de mais um laço, desta vez para repetir o posicionamento de sementes 5 vezes, ao longo do eixo vertical do palco.

Situações como esta requerem que utilizemos dois laços de repetição, encadeados. A Figura 1.108 apresenta um programa para resolver o problema proposto.

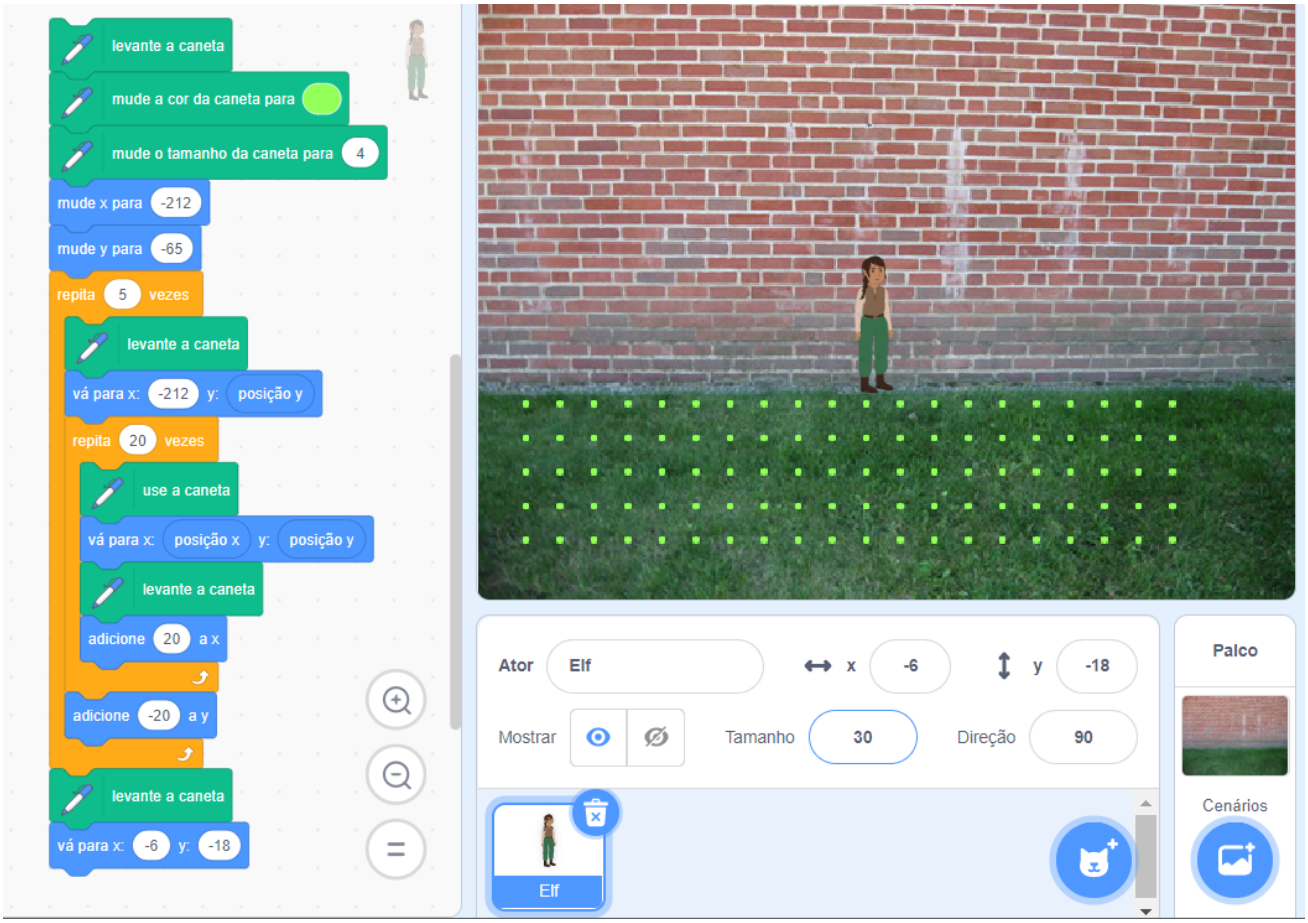

Figura 1.108: Um programa atualizado para implementar a distribuição das 100 sementes em 5 fileiras, com o resultado de sua execução no palco.

Vamos analisar o novo código. Alguns de seus detalhes são essenciais para a produção do resultado correto:

- Observe o laço mais interno. Ele é quase o mesmo do código anterior, ou seja, é o trecho de código responsável por gerar uma fileira de 20 sementes. A única mudança é que y agora não é mais constante.
- Queremos criar 5 filas dessas fileiras de 20 sementes. Para isso, encaixamos o trecho de código responsável por gerar uma fileira dentro de um novo laço de repetição, para repetir o processo 5 vezes.
- O valor da coordenada y também foi inicializado neste novo programa.
- Perceba que antes de entrar no laço mais interno, a coordenada x é resetada para a posição onde o Elfo inicia o percurso em cada fileira.
- A atualização da coordenada y é feita logo após o Elfo soltar a última semente de uma fileira, preparando o posicionamento dele para o próximo local.

A Figura 1.109 ilustra as observações listadas acima.

■

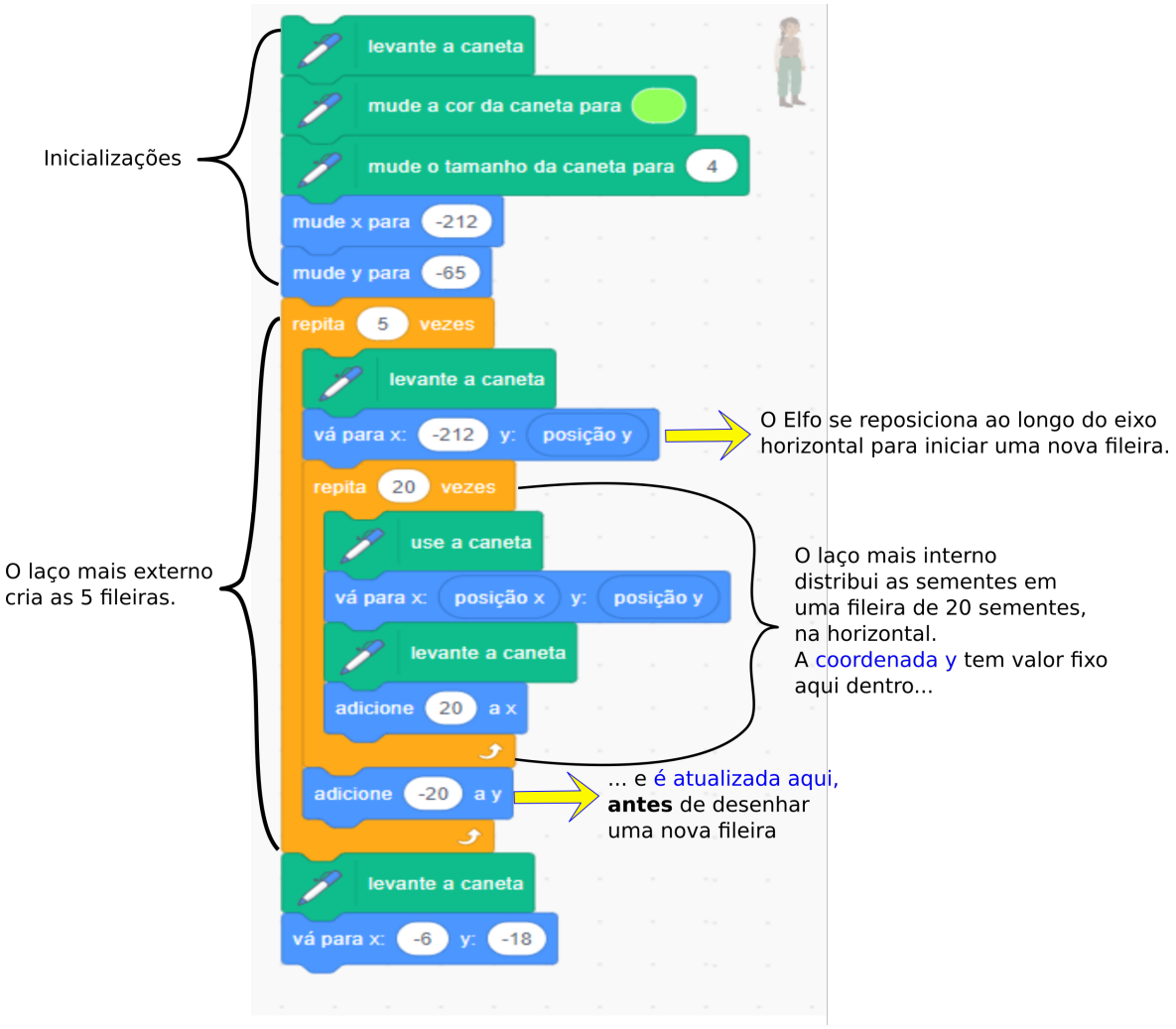

Figura 1.109: Detalhamento das instruções do programa feito para o Exercício 1.46.

Obs Quando encadeamos laços de repetição, o laço mais interno é totalmente executado antes de seguir para o próximo passo do laço mais externo.

■ **Exemplo 1.10** A Figura 1.110 mostra um programa simples que faz o ator dizer pares de números, variando seus valores conforme os comandos de atualização dentro dos laços de repetição. Para facilitar a análise, as variáveis receberam nomes de "varLaçoInterno" e "varLaçoExterno".
O laço mais externo realiza duas repetições. Assim, a variável varLaçoExterno assumirá os valores 0 e 1. Ela é atualizada dentro do laço externo e fora do laço interno.
O laço interno atualiza somente a variável varLaçoInterno, que assumirá valores de 0 a 4. Dentro do laço interno, o comando "diga" fará o ator falar os pares de números, que correspondem a valores armazenados nas duas variáveis, na ordem var. Para o programa da Figura 1.110, especificamente, o ator vai falar, nesta ordem, os pares na ordem apresentada na Tabela 1.1:
■

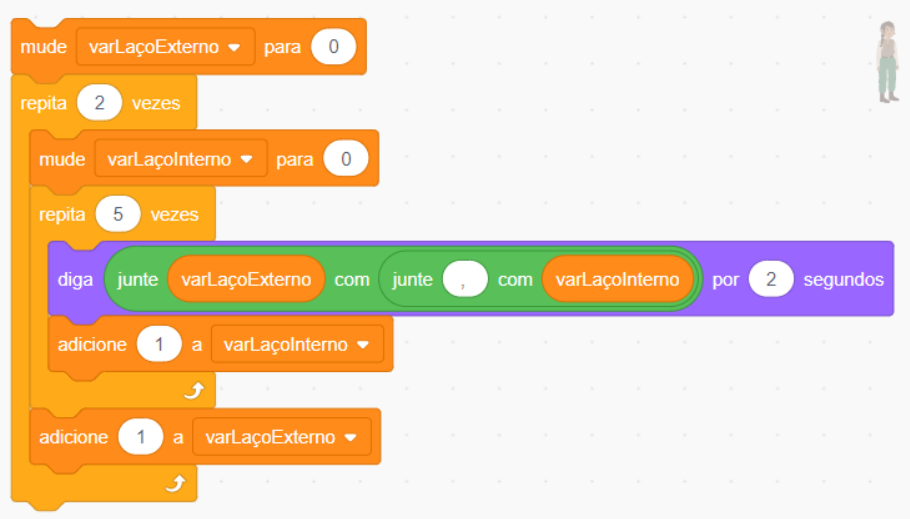

Figura 1.110: Um programa para exemplificar o comportamento de laços encadeados.

varLaçoExterno	varLaçoInterno
0	0
0	1
0	2
0	3
0	4
1	0
1	1
1	2
1	3
1	4

Tabela 1.1: Lista de pares gerados com os laços de repetição encadeados no programa da Figura 1.110.

1.7 – Funções

Na categoria "Operadores" do Scratch, podemos encontrar alguns blocos que realmente simplificam nosso trabalho como programadores. Tais blocos podem ser vistos na Figura 1.68.

Com esses blocos, podemos apenas fornecer um valor de entrada e obter um resultado, sem nos preocupar com as instruções que fazem aquelas funções produzirem os resultados de que precisamos.

A boa notícia é que nós também podemos criar nossas próprias funções e utilizá-las em nossos programas. Isso facilita a organização do programa, principalmente quando se trata da implementação de uma aplicação que realiza múltiplas tarefas.

Definições

Nesta seção, vamos aprender como podemos criar nossas próprias funções. A primeira coisa a fazer é entender o que é uma função.

Definição 1.7.1 Um **Função** é um conjunto de instruções que tenham:

- Um começo, meio e fim bem definidos
- Um propósito muito bem delimitado (ou seja, não mistura tarefas, dedicando-se a uma só).

Com base na definição 1.7.1 e nas funções que já encontramos disponíveis no Scratch, compreendemos que uma função funciona como uma espécie de "caixa preta", na qual inserimos algum valor de entrada e recebemos um resultado, sem nos importar com a maneira como o processamento dos dados de entrada foi feito. Desconhecemos as instruções que estão dentro dela.

Com isso, uma função garante uma boa legibilidade do seu código, deixando-o mais enxuto e mais organizado. Desta forma, atendemos a dois princípios importantes em programação: **abstração** e **decomposição**, definidos a seguir:

Definição 1.7.2 **Abstração** é a capacidade de suprimir detalhes. É a habilidade de enxugar o código, mantendo sua funcionalidade, tratando alguns conjuntos de instruções como uma "caixa preta": não se tem interesse no que tem dentro dela – apenas queremos utilizá-la. Exemplos disso são as funções já disponibilizadas no Scratch, vistas na Figura 1.68.

Definição 1.7.3 **Decomposição** é o processo de decompor o código em módulos, que têm um sentido próprio. O(A) programador(a), ao lê-lo, sabe o que aquele conjunto agrupado de instruções faz. É uma tarefa que tem início, meio e fim muito bem definido, sem misturar diferentes propósitos.

Além das definições apresentadas, é necessário conhecermos alguns termos e conceitos:

- Uma função pode receber valores de entrada, que chamaremos de **parâmetros**.
- Uma função pode também produzir uma saída, que chamaremos de **valor de retorno da função**.
- Uma função é escrita como um conjunto separado de instruções específicas para a realização de uma tarefa. Com isto, podemos ter, em nossos programas, vários conjuntos de blocos de instruções separados, um para cada função!
- Além das funções, nosso programa precisará de um conjunto de instruções que será o nosso **programa principal**. Normalmente, é a partir do programa principal que as outras funções são chamadas.
  - Chamar uma função é uma instrução especial que ordena que uma determinada função seja executada.
  - A chamada a uma função desvia o fluxo de execução para que as instruções daquela função sejam executadas
  - Quando uma função termina sua execução, o fluxo de execução do programa volta para o ponto onde a função foi chamada.

Obs Se tudo isso parece um pouco complicado, não se preocupe. Vamos ganhar familiaridade com esses termos e conceitos por meio de exercícios práticos!

Mecanismos de funcionamento de funções em um programa.

Para melhor compreender os conceitos apresentados, vamos acompanhar o Exemplo 1.11.

■ **Exemplo 1.11** Observe o programa da Figura 1.111. Ele recebe um nome e o exibe de trás pra frente.

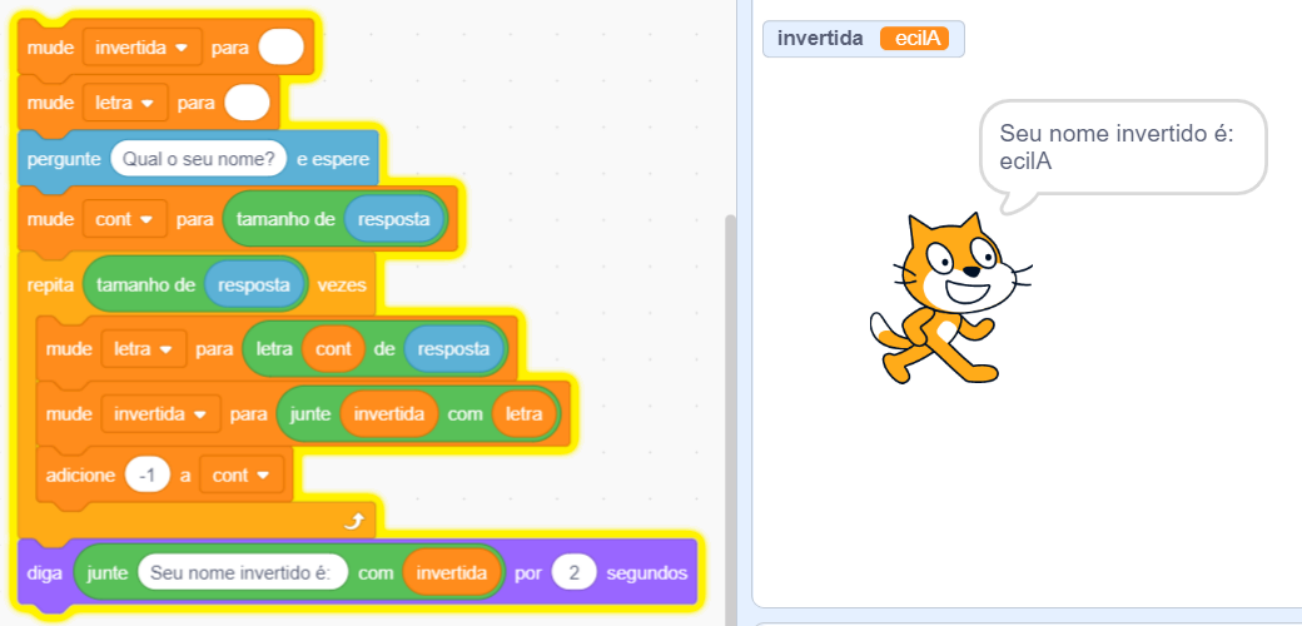

Figura 1.111: Um programa para inverter um nome fornecido via teclado.

O programa contém todos os seus comandos escritos em sequência, em um único conjunto de blocos de instruções encaixados. Podemos perceber que algumas instruções são específicas da tarefa de inverter o nome digitado, e outras instruções são acessórias, para completar a aplicação. Assim, podemos "isolar" as instruções específicas para a tarefa de inversão do nome em um bloco separado, e deixar o programa geral um pouco mais enxuto.

Esse bloco separado, focado em uma única ação específica, com um começo, meio e fim muito bem definidos, será nossa **função**. A Figura 1.112 mostra em destaque o conjunto de instruções para a inversão do nome separadas.

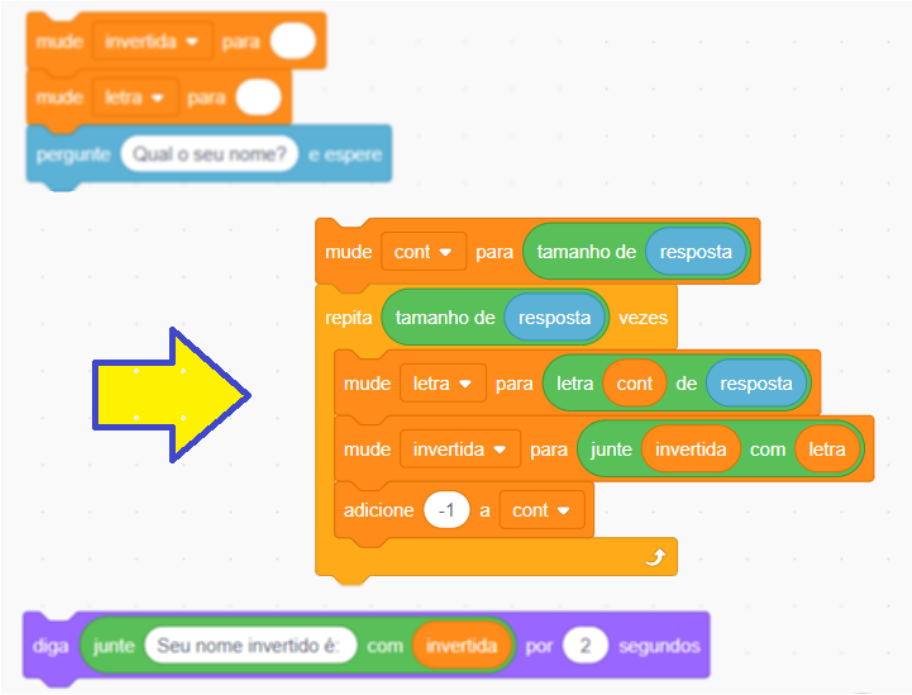

Figura 1.112: Instruções específicas para a tarefa de inverter um nome digitado, destacadas do restante do programa.

Agora, vamos ver como separar esse conjunto de instruções dentro de um módulo, ou de uma função, no Scratch. Os passos necessários para organizar o código em que estamos trabalhando estão descritos a seguir:

- **Passo 1: criar um novo bloco**.
  - Neste passo, vamos clicar sobre o botão "Criar um bloco", na categoria "Meus blocos". Esse novo bloco criado será a nossa "caixa preta", a ser utilizada no programa principal.
  - Como vamos querer receber um parâmetro de entrada (que será um nome a ser invertido), vamos escolher a opção de criar um bloco do tipo "Adicionar uma entrada número ou texto", que é a opção em destaque na Figura 1.113.

Figura 1.113: Criação de bloco para agrupar as instruções referentes à tarefa de inversão de texto, discutida no Exemplo 1.11.

  - Para finalizar a criação do bloco, vamos dar um nome a ele. Além disso, vamos dar um nome significativo para o parâmetro de entrada. A Figura 1.114 mostra os nomes escolhidos para este exemplo.

Figura 1.114: Preenchimento do nome do bloco e do nome do parâmetro da função que vai agrupar as instruções referentes à tarefa de inversão de texto, discutida no Exemplo 1.11.

- **Passo 2: inserir instruções no novo bloco criado.**
  - Após a criação do novo bloco, surgirá um bloco na área de trabalho, e outro na categoria "Meus Blocos". O bloco da área de trabalho receberá as instruções referentes à tarefa específica. O bloco que está na categoria "Meus Blocos" será utilizado no seu programa principal, deixando-o mais sucinto e enxuto. Os novos blocos criados podem ser vistos em destaque na Figura 1.115.

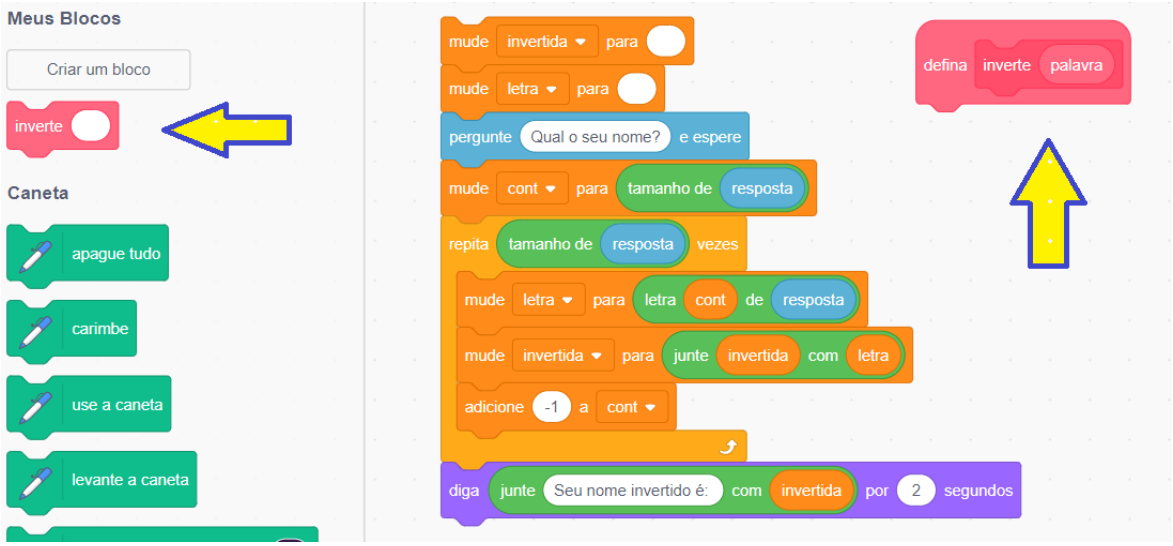

Figura 1.115: Novos blocos que surgirão na área de trabalho e na categoria "Meus Blocos", após a criação de um novo bloco.

- **Passo 3: mover os comandos apropriados para o bloco "Defina", na área de trabalho.**
  - Neste passo, vamos mover os blocos que identificamos como específicos para a tarefa pretendida, para que se encaixem no bloco "Defina". A Figura 1.116 mostra a movimentação a ser feita nos blocos de instruções.
- **Passo 4: no programa principal, substituir o conjunto de instruções retiradas por um bloco correspondente à função criada.**
  - O bloco em questão pode ser encontrado na categoria "Meus Blocos". A Figura 1.117 mostra o posicionamento dos blocos após esses ajustes.

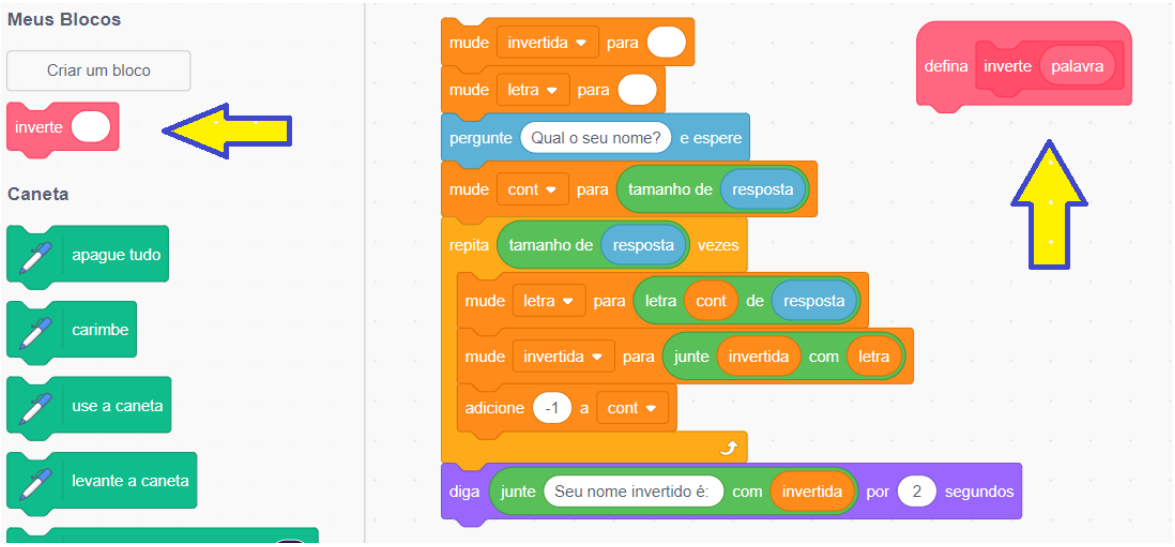

Figura 1.116: Novos blocos que surgirão na área de trabalho e na categoria “Meus Blocos”, após a criação de um novo bloco.

– O bloco que foi inserido no programa principal corresponde a uma **chamada à função** inverte.

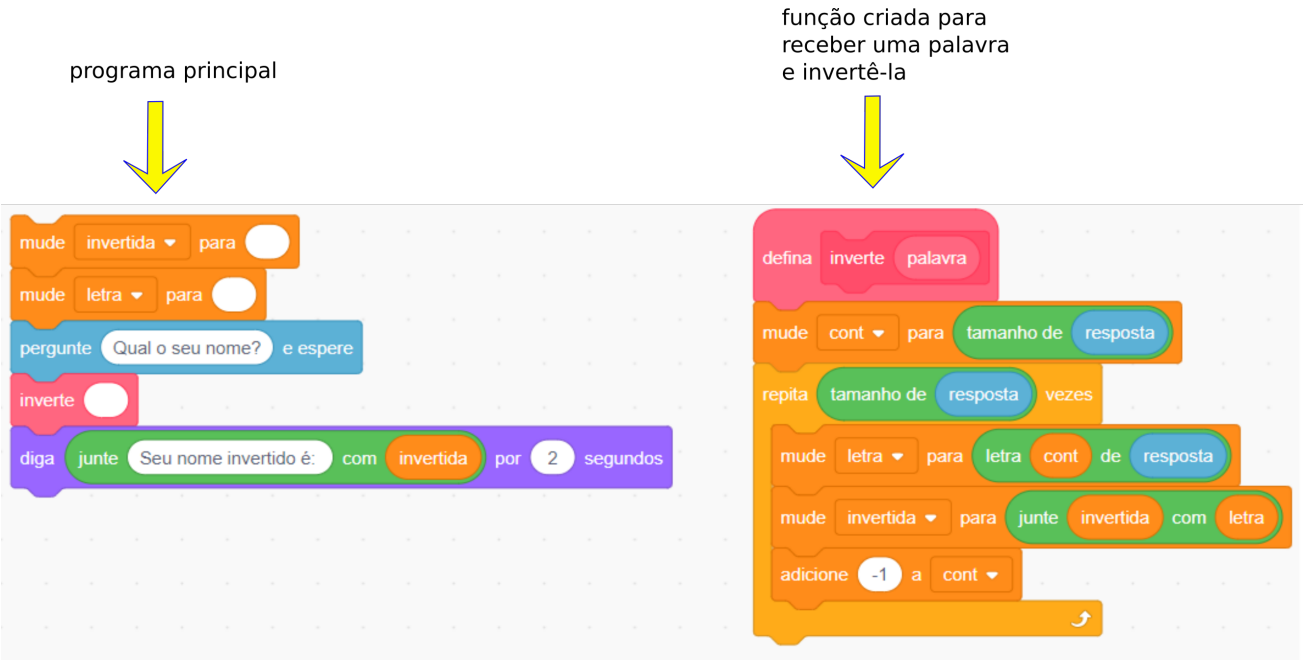

Figura 1.117: Reorganização do código, com o programa principal e a função inverte separados.

- **Passo 5: inserir o dado que será passado na chamada à função.**
  - Observe que no programa principal, o bloco referente à chamada da função inverte está com um espaço em branco. Ali, deve ser colocado o dado que será passado para a função processar.
  - Como neste programa, queremos inverter um nome digitado, passaremos como dado para a função a variável “resposta”.
- **Passo 6: ajustar as variáveis no código da função.**
  - O código da função não estará mais lidando com a variável resposta. Ela vai lidar com o parâmetro de entrada.

– Neste exemplo, o parâmetro de entrada tem o nome de “palavra”. Assim, “palavra” é uma variável interna à função, que recebe o mesmo valor do parâmetro passado na chamada a ela. A Figura 1.118 mostra o código da função inverte com os devidos ajustes nas variáveis a serem processadas.

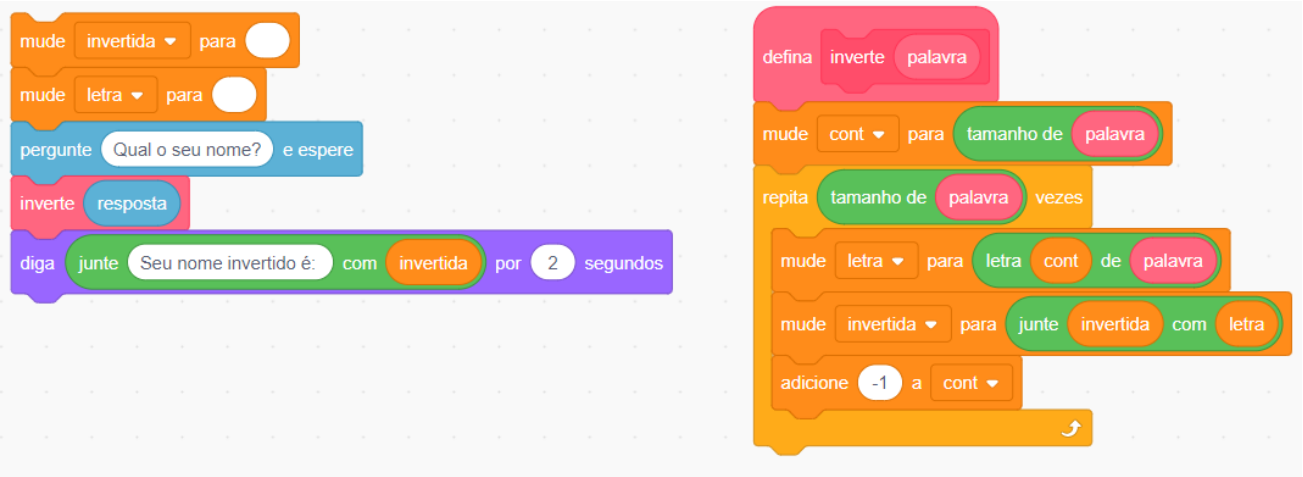

Figura 1.118: Versão finalizada do código, com todos os devidos ajustes.

■

Algumas perguntas podem surgir: por que criar essa variável interna em uma função? Bem, uma das grandes vantagens de criar funções é o fato de o nosso programa poder chamá-la mais de uma vez. Isso é vantajoso, pois podemos utilizar e reutilizar uma função, mesmo que ela tenha sido escrita apenas uma vez.

Porém, vamos pensar um pouco, por exemplo, na função “raiz quadrada”. Em um momento, um programa pode chamar essa função passando como entrada um valor 25, e em um outro momento passando um valor diferente, digamos, 9. Como cada vez que um programa chama uma função, os valores de entrada podem ser diferentes, a função precisa ter dentro dela uma **variável** para assumir esses diferentes valores a cada chamada. Isso confere flexibilidade e reusabilidade às funções.

A Figura 1.119 ilustra o mecanismo geral de funcionamento de uma função dentro de um programa.

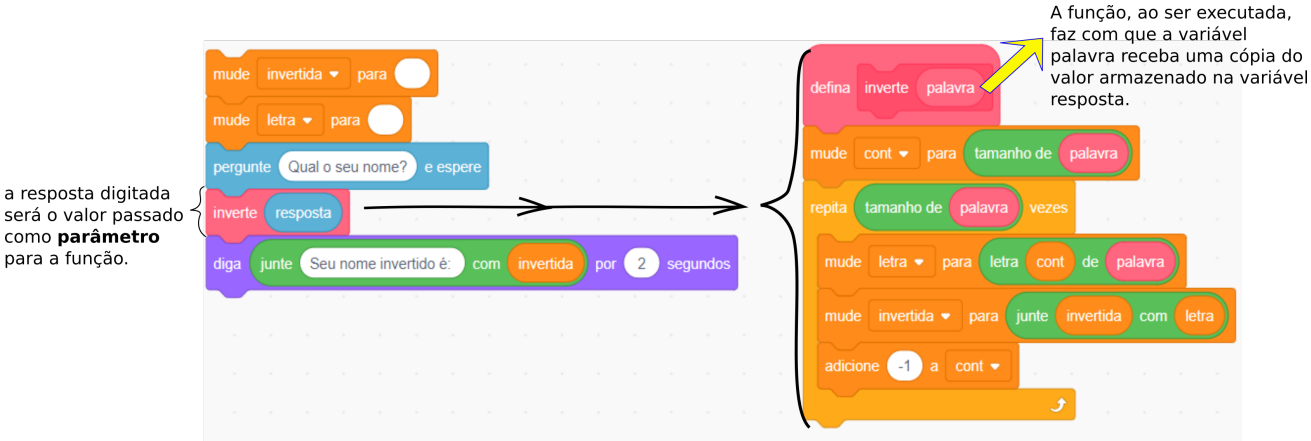

Figura 1.119: Código completo do Exemplo 1.11, mostrando o mecanismo de chamada a uma função.

Vamos estudar mais uma opção de função que podemos criar no Scratch. Desta vez, ao invés de passar um número ou texto como parâmetro, vamos passar o resultado de uma expressão booleana. O Exemplo 1.12 descreve uma situação em que esse tipo de função pode ser utilizado.

■ **Exemplo 1.12** Observe o programa da Figura 1.120. Trata-se de um programa simples, que recebe do(a) usuário(a) um valor numérico e diz se o número informado é par ou ímpar.

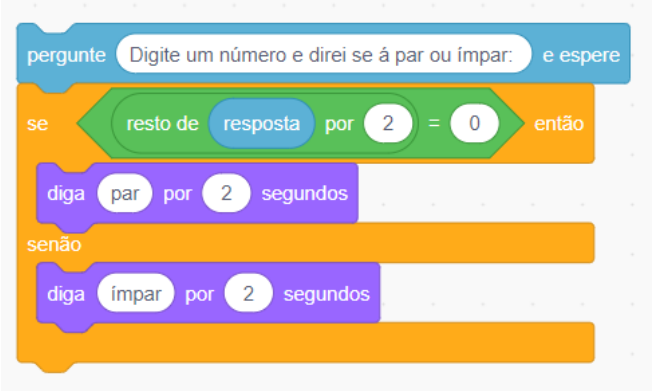

Figura 1.120: Um programa para determinar se um dado número é par ou ímpar.

Analisando os blocos de instruções, podemos identificar aqueles que são específicos para a tarefa de identificar a paridade do número fornecido. Estes serão os blocos que iremos mover para dentro de uma função. Será necessário, então, efetuarmos o seguinte passo-a-passo:

- **Passo 1: criar o bloco para a função.**
  - Escolheremos, desta vez, o bloco do tipo "Adicionar uma entrada – booleano".
  - Feito isso, precisar dar um nome para a função e para a expressão booleana de entrada.
- **Passo 2: mover as instruções para o bloco "Defina", visível na área de trabalho.**
  - Devemos mover somente as instruções referentes ao resultado do teste de paridade feito no programa principal.
  - Você pode verificar o resultado final deste passo-a-passo na Figura 1.121.
- **Passo 3: Inserir, no programa principal, o bloco referente à chamada da função.**
- **Passo 4: Ajustar a passagem do parâmetro para a função, no bloco que realiza a chamada à função.**
- **Passo 5: Ajustar a variável usada dentro da definição da função.**

Após efetuar os passos acima, seu programa modificado deverá estar como é mostrado na Figura 1.121.

■

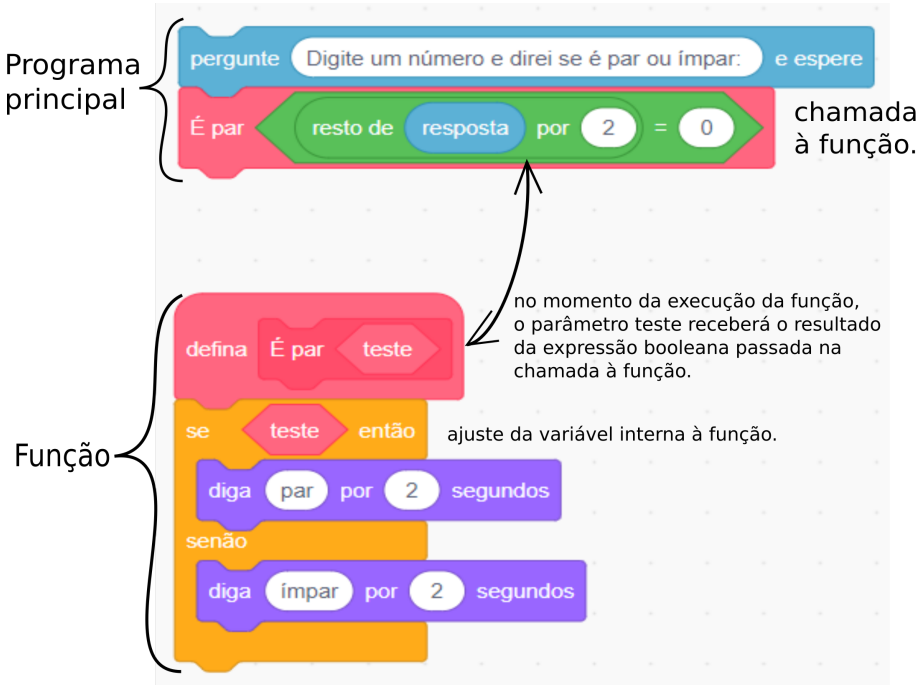

Figura 1.121: Um programa que contém uma função para determinar se um dado número é par ou ímpar.

Os dois tipos de funções vistos até aqui recebem um parâmetro para funcionar. No Exemplo 1.11, podemos ver uma função que recebe uma palavra como parâmetro de entrada; já no Exemplo 1.12, a função criada recebeu como parâmetro o resultado de uma expressão booleana. Vamos completar nosso estudo sobre funções, conhecendo mais um tipo de função. São funções que agrupam instruções, mas não necessitam de parâmetros de entrada para funcionar. No Scratch, este tipo de função está disponível na terceira opção que vemos na janela "Criar um Bloco". A Figura 1.122 mostra essa opção em destaque.

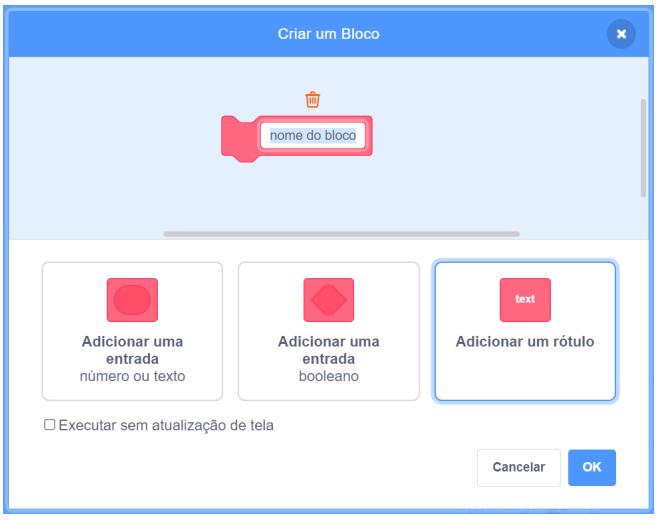

Figura 1.122: Criação de um bloco do tipo rótulo.

O processo para criação e utilização é bem parecido com os demais processos já vistos. A única diferença, é que não teremos parâmetros para ajustar dentro da função. A Figura 1.123 mostra a adição de um bloco rotulado como "muda cenário", e realiza a tarefa de alternar entre os 4 cenários inseridos no programa, com um intervalo de 1 segundo entre cada troca de cenário.

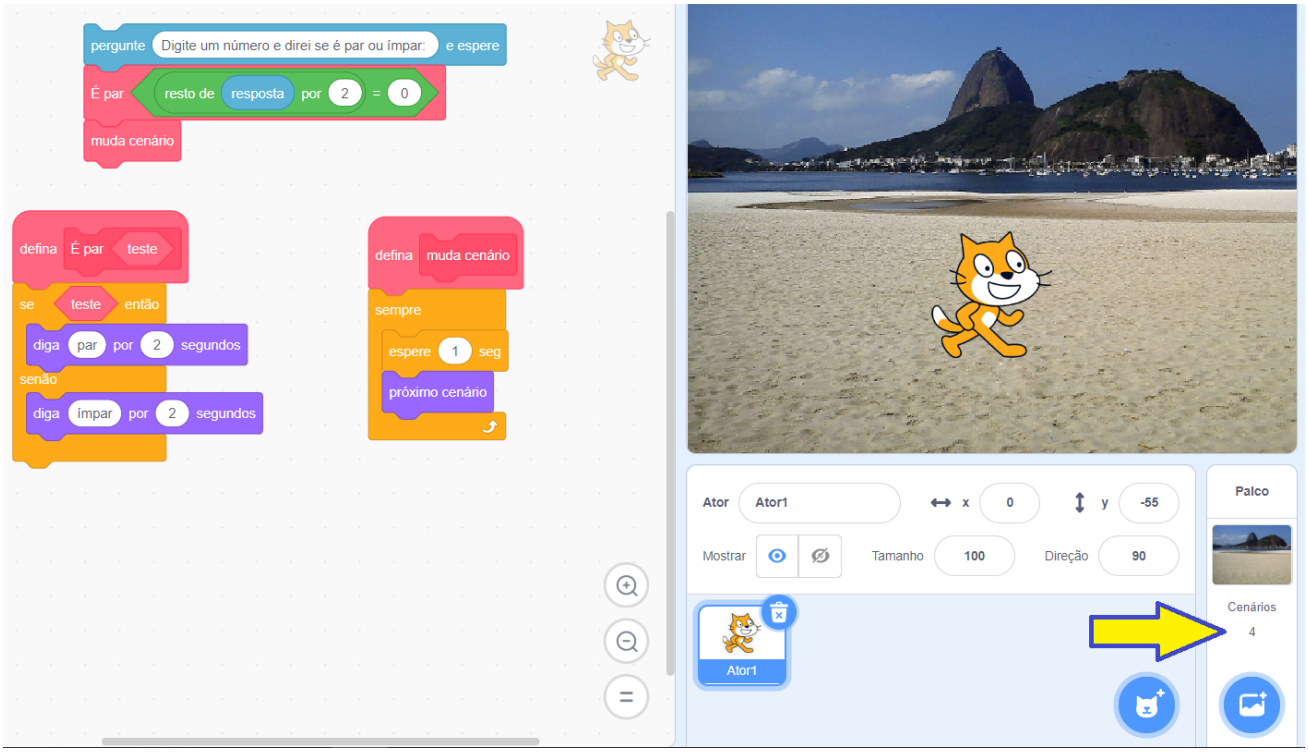

Figura 1.123: Programa com duas funções. A função "muda cenário" é do tipo que não recebe parâmetro de entrada.

Exercício 1.47 Crie duas funções: uma para exibir o dobro de um valor fornecido pelo(a) usuário(a) e outra para exibir o somatório de 1 até o número informado. Crie um programa principal para chamar uma dessas funções, a depender de uma opção escolhida pelo(a) usuário(a). Por exemplo, se a opção escolhida for "1", o programa irá executar somente a função que exibe o dobro do valor informado; se a opção escolhida for "2", o programa exibirá somente o resultado do somatório de 1 até o número informado.

Solução. Um programa completo, com as duas funções pedidas e o programa principal, pode ser visto na Figura 1.124.

■

Exercício 1.48 Faça um programa que contenha e use uma função que exiba os n primeiros termos de uma série de Fibonacci. **OBS:** uma série de Fibonacci é uma sequência de números tal que:

- O primeiro termo é 1;
- O segundo termo é 1;
- O terceiro termo e os termos seguintes a ele são o resultado da soma dos dois termos anteriores a ele.

Exemplo: para n = 10, os 10 primeiros termos são 1, 1, 2, 3, 5, 8, 13, 21, 34, 55. Assim, para ficar interessante, seu programa deverá pedir que o valor de n seja superior a 2.

Solução. Para solucionar o problema, vamos incluir, no programa principal, uma verificação para saber se o número informado pelo(a) usuário(a) foi maior que 2. A função que gera a sequência só será chamada se essa condição for satisfeita. A função tem como estratégia a criação de duas variáveis, para poder recuperar, a cada novo termo, os valores dos dois termos anteriores. Descoberto o novo termo, essas variáveis são atualizadas para possibilitar a correta obtenção de um novo termo, caso seja necessário. Observe que o laço de repetição é executado n - 2 vezes

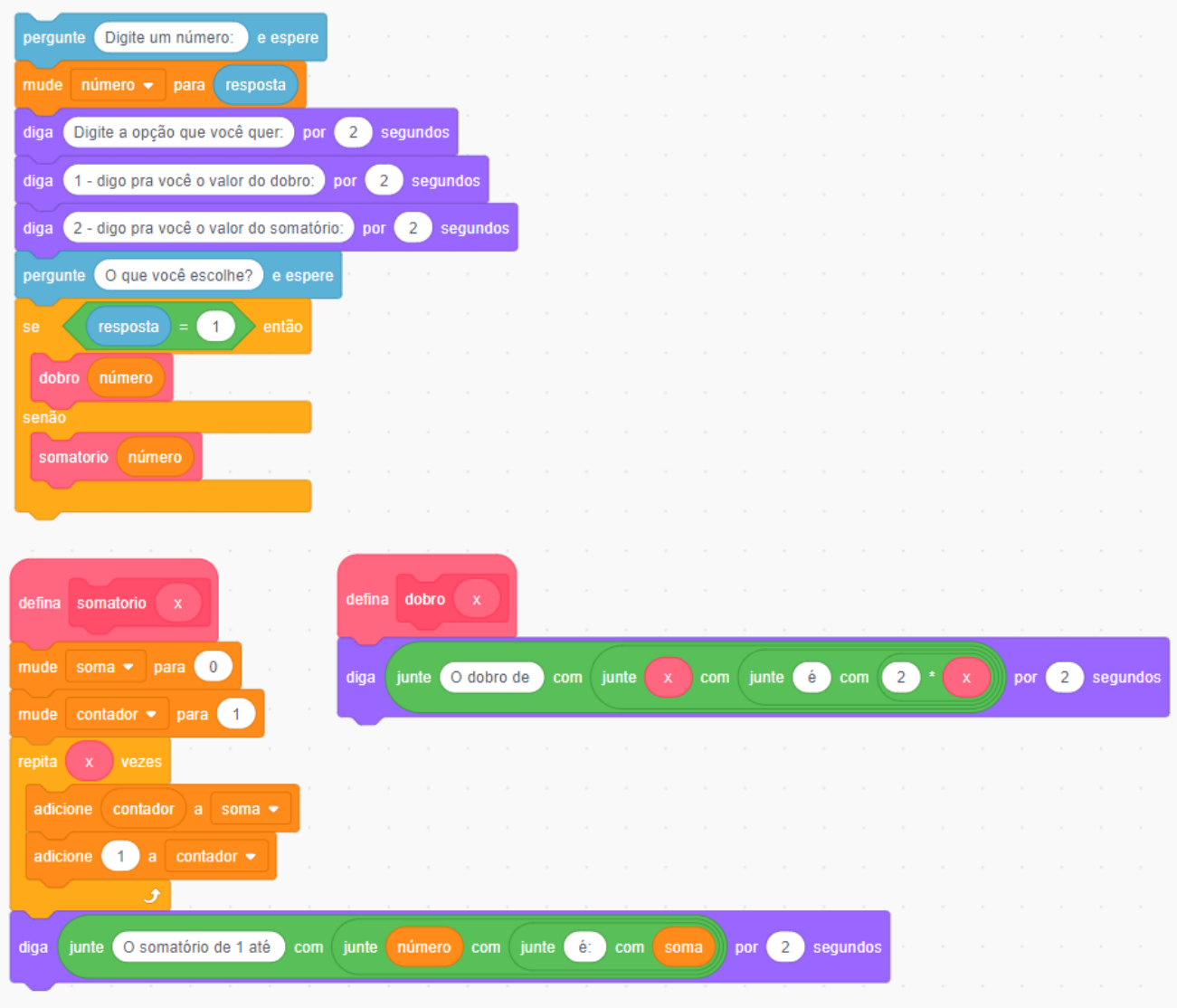

Figura 1.124: Um programa para solucionar o problema proposto no Exercício 1.47.

apenas, pois os dois primeiros termos da série já são conhecidos. A Figura 1.125 mostra um programa que soluciona o problema proposto no Exercício 1.48.

■

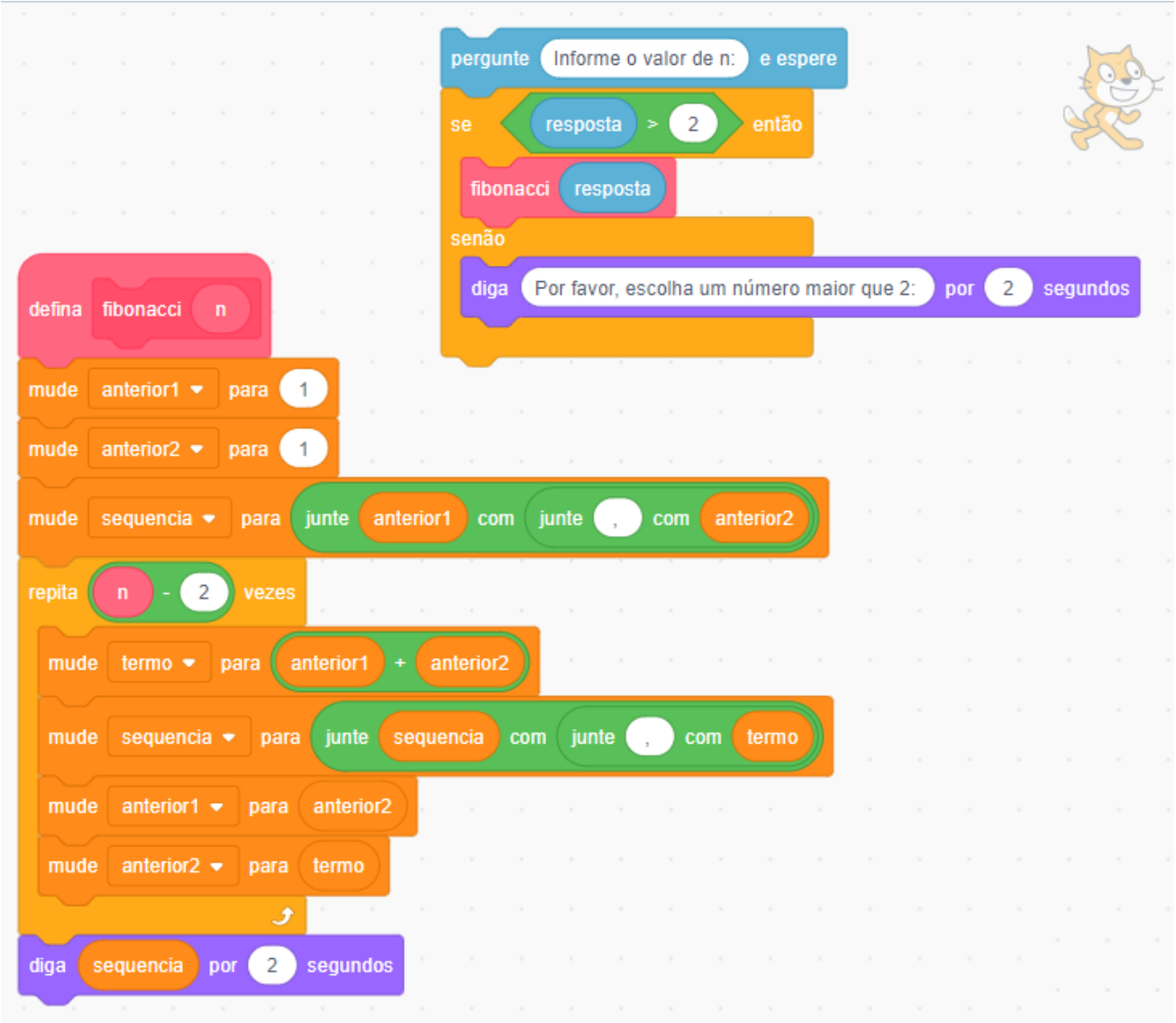

Figura 1.125: Um programa para solucionar o problema proposto no Exercício 1.48.

1.8 – Interatividade

Os sistemas computacionais que encontramos à nossa disposição no dia-a-dia são em sua maioria interativos: eles respondem a comandos que damos a eles em tempo real (ou seja, durante o tempo em que o programa está em execução).

Agora, vamos comparar esses sistemas com as aplicações que desenvolvemos até o momento: os programas que fizemos até aqui não exploraram um nível de interatividade muito elevado: a maioria da interação ficou limitada a uma entrada de dados via teclado pelo(a) usuário(a) em um momento determinado pela aplicação. Este tipo de interação é muito limitado.

Nesta seção, vamos aprender a aprimorar as habilidades interativas de nossas aplicações computacionais! Então, para continuarmos, é importante compreendermos bem o que é **Interatividade**.

Definição 1.8.1 Chamamos de **Interatividade** a capacidade de um sistema de reagir a estímulos externos no momento em que eles ocorrem.

Para ser interativo, um programa precisa ser capaz de responder, durante a sua execução, a comandos vindos do(a) usuário(a) por meio de diferentes dispositivos de entrada. Cada comando desse tipo é chamado de um **evento**.

Definição 1.8.2 Chamamos de **Evento** cada tipo de estímulo que pode atingir um sistema, vindo de uma entidade externa.

No caso de sistemas computacionais, os eventos são todas as ações que um(a) usuário(a) pode realizar enquanto o programa está em execução. Os eventos mais comuns que iremos tratar em nossos programas são:

- **de mouse**: clique, arrasto, duplo-clique;
- **de teclado**: tecla pressionada, pressionamento de duas teclas ao mesmo tempo, pressionamento de teclas especiais (setas de direção, barra de espaço, etc).

Vale ressaltar que há outros eventos além dos listados acima. Aplicações mais sofisticadas podem lidar com eventos de *joystick* e outros dispositivos de entrada.

Obs Lembre-se de que os jogos digitais são exemplos de aplicações computacionais interativas. Usando o Scratch, podemos criar jogos muito legais!

O segredo de programas interativos é a implementação do tratamento dos eventos que podem ocorrer durante sua execução. Assim, a linguagem de programação precisa disponibilizar recursos para o tratamento de eventos, que iremos discutir a seguir.

1.8.1 – Tratamento de Eventos

A maioria das linguagens de programação oferecem ao(à) projetista funcionalidades que permitem programar como responder a eventos. Nesta seção, vamos conhecer os blocos de instruções que nos permitirão tratar eventos, para podermos criar aplicações interativas.

No Scratch, os blocos de instrução referentes a tratamento de eventos podem ser encontrados na categoria “Eventos”, como mostra a Figura 1.126. Perceba que esses blocos estão prontos para tratar alguns eventos predefinidos e bem estabelecidos.

Figura 1.126: Blocos de instrução da categoria “Eventos”.

Mas, não se preocupe, caso você queira implementar algum conjunto de ações que devam ser executados mediante uma outra condição que não sejam os eventos predefinidos. Para casos bem específicos para a sua aplicação, podemos usar o bloco de instrução "espere até que...", ilustrado na Figura 1.127 e disponível na categoria "Controle". Com esse bloco, podemos programar uma sequência de ações que seriam disparadas somente no momento em que uma determinada condição, bem específica da sua aplicação, ocorresse. O Exercício 1.53 fará uso dessa instrução.

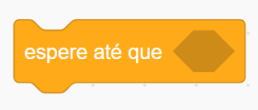

Figura 1.127: Bloco de instrução que nos permite verificar uma condição específica para disparar algumas ações.

Uma outra forma de deixar nossas aplicações mais interativas é explorar as possibilidades na categoria "Movimento". É possível, por exemplo, criar uma aplicação que faça um ator seguir o ponteiro do mouse em tempo de execução. Vale a pena checar também os blocos de instrução da categoria "Sensores".

Vamos praticar como criar aplicações interativas por meio de exercícios. Mãos à obra!

Exercício 1.49 Aplique o tratamento de evento de clique de mouse para fazer uma bailarina começar a dançar sobre o palco de um teatro.

Solução. Para este exercício, escolhemos um palco de um teatro para ser o cenário e inserimos o ator "Ballerina", disponíveis no Scratch. Incluímos na área de trabalho o bloco de instrução "quando este ator for clicado" e programamos as ações necessárias para troca de fantasias e encaixamos nele. O programa completo pode ser visto na Figura 1.128.

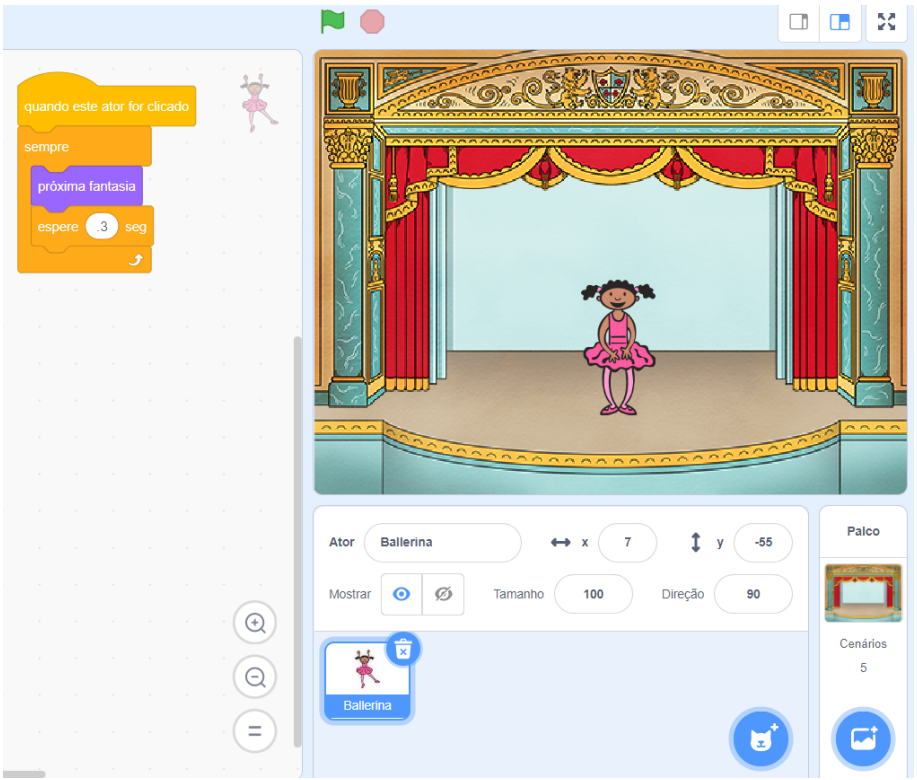

Figura 1.128: Um programa para solucionar o problema proposto no Exercício 1.49.

■

Exercício 1.50 Modifique o programa do Exercício 1.49 para que ela pare de dançar, caso a barra de espaço seja pressionada.

Solução. Para atender ao que foi pedido, será necessário retirar o laço de repetição "sempre". Vamos trocá-lo por um laço de repetição com uma condição de parada. Essa condição será justamente o pressionamento da tecla barra de espaço. Uma modificação válida para o programa, de modo a atender ao que foi pedido no exercício, pode ser vista na Figura 1.129.

Figura 1.129: Um programa para solucionar o problema proposto no Exercício 1.50.

■

Exercício 1.51 Vamos treinar o uso de tratamento de eventos de teclado: faça uma aplicação que permita um robô vigiar o corredor de uma escola. Ele se movimentará com comandos de teclas de direção para se locomover ao longo do corredor. A Figura 1.130 mostra o cenário em que o robô se encontra para cumprir sua missão de vigilante.

Para tornar o programa mais interessante, faça o robô mudar de pose conforme a direção para onde ele se movimenta. Este robô, em particular, tem 4 fantasias disponíveis no Scratch.

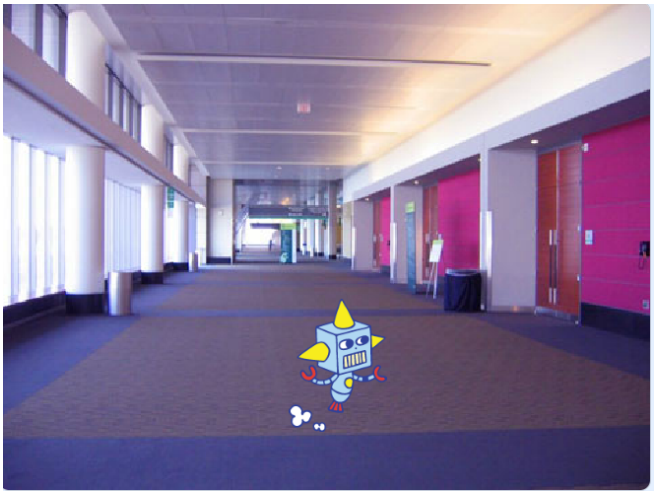

Figura 1.130: Cenário e ator escolhidos para o Exercício 1.51.

Solução. Para o programa pedido no Exercício 1.51, vamos adicionar 4 tratamentos de eventos de teclado: um para cada tecla de direção. Para cada uma delas, foi feita uma programação escolhendo uma fantasia apropriada para a direção de movimento. Para a movimentação feita com as teclas direcionais para cima e para baixo, foram adicionadas instruções para alterar o tamanho do robô, para dar uma ilusão de profundidade. O programa completo pode ser visto na Figura 1.131.

■

Figura 1.131: Um programa para solucionar o problema proposto no Exercício 1.51.

Exercício 1.52 Crie uma aplicação interativa que permita o(a) usuário(a) desenhar. O palco exibirá um lápis que vai seguir sempre a posição do mouse, desenhando linhas. Ao se pressionar a barra de espaços, a cor do traço muda aleatoriamente. Para aumentar a espessura do traço, o(a) usuário(a) pode pressionar a tecla seta para cima, e para diminuir a espessura do traço, o(a) usuário(a) pode pressionar a tecla seta para baixo.

Solução. Para implementar esta aplicação, serão necessários três eventos de teclado, um para cada tecla mencionada no enunciado da questão. O programa será inicado quando o(a) usuário(a) clicar sobre a bandeirinha verde, e a tarefa de pintura se inicia. O(A) usuário(a) poderá mudar a espessura e a cor do traço a qualquer momento, durante a execução do programa. A Figura 1.132 mostra um programa para solucionar o problema proposto.

■

Exercício 1.53 Nano é um ator que dá o maior valor a uma merenda, como se pode ver na Figura 1.133. Ele está prestes a se servir de uma tigela de frutas. Faça um programa que permita o Nano comer cada uma das 4 frutas disponíveis, fazendo com que a tigela vá esvaziando a cada fruta retirada.

Solução. O personagem escolhido para esta aplicação possui poses que simulação a ação de pegar um objeto e levar à boca. A tigela e as frutas são também atores. A solução para este exercício envolve um truque de fazer as frutas ficarem invisíveis, em momentos sincronizados com a animação do personagem, para simular a retirada delas da tigela.

A mudança na visibilidade de cada fruta é controlada por uma espécie de evento, que é muito específico para esta aplicação. O evento em questão seria o momento em que determinada fruta

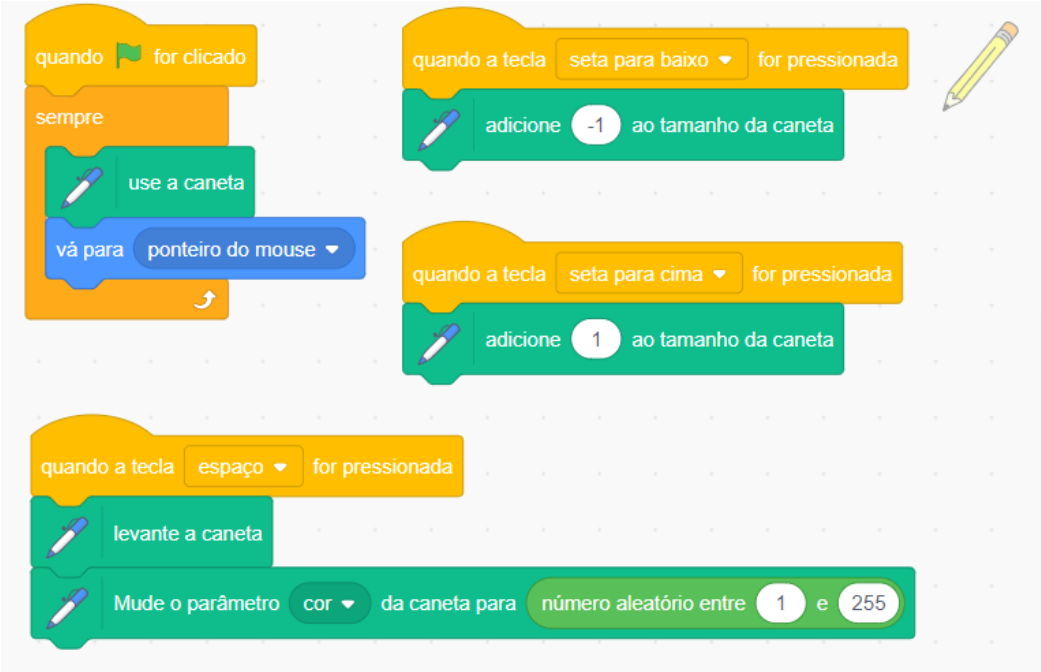

Figura 1.132: Um programa para solucionar o problema proposto no Exercício 1.52.

Figura 1.133: Personagem Nano, do Exercício 1.53.

tenha sido retirada para ser comida. Assim, vamos tratar como evento, 4 momentos específicos: 1, 2, 3 e 4, referentes às retiradas de cada uma das 4 frutas. A programação do personagem pode ser vista na Figura 1.134.

Cada fruta teve uma programação específica para determinar sua vez de ficar invisível. A Figura 1.135 mostra a programação do ator Strawberry, que é a primeira fruta a sumir da tigela.

As demais frutas possuem programação semelhante, mudando apenas o valor de comparação da variável "quantFrutas":

- Para a laranja, o valor foi 2;
- Para a maçã, 3;
- Para as bananas, o valor de comparação no teste foi 4.

Observe que o tratamento de eventos neste exercício foi por meio da utilização do bloco de instrução "espere até que". ■

Figura 1.134: Programação do personagem Nano, do Exercício 1.53.

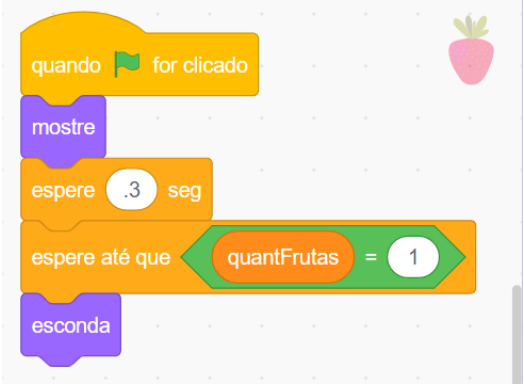

Figura 1.135: Programação do ator Strawberry, do Exercício 1.53.

2 Programação Convencional

No primeiro capítulo, estudamos uma linguagem de programação visual. Com ela, a montagem dos nossos programas era feita por meio da escolha, organização e encaixe de blocos de instruções disponibilizados pela linguagem. No caso específico deste curso, utilizamos a linguagem de programação Scratch. No entanto, vale lembrar que há outras linguagens de programação que funcionam de maneira semelhante, como o Google Blockly[1] e o App Inventor[2].

Nosso objetivo, neste capítulo, é conhecer uma outra forma de construir programas: desta vez, ao invés de utilizar blocos de instruções prontos, vamos construir programas por meio de instruções digitadas por nós. Vamos colocar o teclado para trabalhar!

Obs Denominaremos as linguagens de programação que funcionam dessa forma de **Linguagens de Programação Convencionais**.

A quantidade de linguagens de programação convencionais atualmente em uso é enorme! Na Figura 2.1, vemos os nomes de algumas das linguagens de programação em uso, e a grande maioria dos nomes presentes na figura são de linguagens de programação convencionais.

Figura 2.1: Figura ilustrativa com nomes de linguagens de programação.

[1] Você pode experimentar o Blockly no endereço https://blockly-demo.appspot.com/static/demos/code/index.html.
[2] Vocês terão a oportunidade de trabalhar com essa linguagem no Módulo 3 deste curso.

A programação convencional é, na verdade, a forma mais comum de se programar no mercado de trabalho voltado ao desenvolvimento de grandes sistemas. Nas seções que seguem, vamos aplicar o que já aprendemos sobre a construção de algoritmos e programas utilizando uma linguagem convencional. A linguagem escolhida para nosso estudo é o Python[3], por oferecer muitas facilidades de uso e por sua ampla gama de aplicações em diferentes áreas do conhecimento, como Desenvolvimento de Jogos, Estatística, Sistemas Financeiros, além de sistemas voltados às áreas de Ciência de Dados e de Inteligência Artificial.

2.1 – Conceitos de linguagem de programação convencional

O desenvolvimento de programas utilizando uma linguagem de programação visual nos permitiu amadurecer o **Pensamento Computacional**, direcionando nossa atenção para a **lógica** necessária para a solução de problemas.
Com o aprimoramento desta habilidade, poderemos incluir agora mais um elemento importante no nosso aprendizado: a atenção à **sintaxe** da linguagem convencional escolhida.

Definição 2.1.1 **Sintaxe** é o conjunto de regras que determinam as relações entre as palavras de uma língua para a formação de frases.

A Definição 2.1.1 descreve o conceito de sintaxe no contexto da linguagem natural. Dentro desse contexto, identificamos os elementos sintáticos de uma frase como sujeito, predicado, objeto direto, entre outros. Tais elementos possibilitam a compreensão do que se pretende comunicar com uma frase.

Já no contexto das linguagens de programação, a sintaxe será o conjunto de regras que determinam as possibilidades de associação de **palavras reservadas** da linguagem de programação para a formação de comandos e instruções que podem ser compreendidos e executados por um computador.

Definição 2.1.2 Denominamos **Palavra Reservada** todas as palavras predefinidas em uma linguagem de programação, para escrever comandos, estruturas básicas e estruturas de controle.

Vale ressaltar que cada linguagem de programação tem seu próprio conjunto de palavras reservadas. Por isso, ao começar a programar em uma nova linguagem de programação, é importante conhecer sua sintaxe.

2.1.1 – Noções Básicas

Para começar, vamos compreender como a linguagem convencional que vamos utilizar – o Python – funciona. Em nossas atividades práticas, nossa rotina será, tipicamente:

(a) Escrever um programa, utilizando um Ambiente de Programação compatível com o Python.
- Para os exemplos deste material, utilizaremos o ambiente de programação Thonny[4].
- Os programas que vamos escrever serão salvos com a extensão ".py".
- Os programas escritos por nós são denominados **código-fonte**.

(b) Executar o programa.
- Nesse momento, ocorrem algumas operações sobre o código-fonte que você escreveu, para verificar se há erros de sintaxe.

[3]Disponível em https://www.python.org/
[4]Disponível para download em https://thonny.org/

- O seu programa só é executado se não houver erros.

A Figura 2.2 mostra o ambiente de programação que vamos utilizar, com destaque para as áreas onde vamos escrever nossos códigos-fonte e onde vamos observar o resultado da execução de nossos programas.

Figura 2.2: Ambiente de programação Thonny, para a linguagem Python.

As operações referentes à verificação de erros no código-fonte e à execução do programa por um computador são efetuadas por Softwares Básicos (que vimos no Módulo 1 deste curso) e que realizam:

(a) A análise do código, para verificar se há erros de sintaxe;
(b) Não havendo erros de sintaze, o software básico gera uma versão traduzida do seu código para uma linguagem que o computador seja capaz de compreender e executar, chamada de **Linguagem de Máquina**.

No caso específico da linguagem Python, o programa escrito por nós passará pelo seguinte processo:

- O código-fonte passa por um software **compilador**, que vai traduzir o código escrito na linguagem Python para um código que leva o nome de *Bytecode*;
- O *Bytecode* gerado é executado em uma **máquina virtual**, que é um software que vai traduzir o Bytecode para a linguagem de máquina;
- O computador recebe o código em linguagem de máquina e o executa.

A Figura 2.3 ilustra o processo.

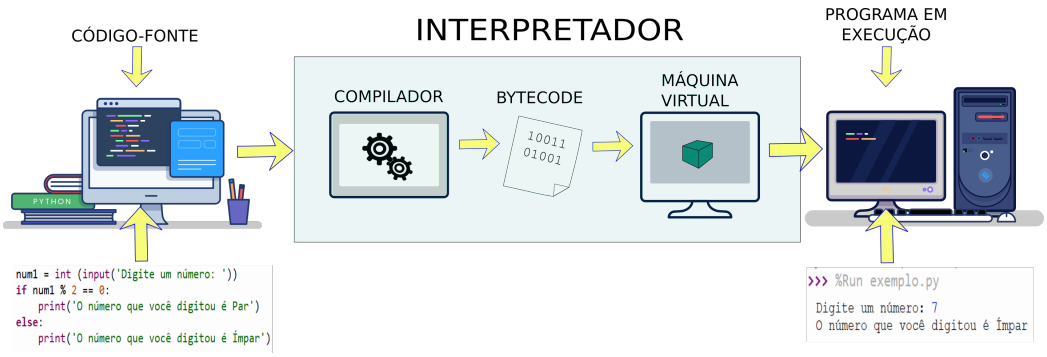

Figura 2.3: Processo de tradução e execução de um código escrito em Python.

O papel do(a) programador(a), no processo apresentado na Figura 2.3, é de escrever o código-fonte e, ao concluir, dar um comando para que o código seja analisado e, se estiver tudo correto, ser executado. Isso é feito de maneira bem simples. No caso do ambiente de programação que estamos utilizando – o Thonny – o comando para realizar essas operações sobre o seu código-fonte é um clique sobre o botão "Executar programa atual". Alternativamente, podemos simplesmente pressionar a tecla F5 para realizar esta tarefa. A Figura 2.4 mostra onde podemos encontrar o botão "Executar programa atual" na interface do Thonny.

Figura 2.4: Botão “Executar programa atual” na interface do Thonny.

Sintaxe

Quando construímos nossos programas utilizando uma linguagem de programação visual, não há a necessidade de se analisar o código para verificar se há erros de sintaxe. Isso acontece porque os encaixes dos blocos de instruções são feitos de modo a impedir uma combinação errada de comandos no programa.

Na programação convencional, entretanto, as instruções serão digitadas por nós. Assim, será essencial conhecermos as palavras reservadas da linguagem e utilizá-las corretamente, respeitando a sua sintaxe.

Para se ter uma ideia do que acontece quando digitamos algum comando errado, a Figura 2.5 mostra um exemplo de erro: o comando “print”, na última linha do código, está escrito de maneira incorreta (está escrito “prin”). Por conta disto, ao executar o programa, o compilador acusa o problema. A seta vertical na figura aponta o erro de sintaxe e as setas horizontais destacam as mensagens de erro que o ambiente de programação exibe ao(à) programador(a).

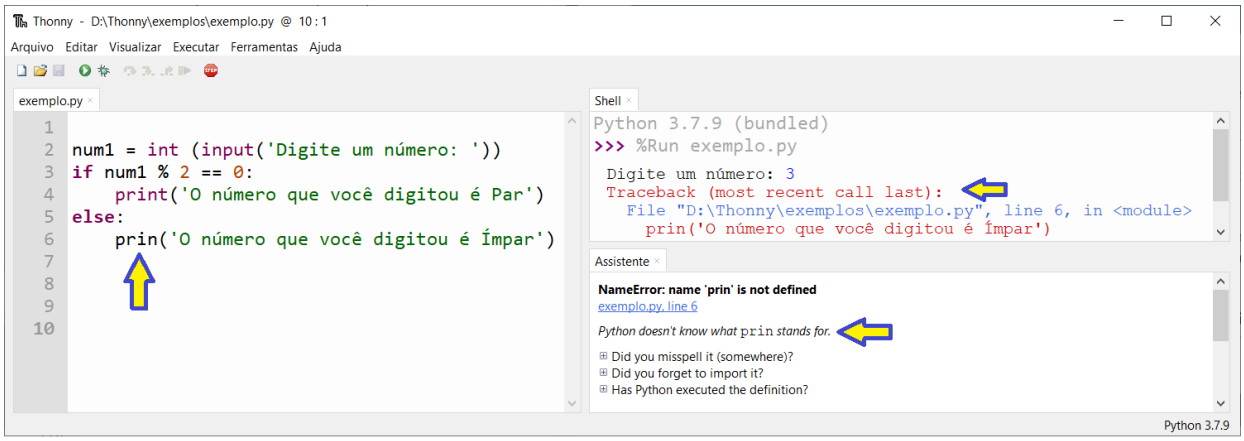

Figura 2.5: Um erro no código-fonte e a exibição de mensagens de erro no ambiente de programação Thonny.

Os detalhes da sintaxe da linguagem Python serão apresentados ao longo do restante deste capítulo, onde, em cada seção, serão apresentadas as palavras reservadas e as regras de sintaxe pertinentes a cada assunto abordado.

De antemão, dá para perceber que teremos muita coisa nova para aprender. Assim, seria muito legal se conhecêssemos uma forma de tomar nota das coisas novas que vamos aprendendo dentro dos próprios programas. A subseção a seguir vai tratar deste assunto, especificamente.

Comentários dentro de um programa

Antes de darmos continuidade aos nossos estudos, vale ressaltar um recurso bastante útil: os **Comentários**.

Na sintaxe do Python, os comentários não são interpretadas como comandos, e portanto, não são executados. Eles servem para dois propósitos bem interessantes:

- Auxiliam na documentação do programa, permitindo que se anote informações como versão do programa, nome do(a) autor(a), propósito geral do programa, etc.
- Auxiliam na suspensão temporária da execução de um determinado comando do programa, para testar alguma modificação sem necessariamente ter que apagar o comando.

Os comentários podem ser inseridos em um código Python de duas maneiras: como linhas ou como blocos de linhas.

Para adicionar apenas uma linha de comentário, a sintaxe do Python requer que se inicie a linha com o caractere cerquilha (#):

```
# Esta é uma linha de comentário no meu código.
```

Se desejarmos adicionar um bloco de linhas de comentários, a sintaxe do Python requer que se utilize aspas triplas marcando o início e o fim do bloco de comentários. As aspas triplas podem ser do tipo ' ' ' ou " " ":

```
'''Este é um bloco de linhas de comentários
no meu programa, utilizando as aspas simples três vezes.
Este recurso é bastante útil para a inserção
de várias informações que devem ficar agrupadas no código.'''
```

```
"""Este também é um bloco de linhas de comentários
no meu programa,
só que agora estou utilizando as aspas duplas três vezes."""
```

Para compreender melhor a função de uma linha ou bloco de comentários, observe os Exemplos 2.1, 2.2 e 2.3:

■ **Exemplo 2.1** O programa a seguir exibe na tela o valor armazenado em uma variável de nome x:

```
x = 3
print(x)
```

Caso eu julgue necessário adicionar uma anotação para me ajudar a lembrar o que o programa faz, ou até mesmo para comunicar um colega de trabalho sobre o que o programa faz, eu posso adicionar uma linha de comentário ao programa:

```
#Este programa exibe na tela o valor armazenado em x
x = 3
print(x)
```

O Exemplo 2.2 apresenta a utilização de um bloco de comentários:

■ **Exemplo 2.2** O programa a seguir exibe um nome armazenado em uma variável.

```
x = "José"
print(x)
```

Esse programa foi um dos vários que um aluno precisou fazer para resolver uma lista de exercícios passados pela professora. Para facilitar a identificação da questão da lista de exercícios que este programa resolve, e a identificação do autor do programa, o aluno resolveu inserir essas informações como bloco de comentários no programa dele. Ele escolheu utilizar as aspas simples três vezes para demarcar o bloco de linhas de comentário:

```
'''Programa referente à questão 2 da lista de exercícios.
Autor do programa: José
Matrícula: 21150
Turma: B'''

x = "José"
print(x)
```

Para finalizar, vamos observar um exemplo em que linhas de comentário servem para realizar testes rápidos em nossos códigos, sem a necessidade de se fazer grandes alterações no mesmo.

■ **Exemplo 2.3** José quis juntar os dois programas dos Exemplos 2.1 e 2.2 em um só. O programa ficou como se vê abaixo:

```
x = 3
x = "José"
print(x)
```

Ao executar o programa, o resultado exibido na tela é `José`. Ele gostaria de testar a execução do código, fazendo com que ele passe a exibir o valor 3. Como José é muito cauteloso, ele não gostaria de retirar nenhuma instrução do programa dele – vai que ele pode precisar depois?

Então, para testar uma execução do código fazendo com que o programa exiba o valor 3, uma saída possível seria suspender temporariamente a execução da instrução da linha 2. Para isso, basta fazer com que o comando da linha 2 vire um comentário:

```
x = 3
# x = "José"
print(x)
```

A instrução na linha 2 deixa de ser reconhecida como um comando, e a execução do programa resultará na exibição do valor 3. Caso José queira recuperar a linha 2 para que seja novamente reconhecida como um comando a ser executado, basta retirar a cerquilha do início da linha.

2.2 – Variáveis e Constantes

Vimos no Módulo 1 deste curso que uma instrução de um programa, para ser executada, precisa estar carregada na Memória Principal do computador. Assim, a memória principal precisa conter:

(a) As instruções de um programa em execução;
(b) Os dados que essas instruções processam.

Para guardar **dados** na memória principal do computador, utilizaremos as **Variáveis** e as **Constantes**.

2.2.1 – Variáveis

As **variáveis** são o meio pelo qual armazenamos e acessamos dados específicos na memória. Para oferecer essa funcionalidade, as variáveis precisam ter:

- um nome;
- um tipo;
- um valor.

O **nome** da variável é a palavra que usamos nas instruções de nossos algoritmos para acessar e manipular um dado específico armazenado na memória principal. Para criar o nome de uma variável em um programa escrito na linguagem Python, é necessário seguir algumas regras de nomenclatura, descritas a seguir:

- O nome de uma variável pode iniciar com uma letra ou com o símbolo de undescore.
- O nome de uma variável não pode iniciar com um dígito numérico.
- O nome de uma variável pode conter apenas caracteres alfanuméricos e símbolos de underscore (A-z, 0-9, e _).
- O Python faz distinção entre letras maiúsculas e minúsculas. Assim, as variáveis idade, Idade e IDADE são três variáveis diferentes.

Obs Chamamos as linguagens de programação que fazem distinção entre maiúsculas e minúsculas de **Linguagens de Programação Sensíveis à Caixa**. O termo original em inglês é "*Case-Sensitive Programming Language*".

Exemplos de nomes **válidos** para variáveis são: soma, resultado, num1, valor2, `x_2`, totalVendas, `producao_mes`, x123, somaTotal, ABC.

Obs **IMPORTANTE**: não é permitido usar espaço em branco no nome de uma variável.

Exemplos de nomes **não válidos** para variáveis são: 1valor, senha do usuario, 123, AB C, producao mes.

Obs Embora sejam **válidos** nomes como ABC e x123 para variáveis, é recomendado adotar nomes mais significativos para as variáveis. Alguns bons exemplos são: totalVendas, somatorio, nomeUsuario.

O **tipo** da variável indica a natureza dos dados que ela irá armazenar. Além disso, é pelo tipo da variável que o Python saberá que operações são aplicáveis aos dados a serem processados. Os tipos de dados que estudaremos neste curso são apresentados na seção 2.3.
As operações aplicadas a uma variável, no Python, são:

- Criação/Inicialização de variáveis;
- Comando de atribuição de valores a variáveis.

Obs Observe que, na linguagem Python, não se faz a declaração de uma variável, como fizemos com os algoritmos escritos na forma de Descrição Narrativa.

O **valor** armazenado em uma variável, no Python, poderá ser qualquer valor válido, dentre os tipos de dados apresentados na Seção 2.3.

No contexto da linguagem de programação Python, é muito comum referir-se aos dados armazenados em uma variável utilizando o termo **objeto**. Este será o termo que utilizaremos no restante deste material para nos referir a dados armazenados em variáveis.

Criação/Inicialização de Variáveis

No Python, **as variáveis não são declaradas**. Podemos simplesmente escrever o nome de uma variável e já inicializá-la no primeiro momento em que vamos utilizá-la. O Exemplo 2.4 mostra a criação e inicialização de algumas variáveis para um programa específico.

■ **Exemplo 2.4** Considere um programa que calcula a soma de dois valores, exibindo o resultado ao final. O programa cria três variáveis: `x`, `y` e `soma`. As variáveis `x` e `x` serão inicializadas com os objetos a serem somados. A variável `soma` receberá o resultado da operação de adição dos objetos armazenados em `x` e `y`.

```
x = 5
y = 2
soma = x + y
print(soma)
```

Os comandos das duas primeiras linhas do programa apresentado correspondem às operações de criação e inicialização de variáveis em Python.

Comando de Atribuição de Valores a Variáveis

A qualquer tempo no fluxo de execução de instruções, podemos precisar modificar (ou atualizar) o objeto armazenado em uma variável. O **comando de atribuição** efetua o armazenamento de um novo objeto a uma dada variável.

Para exemplificar, considere uma variável `x` que deverá armazenar o objeto 2 nela. A sintaxe do Python para o comando de atribuição desse objeto à variável `x` é:

```
x = 2
```

onde o sinal de igualdade denota a operação de atribuição de um objeto a uma variável.

Podemos atribuir a uma variável um objeto com valor conhecido (uma constante), um objeto armazenado em outra variável ou até mesmo uma expressão.

O Exemplo 2.5 lista alguns comandos válidos para atribuição de valores a variáveis.

■ **Exemplo 2.5** Alguns comandos válidos para atribuição de valores a variáveis podem ser vistos na Tabela 2.1.

```
idade = 15
minhaIdade = idade
preco = 10.49
precoAtual = preco + 1000
nomeCompleto = "Francisco José da Silva"
opcao = 'N'
venceu = False
pago = True
```

Tabela 2.1: Exemplos de comandos de atribuição de objetos a variáveis na sintaxe do Python.

Obs Algumas observações importantes sobre o comando de atribuição de um objeto a uma variável:

- A notação `x = 2` é lida como "x recebe 2".
- Uma variável só guarda um valor por vez. Assim, quando aplicamos um comando de atribuição a uma variável, o antigo objeto armazenado nela será perdido. O Exemplo 2.6 ilustra esse processo.
- Se desejarmos atribuir o resultado de uma expressão a uma variável, é importante observar se os tipos dos objetos são compatíveis com as operações a serem realizadas na expressão. Por exemplo, se desejamos atribuir a uma variável de nome `soma` o resultado da adição de uma variável `x` armazenando um objeto numérico com outra variável `y` armazenando um objeto do tipo texto, a operação de adição não será possível, e a instrução de atribuição não será realizada.

■ **Exemplo 2.6** Considere a seguinte sequência de comandos:

```
matricula = 100
matricula = 95
```

Ao final da execução dos comandos acima, o objeto armazenado na variável `matricula` é o de valor 95. O objeto de valor 100, armazenado anteriormente, é descartado da memória do computador e não pode ser recuperado após a execução do comando da linha 2. ■

Exercício 2.1 Indique o valor armazenado em cada variável **ao final da execução** das sequências de comandos a seguir:

(a)
```
nome = "Maria"
nome = "Antônio"
nome = "Luiza"
```

(b)
```
preco = 50.00
preco = 49.99
```

(c)
```
taPago = False
taPago = True
```

Solução. Os valores armazenados ao final da execução dos comandos listados no Exercício 2.1 são:

(a) a variável `nome` tem somente o objeto "Luiza" armazenado nela.
(b) a variável `preco` tem somente o objeto 49.99 armazenado nela.

(c) a variável `taPago` tem somente o valor `True` armazenado nela.

■

2.2.2 – Constantes

Enquanto as variáveis servem como um lugar para armazenar e acessar dados, as constantes são os dados propriamente ditos.

A Tabela 2.2 traz mais alguns exemplos, destacando as diferenças de sintaxe para denotar um objeto de valor numérico, de texto e lógico.

Sintaxe	**Tipo**
123	numérico inteiro
3.1415	numérico real
9	numérico inteiro
“123”	texto
“3.1415”	texto
“9”	texto
’9’	texto
True	lógico
False	lógico
’False’	texto
“False”	texto

Tabela 2.2: Diferenças entre os tipos de constantes conforme a sintaxe do Python.

Obs A sintaxe do Python aceita o uso de aspas simples ou duplas para denotar objetos do tipo texto. Não existe o tipo caractere em Python.

2.3 – Tipos de Dados

O tipo determina a natureza de um dado a ser armazenado na memória do computador. Assim, como vimos no Módulo 1 deste curso, sabemos que há tipos para armazenar dados numéricos, de texto e dados lógicos. Além de definir a natureza do dado, o tipo serve também para determinar quanto espaço será necessário separar na memória principal para armazenar o dado. Como exemplo, o espaço armazenado por um valor numérico real será maior que o espaço necessário para armazenar um valor numérico inteiro, por conta da informação adicional de casas decimais que o valor numérico real precisa armazenar.

Na linguagem Python, os tipos de dados mais utilizados, e que abordaremos neste material, estão listados na Tabela 2.3:

Obs Há outros tipos de dados definidos para o Python, porém vamos limitar nossa discussão para a listagem apresentada neste material.

Obs De agora em diante, chamaremos de **objeto** os dados armazenados em variáveis de quaisquer tipos.

Tipo	Palavra reservada
texto	`str`
numérico – inteiro	`int`
numérico – real	`float`
lógico	`bool`
sequência	`list`
sequência	`range`
mapeamento	`dict`
conjunto	`set`

Tabela 2.3: Tipos de dados mais comuns da linguagem Python

A lista de tipos apresentada acima contém alguns tipos que já discutimos anteriormente no Módulo 1 – texto, numérico e lógico –, e outros que são novidade: sequência (`list` e `range`), mapeamento (`dict`) e conjunto (`set`). Nas subseções a seguir, vamos conhecer um pouco melhor os tipos referentes a armazenamento de objetos em sequências, em estruturas de mapeamento e em conjuntos.

Tipo sequência: `list`

Assim como vimos na linguagem Scratch, em alguns programas, é desejável se armazenar mais de um valor em uma única variável. Variáveis do tipo `list` possibilitam esse tipo de armazenamento. O conteúdo de uma lista tem as seguintes características:

- Os objetos dentro de uma lista podem ser repetidos;
- Os objetos dentro de uma lista são modificáveis;
- Cada objeto em uma lista tem a informação sobre a sua posição dentro da lista.
  - Assim, cada objeto da lista pode ser encontrado por meio de sua posição, ou **índice**.
  - O primeiro objeto de uma lista tem índice 0; o segundo objeto tem índice 1, e assim por diante.
  - Chamamos a lista de uma estrutura de armazenamento **indexada**.
- Os objetos de uma mesma lista podem ser de diferentes tipos.
- Escrevemos uma lista com seus objetos entre colchetes, e separados por vírgulas. O Exemplo 2.7 apresenta a sintaxe apropriada para listas.

Para melhor compreender o conceito de índice em uma lista, observe o Exemplo 2.7.

■ **Exemplo 2.7** Considere uma lista contendo 5 objetos: 3, 7.9, "Maria", -3 e False. A sintaxe para escrever essa lista em Python é:

```
[3, 7.9, ''Maria'', -3, False]
```

Precisamos, ainda, de uma variável para armazenar essa lista. Assim, usaremos uma variável `x` para receber essa lista:

```
x = [3, 7.9, ''Maria'', -3, False]
```

Cada um dos objetos dessa lista podem ser acessados individualmente, por meio de seu índice. O primeiro elemento sempre terá índice 0, e os demais objetos terão índices que aumentam de uma unidade a cada próximo objeto na sequência.

A Figura 2.6 apresenta a estrutura de armazenamento, mostrando os índices de cada objeto contido nela.

Figura 2.6: Uma lista em Python contendo 5 objetos de diferentes tipos. Os índices de cada objeto estão indicados em vermelho.

Observando a Figura 2.6, podemos perceber que cada índice está escrito entre colchetes. Esta é a sintaxe para nos referir ao índice de um objeto dentro de uma lista. Assim, para acessar, por exemplo, o primeiro elemento da lista `x`, devemos escrever:

```
x[0]
```

Observe, também, que o último objeto da lista `x` do Exemplo 2.7 tem índice `4`, apesar de termos 5 objetos. Isto se deve pelo fato de os índices começarem com 0, sempre.

Exercício 2.2 Dada a lista em um programa escrito na linguagem Python:

```
minhaLista = [7, 5, 7, "Oito", 3.25 ]
```

e sabendo que o comando para exibir um determinado valor na tela é o comando `print()`, escreva uma instrução para exibir na tela:

(a) O segundo objeto da lista;
(b) O terceiro objeto da lista;

Solução. As respostas para os itens (a) e (b) do Exercício 2.2 são apresentadas a seguir:

(a) O comando para exibir na tela o segundo objeto armazenado na variável `minhaLista` é:
`print(minhaLista[1])`
(b) O comando para exibir na tela o terceiro objeto armazenado na variável `minhaLista` é:
`print(minhaLista[2])`

■

Tipo sequência: `range`

O tipo `range` armazena uma sequência de números inteiros. No momento da criação de um range, são dados os limites inferior e superior da sequência. Podemos ver uma sequência desse tipo no Exemplo 2.8.

■ **Exemplo 2.8** Criação de uma sequência de inteiros de 3 a 7. Note, pelo exemplo, que os limites da sequência incluem o valor inferior e excluem o valor superior. Assim, para a criação da sequência de 3 a 7, são fornecidos os valores 3 e 8.

```
x = range(3, 8)
```

Tipo mapeamento: `dict`

O tipo `dict` armazena objetos utilizando um mapeamento constituído por pares **chave:valor**. A chave cria uma associação com os objetos dentro de um dicionário. Assim, para encontrar um objeto em um dicionário, podemos procurar por ele utilizando sua chave. Vale ainda ressaltar que, dentro de um mesmo dicionário, podemos ter objetos de diferentes tipos.

O Exemplo 2.9 mostra um dicionário criado com 3 pares do tipo chave:valor. No exemplo, é possível observar a sintaxe para denotar dicionários em Python, utilizando chaves { }.

■ **Exemplo 2.9** Criação de um dicionário `x` com 3 pares chave: valor. No exemplo, as chaves são `"disciplina"`, `"ano"` e `"turma"`.

```
x = {''disciplina'':''Programação'', ''ano'':2021, ''turma'':''B''}
```

■

Obs Em um dicionário, cada chave é única: dois objetos não podem ter a mesma chave!

Exercício 2.3 Considere o dicionário

```
{x = {"fabricante":"Volkswagen", "modelo":"fusca", "ano": 1980}
```

Sabendo que o comando `print()` exibe na tela o valor armazenado em uma variável, escreva comandos em Python para realizar as seguintes tarefas:

(a) A exibição do conteúdo completo do dicionário;
(b) A exibição do objeto referente à chave "modelo" do dicionário;
(c) A exibição do objeto referente à chave "ano" do dicionário;

Solução. Os comandos pedidos nos itens (a) a (c) do Exercício 2.3 estão enumerados abaixo:

(a) Para a exibição do conteúdo completo armazenado no dicionário, o comando é:
`print(x)`
(b) Para a exibição do objeto referente à chave "modelo", o comando é:
`print(x["modelo"])`
(c) Para a exibição do objeto referente à chave "modelo", o comando é:
`print(x["ano")]`

■

Obs Podemos observar no Exercício 2.3 que as chaves de um dicionário funcionam como os índices de uma lista, para permitir o acesso a um objeto armazenado. A diferença é que os índices de uma lista são uma sequência ordenada de números, começando com índice 0. Já no dicionário, podemos utilizar qualquer objeto como chave para acessar seus elementos.

Tipo conjunto: `set`

O tipo `set` em Python armazena um conjunto de objetos, observando as caraterísticas de um conjunto:

- O conjunto não possui itens repetidos;
- Os objetos de um conjunto não são ordenados;
- Os objetos de um conjunto não são modificáveis
  - No entanto, podemos excluir objetos de um conjunto e inserir outros.
- Os objetos de um conjunto não são indexados.
- Os objetos de um conjunto podem ser de tipos diferentes. Assim, um conjunto pode conter valores numéricos, textos e até mesmo outros conjuntos ou listas e dicionários dentro dele.

O Exemplo 2.10 mostra um conjunto criado com 5 objetos. No exemplo, é possível observar a sintaxe para denotar conjuntos em Python, utilizando chaves { }.

■ **Exemplo 2.10** Criação de um conjunto `x` com 5 objetos:

```
x = {0, ‘‘Maria’’, 6, 3.14, True}
```

O tipo de um dado determina também quais operações podem ser efetuadas sobre ele. Maiores detalhes sobre as operações aplicáveis aos tipos `list`, `range`, `dict` e `set` serão discutidos na Seção 2.5.

2.3.1 – Definindo o tipo de uma variável

No Python, o tipo de uma variável pode ser definido de diferentes formas:

(a) Por atribuição de uma valor a uma variável;
(b) Por **conversão de tipos**, que pode ser:
  - Explícita (também chamada de *typecast*);
  - Implícita.

Maiores detalhes sobre as conversões de tipos mencionadas nos itens (a) e (b) serão apresentados a seguir.

Determinando o tipo por atribuição de valor

Em Python, a operação de atribuição de valor a uma variável é uma das formas de determinar o tipo da variável em questão. O Exemplo 2.11 mostra alguns casos.

■ **Exemplo 2.11** Considere uma variável de nome `x` em um código Python. Essa variável pode receber valores de quaisquer tipos. Assim, se escrevermos a instrução de atribuição

```
x = 10
```

a variável `x` será do tipo numérico `int`.

Considere, ainda, que a mesma variável `x` receba agora um valor de texto, digamos, “Maria”, dentro do mesmo código-fonte que tinha atribuído o valor 10 a `x`. O comando de atribuição que

realiza essa operação é:

```
x = ‘‘Maria’’
```

a variável `x` será agora do tipo texto `str`.

...e como saber o tipo de uma variável?

Diante da flexibilidade que uma variável tem para assumir diferentes tipos ao longo de um mesmo programa, o Python disponibiliza um comando para consultarmos o tipo de uma variável em um dado momento do programa. A sintaxe para este comando é:

```
type(x)
```

onde `x` é o objeto cujo tipo queremos descobrir.

A Figura 2.7 mostra um exemplo de programa utilizando a instrução `type()` para exibir o tipo de uma variável `x` e o resultado de sua execução, destacado com uma seta.

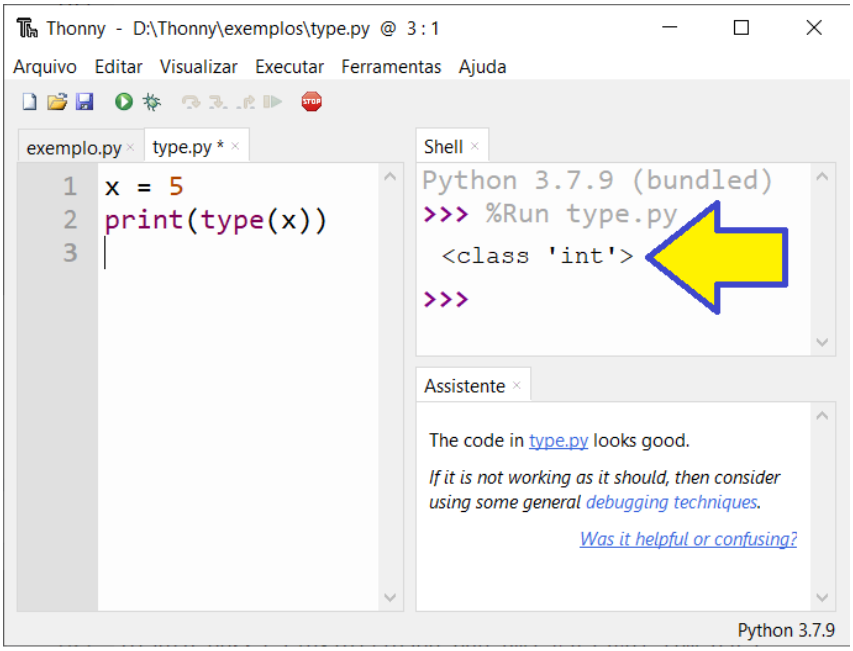

Figura 2.7: Um programa que utiliza a instrução `type()`, com o resultado da execução visível no ambiente de programação Thonny.

Determinando o tipo por conversão de tipos

A linguagem Python nos permite alterar o tipo de uma dada variável, independentemente do valor atribuído a ela.
Há duas maneiras de se fazer isso:

(a) Conversão explícita (ou *typecasting*);
(b) Conversão implícita.

Conversão de tipos explícita (ou *typecasting*)

As operações de conversão explícita de tipos são realizadas por meio da utilização de comandos já definidos na linguagem. A Tabela 2.4 mostra os comandos para conversão explícita de tipos, junto com exemplos de sua utilização e resultados produzidos em cada exemplo:

Comando	Exemplo	Valor armazenado em `x`
`str()`	`x = str(4)`	Um texto cujo conteúdo é "4".
`int()`	`x = int(2.5)`	O valor inteiro 2.
`float()`	`x = float(2)`	O valor real 2.0.
`bool()`	`x = bool(5)`	O valor `True`.
`list()`	`x = list(range(8))`	Uma lista de valores de 0 a 7.
`range()`	`x = range(4)`	Uma sequência numérica de 0 a 3.
`dict()`	`x = dict(nome="Ed", mat=2150)`	Um dicionário {nome:"Ed", mat:2150}.
`set()`	`x = set([1, 2, 2, 3, 3, 3])`	Um conjunto com os elementos {1, 2, 3}.

Tabela 2.4: Comandos para conversão explícita de tipos

Exercício 2.4 Estou confusa com um código em Python que escrevi. Vou mostrar aqui para pedir ajuda:

```
minhaSequencia = [7, 5, 7, "Oito", 3.25]
novoObjeto = set(minhaSequencia)
print(novoObjeto)
```

O problema que aconteceu foi o seguinte: quando executei o código, vi que o conteúdo do objeto `novoObjeto` "sumiu" com um dos elementos de `minhaSequencia`! A variável `minhaSequencia` tem 5 objetos dentro dela, e a variável `novoObjeto` possui apenas 4. E para aumentar a confusão, o conteúdo de `novoObjeto` saiu todo embaralhado, mostrando os objetos em uma ordem diferente da ordem dos objetos de `minhaSequencia`. Alguém pode me explicar o que está acontecendo com esse código?

Solução. Primeiramente, é importante informar a autora do código de que não há nada de errado com o código. Tudo está funcionando **exatamente** como deveria. O que a autora do código pode não estar percebendo é que as variáveis `minhaSequencia` e `novoObjeto` são de tipos diferentes: `minhaSequencia` é uma **lista** e `novoObjeto` é um **conjunto**. A instrução `set(minhaSequencia)` converteu o tipo de `minhaSequencia` para conjunto e armazenou na variável `novoObjeto`. Assim, a instrução que exibe o conteúdo de `novoObjeto` na tela vai respeitar as características do tipo `set`: não haverá objetos repetidos, e os objetos não seguem uma ordem determinada. Inclusive, a cada vez que o programa foi executado, o conteúdo de `novoObjeto` será exibido em ordens diferentes. ■

Exercício 2.5 Dados os comandos abaixo, informe os resultados de suas execuções:

(a)
```
minhaSequencia = [7, 5, 7, "Oito", 3.25]
print(type(minhaSequencia))
```

(b)
```
minhaSequencia = [7, 5, 7, "Oito", 3.25]
print(type(set(minhaSequencia)))
```

(c)
```
minhaSequencia = {7, 5, 7, "Oito", 3.25}
print(type(minhaSequencia))
```

(d)
```
x = range(5)
print(x)
print(type(x))
```

(e)
```
x = list(range(5))
print(x)
print(type(x))
```

(f)
```
x = 5
y = 2
print(int(x/y))
```

(g)
```
x = 8
x = str(x)
print(x)
print(type(x))
```

Solução. As respostas para os itens (a) a (g) do Exercício 2.5 podem ser vistas abaixo:

(a) O programa vai exibir na tela o resultado `<class 'list'>`.
(b) O programa vai exibir na tela o resultado `<class 'set'>`.
(c) O programa vai exibir na tela o resultado `<class 'set'>`.
(d) O programa vai exibir na tela o resultado
`range(0, 5)`
`<class 'range'>`.
(e) O programa vai exibir na tela o resultado
`[0, 1, 2, 3, 4]`
`<class 'list'>`.
(f) O programa vai exibir na tela o resultado 2.
(g) O programa vai exibir na tela o resultado
`8`
`<class 'str'>`.

■

Conversão de tipos implícita

Este tipo de conversão não é realizada por comandos escolhidos por um(a) programador(a); ela é realizada de maneira automática pelo Python. Para melhor compreender o processo, observe o Exemplo 2.12.

■ **Exemplo 2.12** Considere as instruções:

```
x = 2
y = 3.5
soma = x + y
```

Pelas instruções escritas no Exemplo 2.12, percebe-se, facilmente, que:

- A variável `x` é do tipo `int`;

- A variável `y` é do tipo `float`;

Os tipos de `x` e `y` foram determinados pelas operações de atribuição de valores a elas. À variável `soma` foi atribuído o resultado de uma operação de adição envolvendo as variáveis `x` e `y`, sendo cada uma de um tipo diferente. E agora, qual será o tipo da variável `soma`?

A resposta é: quando uma variável recebe o resultado de uma expressão envolvendo objetos de tipos diferentes, o Python determinará o tipo dessa variável como sendo o tipo que ocupa maior espaço na memória. Isso é feito para evitar a perda de dados. O tipo float, por precisar armazenar informação de casas decimais, ocupa mais espaço na memória que o tipo int, que não tem casas decimais para armazenar. Assim, a variável `soma` será do tipo `float`.

Exercício 2.6 Qual o tipo da variável `x` no programa abaixo?

```
x = '5'
```

Solução. A variável `x` é do tipo texto (`str`), pois o valor atribuído a ela está escrito como um texto, e não como um valor numérico. ■

2.4 – Entrada e Saída de Dados

O Python disponibiliza instruções para permitir a interação com o(a) usuário(a) de um programa, por meio de entradas de dados fornecidos pelo(a) usuário(a) via teclado e pela exibição, em tela, de mensagens e valores correspondentes a resultados do processamento dos dados.

Nesta seção, vamos conhecer os comandos que utilizaremos para indicar operações de entrada e saída em nossos programas escritos na linguagem Python.

Comando de Entrada

A instrução em Python que permite a **entrada de dados** é o comando `input()`. A sintaxe completa dessa instrução será apresentada em breve. Porém, para melhor compreendê-la, é interessante antes conhecermos como funciona o processo de entrada de dados em um programa escrito em Python. Uma sequência de ações é listada no Exemplo 2.13.

■ **Exemplo 2.13** O processo de entrada de dados em um programa escrito em Python se dá como segue:

- Mediante solicitação emitida pelo sistema, o(a) usuário(a) do programa digita um valor utilizando o teclado;
- Após digitar o valor, o(a) usuário(a) pressiona a tecla ENTER;
- Ao pressionar ENTER, o valor informado via teclado é armazenado em uma variável;
- A variável em questão recebe o valor informado via teclado por meio do comando `input()`.
- Precisaremos, portanto, de uma variável para receber o resultado produzido pela instrução `input()`.
- Os parênteses do comando `input()` são o espaço onde devemos escrever a mensagem solicitando o dado de entrada do(a) usuário(a).

Desta forma, a sintaxe do comando de entrada é:

```
variavel = input("Mensagem de solicitação de dados")
```

■

Obs IMPORTANTE: a instrução `input()` faz com que o valor obtido pelo teclado seja armazenado na variável sempre como um `str`. Caso se deseje tratar os dados de entrada com sendo de outro tipo, podemos sempre utilizar os comandos de conversão de tipos, apresentados na Seção 2.3.1

■ **Exemplo 2.14** Considere um programa que obtém dois valores de um(a) usuário(a) via teclado e informa os tipos de cada um deles. O código-fonte desse programa é apresentado a seguir:

```
x = input("Digite um valor qualquer:")
y = input("Agora, digite um outro valor:")
print(type(x))
print(type(y))
```

As linhas 1 e 2 realizam a exibição de mensagens de solicitação de dados e o devido armazenamento dos dados fornecidos via teclado nas variáveis indicadas em cada instrução `input()`.

Ao executar o programa, será possível observar que, mesmo o(a) usuário(a) fornecendo dados numéricos pelo teclado, os tipos das variáveis `x` e `y` serão, invariavelmente, `str`. ■

Comando de Saída

Um programa produz resultados que são de interesse para alguém. Assim, é necessário que a linguagem de programação utilizada disponibilize instruções para fornecer a saída (exibição) dos resultados produzidos.

Uma saída de dados pode ocorrer de diversas formas: por impressora, por caixa de som ou por exibição de textos e gráficos em monitor. Neste material, vamos considerar a opção de produzir a exibição de textos em monitor. Para tal, o Python disponibiliza a instrução `print()`.
No Python, a sintaxe da instrução `print()` é:

```
print(saida)
```

onde **`saida`** são as informações referentes aos resultados e textos que desejamos que o programa exiba em tela.

Há, portanto, diferentes maneiras de utilizar a instrução `print()`. A Tabela 2.5 lista as diferentes possibilidades de utilização desse comando.

Exibição	Informação entre parênteses	Exemplo
Texto simples	texto entre aspas simples	`print('Olá!')`
Texto simples	texto entre aspas duplas	`print("Beleza?")`
Valor de uma variável	nome da variável	`print(soma)`
Resultado de um cálculo	expressão a ser calculada	`print(10 + 5)`
Textos e valores de variáveis	texto e variáveis, separados por vírgulas	`print("Resultado:", soma)`
Textos e resultados de expressões	texto e expressão, separados por vírgulas	`print('Média:', (x+y)/2)`

Tabela 2.5: Diferentes maneiras de utilizar a instrução `print()`.

Obs O Python oferece uma alternativa para organizar melhor a montagem de uma saída (exibição em tela), de maneira a juntar textos e outros objetos. A função que permite fazer essa organização da saída é a `format()`.

Função `format()`

Como foi visto na Figura 2.5, uma saída pode combinar texto e objetos de outros tipos. Na maioria dos casos, os objetos de outros tipos não são conhecidos a priori pelo(a) programador(a).

■ **Exemplo 2.15** Tomemos uma saída que exibe a mensagem de resultado:

```
Os números obtidos foram 3 e 4, e a multiplicação entre eles resulta em 12.
```

■

Utilizando a instrução `print()` para exibir a saída do Exemplo 2.15, precisaríamos escrever um código como segue:

```
print("Os números obtidos foram ", num1, "e ", num2, "e a multiplicaçã
    o entre eles resulta em ", num1 * num2)
```

Podemos perceber que a mensagem que queremos exibir foi "quebrada" dentro da instrução `print()`, para alternar entre momentos de exibição de texto e momentos de exibição de objetos de outros tipos.

Uma alternativa a essa "quebra" é escrever o texto completo, deixando **lacunas** nos locais onde desejamos encaixar os valores a serem obtidos durante a execução do programa. As lacunas serão marcadas no texto com um par de chaves {}. O texto do nosso exemplo ficaria escrito dentro da instrução `print()` assim:

```
"Os números obtidos foram {} e {}, e a multiplicação entre eles resulta em
{}."
```

Do ponto de vista de quem está escrevendo o programa, essa forma de escrever a saída é uma alternativa mais intuitiva e facilita a visualização do que se deseja exibir.

O preenchimento das lacunas será feito por meio da função `format()`, cuja sintaxe é apresentada a seguir:

```
texto.format(objetos)
```

onde `texto` é a variável que armazena o texto completo, marcado com as lacunas, e `objetos` pode ser um objeto ou mais de um objeto. No caso de haver mais de um objeto, vamos separá-los por vírgulas.

Para exemplificar, o código para produzir a saída do Exemplo 2.15 é:

```
saida = "Os números obtidos foram {} e {}, e a multiplicação entre
    eles resulta em {}."
print(saida.format(num1, num2, num1*num2))
```

Exercício 2.7 Sem utilizar a função `format()`, faça um programa em Python que obtenha do(a) usuário(a) dois números e exiba um relatório com as seguintes informações:

- Uma mensagem de boas-vindas ao sistema;
- Os valores fornecidos pelo(a) usuário(a);
- O resultado da multiplicação entre os dois valores fornecidos;
- Uma mensagem de agradecimento por usar o sistema.

Solução. Um programa para resolver o problema proposto no Exercício 2.7 é apresentado abaixo. Note que, para realizar a operação de multiplicação, os objetos armazenados em `num1` e `num2` precisam ter seus tipos convertidos de `str` para `int` ou `float` antes. Neste exercício, escolhemos usar o tipo `float`, para prevenir eventuais erros, caso o(a) usuário(a) digite um número com casas decimais.

```
print("Bem-vindo(a) ao nosso fabuloso sistema!")
num1 = input("Por gentileza, forneça um número:")
num2 = input("Agora, por obséquio, forneça um outro número:")
print("Os números fornecidos foram:", num1, " e ", num2)
print("e o resultado da multiplicação entre esses dois números é: ",
   float(num1) * float(num2))
print("Muito obrigada por usar nosso sistema!")
```

■

Exercício 2.8 Agora, para comparar, refaça o programa do Exercício 2.7, utilizando a função `format()`.

Solução. O código pedido pode ser visto logo abaixo:

```
print("Bem-vindo(a) ao nosso fabuloso sistema!")
num1 = input("Por gentileza, forneça um número:")
num2 = input("Agora, por obséquio, forneça um outro número:")
saida = "Os números fornecidos foram: {} e {}, e o resultado da
   multiplicação entre esses dois números é: {}"
print(saida.format(num1, num2, float(num1)*float(num2)))
print("Muito obrigada por usar nosso sistema!")
```

Alternativamente, podemos armazenar o resultado da multiplicação em uma variável, para deixar os parâmetros da função `format()` mais enxutos e melhores de se ler:

```
print("Bem-vindo(a) ao nosso fabuloso sistema!")
num1 = input("Por gentileza, forneça um número:")
num2 = input("Agora, por obséquio, forneça um outro número:")
mult = float(num1) * float(num2)
saida = "Os números fornecidos foram: {} e {}, e o resultado da
   multiplicação entre esses dois números é: {}"
print(saida.format(num1, num2, mult))
print("Muito obrigada por usar nosso sistema!")
```

■

Índices no comando `format()`

Podemos considerar variações na utilização dos parâmetros da função `format()` para o preenchimento das lacunas de um texto. Tais variações são possíveis por meio da utilização de **índices** dentro das chaves. Os índices, neste caso, são associados aos objetos na lista de parâmetros da função `format`. A Figura 2.8 ilustra a associação dos índices aos parâmetros da função.

Figura 2.8: Índices dos parâmetros da função `format()`.

■ **Exemplo 2.16** O código abaixo mostra como utilizar índices no comando `format()`:

```
quantidade = 4
codigoItem = 1021
preco = 15
compra = "Solicito {0} unidades do produto com código {1} pela promoçã
    o de {2:.2f} Reais."
print(compra.format(quantidade, codigoItem, preco))
```

A saída produzida por esse programa é:

```
Solicito 4 unidades do produto com código 1021 pela promoção de 15.00 Reais.
```

■

Obs Observe, no código do Exemplo 2.16, que a variável de índice 2 sofreu uma formatação adicional. A notação `:.2f` serve para exibir o valor como um número real com duas casas decimais.

Os índices são muito úteis quando desejamos repetir um dado armazenado em uma variável dentro do texto.

■ **Exemplo 2.17** O código a seguir mostra a utilidade dos índices quando se deseja repetir o valor armazenado em uma variável dentro das lacunas de um texto.

```
nome = "Léo"
idade = 16
texto = "Dentre os meus amigos, o {0} é o mais responsável. {0} já tem
    {1} anos!"
print(texto.format(nome, idade))
```

A saída produzida por esse programa é:

```
Dentre os meus amigos, o Léo é o mais responsável.  Léo já tem 16 anos!
```

■

2.5 – Organização e manipulação de dados

Na Seção 2.3, conhecemos alguns tipos de dados novos, que permitem o armazenamento de sequências e conjuntos de valores, além de uma estrutura que organiza dados por um par chave:valor. Nesta seção, vamos conhecer maiores detalhes sobre essas estruturas de armazenamento disponibilizadas na linguagem Python, que facilitam bastante o trabalho do(a) programador(a).

Como cada um desses tipos a serem discutidos aqui já foram apresentados na Seção 2.3, vamos dedicar esta seção para descrever maiores detalhes sobre as **operações** definidas para cada um dos tipos.

O Python assume duas sintaxes específicas para realizar as operações que discutiremos nesta seção. Essas diferenças se devem ao fato de que algumas operações são funções aplicáveis a objetos de mais de um tipo, enquanto que outras são definidas especificamente para o tipo de um determinado objeto.

Para operações disponibilizadas em funções que podem ser utilizadas com objetos de diferentes tipos, a sintaxe adotada é:

```
funcao(parametros)
```

onde `funcao` é o nome da função disponibilizada pelo Python para efetuar uma determinada operação, e `parametros` será um objeto ou mais de um, passado(s) como parâmetro(s) para a função.

Obs Essa sintaxe já é conhecida por nós. Ela é a sintaxe aplicada a funções que já utilizamos, como a `print()`, a `type()` e as funções para conversão explícita de tipo (`int()`, `str()` entre outras).

Para operações específicas de um dado objeto, a sintaxe leva a forma geral:

```
nomeDoObjeto.função(parâmetros)
```

A sintaxe para essa notação indica que desejamos chamar uma determinada função a partir de um dado objeto, separando o nome do objeto da função utilizando um ponto.

Obs Essa outra notação pode até parecer novidade, mas nós também já a utilizamos. No caso, foi quando estudamos a função `format()`, apresentada na Seção 2.4

2.5.1 – Sequências

Os dados organizados como sequências de valores podem ser armazenados como listas (tipo `list`), ou como um intervalo de valores (tipo `range`).

Listas (Objetos do tipo `list`)

Listas são capazes de armazenar múltiplos objetos em uma única variável. Cada objeto tem um índice, que indica sua posição dentro da lista. O índice 0 refere-se ao primeiro elemento de uma lista. Um objeto do tipo `list` é representado por colchetes.

Nesta subseção, vamos abordar as operações que podemos efetuar sobre listas.

Função `len()`

A função `len()` informa a quantidade de objetos de uma lista.

A sintaxe para esta função é:

```
len(nomeDaLista)
```

■ **Exemplo 2.18** Considere o programa abaixo:

```
lista = [8, "Maria", True, -5.6]
print(len(lista))
```

A saída desse programa será:

```
4
```

Função `list()`

Vimos anteriormente que a função `list()` pode ser utilizada para alterar o tipo de um objeto para `list`. Essa função serve também para criar uma nova lista. A sintaxe para criar uma nova lista com o comando `list()` é:

```
variavel = list((objetos))
```

onde `objetos` pode ser um ou mais objetos. Havendo mais de um, eles devem ser separados por vírgulas.

■ **Exemplo 2.19** O código abaixo gera uma lista com três elementos e a exibe na tela:

```
    minhaLista = list((45, "Olá", 6.21))
    print(minhaLista)
```

A saída produzida pelo programa é:

```
[45, 'Olá', 6.21]
```

■

Obs Observe o duplo par de parênteses na criação de uma lista usando o comando `list()`.

Acessando objetos de uma lista

Utilizamos os índices dos objetos para acessá-los dentro de uma lista. A sintaxe para acesso a um objeto é:

```
variavel[indice]
```

onde `variavel` é a variável que armazena a lista, e `indice` é a posição do objeto a ser acessado.

■ **Exemplo 2.20** O código a seguir cria uma lista e exibe o seu primeiro objeto:

```
minhaLista = list(("Oitenta", 7.5, False))
print(minhaLista[0])
```

A saída produzida por esse programa é:

```
'Oitenta'
```

■

Podemos especificar um intervalo de índices para acessar uma parte dos objetos contidos em uma lista. A sintaxe para especificar um intervalo de índices pode ser vista nos Exemplos 2.21, 2.22 e 2.23.

■ **Exemplo 2.21** Dada a lista:

```
minhaLista = ["Taíba", "Iguape", "Pecém", "Cumbuco", "Lagoinha", "
   Bitupitá", "Paracuru", "Mundaú"]
```

O código para acessar e exibir do terceiro ao quinto objeto de `minhaLista` é:

```
print(minhaLista[2:5])
```

A saída produzida por esse programa será:

```
['Pecém', 'Cumbuco', 'Lagoinha']
```

■

■ **Exemplo 2.22** Dada a lista do Exemplo 2.21, o código para acessar e exibir os quatro primeiros objetos de `minhaLista` é:

```
print(minhaLista[:4])
```

A saída produzida por esse programa será:

```
['Taíba', 'Iguape', 'Pecém', 'Cumbuco']
```

■

■ **Exemplo 2.23** Dada a lista do Exemplo 2.21, o código para acessar e exibir os objetos, do terceiro até o fim de `minhaLista` é:

```
print(minhaLista[2:])
```

A saída produzida por esse programa será:

```
['Pecém', 'Cumbuco', 'Lagoinha', 'Bitupitá', 'Paracuru', 'Mundaú']
```

■

Exercício 2.9 Dada a lista:

```
x = ['azul', 'verde', 'vermelho', 'roxo', 'lilás', 'branco', 'rosa',
    'preto', 'amarelo', 'marrom']
```

Utilize a função `len()` para exibir o último objeto da lista.

Solução. Vimos que, como os índices de uma lista começam sempre com 0, o último elemento de uma lista sempre terá como índice o valor da quantidade de elementos, menos 1. Assim, o código para atender ao que foi pedido no Exercício 2.9 é:

```
x = ['azul', 'verde', 'vermelho', 'roxo', 'lilás', 'branco', 'rosa', '
   preto', 'amarelo', 'marrom']
print(x[len(x)-1])
```

■

A solução que acabamos de ver no Exercício 2.9 envolve uma estratégia que funciona em inúmeras linguagens de programação. O Python oferece, ainda, uma opção adicional para resolver o problema de se acessar o último objeto de uma lista, bem como para o penúltimo e os objetos anteriores a ele. No Python, é possível acessar os últimos objetos de uma lista utilizando **índices negativos**: o último objeto pode ser acessado pelo índice -1, o penúltimo pelo índice -2, e assim por diante.

Exercício 2.10 Dada a lista:

```
x = ['azul', 'verde', 'vermelho', 'roxo', 'lilás', 'branco', 'rosa',
    'preto', 'amarelo', 'marrom']
```

Utilize índices negativos para exibir:

(a) o último objeto da lista.
(b) o penúltimo objeto da lista.
(c) o antepenúltimo objeto da lista.

Solução. O código completo para resolver os itens (a) a (c) do Exercício 2.10 pode ser visto abaixo:

```
x = ['azul', 'verde', 'vermelho', 'roxo', 'lilás', 'branco', 'rosa', '
    preto', 'amarelo', 'marrom']
saida = "O último objeto da lista é {}, o penúltimo é {} e o antepenú
    ltimo é {}."
print(saida.format(x[-1], x[-2], x[-3])
```

■

Verificando a existência de um objeto em uma lista

O Python possibilita a checagem se um determinado objeto está ou não presente em uma lista. Para isso, utilizamos uma estrutura de decisão e a palavra reservada `in`. O Exemplo 2.24 mostra como verificar a existência de um objeto em uma lista:

■ **Exemplo 2.24** O código abaixo cria uma lista e checa se o objeto 'pardal' se encontra nela:

```
p = ['beija-flor', 'galo de campina', 'bem-te-vi', 'pardal', 'sanhaço-
    azul', 'soldadinho-do-Araripe', 'canário']

if 'pardal' in p:
    print('Pardal encontrado')
else:
    print('Pardal não encontrado!')
```

■

Obs As palavras `if` e `else` do código apresentado no Exemplo 2.24 são palavras reservadas para **estruturas de decisão**, que estudaremos na Seção 2.7.

Modificando objetos de uma lista

No Python, é possível modificar objetos contidos em uma lista. Para isso, usamos uma combinação de comando de acesso (uso dos colchetes) com o comando de atribuição. A sintaxe do Python permite modificar um único objeto de uma lista, bem como um intervalo de objetos de uma só vez. Os diferentes casos serão abordados por meio dos Exemplos 2.25 a 2.27.

■ **Exemplo 2.25** O código apresentado neste exemplo cria uma lista, modifica o terceiro objeto dela e exibe como a lista ficou ao final da operação de alteração:

```
x = ['Ed', 'Sá', 'Dé', 'Jo']
x[2] = 'Ju'  #modifica o terceiro objeto da lista para 'Ju'
print(x)     #exibe todo o conteúdo da lista x
```

A saída produzida por esse programa será:

```
['Ed', 'Sá', 'Ju', 'Jo']
```

■

Obs Lembre-se: o i-ésimo objeto de uma lista tem sempre índice igual a $i - 1$. Assim, no Exemplo 2.25, o terceiro objeto da lista tem índice 2.

O Python permite trocar um objeto de uma lista por mais de um. O Exemplo 2.26 mostra um progrma que troca o segundo objeto de uma lista por outros três novos objetos.

■ **Exemplo 2.26** O código apresentado neste exemplo cria uma lista e troca o segundo objeto por outros três, exibindo como a lista ficou ao final da operação de alteração:

```
x = ['Ed', 'Sá', 'Dé', 'Jo']
x[1:2] = ['Ju', 'Du', 'Di']  #modifica o segundo objeto da lista para
   outros três objetos: 'Ju', 'Du' e 'Di'
print(x)  #exibe todo o conteúdo da lista x
```

A saída produzida por esse programa será:

```
['Ed', 'Ju', 'Du', 'Di', 'Dé', 'Jo']
```

■

Não há restrição em relação à quantidade de objetos envolvidos em uma troca dentro de uma lista. O Exemplo 2.27 mostra um programa que troca o segundo e o terceiro objetos de uma lista por um só objeto.

■ **Exemplo 2.27** O código apresentado neste exemplo cria uma lista e troca o segundo e o terceiro objetos por um só objeto, exibindo como a lista ficou ao final da operação de alteração:

```
x = ['Ed', 'Sá', 'Dé', 'Jo']
x[1:3] = ['Ju']  #modifica o segundo e o terceiro objetos da lista
   para o objeto 'Ju'
print(x)  #exibe todo o conteúdo da lista x
```

A saída produzida por esse programa será:

```
['Ed', 'Ju', 'Jo']
```

■

Inserindo objetos em uma lista

O Python disponibiliza duas funções para inserirmos objetos em uma lista:

- Comando `insert()`
- Comando `append()`

O comando `insert()` permite a inserção de um objeto em uma lista, especificando a posição onde esse novo objeto deverá ficar.

A sintaxe para este comando é:

```
variavel.insert(indice, objeto)
```

onde `variavel` é a variável que armazena a lista, `indice` é a posição onde se deseja colocar o objeto e `objeto` é o objeto a ser inserido na lista.

■ **Exemplo 2.28** O código a seguir cria uma lista, adiciona o objeto 'Crato' na quarta posição da lista. O conteúdo total da lista após a operação de inserção é exibido na tela:

```
x = ['Caucaia', 'Camocim', 'Maracanaú', 'Lavras da Mangabeira', '
    Fortaleza', 'Itarema', 'Crateús']
x.insert(3, 'Crato')
print(x)
```

A saída produzida por esse programa será:

```
['Caucaia', 'Camocim', 'Maracanaú', 'Crato', 'Lavras da Mangabeira',
'Fortaleza', 'Itarema', 'Crateús']
```

■

O comando `append()`, por sua vez, insere um novo objeto na lista, mas somente no final dela.

A sintaxe para este comando é:

```
variavel.append(objeto)
```

onde `variavel` é a variável que armazena a lista e `objeto` é o objeto a ser inserido na lista.

■ **Exemplo 2.29** O código a seguir cria uma lista e adiciona o objeto 'Tauá' ao final da lista. O conteúdo total da lista após a operação de inserção é exibido na tela:

```
x = ['Caucaia', 'Camocim', 'Maracanaú', 'Crato', 'Lavras da
    Mangabeira', 'Fortaleza', 'Itarema', 'Crateús']
x.append('Tauá')
print(x)
```

A saída produzida por esse programa será:

```
['Caucaia', 'Camocim', 'Maracanaú', 'Crato', 'Lavras da Mangabeira',
'Fortaleza', 'Itarema', 'Crateús', 'Tauá']
```

■

Juntando listas

Dadas duas listas, `x` e `y`, a sintaxe para unir a segunda lista à primeira é:

```
x.extend(y)
```

indicando que os objetos de `y` serão todos adicionados ao final da lista `x`.

■ **Exemplo 2.30** O código a seguir cria duas listas e adiciona todo o conteúdo da segunda lista na primeira lista. Comandos de exibição foram incluídos para facilitar a compreensão do código:

```
primeira = ['eu', 'tu', 'ele', 'ela']
segunda =['nós', 'vós', 'eles', 'elas']
print("Primeira lista:  ", primeira)
print("Segunda lista:  ", segunda)
primeira.extend(segunda)
print("Primeira lista após o comando extend: ", primeira)
```

A saída produzida por esse programa será:

```
Primeira lista:  ['eu', 'tu', 'ele', 'ela']
Segunda lista:  ['nós', 'vós', 'eles', 'elas']
Primeira lista após o comando extend:  ['eu', 'tu', 'ele', 'ela', 'nós',
'vós', 'eles', 'elas']
```

■

Alternativamente, podemos usar o operador +, que concatena, ou une, duas listas, na ordem que você escolher. O resultado da operação é armazenado em uma terceira lista.
A sintaxe para unir duas listas usando o operador `+` é:

```
lista3 = lista1 + lista2
```

A ordem das listas 1 e 2 é de escolha do(a) programador(a).

■ **Exemplo 2.31** O código a seguir cria duas listas de objetos do tipo `str`. Em seguida, uma terceira lista é criada, contendo os objetos da segunda lista seguidos dos elementos da primeira lista. Ao final, o conteúdo da terceira lista é exibido em tela:

```
primeira = ['eu', 'gosto']
segunda =['de', 'programar']
terceira = segunda + primeira
print(terceira)
```

A saída produzida por esse programa será:

```
['de', 'programar', 'eu', 'gosto']
```

■

Removendo objetos de uma lista

O Python disponibiliza várias maneiras para se remover objetos de uma lista. A depender do que desejamos como resultado, podemos escolher entre:

- Utilizar a função `remove()`;
- Utilizar a função `pop()`;
- Utilizar a palavra reservada `del`;
- Utilizar a função `clear()`.

O comando **`remove()`** retira um determinado objeto de uma lista.

A sintaxe para este comando é:

```
variavel.remove(objeto)
```

onde `variavel` é a variável que armazena a lista e `objeto` é o objeto a ser removido da lista.

■ **Exemplo 2.32** O código a seguir cria uma lista e remove o objeto 'sal'da lista. O conteúdo total da lista após a operação de remoção é exibido na tela:

```
lista = ['açúcar', 'farinha', 'sal', 'rapadura',  'macarrão', 'tomate'
   ]
lista.remove('sal')
print(lista)
```

A saída produzida por esse programa será:

```
['açúcar', 'farinha', 'rapadura', 'macarrão', 'tomate']
```

■

Agora, querendo remover um objeto informando o seu índice, o comando a ser utilizado é o **`pop()`**. A sintaxe para esse comando é:

```
variavel.pop(indice)
```

onde `variavel` é a variável que armazena a lista e `indice` é a posição do objeto a ser removido da lista.

■ **Exemplo 2.33** O código a seguir cria uma lista e remove o quinto objeto da lista. O conteúdo total da lista após a operação de remoção é exibido na tela:

```
lista = ['açúcar', 'farinha', 'sal', 'rapadura',  'macarrão', 'tomate'
   ]
lista.pop(4)
print(lista)
```

A saída produzida por esse programa será:

```
['açúcar', 'farinha', 'sal', 'rapadura', 'tomate']
```

■

Uma outra forma de remover um objeto de uma lista por meio de seu índice é utilizar a palavra reservada `del`. A sintaxe para sua utilização pode ser vista abaixo:

```
del variavel[indice]
```

onde `variavel` é a variável que armazena a lista e `indice` é a posição do objeto a ser removido da lista.

■ **Exemplo 2.34** O código a seguir cria uma lista e remove o segundo objeto da lista. O conteúdo total da lista após a operação de remoção é exibido na tela:

```
lista = ['açúcar', 'farinha', 'sal', 'rapadura', 'macarrão', 'tomate'
    ]
del lista[1]
print(lista)
```

A saída produzida por esse programa será:

```
['açúcar', 'sal', 'rapadura', 'macarrão', 'tomate']
```

■

O comando **`clear()`** retira todos os objetos de uma lista.

A sintaxe para esse comando é:

```
x.clear()
```

onde `x` é a variável que armazena a lista.

■ **Exemplo 2.35** O código a seguir cria uma lista e remove todos os seus objetos. Após a remoção, um comando de exibição da lista em tela mostra que a lista ficou vazia:

```
lista = ['açúcar', 'farinha', 'sal', 'rapadura', 'macarrão', 'tomate'
    ]
lista.clear()
print(lista)
```

A saída produzida por esse programa será:

```
[]
```

■

Ordenando uma lista

O Python disponibiliza opções para ordenar uma lista de maneira automática. Com elas, podemos ordenar listas tanto em ordem direta como inversa. Para esse tipo de ordenação, podemos ter, por exemplo, uma lista de objetos númericos e ordená-los em ordem crescente (direta) ou decrescente (inversa).

Além disso, é possível reverter a ordem dos objetos de uma lista, sem fazer ordenação dos objetos. Esta operação apenas troca os objetos de lugar, de modo que o primeiro se torne o último, o segundo se torne o penúltimo e assim por diante.

A depender do que desejamos como resultado, podemos utilizar:

- A função `sort()`, para ordenação:
  - Crescente ou decrescente para números;
  - Alfabética direta ou inversa para textos.
- A função `reverse()`, para reverter uma lista, como se fosse um "espelhamento" dela.

O comando **`sort()`** realiza a ordenação direta ou inversa dos objetos de uma lista, sejam eles do tipo `str` ou quaisquer tipos numéricos.

A sintaxe para **ordenação direta** (para números ou textos) é:

```
x.sort()
```

onde `x` é a variável que armazena a lista.

■ **Exemplo 2.36** O código a seguir cria uma lista de objetos do tipo `str`. No momento da criação da lista, os objetos estão fora de ordem alfabética. Em seguida, é aplicado o comando `sort()` sobre a lista e o resultado final é exibido na tela:

```
lista = ['açúcar', 'farinha', 'sal', 'rapadura',  'macarrão', 'tomate'
   ]
lista.sort()
print(lista)
```

A saída produzida por esse programa será:

```
['açúcar', 'farinha', 'macarrão', 'rapadura', 'sal', 'tomate']
```

■

A sintaxe para **ordenação inversa** (para números ou textos) é:

```
x.sort(reverse=True)
```

onde `x` é a variável que armazena a lista.

■ **Exemplo 2.37** O código a seguir cria uma lista de objetos do tipo `int`. No momento da criação da lista, os objetos estão fora de ordem numérica. Em seguida, é aplicado o comando `sort()`,

com o parâmetro `reverse = True` sobre a lista e o resultado final é exibido na tela:

```
lista = [7, 5, -2, 9, 0, -4]
lista.sort(reverse = True)
print(lista)
```

A saída produzida por esse programa será a lista com seus objetos ordenados em ordem decrescente:

```
[9, 7, 5, 0, -2, -4]
```

■

Obs

É importante lembrar que o Python faz distinção entre maiúsculas e minúsculas. Textos iniciando com maiúsculas serão ordenados antes dos que iniciam com minúsculas. Assim, se a lista contiver textos iniciando em maiúsculas e outros em minúsculas, o resultado produzido pelo comando `sort()` pode não corresponder ao desejado.

Há, porém, uma forma de contornar esse problema. Basta passar o parâmetro `(key = str.lower)` para o comando `sort()`.

■ **Exemplo 2.38** Observe a lista abaixo:

```
lista = ['caderno', 'Mochila', 'livro', 'Caneta']
```

Se aplicássemos o comando `sort()` à lista, teríamos, como resultado:

```
['Caneta', 'Mochila', 'caderno', 'livro']
```

Aplicando o comando `sort(key = str.lower)` à lista, teríamos, como resultado:

```
['caderno', 'Caneta', 'livro', 'Mochila']
```

■

O comando `reverse()` inverte as posições dos objetos dentro de uma lista, de modo que o primeiro se torne o último, o segundo o penúltimo e assim por diante.

A sintaxe para **ordenação direta** (para números ou textos) é:

```
x.reverse()
```

onde `x` é a variável que armazena a lista.

■ **Exemplo 2.39** O código a seguir cria uma lista de objetos do tipo `float`. Para este exemplo, a ordem dos objetos não é importante, pois não vamos realizar ordenação crescente nem decrescente sobre os objetos. Em seguida, é aplicado o comando `reverse()` sobre a lista e o resultado final é exibido na tela:

```
lista = [3.5, 1.2, 3.2, 0.3, 0.1, 3.1, 3.4]
lista.reverse()
print(lista)
```

A saída produzida por esse programa será:

```
[3.4, 3.1, 0.1, 0.3, 3.2, 1.2, 3.5]
```

■

Fazendo uma cópia de uma lista

Se desejarmos criar uma nova lista que contenha os mesmos objetos de uma lista existente, podemos utilizar uma das seguintes funções do Python:

- `copy()`;
- `list()`.

O comando **`copy()`** insere todos os objetos de uma dada lista dentro de uma nova lista criada. A sintaxe para esse comando é:

```
novaLista = listaExistente.copy()
```

■ **Exemplo 2.40** O código a seguir cria uma lista de objetos do tipo `str`. Em seguida, uma nova lista é criada, contendo uma cópia do conteúdo da primeira lista criada. Por fim, o programa exibe o conteúdo da segunda lista criada:

```
listaOriginal = ['Márcia','Cristina','Ana']
copiaDaLista = listaOriginal.copy()
print(copiaDaLista)
```

A saída produzida por esse programa será, naturalmente:

```
['Márcia', 'Cristina', 'Ana']
```

■

Alternativamente, podemos utilizar o comando de criação de uma lista, o comando **`list()`**. A sintaxe para esse comando é:

```
novaLista = list(listaExistente)
```

■ **Exemplo 2.41** O código a seguir cria uma lista de objetos do tipo `int`. Em seguida, uma nova lista é criada, contendo uma cópia do conteúdo da primeira lista criada. Por fim, o programa exibe o conteúdo da segunda lista criada:

```
listaOriginal = [1, 6, 3, 1]
copiaDaLista = list(listaOriginal)
print(copiaDaLista)
```

A saída produzida por esse programa será, naturalmente:

```
[1, 6, 3, 1]
```

■

Contando ocorrências de um objeto em uma lista

O Python disponibiliza uma função que nos permite verificar, rapidamente, quantas vezes um dado objeto ocorre dentro de uma lista.
A sintaxe para esse comando é:

```
x.count(objeto)
```

onde `x` é a variável que armazena a lista, e `objeto` é o objeto a ser procurado e contado dentro da lista.

■ **Exemplo 2.42** O código a seguir cria uma lista de objetos do tipo `str`. Em seguida, queremos contar quantas vezes o objeto 'a' ocorre na lista. Por fim, o programa exibe o resultado da contagem:

```
lista = ['b', 'a', 'n', 'a', 'n', 'a']
quantidade = lista.count('a')
print(quantidade)
```

A saída produzida por esse programa será:

```
3
```

■

Obtendo o índice da primeira ocorrência de um objeto em uma lista

O Python disponibiliza uma função que nos permite verificar, rapidamente, a posição da primeira ocorrência de um dado objeto dentro de uma lista.
A sintaxe para esse comando é:

```
x.index(objeto)
```

onde `x` é a variável que armazena a lista, e `objeto` é o objeto a ser procurado dentro da lista.

■ **Exemplo 2.43** O código a seguir cria uma lista de objetos do tipo `str`. Em seguida, queremos saber em que posição o objeto 'a' ocorre pela primeira vez na lista. Por fim, o programa exibe o resultado da busca:

```
lista = ['b', 'a', 'n', 'a', 'n', 'a']
posicao = lista.index('a')
print(posicao)
```

A saída produzida por esse programa será:

```
1
```

■

Intervalos (Objetos do tipo `range`)

Para armazenar uma sequência numérica ordenada que define um **intervalo** de valores, uma boa opção é usar o tipo `range`.

Esse objeto é bastante simples, porém muito útil, especialmente para facilitar a implementação de laços de repetição.

As operações que veremos aqui se resumem nas diferentes opções de criação de um objeto do tipo `range`.

Criando um intervalo

Um intervalo é definido por:

- Um valor inicial;
- Um valor final;
- Um valor de *passo*.

E para criar um intervalo, utilizamos a função `range()`.

A sintaxe para essa função é:

```
range(inicio, fim, passo)
```

onde `inicio` é o valor inicial do intervalo. Este valor pertence ao intervalo; `fim` é o valor numérico para marcar o final do intervalo. Este valor não pertence ao intervalo. Por fim, `passo` é a diferença entre um valor do intervalo e seu antecessor e seu sucessor.

■ **Exemplo 2.44** O código a seguir cria uma sequência de números para criar um intervalo de 5 até 10. A diferença entre um número e seu antecessor e sucessor dentro do intervalo é de uma unidade. Ao final, o programa exibe o conteúdo do intervalo:

```
intervalo = range(5, 11, 1)
print(intervalo)
```

A saída produzida por esse programa será:

```
range(5, 11)
```

A informação exibida em tela não ajudou muito a visualizar o conteúdo desse intervalo. Vamos experimentar o seguinte código:

```
intervalo = range(5, 11, 1)
for i in intervalo:
    print(i)
```

Com a adição desse novo comando, a saída do programa produziu a exibição de cada número pertencente ao intervalo:

```
5
6
7
8
9
10
```

Observe que o número 11 não pertence ao intervalo.

Neste novo código, vemos uma palavra reservada que já tinha sido apresentada nesta seção: a palavra reservada `in`, que utilizamos anteriormente para verificar se um determinado objeto se encontrava em uma lista. Aqui, neste exemplo, ela é utilizada para obter cada um dos números dentro do intervalo e exibi-los em tela.
A palavra reservada `for`, por sua vez, é novidade: ela é a sintaxe para um dos laços de repetição disponibilizados na linguagem Python. Estudaremos laços de repetição em Python na Seção 2.8. ■

O valor do passo pode ser negativo. Isso resultará na criação de um intervalo com números em ordem decrescente.

■ **Exemplo 2.45** O código a seguir cria uma sequência de números em ordem decrescente para criar um intervalo de 5 até 1. A diferença entre um número e seu antecessor e sucessor dentro do intervalo é de uma unidade. Ao final, o programa exibe o conteúdo do intervalo:

```
intervalo = range(5, 0, -1)
for i in intervalo:
    print(i)
```

A saída produzida por esse programa será:

```
5
4
3
2
1
```

Observe que o número 0 não pertence ao intervalo.

■

■ **Exemplo 2.46** O código a seguir cria uma sequência de números em ordem decrescente para criar um intervalo de 30 até 21. A diferença entre um número e seu antecessor e sucessor dentro do intervalo é de três unidades. Ao final, o programa exibe o conteúdo do intervalo:

```
intervalo = range(30, 20, -3)
for i in intervalo:
    print(i)
```

A saída produzida por esse programa será:

```
30
27
24
21
```

Observe que o número 20 não pertence ao intervalo.

■

Ao criar um intervalo com a função `range()`, podemos suprimir a informação de *passo* quando seu valor for igual a 1.

■ **Exemplo 2.47** Podemos, então, simplificar a criação do intervalo do Exemplo 2.44, escrevendo:

```
intervalo = range(5, 11)
```

O comando acima produzirá o mesmo intervalo gerado no Exemplo 2.44. ■

2.5.2 – Mapeamento (Dicionários)

O Python provê uma estrutura de armazenamento que organiza seus objetos utilizando uma associação **chave:valor**. São os objetos do tipo `dict`, que chamamos de **dicionário**.

A maneira como os dicionários são organizados nos oferece uma alternativa interessante para acessar objetos armazenados dentro dele. As chaves funcionam como índices, que nos ajudam a encontrar valores dentro de um dicionário. A vantagem sobre um índice de uma lista, por exemplo, é que a chave pode ter o nome ou valor que desejarmos, dando mais flexibilidade à sua utilização.

Acessando objetos de um dicionário

As chaves de um dicionário exercem uma função semelhante à dos índices de uma lista. Assim, a sintaxe para acesso a objetos de um dicionário é semelhante. A diferença é que, ao invés de escrever um número de índice, vamos escrever nos colchetes a chave para acesso ao objeto. O Exemplo 2.48 mostra uma operação de acesso a um objeto armazenado em um dicionário.

■ **Exemplo 2.48** O código abaixo cria um dicionário e imprime o objeto referente à informação de `"cidade"`:

```
info = {"país": "Brasil", "estado":"Ceará", "cidade":"Fortaleza", "ano
    ":2021}
print(info["cidade"])
```

A saída produzida por esse programa é:

```
Fortaleza
```

■

Uma outra maneira de acessar um objeto de um dicionário é por meio da utilização do comando **`get()`**.

A sintaxe para esse comando é:

```
variavel.get("chave")
```

onde `variavel` é o nome da variável que armazena o dicionário e `"chave"` é o localizador do objeto que se deseja buscar.

■ **Exemplo 2.49** Vamos reescrever o código do Exemplo 2.48, utilizando o comando `get()` para realizar a mesma tarefa:

```
info = {"país": "Brasil", "estado":"Ceará", "cidade":"Fortaleza", "ano
    ":2021}
print(info.get("cidade"))
```

A saída produzida por esse programa é:

```
Fortaleza
```

■

Obtendo uma lista de chaves de um dicionário

O Python permite armazenar, em uma lista, as chaves de um dado dicionário. O comando que permite fazer isso é **`keys()`** e a sintaxe para ele é:

```
varLista = varDict.keys()
```

onde `varLista` é a variável que armazenará a lista contendo as chaves, e `varDict` é a variável que está armazenando o dicionário de onde serão obtidas as chaves.

■ **Exemplo 2.50** O código a seguir cria um dicionário e uma lista contendo as chaves do dicionário criado. Ao final, o programa exibe o conteúdo da lista em tela:

```
info = {"país": "Brasil", "estado":"Ceará", "cidade":"Fortaleza", "ano
    ":2021}
chaves = info.keys()
print(chaves)
```

A saída produzida por esse programa é:

```
dict_keys(['país', 'estado', 'cidade', 'ano'])
```

■

Obtendo uma lista de valores de um dicionário

O Python permite armazenar, em uma lista, os valores de um dado dicionário. O comando que permite fazer isso é **`values()`** e a sintaxe para ele é:

```
varLista = varDict.values()
```

onde `varLista` é a variável que armazenará a lista contendo os valores, e `varDict` é a variável que está armazenando o dicionário de onde serão obtidos os valores.

■ **Exemplo 2.51** O código a seguir cria um dicionário e uma lista contendo os valores do dicionário criado. Ao final, o programa exibe o conteúdo da lista em tela:

```
info = {"país": "Brasil", "estado":"Ceará", "cidade":"Fortaleza", "ano
    ":2021}
valores = info.values()
print(valores)
```

A saída produzida por esse programa é:

```
dict_values(['Brasil', 'Ceará', 'Fortaleza', 2021])
```

■

Obtendo a quantidade de objetos em um dicionário

A função `len()`, que vimos anteriomente para listas, serve também para informar a quantidade de objetos em um dicionário.

■ **Exemplo 2.52** O código abaixo cria um dicionário e exibe a quantidade de objetos dentro dele:

```
info = {"país": "Brasil", "estado":"Ceará", "cidade":"Fortaleza", "ano
    ":2021}
print(len(info))
```

A saída produzida por esse programa é:

```
4
```

■

Modificando o conteúdo de um dicionário

Os objetos armazenados em um dicionário podem ser modificados. Podems fazer isso de duas maneiras:

- Acessando o objeto por meio de sua chave e atribuindo um novo valor;
- Utilizando a função update().

Um exemplo de acesso e atribuição de novo valor a um objeto de um dicionário é apresentado no Exemplo 2.53.

■ **Exemplo 2.53** O código abaixo cria um dicionário e atualiza a informação referente ao "ano". Ao final, o programa exibe em tela o dicionário atualizado:

```
info = {"país": "Brasil", "estado":"Ceará", "cidade":"Fortaleza", "ano
    ":2021}
info["ano"] = 2022
print(info)
```

A saída produzida por esse programa é:

```
{'país':'Brasil', 'estado':'Ceará', 'cidade':'Fortaleza', 'ano':2022}
```

■

Uma outra maneira de modificar o conteúdo de um dicionário é por meio da utilização do comando **update()**.

A sintaxe para esse comando é:

```
variavel.update({chave:valor})
```

onde `variavel` é o nome da variável que armazena o dicionário.

■ **Exemplo 2.54** O código abaixo cria um dicionário e atualiza a informação referente à "cidade". Ao final, o programa exibe em tela o dicionário atualizado:

```
info = {"país": "Brasil", "estado":"Ceará", "cidade":"Fortaleza", "ano
    ":2021}
info.update({"cidade":"Caucaia"})
print(info)
```

A saída produzida por esse programa é:

```
{'país':'Brasil', 'estado':'Ceará', 'cidade':'Caucaia', 'ano':2021}
```

■

Adicionando objetos a um dicionário

As operações que aprendemos sobre modificação de objetos de um dicionário servem também para adicionar objetos. Ao não encontrar a chave mencionada na operação de acesso ou no comando `update()`, o Python adiciona a chave no dicionário.

Um exemplo de inclusão de um novo objeto em um dicionário por meio de acesso e atribuição de novo valor é apresentado no Exemplo 2.55.

■ **Exemplo 2.55** O código abaixo cria um dicionário e adiciona a informação de "safra". Ao final, o programa exibe em tela o dicionário atualizado:

```
producao = {"fruta": "uva", "estado":"Ceará", "municipio":"Jaguaribe"}
producao["safra"] = 2021
print(producao)
```

A saída produzida por esse programa é:

```
{'fruta':'uva', 'estado':'Ceará', 'municipio':'Jaguaribe', 'safra':2021}
```

■

Podemos adicionar objetos em um dicionário utilizando também o comando **update()**. A sintaxe é exatamente a mesma para modificar um objeto de um dicionário. A única diferença é que o parâmetro passado para a função update() será uma chave que ainda não existe no dicionário.

■ **Exemplo 2.56** O código abaixo cria um dicionário e inclui uma informação referente à "tamanho" Ao final, o programa exibe em tela o dicionário atualizado:

```
info = {"código": 21014, "modelo":"regata", "cor":"azul"}
info.update({"tamanho":"M"})
print(info)
```

A saída produzida por esse programa é:

```
{'código':21014, 'modelo':'regata', 'cor':'azul', 'tamanho':'M'}
```

■

Removendo objetos de um dicionário

As opções que o Python disponibiliza para removermos objetos de um dicionário são semelhantes àquelas que vimos para listas. A depender do que desejamos como resultado, podemos escolher entre:

- Utilizar a função pop();
- Utilizar a palavra reservada del;
- Utilizar a função clear().

O comando **pop()** remove um objeto do dicionário, localizando-o por sua chave associada. A sintaxe para esse comando é:

```
variavel.pop(chave)
```

onde variavel é a variável que armazena o dicionário e chave é o localizador do objeto a ser removido do dicionário.

■ **Exemplo 2.57** O código a seguir cria um dicionário e remove o objeto referente à informação de "ano". O conteúdo total do dicionário após a operação de remoção é exibido na tela:

```
info = {"país": "Brasil", "estado":"Ceará", "cidade":"Fortaleza", "ano
    ":2021}
info.pop("ano")
print(info)
```

A saída produzida por esse programa será:

```
{'país':'Brasil', 'estado':'Ceará', 'cidade':'Fortaleza'}
```

■

Uma outra forma de remover um objeto de um dicionário é por meio da utilização da palavra reservada `del`. A sintaxe para sua utilização pode ser vista abaixo:

```
del variavel[chave]
```

onde `variavel` é a variável que armazena o dicionário e `chave` é o localizador do objeto a ser removido.

■ **Exemplo 2.58** O código a seguir cria um dicionário e remove o objeto referente à informação "modelo" do dicionário. O conteúdo total do dicionário após a operação de remoção é exibido na tela:

```
info = {"código": 21014, "modelo":"regata", "cor":"azul"}
del info["modelo"]
print(info)
```

A saída produzida por esse programa será:

```
{'código':21014, 'cor':'azul'}
```

■

Para esvaziar um dicionário, usamos o comando **`clear()`**.

A sintaxe para esse comando é:

```
x.clear()
```

onde `x` é o nome da variável que armazena o dicionário.

■ **Exemplo 2.59** O código a seguir cria um dicionário e remove todos os seus objetos. Após a remoção, um comando de exibição do dicionário em tela mostra que o dicionário ficou vazio:

```
info = {"país": "Brasil", "estado":"Ceará", "cidade":"Fortaleza", "ano
    ":2021}
info.clear()
print(info)
```

A saída produzida por esse programa será:

```
{}
```

■

Obs Utilizando o comando `del`, podemos eliminar o dicionário completamente. Assim, o comando

```
del variavel
```

onde `variavel` é uma variável que armazena o dicionário, tem um efeito diferente do comando `clear()`. Com o comando `clear()`, o dicionário ainda existe – ele só não tem objetos dentro dele. Já com o comando `del`, o dicionário deixa de existir!

Fazendo uma cópia de um dicionário

O processo de criação de uma cópia de um dicionário em Python é semelhante ao processo de criação de uma cópia de uma lista.

As funções que utilizaremos para criar uma cópia de um dicionário serão:

- `copy()`;
- `dict()`.

O comando **`copy()`** insere todos os objetos de um dado dicionário dentro de um novo dicionário criado.
A sintaxe para esse comando é:

```
novoDict = dictExistente.copy()
```

■ **Exemplo 2.60** O código a seguir cria um dicionário de objetos do tipo `str`. Em seguida, um novo dicionário é criado, contendo uma cópia do conteúdo do primeiro dicionário criado. Por fim, o programa exibe o conteúdo do segundo dicionário criado:

```
dictOriginal = {"nome":"Márcia","idade":13, "turma":"A"}
copiaDoDict = dictOriginal.copy()
print(copiaDoDict)
```

A saída produzida por esse programa será, naturalmente:

```
{ 'nome':'Márcia', 'idade':13, 'turma':'A'}
```

■

Alternativamente, podemos utilizar o comando de criação de um dicionário, o comando **`dict()`**.
A sintaxe para esse comando é:

```
novoDict = dict(dictExistente)
```

■ **Exemplo 2.61** O código a seguir cria uma lista de objetos do tipo `int`. Em seguida, uma nova lista é criada, contendo uma cópia do conteúdo da primeira lista criada. Por fim, o programa exibe o conteúdo da segunda lista criada:

```
dictOriginal = {"cor":"preta", "modelo":"feminino", "tamanho":"P"}
copiaDoDict = dict(dictOriginal)
print(copiaDoDict)
```

A saída produzida por esse programa será, naturalmente:

```
{'cor':'preta', 'modelo':'feminino', 'tamanho':'P'}
```

■

Aninhando dicionários

Os objetos de um dicionário podem também ser do tipo `dict`. Vamos ver, por meio de exemplos, como podemos organizar dicionários dentro de dicionários.

■ **Exemplo 2.62** O código abaixo cria dois dicionários dentro de um dicionário:

```
turma = {
"aluno1":{"nome":"Lívia", "matrícula":2110},
"aluno2":{"nome":"Lucas", "matrícula":2114},
}
```

Perceba que `"aluno1"` é uma chave para um objeto do tipo dicionário, cujo conteúdo é `{"nome":"Lívia", "matrícula":2110}`. De maneira similiar, `"aluno2"` é uma chave para um objeto do tipo dicionário, cujo conteúdo é `{"nome":"Lucas", "matrícula":2114}`. ■

■ **Exemplo 2.63** Podemos organizar a criação do mesmo dicionário do Exemplo 2.62 de uma maneira diferente. Desta vez, vamos criar cada dicionário separadamente e depois juntar os dois dicionários em um maior. O código abaixo mostra como fazer isso.

```
aluno1 = {"nome":"Lívia", "matrícula":2110}
aluno2 = {"nome":"Lucas", "matrícula":2114}
turma = {"aluno1":aluno1, "aluno2":aluno2}
```

Obs Perceba a diferença de notação entre os nomes de variáveis `aluno1` e `aluno2` e as chaves `"aluno1"` e `"aluno2"` no código deste Exemplo.

■

2.5.3 – Conjuntos

Conjutnos em Python são armazenados em objetos do tipo `set`. Como foi apresentado na Seção 2.3, os conjuntos não possuem objetos repetidos e seus elementos não são modificáveis. Vamos conhecer, agora, as principais operações efetuadas sobre conjuntos no Python.

Criando um conjunto

No Python, podemos criar um conjunto por meio de um comando de atribuição a uma variável. O código abaixo mostra um exemplo:

```
conj = {"feijão", "ervilha", "lentilha"}
```

Uma outra forma é utilizar a função **set()**. A sintaxe desse comando para a criação de um novo conjunto é:

```
conj = set(("feijão", "ervilha", "lentilha"))
```

Obs Note os duplos pares de parênteses na utilização do comando **set() para a criação de um novo conjunto.**

Obtendo o tamanho de um conjunto

A função **len()**, que vimos anteriomente para listas e dicionários, serve também para informar a quantidade de objetos em um conjunto.

■ **Exemplo 2.64** O código abaixo cria um conjunto e exibe a sua cardinalidade (ou seja, seu tamanho):

```
conjunto = {"carambola", "cajarana", "caju", "coco"}
print(len(conjunto))
```

A cardinalidade do conjunto exibida pelo programa é:

4

■

Verificando se um objeto pertence a um conjunto

O Python possibilita a checagem se um determinado objeto pertence ou não a um conjunto. Para isso, utilizamos uma estrutura de decisão e a palavra reservada **in**. O Exemplo 2.65 mostra como verificar se um dado objeto pertence a uma lista:

■ **Exemplo 2.65** O código abaixo cria um conjunto e checa se o objeto 'canário' pertence a ele:

```
conj = {'beija-flor', 'galo de campina', 'bem-te-vi', 'pardal', 'sanha
    ço-azul', 'soldadinho-do-Araripe', 'canário'}

if 'canário' in conj:
    print('Pássaro pertence ao conjunto')
else:
    print('Pássaro não pertence ao conjunto')
```

■

Obs Reforçamos o lembrete de que as palavras `if` e `else` do código apresentado no Exemplo 2.65 são palavras reservadas para **estruturas de decisão**, que estudaremos na Seção 2.7.

Exercício 2.11 Fui testar o código do Exemplo 2.65 com um outro elemento do conjunto, só que o programa não funcionou! Eu testei com o soldadinho do araripe e o programa exibiu a mensagem dizendo que o pássaro não pertence ao conjunto. Eu vejo o nome do pássaro dentro do conjunto, mas meu código não encontra. Vou mostrar meu código aqui para ver se vocês podem me ajudar:

```
conj = {'beija-flor', 'galo de campina', 'bem-te-vi', 'pardal', '
   sanhaço-azul', 'soldadinho-do-Araripe', 'canário'}

if 'soldadinho-do-araripe' in conj:
    print('Pássaro pertence ao conjunto')
else:
    print('Pássaro não pertence ao conjunto')
```

Até os hífens eu lembrei de escrever para deixar o nome do pássaro bem direitinho. O que pode estar errado?

Solução. Seu código está funcionando como deveria, ele está correto. De fato, o objeto `'soldadinho-do-araripe'` que você mandou pesquisar, não pertence ao conjunto. Você precisa lembrar que o Python faz distinção entre maiúsculas e minúsculas, portanto o objeto `'soldadinho-do-araripe'` é diferente do objeto `'soldadinho-do-Araripe'`. Este último sim, é um objeto que pertence ao conjunto criado no código. ■

Adicionando objetos a um conjunto

O Python permite a inserção de um ou mais objetos em um conjunto. A depender do que se deseja adicionar a um conjunto, podemos utilizar as funções:

- `add()`;
- `update()`

O comando `add()` é utilizado para se adicionar um objeto a um conjunto. A sintaxe para esse comando é:

```
variavel.add(objeto)
```

onde `variavel` é a variável que armazena o conjunto e `objeto` é o novo elemento a ser inserido no conjunto.

■ **Exemplo 2.66** O código abaixo cria um conjunto e inclui o objeto "caqui". Ao final, o programa exibe em tela o conjunto atualizado:

```
conjunto = {"carambola", "cajarana", "caju", "coco"}
conjunto.add("caqui")
print(conjunto)
```

A saída produzida por esse programa não será a mesma sempre. Os elementos serão sempre os mesmos, porém a ordem em que eles aparecem no conjunto vai mudar cada vez que o programa for executado. Um conjunto não estabelece uma ordem para seus elementos. A saída apresentada abaixo é uma das saídas possíveis:

```
{'coco', 'caju', 'cajarana', 'caqui', 'carambola'}
```

■

Agora, para adicionar objetos de um outro conjunto em um dado conjunto, vamos utilizar o comando **`update()`**. A sintaxe desse comando é:

```
receptor.update(doador)
```

onde `receptor` é o conjunto que vai receber os elementos do conjunto `doador`.

■ **Exemplo 2.67** O código abaixo cria dois conjuntos e adiciona os elementos do segundo conjunto no primeiro. Ao final, o programa exibe em tela o primeiro conjunto, agora atualizado:

```
primeiro = {3, 7, 9}
segundo = {0, 2, 5, 8}
primeiro.update(segundo)
print(primeiro)
```

Uma saída produzida por esse programa é:

```
{0, 2, 3, 5, 7, 8, 9}
```

■

Removendo objetos de um conjunto

As opções que o Python disponibiliza para removermos objetos de um conjunto estão listadas abaixo:

- Utilizar a função `remove()`;
- Utilizar a função `discard()`;
- Utilizar a função `clear()`;
- Utilizar a palavra reservada `del`.

Tanto a função **`remove()`** quanto **`discard()`** removem um elemento do conjunto. A sintaxe para cada um desses comandos é apresentada abaixo:

```
variavel.remove(objeto)
variavel.discard(objeto)
```

onde `variavel` é a variável que armazena o conjunto e `objeto` é o objeto a ser removido.

■ **Exemplo 2.68** O código a seguir cria um conjunto e remove o objeto "lousa" dele. O conteúdo total do conjunto após a operação de remoção é exibido na tela:

```
conjunto = {"apagador", "carteira", "lousa", "pincel"}
conjunto.remove("lousa")
print(conjunto)
```

Uma saída produzida por esse programa é:

```
{'carteira', 'pincel', 'apagador'}
```

■

O comando **discard()** tem comportamento semelhante, só que é um pouquinho melhor. Vamos entender após analisar o Exemplo 2.69.

■ **Exemplo 2.69** O código a seguir cria um conjunto e remove o objeto "abacate" dele. O conteúdo total do conjunto após a operação de remoção é exibido na tela:

```
conjunto = {"mamão", "banana", "laranja"}
conjunto.discard("abacate")
print(conjunto)
```

Uma saída produzida por esse programa é:

```
{'banana', 'laranja', 'mamão'}
```

Ao que parece, o comando **discard()** faz a mesma coisa que o comando **remove()**. Porém, para entender a diferença, vamos reescrever o código acima, agora usando o comando **remove()**:

```
conjunto = {"mamão", "banana", "laranja"}
conjunto.remove("abacate")
print(conjunto)
```

Ao tentar executar o programa, o Python exibe uma mensagem de erro. O comando `remove()` não funciona se tentarmos remover um objeto que não pertence ao conjunto. Já o comando `discard()` não produz mensagem de erro nesse caso. Ele simplesmente não retira nada do conjunto. Aí está a diferença entre `remove()` e `discard()`. ■

Para esvaziar um conjunto, usamos o comando **clear()**.

A sintaxe para esse comando é:

```
x.clear()
```

onde `x` é o nome da variável que armazena o conjunto.

■ **Exemplo 2.70** O código a seguir cria um conjunto e remove todos os seus objetos. Após a remoção, um comando de exibição do conjunto em tela mostra que o conjunto agora ficou vazio:

```
conjunto = {4, 2, 0, "Ceará", 7.32, 14, "cinco"}
conjunto.clear()
print(conjunto)
```

A saída produzida por esse programa será:

```
set()
```

■

Obs Assim como acontece com dicionários, utilizando o comando `del`, podemos eliminar um conjunto completamente. Assim, o comando

```
del x
```

onde `x` é uma variável que armazena o conjunto, tem um efeito diferente do comando `clear()`. Com o comando `clear()`, o conjunto ainda existe – só que ele é um conjunto vazio. Já com o comando `del`, o conjunto deixa de existir!

Fazendo uma cópia de um conjunto

O processo de criação de uma cópia de um conjunto em Python é feito com o uso da função `copy()`.
O comando `copy()` insere todos os objetos de um dado conjunto dentro de um novo conjunto criado.
A sintaxe para esse comando é:

```
novoConjunto = conjuntoExistente.copy()
```

■ **Exemplo 2.71** O código a seguir cria um conjunto de objetos do tipo `str`. Em seguida, um novo conjunto é criado, contendo uma cópia do conteúdo do primeiro conjunto criado. Por fim, o programa exibe o conteúdo do segundo conjunto:

```
conjOriginal = {"Fortaleza", "Natal", "Recife"}
copiaDoConjunto = conjOriginal.copy()
print(copiaDoConjunto)
```

Uma saída produzida por esse programa é:

```
{'Fortaleza', 'Recife', 'Natal'}
```

■

Efetuando operações sobre conjuntos

O Python disponibiliza funções para efetuarmos as operações de união, interseção, diferença e diferença simétrica entre conjuntos.
Para efetuar a **união** entre conjuntos, utilizamos a função **`union()`**. A sintaxe para esse comando é:

```
conj_união = conj1.union(conj2)
```

onde `conj_união` é um novo conjunto que armazenará o resultado da operação de união entre `conj1` e `conj2`.

■ **Exemplo 2.72** O código a seguir cria dois conjuntos e um terceiro conjunto, que contém os elementos resultantes da união dos dois primeiros conjuntos criados. Após a operação de união, um comando de exibição mostra o conjunto-união resultante:

```
conjunto1 = {4, 0, "Ceará", 14, "cinco"}
conjunto2 = {"Bahia", 5, "cinco", "treze", 14}
conjuntoUniao = conjunto1.union(conjunto2)
print(conjuntoUniao)
```

Uma saída produzida por esse programa é:

```
{0, 'Ceará', 4, 5, 'treze', 14, 'cinco', 'Bahia'}
```

Obs Observe que o conjunto-união não repetirá os elementos comuns aos dois conjuntos.

■

Para efetuar a **interseção** entre conjuntos, utilizamos a função **`intersection()`**. A sintaxe para esse comando é:

```
conj_interseção = conj1.intersection(conj2)
```

onde `conj_interseção` é um novo conjunto que armazenará o resultado da operação de interseção entre `conj1` e `conj2`.

■ **Exemplo 2.73** O código a seguir cria dois conjuntos e um terceiro conjunto, que contém os elementos resultantes da interseção dos dois primeiros conjuntos criados. Após a operação de interseção, um comando de exibição mostra o conjunto-interseção resultante:

```
conjunto1 = {4, 0, "Ceará", 14, "cinco"}
conjunto2 = {"Bahia", 5, "cinco", "treze", 14}
conjuntoIntersecao = conjunto1.intersection(conjunto2)
print(conjuntoIntersecao)
```

Uma saída produzida por esse programa é:

```
{'cinco', 14}
```

■

Para efetuar a **diferença** entre conjuntos, utilizamos a função **`difference()`**. A sintaxe para esse comando é:

```
conj_diferença = conj1.difference(conj2)
```

onde `conj_diferença` é um novo conjunto que terá como elementos aqueles que estão em `conj1` e não estão em `conj2`.

■ **Exemplo 2.74** O código a seguir cria dois conjuntos e um terceiro conjunto, que contém os elementos que estão no primeiro conjunto e não estão no segundo. Após a operação de diferença do primeiro pelo segundo, um comando de exibição mostra o conjunto-diferença resultante:

```
conjunto1 = {4, 0, "Ceará", 14, "cinco"}
conjunto2 = {"Bahia", 5, "cinco", "treze", 14}
diferenca = conjunto1.difference(conjunto2)
print(diferenca)
```

Uma saída produzida por esse programa é:

```
{0, 'Ceará', 4}
```

■

O Python disponibiliza, também, uma função para obter a diferença simétrica entre dois conjuntos. O nome da função é **`symmetric_difference()`** e sua sintaxe é:

```
conj_difSimetrica = conj1.symmetric\_difference(conj2)
```

onde `conj_difSimetrica` é um novo conjunto que armazenará o resultado da operação de diferença simétrica entre `conj1` e `conj2`.

Obs A diferença simétrica entre dois conjuntos é um conjunto que contém somente os elementos que não são comuns a ambos os conjuntos envolvidos na operação.

■ **Exemplo 2.75** O código a seguir cria dois conjuntos e um terceiro conjunto, que contém os elementos resultantes da diferença simétrica entre os dois primeiros conjuntos criados. Após a operação de diferença simétrica, um comando de exibição mostra o conjunto-diferença-simétrica resultante:

```
conjunto1 = {4, 0, "Ceará", 14, "cinco"}
conjunto2 = {"Bahia", 5, "cinco", "treze", 14}
conjuntoDifSimetrica = conjunto1.symmetric_difference(conjunto2)
print(conjuntoDifSimetrica)
```

Uma saída produzida por esse programa é:

```
{0, 4, 5, 'treze', 'Ceará', 'Bahia'}
```

■

Verificando relações entre conjuntos

O Python disponibiliza funções para checagem de relação de continência entre conjuntos. As funções disponíveis são:

- `issubset()`;
- `isupperset()`;
- `isdisjoint()`;

Para checar se um conjunto A é subconjunto de um conjunto B, a sintaxe do Python para efetuar essa operação é:

```
A.issubset(B)
```

Esta operação produz como resultado um valor lógico (`True` ou `False`).

■ **Exemplo 2.76** O código a seguir cria dois conjuntos e exibe na tela a informação se o primeiro conjunto é subconjunto do segundo:

```
primeiro = {1, 2, 3, 4, 5}
segundo = {2, 4}
print(primeiro.issubset(segundo))
```

A saída produzida por esse programa é:

```
False
```

■

Para checar se um conjunto A contém um conjunto B, a sintaxe do Python para efetuar essa operação é:

```
A.issuperset(B)
```

Esta operação produz como resultado um valor lógico (`True` ou `False`).

■ **Exemplo 2.77** O código a seguir cria dois conjuntos e exibe na tela a informação se o primeiro conjunto contém o segundo:

```
primeiro = {1, 2, 3, 4, 5}
segundo = {2, 4}
print(primeiro.issuperset(segundo))
```

A saída produzida por esse programa é:

```
True
```

■

Para checar se um conjunto A e um conjunto B são disjuntos, a sintaxe do Python para efetuar essa operação é:

```
A.isdisjoint(B)
```

Esta operação produz como resultado um valor lógico (`True` ou `False`).

■ **Exemplo 2.78** O código a seguir cria dois conjuntos e exibe na tela a informação se o primeiro conjunto e o segundo são disjuntos:

```
primeiro = {1, 2, 3, 4, 5}
segundo = {2, 4}
print(primeiro.isdisjoint(segundo))
```

A saída produzida por esse programa é:

```
False
```

■

2.6 – Operadores

Em alguns dos exemplos e exercícios feitos até este ponto do material, podemos ver instruções contendo alguns operadores. Nesta seção, vamos conhecer sintaxe do Python para os operadores:

- Aritméticos;
- Relacionais;
- Lógicos.

Além disso, vamos conhecer a sintaxe do Python para as operações de:

- Atribuição;
- Pertinência.

2.6.1 – Operadores Aritméticos

São operadores aplicáveis a objetos de tipo numérico.
Os operadores aritméticos disponíveis na linguagem Python são apresentados na tabela 2.6:

Operador	Nome	Exemplo
+	adição	5 + 2
-	subtração	x - 10
*	multiplicação	2 * y
/	divisão	x/3
//	divisão inteira	x // y
%	resto da divisão inteira	5 % 3
**	exponenciação	4 ** 2

Tabela 2.6: Operadores aritméticos da linguagem Python.

A tabela 2.7 Apresenta a ordem de precedência dos operadores aritméticos:

Operador	Ordem de Precedência
`**`	Primeira
`*`, `/`, `//` e `%`	Segunda
`+` e `-`	Terceira

Tabela 2.7: Ordens de precedência dos operadores aritméticos.

Obs Lembrem-se de que podemos sempre modificar a ordem de precedência de operadores aritméticos utilizando parênteses.

Na Tabela 2.6, podemos ver operadores que já nos são bem familiares, e outros que merecem uma discussão mais detalhada. Vamos discutir a seguir os operadores de **divisão inteira** e de **resto da divisão inteira**.

Operador de divisão inteira

A divisão de um valor por outro contém os seguintes elementos:

- Dividendo;
- Divisor;
- Quociente;
- Resto.

Em Python, o operador de divisão é o `/`, e produz como resultado a divisão real de um número por outro. Por exemplo, a operação de divisão

```
5 / 2
```

resultará no valor 2.5.

Em algumas aplicações, no entanto, podemos estar interessados em obter somente a parte inteira da divisão de um número por outro. O operador que produz esse tipo de resultado é o `//`. Com este operador, a expressão

```
5 // 2
```

resultará no valor 2.

Em outras palavras, o operador de divisão inteira obtém o quociente de uma divisão de um número por outro. Por exemplo, a operação 21 // 5 resultará em 4, que é o quociente, como podemos ver abaixo:

$$\begin{array}{r|l} 21 & 5 \\ \cline{2-2} (1) & \mathbf{4} \end{array}$$

Operador de resto da divisão inteira

O Python contém um operador para permitir a obtenção do valor do resto da divisão de um valor por outro. Vimos, no Capítulo 1, que esta operação é útil em aplicações que envolvem aritmética modular, por exemplo. O operador que fornece o resto da divisão de um número por outro é %. Com este operador, a expressão

```
5 % 2
```

resultará no valor 1, pois

$$\begin{array}{r|l} 5 & 5 \\ (\mathbf{1}) & 2 \end{array}$$

Para praticar, vamos utilizar os operadores aritméticos em alguns programas.

Exercício 2.12 Faça um programa para receber um número entre 10 e 99 e exiba um relatório informando quantas dezenas e quantas unidades compõem o número. Por exemplo, o número 63 faria o programa exibir o seguinte relatório:

```
O número informado é 63.  Ele tem 6 dezenas e 3 unidades.
```

Utilize operadores aritméticos para resolver esse exercício.

Solução. Para separar um número da forma 'dezena + unidade', vamos tomar o número fornecido e dividir por 10. Digamos que o número fornecido seja, por exemplo, 74. A divisão dele por 10 é descrita como:

$$\begin{array}{r|l} 74 & 10 \\ (4) & 7 \end{array}$$

Podemos ver que o quociente da divisão é 7 e o resto é 4. Queremos que nosso programa seja capaz de obter esses dois valores, para colocarmos no relatório pedido. Os operadores que iremos utilizar para obter esses valores serão o de divisão inteira e o de resto da divisão inteira.
Um programa para resolver o problema proposto no Exercício 2.12 pode ser visto no código abaixo:

```
numero = input("Por gentileza, forneça um número entre 10 e 99: ")
dezenas = int(numero) // 10 #O quociente da divisão é a dezena
unidades = int(numero) % 10 #O resto da divisão é a unidade
relatorio = "O número fornecido foi {}. Ele contém {} dezenas e {}
   unidades"
print(relatorio.format(numero, dezenas, unidades))
```

■

Exercício 2.13 Faça um programa que receba um valor em Reais e informe à(ao) usuária(o) como obter esse valor com a menor quantidade possível de cédulas. Assuma que o valor em Reais não contém centavos.

Solução. As cédulas que vamos considerar para este programa são:

- 200;
- 100;
- 50;

- 20;
- 10;
- 5;
- 2.

Para o valor de 1 Real, vamos considerar moedas.

Para se obter a menor quantidade de cédulas possível, devemos começar considerando quantas cédulas do maior possível podemos usar. Assim, começamos tentando usar as cédulas de 200 Reais e, gradativamente, vamos considerando cédulas de menor valor com os valores que ainda sobrarem para serem representados por cédulas.
O programa abaixo é uma das formas de se resolver o problema.

```
numero = input("Por gentileza, informe o valor em Reais: ")
ced200 = int(numero) // 200
ced100 = int(numero) % 200 // 100
ced50 = int(numero) % 200 % 100 // 50
ced20 = int(numero) % 200 % 100 % 50 // 20
ced10 = int(numero) % 200 % 100 % 50 % 20 // 10
ced5 = int(numero) % 200 % 100 % 50 % 20 % 10 // 5
ced2 = int(numero) % 200 % 100 % 50 % 20 % 10 % 5 // 2
moeda1 = int(numero) % 200 % 100 % 50 % 20 % 10 % 5 % 2

relatorio = "O valor em Reais pode ser pago em: {} cédulas de 200; {}
    cédulas de 100; {} cédulas de 50; {} cédulas de 20; {} cédulas de
    10; {} cédulas de 5; {} cédulas de 2 e {} moedas de 1 Real"
print(relatorio.format(ced200, ced100, ced50, ced20, ced10, ced5, ced2
    , moeda1))
```

■

Obs Uma sugestão de exercício é escolher um valor em Reais e tomar nota dos resultados das operações realizadas no código da solução do Exercício 2.13. Desta forma, você poderá encontrar outras expressões aritméticas para resolver o problema proposto.

Exercício 2.14 Escreva um programa em Python que receba um número entre 100 e 999 e exiba o valor correspondente ao "espelho" dele. Por exemplo, se o(a) usuário(a) fornecer o valor 372, o programa exibirá o número 273. Esses dados são numéricos, e não textos, e você deve utilizar operadores aritméticos para solucionar o problema.

Solução. Vamos aproveitar as ideias praticadas nos exercícios anteriores para planejar a solução deste problema.
Sabendo que um número entre 100 e 999 é expresso no formato 'centena + dezena + unidade', nossa estratégia será:

- Utilizar os operadores de divisão inteira e resto da divisão inteira para separar cada dígito do número fornecido, e
- Montar o novo valor com os dígitos trocados conforme pedido.

O código a seguir realiza as operações planejadas para solucionar este exercício:

```
numero = input("Por gentileza, digite um número entre 100 e 999: ")
centena = int(numero) // 100
dezena = int(numero) % 100 // 10
unidade = int(numero) % 100 % 10

#invertendo os papéis de cada dígito no novo número:
espelhado = unidade*100 + dezena*10 + centena

print("O número espelhado é: ", espelhado)
```

■

2.6.2 – Operadores Relacionais

São os operadores utilizados quando desejamos comparar dois valores. O resultado produzido por um operador relacional será sempre do tipo lógico.
Os operadores relacionais disponíveis na linguagem Python são apresentados na tabela 2.8:

Operador	Nome	Exemplo
==	igual	`x == 2`
!=	diferente	`x != 10`
>	maior que	`y > 2`
<	menor que	`x < 3`
>=	maior ou igual a	`x >= y`
<=	menor ou igual a	`y <= 3`

Tabela 2.8: Operadores relacionais da linguagem Python.

Exercício 2.15 Para praticar, indique o resultado das expressões relacionais abaixo:

(a) $4 < 5$
(b) $11 \% 2 == 0$
(c) False != True
(d) True $== 3 >$ 2-1
(e) $5 >= 5$
(f) $3 <= 3$

Solução. As respostas para os itens do Exercício 2.15 podem ser vistas abaixo:

(a) $4 < 5$: **True**
(b) $11 \% 2 == 0$: **False**
(c) False != True: **True**
(d) True $== 3 >$ 2-1: **False**
(e) $5 >= 5$: **True**
(f) $3 <= 3$: **True**

Obs

É importante observar que em uma expressão contendo operações aritméticas e relacionais, resolvemos primeiro as expressões aritméticas e depois as operações relacionais.

■

2.6.3 – Operadores Lógicos

São operadores aplicáveis a objetos do tipo lógico. São também utilizados para compor expressões mais complexas, envolvendo operadores aritméticos e relacionais. O resultado produzido por um operador lógico será sempre do tipo `bool`.
Os operadores lógicos disponíveis na linguagem Python são apresentados na tabela **??**:

Operador	Descrição	Exemplo
`and`	Realiza a operação referente ao conectivo lógico E	`y > 2 and y < 20`
`or`	Realiza a operação referente ao conectivo lógico OU	`x == 5 or x < 5`
`not`	Realiza a operação referente ao conectivo lógico NÃO	`not(x == 1)`

Tabela 2.9: Operadores lógicos da linguagem Python.

A tabela 2.10 Apresenta a ordem de precedência dos operadores lógicos:

Operador	Ordem de Precedência
`not`	Primeira
`and`	Segunda
`or`	Terceira

Tabela 2.10: Ordens de precedência dos operadores lógicos.

Obs Lembrem-se de que podemos sempre modificar a ordem de precedência de operadores lógicos utilizando parênteses.

Exercício 2.16 Indique o resultado das expressões lógicas enumeradas abaixo:

(a) `not False`
(b) `5 == 5 and 5 < 5`
(c) `5 == 5 or 5 < 5`
(d) `3 < 5 and True == (5 == 5 or 5 < 5)`
(e) `2*2 != 8 or 5 != 1+3 and 5 <= 5`
(f) `not (16 > 10+6) and True == 1 > 0 or False`
(g) `5 == 2 + 3 or not True or not False and 8 == 4 * 2`

Solução. Os resultados das expressões do Exercício 2.16 podem ser vistos abaixo:

(a) `not False :  True`
(b) `5 == 5 or 5 < 5 :  True`
(c) `5 == 5 and 5 < 5 :  False`
(d) `3 < 5 and True == (5 == 5 or 5 < 5) :  True`
(e) `2*2 != 8 or 5 != 1+3 and 5 <= 5 :  True`
(f) `not(16 > 10+6) and True == 1 > 0 or False :  True`
(g) `5 == 2 + 3 or not True or not False and 8 == 4 * 2 :  True`

■

2.6.4 – Operadores de Atribuição

Sabemos que uma variável armazena dados que desejamos manipular por meio de instruções de nossos programas. Vimos, anteriormente, o comando de atribuição, que realiza o armazenamento de um dado em uma variável. Como exemplo, a instrução

```
x = 2
```

atribui o valor 2 à variável `x`. Dizemos, normalmente, que "x recebe 2".

Considere, agora, uma operação em que se deseja atualizar o valor armazenado em `x`. Queremos que `x` receba o que tinha antes, mais 1. Assim, a variável `x` passará a armazenar o valor 3.
O comando para realizar essa operação é um comando de atribuição, como se vê abaixo:

```
x = x + 1
```

Com isto, a variável `x` sofreu o que chamamos de um **incremento** do seu valor armazenado. Considere, agora, que desejamos diminuir o valor armazenado em `x` de uma unidade. A variável receberá o valor que tinha antes, menos 1.
O comando para realizar essa operação é um comando de atribuição, como se vê abaixo:

```
x = x - 1
```

Com isto, a variável `x` sofreu o que chamamos de um **decremento** do seu valor armazenado.

Operações de incremento e decremento de variáveis são muito frequentes em programas. Elas permitem que variáveis funcionem como contadores de ocorrência de um determinado evento, como controle de repetições em um laço, entre outros usos. A frequência dessas operações é tão alta, que as linguagens de programação oferecem uma notação especial para elas.

Modificações de incremento e decremento não necessariamente ocorrem para alterar uma variável em uma unidade. Podemos incrementar e decrementar variáveis em mais de uma unidade. Além disso, as operações não se limitam à adição e subtração. É possível realizar atualizações de uma variável utilizando outros operadores. A forma geral do operador em Python para esse tipo de atribuição é:

```
variável op= valor
```

onde `op` pode ser qualquer operador aritmético.

Assim, um operador para incremento de uma variável `x` de uma unidade, em Python, é escrito como

```
x += 1
```

A lista completa de operadores especiais de atribuição envolvendo operadores aritméticos pode ser vista na Tabela 2.11.

Operador	Exemplo	Comando Equivalente
`+=`	`y += 5`	`y = y + 5`
`-=`	`x -= 1`	`x = x - 1`
`*=`	`x *= 2`	`x = x * 2`
`/=`	`d /= 3`	`d = d / 3`
`%=`	`x %= 2`	`x = x % 2`
`//==`	`i //= 2`	`i = i // 2`
`**=`	`e **= 2`	`e = e ** 2`

Tabela 2.11: Operadores especiais de atribuição da linguagem Python.

Exercício 2.17 Considerando a instrução `x = 12`, informe o valor armazenado em `x` nas instruções abaixo:

(a) `x += 3`
(b) `x -= 5`
(c) `x *= 8`
(d) `x /= 5`
(e) `x //= 9`
(f) `x %= 9`
(g) `x **= 2`

Solução. Os valores armazenados em `x` após a execução das instruções enumeradas no Exercício 2.17 estão indicados abaixo:

(a) `x += 3` : `x` armazena o valor 15
(b) `x -= 5` : `x` armazena o valor 7
(c) `x *= 8` : `x` armazena o valor 96
(d) `x /= 5` : `x` armazena o valor 2.4
(e) `x //= 9` : `x` armazena o valor 1
(f) `x %= 9` : `x` armazena o valor 3
(g) `x **= 2` : `x` armazena o valor 144

■

2.6.5 – Operadores de Pertinência

O Python contém operadores para facilitar a checagem se um dado objeto pertence a algum outro objeto. Estes operadores são muito utilizados para verificar a existência de um elemento dentro de um objeto que armazena múltiplos valores. Além disso, esses operadores são úteis para facilitar a escrita de laços de repetição.

Obs Maiores detalhes sobre laços de repetição serão vistos na Seção 2.8.

Os operadores de pertinência do Python podem ser vistos na Tabela 2.12.

Operador	Descrição
`in`	Verifica se um dado objeto se encontra armazenado dentro de um outro objeto
`not in`	Verifica se um dado objeto **não** se encontra armazenado dentro de um outro objeto

Tabela 2.12: Operadores de pertinência da linguagem Python.

Exercício 2.18 Considere a lista

```
lista = list((2, 4, 6, 8))
```

Indique o resultado das operações enumeradas abaixo:

(a) `print(2 in lista)`
(b) `print(0 in lista)`
(c) `print(2 not in lista)`
(d) `print(0 not in lista)`

Solução. Considerando a lista fornecida no Exercício 2.18, os resultados dos comandos nos itens de (a) a (d) são:

(a) `print(2 in lista) :  True`
(b) `print(0 in lista) :  False`
(c) `print(2 not in lista) :  False`
(d) `print(0 not in lista) :  True`

■

2.7 – Estruturas de decisão

Como foi visto no Módulo 1 deste curso, a estrutura de decisão é essencial para descrever processos. Vamos ver, nesta seção, como escrever este tipo de estrutura no Python.

Obs Por determinar os desvios no fluxo de execução de instruções em um algoritmo, a estrutura de decisão recebe, também, o nome de **desvio condicional**.

2.7.1 – Sintaxe

O Python possui somente uma estrutura de decisão, que é iniciada com a palavra-chave `if` e que possui algumas variantes, que veremos a seguir. A sintaxe básica é ilustrada no Exemplo 2.79.

■ **Exemplo 2.79** O seguinte programa lê a idade do usuário a partir do teclado e imprime uma mensagem, informando se o usuário é maior ou menor de idade.

```
x = input('Digite sua idade: ')
x = int(x) # Conversão para inteiro

if x >= 18:
    print('Você é maior de idade.')
else:
    print('Você é menor de idade.')
```

Além da palavra-chave `if` (equivalente a "se", em português), este exemplo contém também a palavra-chave `else`, que é equivalente a "senão", em português. O par de palavras-chaves `if` e `else` forma a estrutura de decisão básica do Python. Após o `if`, temos uma expressão que possui um valor lógico (verdadeiro/falso). Se a expressão for verdadeira (isto é, `x` for maior ou igual a 18), então será impressa na tela a frase "`Você é maior de idade.`". Caso contrário (se `x` for menor que 18), será impressa na tela a frase "`Você é menor de idade.`". ■

Obs A expressão lógica que vem logo após a palavra `if` é chamada de **condição**. A parte `else` da estrutura não possui uma condição. Afinal de contas, a parte `else` corresponde à negação lógica da condição que aparece na parte `if`. É por essa razão que não podemos ter a parte `else` sozinha; ela só pode ser usada em conjunto com a parte `if`.

Como vimos em outro momento, uma estrutura de decisão causa uma divisão do fluxo de execução de um programa. No Exemplo 2.79, a divisão está no fato de que somente uma das duas frases será impressa. Ou seja, ou o programa irá executar a instrução que imprime "`Você é maior de idade.`", ou o programa irá executar a outra instrução "`print`". Não há a possibilidade de que as duas impressões ocorram.

Em algumas situações, não é necessário usar a parte `else` da estrutura de decisão. O Exemplo 2.80 mostra um desses casos.

■ **Exemplo 2.80** O programa a seguir lê um número inteiro a partir do teclado e calcula o módulo (valor absoluto) do número. O número digitado é armazenado na variável `x`. A variável `modulox` é utilizada para armazenar inicialmente o valor de `x`. Se `x` for menor que zero, então o valor de `modulox` é modificado para `-x`. Ao final, é o valor de `modulox` que é impresso como sendo o valor absoluto de `x`. Observe que não há necessidade da parte `else` na estrutura de decisão. Afinal, se `x` for maior ou igual a zero, não é necessário fazer nada especial: o valor armazenado em `modulox` já é igual ao módulo de `x`.

```
x = input('Digite um número inteiro: ')
x = int(x) # Conversão de string para inteiro

modulox = x

if x < 0:
    modulox = -x

print('O módulo de {} é {}.'.format(x, modulox))
```

■

Obs A forma geral da estrutura `if` é: a palavra "`if`", seguida de uma expressão que possui valor lógico (isto é, quando avaliada, a expressão assume o valor verdadeiro ou falso), seguida do caractere dois pontos ("`:`"). Para a parte `else`, temos apenas o caractere "`:`", já que não há condição a ser testada.

Exercício 2.19 Escreva um programa que detecta a hora do dia e que imprime duas mensagens genéricas de "bom dia" (se a hora atual for menor que 18) ou "boa noite" (se a hora for maior ou igua a 18).

Solução. Para detectar a hora atual do computador utilizando o Python, vamos usar uma biblioteca de funções chamada `datetime`. Este é o propósito da instrução "`import datetime as dt`". Além de avisar ao Python para utilizar a biblioteca, esta instrução atribui um nome curto para ela, `dt`. Com isso, precisamos escrever menos na hora de escrever o código para recuperar a hora atual. Na Seção 2.10, vamos discutir com mais detalhes o uso de bibliotecas.

```
import datetime as dt

horaDoDia = dt.datetime.today().hour

if horaDoDia < 18:
    print('Bom dia.')
    print('Tenha um dia produtivo.')
else:
    print('Boa noite.')
    print('Bom descanso.')
```

O comando "`dt.datetime.today().hour`" retorna a hora do computador, na forma de um número inteiro entre 0 e 23. ■

Obs Perceba que o código na solução do Exercício 2.19 possui duas linhas de código tanto na parte do `if` quanto na parte do `else`. Isso vai se tornar mais comum, na medida em que nossos programas forem se tornando mais e mais elaborados.

2.7.2 – Indentação

É importante perceber nos exemplos anteriores que as instruções que estão dentro da estrutura de decisão aparecem deslocadas para a direita em relação às palavras-chaves `if` e `else`. Este deslocamento é chamado de **indentação** e é parte essencial da estrutura de decisão.

É a indentação que determina exatamente quais instruções fazem parte da estrutura de decisão. Logo após o caractere dois pontos da parte `if`, temos uma ou mais linhas de código indentadas. Estas linhas formam o trecho de código que será executado se a condição do `if` for verdadeira.

Da mesma forma, após os dois pontos da parte `else`, temos uma ou mais linhas de código que devem estar indentadas em relação à palavra-chave `else`. São elas que serão executadas se a condição da parte `if` for falsa.

É importante que as palavras-chaves `if` e `else` estejam exatamente no mesmo nível de indentação. As linhas de código controladas pelo `if` devem todas estar no mesmo nível de indentação também, assim como as linhas de código controladas pelo `else`. Caso as regras de indentação não sejam respeitadas, temos duas possibilidades:

(a) O Python irá exibir uma mensagem de erro e o programa não irá executar;

(b) O Python irá interpretar o progama de maneira diferente daquela que o programador pretendia e teremos um **erro lógico**. Isto significa que o programa está escrito corretamente (a sintaxe está de acordo com as regras do Python), mas a tarefa realizada não é aquela que o programador tinha em mente.

O segundo erro é mais grave, já que ele causa uma espécie de erro "silencioso", que só é percebido quando analisamos os resultados produzidos pelo programa. Além disso, este é um tipo de erro que pode não se manifestar em todas as execuções do programa.

Obs Não há nenhum sinal ou palavra-chave para marcar o fim das instruções que são controladas pela parte `if`. É somente a presença da parte `else`, ou a presença de alguma outra instrução que não está indentada, que sinaliza o fim das instruções controladas pelo `if`.

Exercício 2.20 Identifique o erro no seguinte código:

```
x = input('Digite um número inteiro: ')
x = int(x) # Conversão para inteiro

if x > 0:
print('O valor {} é positivo.'.format(x))
else:
    print('O valor {} não é positivo.'.format(x))
```

Solução. A linha de código que aparece imediatamente abaixo da parte `if` da estrutura de decisão não está indentada. O efeito prático disso é que o Python não vai encontrar nenhuma linha de código sendo controlada pela parte `if`. Isto é, para o Python não haverá nenhum código a ser executado quando a condição do `if` for verdadeira. E isso está em conflito com a regra do Python, que é: após os dois pontos devemos ter uma ou mais linhas de código indentadas em relação ao `if`. ■

Obs O erro no Exercício 2.20 viola as regras de sintaxe do Python. Isso quer dizer que o programa está escrito incorretamente e que não poderá ser executado em sua forma atual. Para corrigi-lo, é suficiente indentar a linha que aparece imediatamente abaixo da parte `if`.

Exercício 2.21 Identifique o erro no seguinte código:

```
idade = input('Digite sua idade: ')
idade = int(idade)

if idade > 0:
    print('Você possui {} anos de idade.'.format(idade))
else:
    print('O número digitado foi {}.'.format(x))
print('Esta não é uma idade válida.')
print('A não ser que você tenha usado uma máquina do tempo.')
```

Solução. Este é um erro lógico. O programa atende a todas as regras de sintaxe do Python, mas quando analisamos seu funcionamento percebemos algo estranho. Veja a Figura 2.9.

Perceba que as linhas 8 e 9 não estão indentadas em relação ao `else`. Conforme as regras que discutimos, ao perceber que a linha 8 não está indentada em relação ao `else`, o Python conclui que a única linha de código controlada pelo `else` é a linha 7. Logo, as linhas 8 e 9 serão executadas independente da condição do `if` ser verdadeira ou falsa. Isso não ocasiona um erro de sintaxe, mas as mensagens exibidas são incorretas porque dão a entender que a idade 16 é inválida! Para corrigir o código, é suficiente indentar as linhas 8 e 9. ■

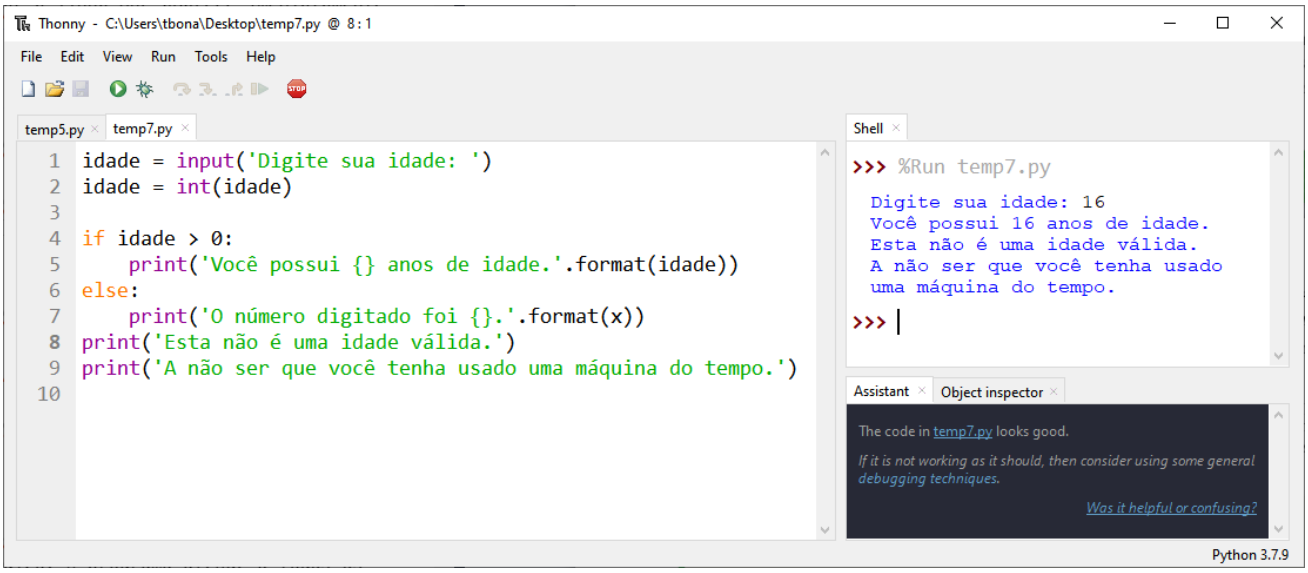

Figura 2.9: Erro lógico causado por indentação.

Como errar é uma boa forma de aprender, vamos ver mais um possível tipo de erro causado por uma indentação inadequada.

Exercício 2.22 Encontre o erro no seguinte código:

```
idade = input('Digite sua idade: ')
idade = int(idade)

if idade > 0:
    print('Você possui {} anos de idade.'.format(idade))
else:
    print('O número digitado foi {}.'.format(x))
print('Esta não é uma idade válida.')
    print('A não ser que você tenha usado uma máquina do tempo.')
```

Solução. Neste caso, temos um erro lógico e também um erro de sintaxe! O erro lógico é o mesmo do exercício anterior. Existe um comando `print` que deveria estar indentado, mas não está!

Mas, existe também um erro de sintaxe. O Python não aceita uma indentação do código sem um motivo. Para ele, a presença de uma estrutura de decisão, ou de uma estrutura de repetição, ou de uma função (assuntos que veremos ainda) são cenários em que é obrigatório usar a indentação para deixar claro quais instruções estão controladas por estas estruturas. Note que a linha do último `print` está indentada, mas sem nenhuma estrutura de decisão envolvida. Afinal, a linha do penúltimo `print` já sinaliza que a estrutura `if-else` foi concluída na linha anterior. Para corrigir esta situação é suficiente indentar a linha do penúltimo `print`. ■

Exercício 2.23 Escreva um programa que pergunta o signo de uma pessoa e imprime algumas informações de horóscopo, de acordo com o signo digitado. Suponha que você tem informações para dois signos apenas, e que o programa irá exibir uma mensagem de desculpas, caso o signo digitado seja diferente.

Solução. Uma possível solução é mostrada abaixo. Nela, exibimos algumas mensagens sobre os signos Áries e Leão, mas mostramos uma mensagem de erro para qualquer outro conteúdo

diferente deste dois.

Primeiramente, vamos entender a lógica do programa. Após ler o signo a partir do teclado, o programa compara o signo lido com a palavra "`áries`". Se a condição do `if` for verdadeira, então o programa imprime três frases sobre o signo de Áries. Caso contrário, iremos para a parte `else` que, por sua vez, contém uma segunda estrutura do tipo `if-else`. Essa é uma novidade!

```
signo = input('Digite seu signo: ')

if signo.lower() == 'áries':

    print('Você é uma pessoa determinada.')
    print('Você valoriza suas amizades.')
    print('Esta semana lhe trará uma grande surpresa.')

else:

    if signo.lower() == 'leão':
        print('Você é uma pessoa justa.')
        print('Você leva suas responsabilidades a sério.')
        print('Você está iniciando um período de sorte no amor.')
    else:
        print('Desculpe. Não tenho informações para seu signo.')
```

O segundo `if-else` funciona de maneira parecida com o primeiro. A condição verifica é se o signo é igual a "`áries`". Se for, então três frases serão impressas. Caso contrário, entramos na parte `else`, que simplesmente imprime uma mensagem de desculpas.

Perceba que todo o código da segunda estrutura `if-else` está completamente indentado em relação à primeira estrutura `if-else`. Isto quer dizer que a indentação é sempre relativa a onde o código aparece: como o segundo `if-else` já está numa região com um nível de indentação, o código controlado pelas partes `if` e `else` aparece com dois níveis de indentação.

Outra novidade neste exemplo é que usamos a operação `signo.lower()`. Esta operação resulta em um nova string, obtida a partir da string `signo`, mas com todas as letras transformadas em minúsculas. Esta operação é útil quando queremos comparar o valor de duas strings, sem levar em conta a diferença entre maiúsculas e minúsculas. Assim, `Áries` é considerado igual a `áriEs`, e assim por diante. ■

2.7.3 – Condições compostas utilizando operadores lógicos

Até o momento trabalhamos com algumas condições bem simples nas nossas estruturas do tipo `if`. Agora, veremos como escrever condições lógicas compostas por dois ou mais testes.

No Python, os conectivos lógicos "E" (conjunção), "OU" (disjunção inclusiva) e "NÃO" (negação) correspondem às palavras `and`, `or` e `not`, nesta ordem. Não há nada de especial no uso destes conectivos no Python. Uma vez que tenhamos em mente a expressão lógica que desejamos testar, basta substituir cada conectivo pela sua palavra correspondente.

■ **Exemplo 2.81** O código a seguir pergunta o peso e a idade do usuário e utiliza a seguinte regra para determinar se o usuário pode ou não doar sangue: para doar, é necessário pesar no mínimo

50 quilos e ter idade entre não inferior a 16 e não superior a 69.

```
peso = input('Digite seu peso: ')
peso = float(peso)
idade = input('Digite sua idade: ')
idade = int(idade)

if peso >= 50 and idade >= 16 and idade <= 69:
    print('Você pode doar sangue.')
else:
    print('Você não pode doar sangue.')
```

■

■ **Exemplo 2.82** Um código levemente diferente realiza a mesma tarefa, mas testando se os dados do usuário violam alguma das condições. Abaixo, vemos apenas a parte do `if-else`.

```
if peso < 50 or idade < 16 or idade > 69:
    print('Você não pode doar sangue.')
else:
    print('Você pode doar sangue.')
```

Perceba que simplesmente fizemos a negação da condição original do Exemplo 2.81. Agora, a parte `if` corresponde ao caso de não doador, enquanto a parte `else` corresponde ao caso de doador. ■

Vejamos um exemplo com uma condição mais elaborada.

■ **Exemplo 2.83** As regras para determinar se um ano é bissexto podem ser escritas da seguinte maneira:

- A cada 4 anos temos um ano bissexto.
- De 100 em 100 anos não é ano bissexto.
- De 400 em 400 anos é ano bissexto.
- Prevalecem as últimas regras em relação às primeiras.

Essa pode parecer uma maneira um pouco complicada de descrever as coisas. Que tal tentarmos reescrever as regras de uma maneira equivalente, mas que fiquem mais parecidas com nossas estruturas de `if-else`? Primeiramente, sabemos que o ano obrigatoriamente precisa ser múltiplo de 4 para ser bissexto; isto é um pré-requisito (ou uma **condição necessária**). Isto significa que qualquer ano que não é múltiplo de 4 é bissexto.

Agora, partindo do fato de que o ano é múltiplo de 4, podemos distinguir dois casos: se o ano é múltiplo de 100 e não é múltiplo de 400, então ele não é bissexto; os demais anos são bissextos.

- Se o ano é múltiplo de 4, então:
  - Se o ano não é múltiplo de 400 e é múltiplo de 100, então ele não é bissexto.
  - Caso contrário, o ano é bissexto.
- Caso contrário, o ano não é bissexto.

Agora, parece possível transcrever de maneira bem direta as condições que escrevemos em português para expressões correspondentes em Python.

```
ano = input('Digite o ano desejado: ')
ano = int(ano)

if ano % 4 == 0:
    if ano % 100 == 0 and not ano % 400 == 0:
        print('O ano {} não é bissexto.'.format(ano))
    else:
        print('O ano {} é bissexto.'.format(ano))
else:
    print('O ano {} não é bissexto.'.format(ano))
```

Obs Lembre-se de que o operador `%` do Python nos fornece o resto da divisão inteira. Portanto, quando escrevemos, por exemplo, "`ano % 100 == 0`", estamos verificando se a variável `ano` é múltipla de 100 (isto é, ao dividí-la por 100, obtemos resto zero).

■

2.7.4 – A parte `elif` da estrutura de decisão

Além das partes `if` e `else`, o Python possui uma terceira palavra-chave: `elif`. O `elif` é um recurso interessante quando queremos testar algumas possibilidades diferentes dentro da mesma estrutura de decisão. Enquanto a estrutura `if-else` nos permite dividir o fluxo de execução em duas partes, com o `elif` podemos dividir o fluxo de execução em duas, três, ou mais partes.

A parte `elif` deve estar na mesma indentação da parte `if` da estrutura de decisão. Além disso, o `elif` possui uma sintaxe semelhante à da parte `if`: ela é seguida por uma condição, um caractere "`:`", e uma ou mais linhas indentadas em relação à palavra `elif`.

Obs A palavra `elif` é uma abreviação de "`else if`". Assim como acontece com a parte `else`, a parte `elif` não pode ser utilizada sozinha; ela só pode ser usada em conjunto com o `if`.

Exercício 2.24 Reescreva o programa do Exercício 2.23, utilizando a parte `elif`.

Solução. O programa a seguir realiza a mesma tarefa da solução do Exercício 2.23, mas de uma maneira mais breve e talvez mais fácil de entender.

```
signo = input('Digite seu signo: ')

if signo.lower() == 'áries':
    print('Você é uma pessoa determinada.')
    print('Você valoriza suas amizades.')
    print('Esta semana lhe trará uma grande surpresa.')
elif signo.lower() == 'leão':
    print('Você é uma pessoa justa.')
    print('Você leva suas responsabilidades a sério.')
    print('Você está iniciando um período de sorte no amor.')
else:
    print('Desculpe. Não tenho informações para seu signo.')
```

Ao usarmos a estrutura `elif` no código acima, estamos distinguindo entre duas possibilidades que são coerentes com a ideia de que `signo.lower()` não é igual a `'áries'`:

(a) `signo.lower()` é igual a `'leão'`; ou
(b) `signo.lower()` é diferente de `'leão'`.

O `elif` propriamente dito trata do caso em que `signo.lower()` é diferente de `'áries'` e `signo.lower()` é igual a `'leão'`, enquanto o `else` trata do caso em que `signo.lower()` é diferente de `'áries'` e também diferente de `'leão'`. ■

Agora, imagine que você deseja criar um programa que imprime um horóscopo para cada signo. É possível utilizar várias partes `elif` em sequência! Ou seja, o `elif` pode aparecer mais de uma vez na mesma estrutura de decisão, permitindo várias divisões do fluxo de execução. Vamos ver um exemplo.

Exercício 2.25 Escreva um programa que lê dois números reais a partir do teclado e oferece ao usuário a opção de fazer a soma dos dois números, a diferença do primeiro pelo segundo, ou o produto entre os dois.

Solução. Uma solução deste exercício é usar uma condição para cada uma das opções possíveis. O código a seguir faz isso.

```
x = input('Digite o primeiro número real: ')
y = input('Digite o segundo número real: ')

x = float(x) # Conversão para número real
y = float(y) # Conversão para número real

print('Escolha uma das opções:')
print('    (a) soma;')
print('    (b) diferença do primeiro pelo segundo;')
print('    (c) produto.')
opcao = input('Opção: ')

if opcao == 'a':
    print('Soma = {}'.format(x + y))
elif opcao == 'b':
    print('Soma = {}'.format(x - y))
elif opcao == 'c':
    print('Soma = {}'.format(x * y))
else:
    print('Lamento. Opção inválida.')
```

O código acima pode ser descrito da seguinte forma:

- Se a opção escolhida foi (a), então imprima `x + y`;
- Se a opção escolhida foi (b), então imprima `x - y`;
- Se a opção escolhida foi (c), então imprima `x * y`;
- Se a opção escolhida não foi (a), nem (b), nem (c), então imprima uma mensagem de erro.

■

2.7.5 – O `if-else` em linha

O Python possui uma versão interessante da estrutura de decisão `if-else` que nos permite simplificar bastante nosso código em algumas situações. Esta estrutura é chamada de **`if-else` em linha** porque ela é escrita de tal maneira que pode ocupar uma única linha de código. Apesar de ser mais resumida, ela segue o funcionamento do `if-else` que já estudamos. Sua presença na linguagem Python é apenas um acréscimo, cujo objetivo é tentar facilitar a vida do programador.

Considere o Exemplo 2.84.

■ **Exemplo 2.84** O código mostrado abaixo é muito parecido com aquele que discutimos no Exemplo 2.80. Ele calcula o valor absoluto de um número fornecido pelo usuário.

```
x = input('Digite um número inteiro: ')
x = int(x) # Conversão de string para inteiro

if x >= 0:
    modulox = x
else:
    modulox = -x

print('O módulo de {} é {}.'.format(x, modulox))
```

Perceba que a estrutura `if-else` tem o papel de atribuir um valor adequado à variável `modulox`.

O seguinte código é perfeitamente equivalente ao código acima.

```
x = input('Digite um número inteiro: ')
x = int(x) # Conversão de string para inteiro

modulox = x if x >= 0 else -x

print('O módulo de {} é {}.'.format(x, modulox))
```

A linha "`modulox = x if x >= 0 else -x`" deve ser lida da seguinte maneira: "Se `x` for maior ou igual a zero, `modulox` recebe o valor de `x`; caso contrário, `modulox` recebe o valor `-x`". ■

O formato geral do `if-else` em linha é mostrado abaixo, com `condicao` sendo alguma expressão lógica (verdadeiro/falso), e `valor1` e `valor2` sendo quaisquer dois objetos do Python (por exemplo, `int`, `float`, `str`, `list`, etc):

```
    valor1 if condicao else valor2
```

A expressão toda assume um de dois possíveis valores: `valor1` ou `valor2`. Se `condicao` tiver valor verdadeiro, então a expressão toda assume o valor `valor1`; caso `condicao` tenha valor falso, a expressão toda assume o valor `valor2`. Este funcionamento é coerente com o funcionamento da estrutura `if-else` que já estudamos anteriormente, e equivalente ao que escrevemos no código mais acima.

Normalmente, esta variante da estrutura de decisão `if-else` é utilizada quando desejamos escolher um de dois possíveis valores para atribuir a uma mesma variável. Foi exatamente isso que aconteceu no Exemplo 2.84 acima, com a variável `modulox`.

O seguinte exemplo é um pouco mais complicado, mas mostra como este tipo de estrutura de decisão pode simplificar muito a escrita de código.

■ **Exemplo 2.85** Suponha que desejamos obter o **sinal** de um dado número real fornecido pelo usuário. A função sinal pode ser definida da seguinte maneira:

$$\text{sinal}(x) = \begin{cases} 1, & \text{se } x > 0; \\ -1, & \text{se } x < 0; \\ 0, & \text{caso contrário.} \end{cases}$$

O código a seguir lê um número real a partir do tecaldo e calcula seu sinal. Ao final, o sinal do número é impresso na tela.

```
x = input('Digite um número real: ')
x = float(x) # Conversão de string para float

sinalx = 1 if x > 0 else (-1 if x < 0 else 0)

print('O sinal de {} é {}.'.format(x, sinalx))
```

O uso de parênteses na expressão que atribui um valor à variável `sinal` é opcional. Mas, ele torna a leitura da expressão mais fácil. Seu significado pode ser descrito assim: "se `x` for positivo, atribua a `sinalx` o valor 1; se `x` for negativo, atribua a `sinalx` o valor -1"; caso contrário, atribua a `sinalx` o valor zero.

É interessante fazer o exercício de tentar escrever um código equivalente ao mostrado acima, mas sem utilizar a estrutura `if-else` em linha. O número de linhas do programa resultante seguramente será maior do que o do programa acima. ■

Obs

Apesar de ser uma ferramenta poderosa, o `if-else` em linha nem sempre é usado na prática. Seu uso é inteiramente opcional e, às vezes, pode ser mais legível usar uma estrutura comum de `if-else`. A legibilidade de nosso código é sempre algo que deve estar em nossa mente. Afinal, quanto mais claro for nosso código, menor será a chance de haver um erro nele, e melhores serão as chances de encontrarmos e corrigirmos um erro, se existir algum.

2.8 – Estruturas de repetição

Assim como acontece com a estrutura de decisão, as estruturas de repetição são fundamentais para descrever processos, principalmente aqueles mais complexos. Nesta Seção, vamos estudar as formas de escrever laços de repetição em Python.

O Python possui dois tipos de laços de repetição: `for` e `while`, além de dois comandos que flexibilizam o comportamento dos laços de repetição: `break` e `continue`.

2.8.1 – O laço de repetição `for`

O laço `for` é provavelmente o laço de repetição mais frequentemente utilizado no Python. Ele pode ser utilizado em conjunto com qualquer objeto iterável do Python, e se aplica naturalmente a uma grande quantidade de situações.

A sintaxe do `for` é bem simples:

```
for objeto1 in objeto2:
    instrução(ões)
```

No código acima, `objeto1` representa normalmente uma variável e `objeto2` necessariamente deve ser algum objeto iterável (*iterable*, em inglês) do Python. Dentre os tipos de dados que já estudamos, alguns estão nesta categoria: `list`, `str`, `set`, `dict`. Todos estes são formados por várias partes: uma string é formada por caracteres individuais, uma lista por valores individuais, e assim por diante. A expressão `instrução(ões)` representa uma ou mais linhas de código que são controladas pelo laço `for`.

A interpretação da estrutura `for` pode ser escrita da seguinte forma: "para cada valor presente em `objeto2`, atribua o valor à variável `objeto1` e execute `instrução(ões)`".

■ **Exemplo 2.86** O seguinte código imprime os valores presentes na lista `L`:

```
L = [1, 3, 5, 7, 3]

for val in L:
    print(val)
```

Ao executarmos este programa, os valores 1, 3, 5, 7 e 3 são exibidos na tela, nesta ordem. Isto confirma nossa descrição do `for`. Para o valor 1 presente na lista, atribuimos à variável `val` o valor 1 e executamos a única instrução que é controlada pelo `for`: o comando `print(val)`. Depois disso, tratamos o segundo valor na lista, 3. A variável `val` recebe o valor 3 e novamente imprimos seu valor na tela. Dessa forma, imprimimos uma vez cada um dos cinco valores em `L`.

Ou seja, o laço de repetição `for` cria uma situação em que o fluxo de controle do programa fica preso dentro daquele conjunto de linhas, até que todos os elementos da lista `L` tenham sido processados. Quando chegamos nesse ponto, o fluxo de execução do programa abandona a estrutura `for` e segue para a próxima linha de código após o laço. ■

Obs

A variável `val` no Exemplo 2.86 é chamada de **variável de controle** do laço `for`. Para cada valor que a variável de controle assume, temos uma **iteração**, que é sinônimo de **repetição**. Uma iteração completa consiste tanto da atribuição de valor à variável de controle quanto da execução das instruções controladas pelo `for`.

Observe algumas semelhanças entre o `for` e a estrutura de decisão `if`: há um caractere "`:`"ao final da linha do `for`; as linhas controladas pelo `for` ficam abaixo da linha inicial e indentadas. Aqui, a indentação é tão séria quanto no contexto de estruturas de decisão. O código que é controlado pelo `for` é determinado pelo mesmo tipo de indentação que conhecemos.

O Exemplo 2.87 mostra uma situação em que temos mais de uma linha de código controladas pelo laço de repetição `for`.

■ **Exemplo 2.87** O código a seguir imprime uma única vez cada um dos elementos pertencentes à lista. Juntamente com o elemento, imprimimos também o número de vezes que ele aparece na lista `L`.

Figura 2.10: Laço de repetição que controla duas linhas de código.

■

Quando usamos a operação `set(L)`, estamos criando um conjunto (tipo `set`) a partir dos elementos de `L`. Como conjuntos não possuem repetições, o resultado desta operação é um conjunto quatro elementos: o elemento 3 não aparece repetido. Este fato nos garante que o `for` irá imprimir cada elemento de `L` uma única vez. Já a expressão `L.count(val)` nos permite obter o número de ocorrências do elemento `val` na lista `L`.

Perceba a indentação em ambas as linhas de código controladas pelo `for`. As duas estão avançadas em relação à linha inicial do `for`.

Obs Há diferença importante entre os Exemplos 2.86 e 2.87. No Exemplo 2.86, sabemos a ordem com que os elementos serão impressos, pois `L` é uma lista, e listas possuem uma ordem entre seus elementos. Como conjuntos (tipo `set`) não possuem uma ordem entre seus elementos, não temos como antecipar a ordem de impressão dos dados no Exemplo 2.87.

Além de ser utilizado para percorrer elementos de listas e conjuntos, o `for` é muito utilizado também com objetos do tipo `range`. O Exemplo 2.88 mostra um exemplo típico.

■ **Exemplo 2.88** O seguinte código imprime os números de 0 a 9 na tela. Lembre-se de que `range(a,b)` enumera os números inteiros a partir de `a` até `b`, mas sem incluir o valor `b`.

```
for x in range(0,10):
    print(x)
```

■

Para imprimir os números naturais pares menores que 10, podemos escrever simplesmente:

```
for x in range(2,10,2):
    print(x)
```

■ **Exemplo 2.89** Suponha que temos um conjunto de valores numéricos e que desejamos somar todos eles. E suponha que desejamos também calcular e imprimir a média destes valores. O seguinte código realiza esta tarefa, utilizando os valores armazenados em uma lista `S`.

```
S = [4.2, 80, 17, 9.03, 42]

somatorio = 0
for x in S:
    somatorio += x

media = somatorio / len(S)
print('Média = {}'.format(media))
```

No código acima, temos uma variável `somatorio`, que inicialmente possui valor igual a zero. Em cada iteração do laço, atualizamos o valor de `somatorio`, somando a ela mais um dos valores armazenados em `S`. Ao final, temos nesta variável a soma de todos os elementos de `S`, e podemos utilizá-la para calcular a média aritmética destes elementos. ■

Muitas vezes, desejamos selecionar alguns dos elementos de um conjunto ou lista. O Exemplo 2.90 mostra uma situação comum, em que combinamos um laço de repetição `for` com uma estrutura de decisão.

■ **Exemplo 2.90** O seguinte código imprime os valores pares armazenados na lista `A`.

```
A = [5, 4, 15, 26, 9, 6, 3, 1, 8]

for x in A:
    if x % 2 == 0:
        print(x)
```

Perceba a indentação: toda a estrutura de decisão está indentada em relação à linha inicial do `for`. Mas, além disso, a linha `print(x)` está ainda mais à frente, pois ela precisa estar mais indentada em relação à linha inicial do `if`. Aqui, vemos novamente a importância da indentação para comunicar precisamente ao Python o que desejamos. ■

Outra situação muito frequente de uso do laço `for` é quando desejamos enumerar os índices de algum objeto como uma string ou uma lista. Vejamos uma destas situações no Exemplo 2.91.

■ **Exemplo 2.91** O código a seguir imprime todos os índices da string `palavra` que possuem uma vogal. Por simplicidade, não estamos considerando vogais com acentos.

Uma vez que obtemos o tamanho da string `frase` digitada, por meio da operação `len(frase)`, podemos escrever um laço de repetição que gera todos os valores de 0 até `tamanho-1`. Estes são os índices da string `frase`. Em seguida, obtemos a letra que existe naquele índice. Por fim, verificamos se a letra pertence à lista de vogais que desejamos detectar. Se pertencer, imprimimos o índice da string onde a letra aparece. ■

Figura 2.11: Identificando as posições de uma string que contêm vogais.

Exercício 2.26 Escreva um programa que lê uma frase a partir do teclado e cria um dicionário, associando cada caractere que aparece na frase com o número de ocorrências daquele caractere na frase. Ao final, imprima o conteúdo do dicionário, mostrando cada caractere juntamente com seu número de ocorrências.

Solução. No Exercício 2.87, fizemos uso do tipo `set` para armazenar cada elemento único de uma lista, isto é, sem repetições. Podemos fazer exatamente o mesmo com os elementos de uma string.O código da Figura 2.12 realiza a tarefa solicitada.

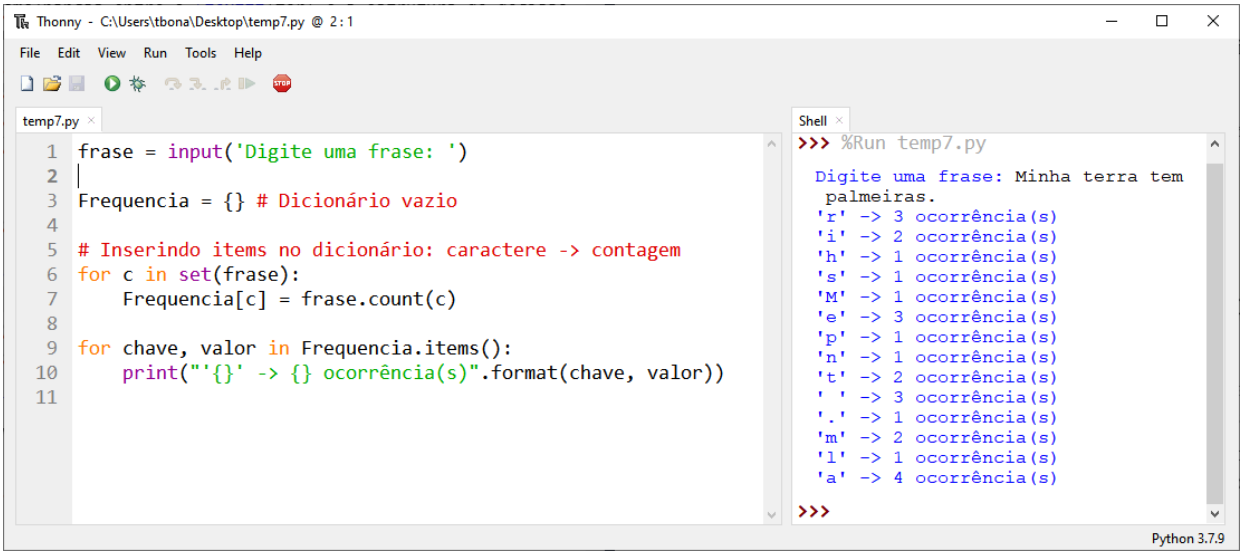

Figura 2.12: Laço `for` para percorrer os elementos de um dicionário: operação `.items()`.

Ao fazer `set(frase)`, o programa cria um conjunto com cada um dos caracteres que aparecem na variável `frase`. Lembre-se de que o Python diferencia caracteres maiúsculos de minúsculos e que ele trata espaços em branco e pontos como caracteres também.

Um laço `for` é dedicado a preencher o dicionário com pares do tipo (`chave`,`valor`), onde `chave` é um caractere e `valor` é o número de ocorrências do caractere, obtido por meio da instrução `frase.count(c)`. Esta variável `c` é a variável de controle do `for`, e em cada iteração ela irá assumir o valor de um dos caracteres que ocorre na frase.

O segundo laço de repetição `for` tem o objetivo de imprimir na tela o conteúdo do dicionário `Frequencia`. Isto é feito por meio da operação `Frequencia.items()`, que fornece um objeto iterável contendo todos os pares (`chave`,`valor`) que estão armazenados em `Frequencia`.

Portanto, o segundo laço de repetição deste exemplo nos apresenta uma situação nova: cada elemento de `Frequencia.items()` é um par, e em vez de uma só variável de controle, temos duas: `chave` e `valor`. ■

2.8.2 – O laço de repetição `while`

O outro laço de repetição do Python é o laço `while`. Podemos dizer que o laço `for` se aplica naturalmente sempre que temos algum tipo de conjunto ou lista (tecnicamente, um objeto iterável) que desejamos percorrer. Já o laço `while` não percorre um objeto iterável, mas permanece repetindo uma sequência de instruções até que certa condição deixe de ser verdadeira. Sua sintaxe é a seguinte:

```
while condição:
    instrução(ões)
```

No código acima, a expressão `condição` corresponde a qualquer expressão lógica. Já a expressão `instrução(ões)` representa uma sequência de uma ou mais linhas de código que são controladas pelo laço de repetição `while`. Assim como no caso do laço de repetição `for`, estas linhas de código controladas pelo laço `while` devem estar indentadas em relação à linha inicial do `while`.

O funcionamento do laço `while` pode ser descrito da seguinte maneira: "enquanto a expressão `condição` tiver valor igual a `True`, execute `instrução(ões)`". Isto significa que, antes de mais nada, o valor de `condição` é avaliado. Se este valor for `True`, teremos uma execução da sequência de instruções em `instrução(ões)`. Esta verificação da condição, seguida da execução das instruções controladas pelo laço constituem uma iteração completa do laço `while`.

Se o valor de `condição` for `False`, as linhas de código em `instrução(ões)` não serão executadas e o fluxo de controle do progama irá para a próxima linha após o laço `while`. Portanto, é o valor de `condição` que determinar até quando o laço `while` permanece executando.

Como a verificação do valor de `condição` acontece no início de cada iteração, é possível que `condição` possua valor `False` logo no início da primeira iteração. Neste tipo de situação, nenhuma iteração é concluída e a parte `instrução(ões)` não é executada nenhuma vez.

■ **Exemplo 2.92** O seguinte código realiza uma tarefa equivalente àquela do Exemplo 2.88: imprime o números inteiros de 0 a 9.

```
x = 10
i = 0
while i < x:
    print(i)
    i = i + 1
```

Note que a condição que controla o laço `while` verifica se a variável `i` é menor que a variável `x`. Ora, inicialmente temos `i` com o valor 0, enquanto `x` tem valor igual a 10. Além disso, perceba que o valor de `x` nunca é modificado ao longo do programa, enquanto o valor de `i` aumenta ao final de cada iteração do laço `while`. Logo, após algumas iterações, o valor de `i` deixará de ser menor que o valor de `x`. É neste momento que a condição `i < x` se torna igual a `False` pela primeira vez, e é este o momento em que o laço `while` se encerra e o fluxo de execução segue para a próxima linha de código após o laço. ■

Obs A condição que controla o funcionamento do laço `while` (a expressão "`i < x`" no Exemplo 2.92 acima) é muitas vezes chamada de **condição de término** do laço. É quando ela se torna igual a `False` que o laço `while` termina.

Exercício 2.27 Reescreva o código do Exemplo 2.90.

Solução. No Exemplo 2.90, o laço `for` percorria diretamente os elementos da lista `A`. Com o laço `while` não podemos fazer exatamente o mesmo. A maneira mais simples de contornar este problema é acessar os elementos da lista utilizando índices, isto é, `A[0]`, `A[1]`, e assim por diante. Podemos criar uma variável para fazer o papel de índice, e fazê-la aumentar gradualmente dentro de um laço `while`. O código a seguir funciona desta maneira.

```
A = [5, 4, 15, 26, 9, 6, 3, 1, 8]

tamanho = len(A)
indice = 0

while indice < tamanho:
    x = A[indice]
    if x % 2 == 0:
        print(x)
    indice = indice + 1
```

Em cada iteração do laço `while`, temos um valor diferente da variável `indice`. Isso pode ser visto no programa porque `indice` inicia com valor igual a zero e, a cada iteração do laço `while`, fazemos `indice = indice + 1`. Quando `indice` se torna igual a `tamanho`, o laço de repetição é interrompido, já que os índices da lista vão desde 0 até `tamanho-1`.

Em cada iteração, guardamos na variável `x` o valor do elemento que ocupa a posição `indice` da lista. Após isso, fazemos o mesmo teste (instrução `if`) e a mesma impressão (`print`) que fizemos no Exemplo 2.90. ■

Embora seja possível reescrever qualquer laço `while` por meio de um laço `for`, e vice-versa, há situações em que o laço `while` se aplica mais naturalmente do que o laço `for`. Vejamos um exemplo.

■ **Exemplo 2.93** O código a seguir lê números inteiros a partir do teclado, até que o usuário digite um valor igual a zero. Qualquer valor positivo ou negativo é aceito, mas quando o valor zero é digitado, o programa simplesmente se encerra, imprimindo uma mensagem de despedida.

```
x = input('Digite um número inteiro (digite 0 para sair): ')
x = int(x)
```

```
while x != 0:
    print('Obrigado.')
    x = input('Digite um número inteiro (digite 0 para sair): ')
    x = int(x)

print('Adeus!')
```

A indentação das linhas controladas pelo laço de repetição `while` segue exatamente a mesma regra que vimos para o laço de repetição `for`. Perceba como as três linhas de código controladas pelo `while` estão todas indentadas em relação à linha inicial do `while`. E perceba como a linha que imprime a mensagem final não está neste mesmo nível de indentação, mas sim no nível de indentação da linha inicial do `while`. Como resultado, esta linha é executada somente quando o fluxo de controle abandona o laço `while`. ■

■ **Exemplo 2.94** No Exercício 2.25, utilizamos uma estrutura de decisão (envolvendo as partes `if`, `elif` e `else`) para construir um tipo de menu na tela. Dependendo da escolha do usuário, o programa faria uma dentre três operações possíveis. Seria interessante, agora que sabemos usar laços de repetição no Python, escrever um menu que seja exibido novamente, até que o usuário escolha uma opção que indica o desejo de encerrar o programa. O código a seguir constrói um menu mais simples do que o do Exercício 2.25, mas que se repete enquanto o usuário não digita a opção de saída.

```
opcao = ''
while opcao != 'd':
    print('Escolha o animal que você deseja ouvir.')
    print('Digite (a) para bode;')
    print('Digite (b) para galinha;')
    print('Digite (c) para peru;')
    print('Digite (d) para sair do programa.')

    opcao = input('Opção: ')

    print('')

    if opcao == 'a':
        print('Béééé...')
    elif opcao == 'b':
        print('Cocoricóóó...')
    elif opcao == 'c':
        print('Gluglugluuu...')
    elif opcao == 'd':
        print('Encerrando o programa...')
    else:
        print('Opção inválida. Tente novamente, por gentileza.')

    print('')

print('Fim do programa.')
```

■

Sobre a condição de término do laço `while`

No laço `for`, normalmente sabemos que o número de iterações a serem realizadas é finito. Isso se dá porque estamos caminhando sobre um objeto iterável, que possui uma quantidade limitada de elementos.

Com o laço `while`, precisamos de certo cuidado para garantir que um dia o laço será encerrado. No Exemplo 2.92, sabíamos que isso aconteceria porque `i` aumentava a cada iteração do `while`, e nunca tinha seu valor diminuído. Além disso, o valor de `x` nunca era modificado. Logo, mais cedo ou mais tarde, a condição "`i < x`" seria igual a `False` e o laço terminaria.

Algo parecido acontecia no Exemplo 2.93. A condição de término do laço era `x != 0` e a cada iteração um novo número era lido a partir do teclado. Isto significa que o fim da execução do laço dependia do valor digitado pelo usuário. Quando este valor fosse igual a zero, o laço terminaria.

Aqui, vale a pena fazer uma advertência. Se o laço `while` não for planejado com cuidado, é possível que a condição de término permanecça sempre igual a `True`, dando origem ao que chamamos de um **laço infinito**. Por exemplo, se removermos a linha "`i = i + 1`" do Exemplo 2.92, temos uma situação em que o valor de `i` nunca é modificado, permanecendo igual a 0 ao longo das iterações. Portanto, teremos uma situação em que o laço `while` nunca terminará, já que não há nada no programa que faça com que, em algum momento, a condição `i < x` passe a ter valor igual a `False`.

2.8.3 – O comando `break`

Nas seções anteriores, estudamos os laços de repetição do Python e como eles terminam. Para o laço `for`, vimos que sua execução termina quando o objeto iterável já teve todos os seus valores inspecionados. Ou seja, quando a variável de controle já assumiu todos valores que deveria assumir. Já no laço `while`, a condição de término está explícita na linha inicial do laço de repetição: o laço termina quando a avaliação daquela condição retorna o valor `False`. Em algumas situações, podemos desejar especificar condições adicionais para o término de um laço de repetição. É nesse tipo de contexto que usamos o comando `break`.

O comando `break` pode ser usado tanto com laços de repetição do tipo `for` quanto com laços do tipo `while`. Quando este comando é executado, o fluxo de execução do programa muda radicalmente: o fluxo é transferido para a primeira linha de código após o laço de repetição que contém o `break`. Vamos ver um exemplo.

■ **Exemplo 2.95** O seguinte código percorre uma lista de números inteiros, em busca de um valor ímpar. Se existir um valor ímpar, uma mensagem é exibida, comunicando este fato. Independentemente da lista conter ou não algum valor ímpar, uma mensagem de despedida é sempre exibida na tela antes do fim do programa.

```
L = [6, 2, 18, 0, 14, 9, 22, 17]

for x in L:
    if x % 2 == 1:
        print('A lista contém um valor ímpar!')
        break

print('Adeus!')
```

Perceba que a indentação do comando `break` comunica que ele é controlado pela estrutura de decisão `if`. Ou seja, somente se for encontrado um número ímpar é que o laço será interrompido pelo `break`. Antes da interrupção propriamente dita, uma mensagem é impressa na tela. Quando o `break` é executado, o fluxo do programa vai diretamente para a instrução "`print('Adeus!')`", mesmo que ainda haja uma ou mais iterações do `for` a serem feitas. Ou seja, o `break` interrompe prematuramente a execução do laço. O `break`, portanto, ignora a condição de término natural do laço de repetição. ■

Obs

No Exemplo 2.95, se removermos o comando `break`, a mensagem de que a lista possui um valor ímpar será impressa duas vezes: uma para o valor 9 e outra para o valor 17. Se for este o intuito (imprimir tantas mensagens quantos forem os números ímpares), então não há razão para usar o `break`.

O exemplo a seguir mostra que o `break` se aplica apenas ao laço de repetição mais próximo dele, isto é, ao laço de repetição dentro do qual o `break` aparece, mesmo que este laço de repetição esteja dentro de outro laço de repetição (isto é, mesmo que tenhamos laços encadeados).

■ **Exemplo 2.96** O código a seguir percorre uma lista de strings e imprime aquelas que possuem a letra x (minúscula).

```
palavras = ['mesa', 'sal', 'caixa',
            'banana', 'cuscuz', 'axé',
            'sapoti', 'cassaco', 'xaxado']

for p in palavras:
    for letra in p:
        if letra == 'x':
            print(p)
            break
```

O código possui dois laços de repetição. Vamos chamar o laço "`for p in palavras:`" de laço externo e o outro laço ("`for letra in p:`") de laço interno. Em cada iteração do laço externo, a variável `p` assume o valor de uma das palavras na lista `palavras`. A cada iteração do laço interno, a variável `letra` assume o valor de um dos caracteres da palavra `p`.

Se o valor atual da variável `letra` for igual a `'x'`, então o valor atual da variável `p` é impresso na tela e o laço interno é interrompido. Isto significa que o fluxo de execução abandona o laço interno, que contém o `break`, mas permanece dentro do laço externo. Isto faz sentido, já que não é necessário inspecionar as demais letras da palavra `p` atual; em vez disso, vamos seguir imediatamente para a próxima palavra. Afinal de contas, já sabemos que a palavra atual possui um x e esta palavra já acabou de ser impressa uma vez na tela. ■

2.8.4 – Contadores

Uma tarefa muito comum quando estamos escrevendo programas é a de contar o número de vezes que um certo evento acontece, ou a de contar o número de vezes que uma certa condição é verdadeira. Nestes casos, é muito comum utilizarmos uma variável inteira para realizar esta contagem. Considere o seguinte exemplo.

■ **Exemplo 2.97** O código a seguir conta quantos números da lista `L` são ímpares. Note que, diferentemente do que aconteceu no Exemplo 2.95, o programa não usa o `break` ao encontrar o primeiro valor ímpar: afinal, o objetivo é contar todos eles.

```
L = [6, 2, 18, 0, 14, 9, 22, 17]
numeroDeImpares = 0

for x in L:
    if x % 2 == 1:
        numeroDeImpares += 1

print('Encontrei {} valor(es) ímpar(es).'.format(numeroDeImpares))
```

A variável `numeroDeImpares` é comumente chamada de **contador**, porque é exatamente isso que ela faz: conta. Neste exemplo, o contador `numeroDeImpares` conta o número de vezes que a condição "`x % 2 == 1`" foi verdadeira, ao longo de toda a execução do laço de repetição. Um contador é facilmente reconhecido porque ele recebe inicialmente o valor zero e tem seu valor aumentado, mas nunca reduzido. ■

Considere a seguinte variante do Exemplo 2.96.

■ **Exemplo 2.98** Suponha que, no Exemplo 2.96, desejássemos também contar o número de palavras que continham a letra x.

Para realizar esta tarefa, é suficiente criar uma variável para atuar como contador, e aumentar em uma unidade o valor desta variável na linha imediatamente antes do comando `break`. O código a seguir mostra essa modificação.

```
palavras = ['mesa', 'sal', 'caixa',
            'banana', 'cuscuz', 'axé',
            'sapoti', 'cassaco', 'xaxado']

contaPalavrasX = 0 # modificação
for p in palavras:
    for letra in p:
        if letra == 'x':
            print(p)
            contaPalavrasX =+ 1 # modificação
            break

print('Foram encontradas {} palavras com x.'.format(contaPalavrasX))
```

Desta forma, além de imprimir a palavra (com o "`print(p)`"), o programa também registra que encontrou uma (ou mais uma) palavra com x. Ao final do laço externo, temos na variável

`contaPalavrasX` o número de palavras contendo a letra x. ■

2.9 – Funções

A maioria das linguagens de programação modernas (tais como Python, C++, Java, etc) fornecem ao programador a capacidade de organizar código em grupos de instruções chamados de **funções**. Uma função nada mais é do que uma sequência de instruções da linguagem que recebe um nome escolhido pelo usuário. Por meio deste nome, o código da função pode ser **chamado** (isto é, executado) a partir de diferentes partes do programa.

Já utilizamos várias funções do Python a esta altura, tais como a função `len`, que, ao ser utilizada, fornece o número de elementos em alguma coleção (um conjunto, uma lista, etc). O próprio `print` é um exemplo de função, que realiza a tarefa de imprimir certo conteúdo na tela. As funções `len` e `print` são chamadas de funções **nativas** do Python. Isto significa que elas estão disponíveis para uso em qualquer programa, sem haver a necessidade de nenhuma ação particular antes de serem utilizadas. Nesta parte do curso, vamos aprender a criar nossas próprias funções.

Sintaxe

O seguinte código em Python mostra a definição de uma função que calcula a média de dois valores que são fornecidos como parâmetros (ou argumentos):

```
def media(a,b):
    return (a + b)/2
```

A instrução `def` é utilizada para comunicar ao Python que estamos definindo uma nova função. Perceba o caractere de dois pontos (":") ao final da linha "`def media(a,b):`". Observe também a indentação da linha que vem logo abaixo do `def`. Estas características indicam que a definição de uma função segue um esquema similar àquele de um `if` ou um `for`: tudo aquilo que é parte da função fica indentado nas linhas abaixo dela. E o fim da indentação marca o fim da função.

Após o uso de `def`, deve ser especificado o nome da função. Este nome deve seguir as mesmas regras para criação de nomes de variáveis. Além destas partes, temos também a lista de parâmetros da função, que aparecem entre parênteses, imediatamente após o nome da função.

Retorno

Além estes elementos, vemos neste exemplo a instrução `return`, que não havíamos utilizado ainda. Ela informa o valor que a função assume.

Fazendo um paralelo com funções matemáticas, uma função em Python tipicamente vai calcular algum valor com base em um ou mais valores. A instrução `return` faz com que a chamada feita à função seja substituída pelo valor retornado. Isto é semelhante ao que acontece quando escrevemos $f(x)$ em uma expressão matemática para representar um valor calculado a partir do argumento x, de acordo com uma regra ou fórmula. A função `media` mostrada neste exemplo é capaz de calcular, com base em dois valores numéricos fornecidos (os **argumentos** da função), um valor que é a média aritmética dos dois argumentos. Este valor calculado e retornado é chamado de **retorno** da função.

É importante notar que a instrução `return` encerra a execução da função: a função é concluída imediatamente e a chamada feita à função é substituída pelo valor de retorno.

Argumentos (ou parâmetros)

Os argumentos `a` e `b` da função `media` são normalmente chamados de **parâmetros**. Eles são variáveis normais do Python e permitem que o código escrito seja genérico – isto é, ele pode ser utilizado para calcular a média aritmética entre quaisquer dois números. Vejamos o Exemplo 2.99.

■ **Exemplo 2.99** O seguinte código calcula a média aritmética entre dois números lidos a partir do teclado.

```
def media(a,b):
    return (a + b)/2

x = float(input('Digite um número: '))
y = float(input('Digite outro número: '))
m = media(x,y)

print('A média de {} e {} é {}.'.format(x, y, m))
```

Perceba que, além da função `media`, temos um código adicional, que lê dois números reais a partir do teclado, chama a função `media` e imprime uma frase na tela. As instruções adicionais, que não fazem parte da definição da função `media`, são o que comumente chamamos de **programa principal**. O objetivo do programa principal costuma ser o de controlar o funcionamento geral do nosso programa, enquanto as funções são responsáveis por tarefas mais específicas. ■

No Exemplo 2.99, chamamos a função `media` com os parâmetros `x` e `y`. Quando isso acontece, os parâmetros `a` e `b` que aparecem na definição da função recebem cópias dos valores que as variáveis `x` e `y` possuem no ponto em que a chamada da função acontece, isto é, na linha `m = media(x,y)`. Isto significa que o código da função será executado com o valor de `a` igual ao valor de `x` e com o valor de `b` igual ao valor de `y`.

A principal vantagem de se escrever uma função é que o código dela é escrito uma única vez, mas pode ser utilizado várias vezes. Na prática, isso significa que nosso programa pode fazer várias chamadas à função `media`, podendo utilizar diferentes parâmetros em cada chamada. Ou seja, podemos realizar chamadas tais como `media(x,y)`, `media(7,4)`, `media(2,x)`, etc, em um mesmo programa. O que acontece na prática é que cada execução da função `media` é independente de qualquer outra chamada feita a ela. Cada vez que `media` é chamada, novas variáveis `a` e `b` são criadas na memória e recebem cópias dos argumentos fornecidos. Ao final desta execução da função, as variáveis `a` e `b`, que só existem dentro da função, deixam de existir.

Exercício 2.28 Vamos escrever uma função que recebe dois valores numéricos como parâmetros e retorna o maior deles.

Solução. Para isso, vamos escolher um nome informativo para a função. Que tal o nome `maximo`? Assim, fica muito claro qual tarefa a função realiza.

Além do nome, precisamos dizer que a função recebe dois parâmetros. Estes parâmetros também precisam de nomes. No código a seguir, estamos usando os nomes `x` e `y`. O código da função é bem direto: se `x` for maior ou igual a `y`, a função retorna o valor de `x`; caso contrário (`else`), a função retorna o valor de `y`.

```
def maximo(x, y):
    if x >= y:
        return x
```

```
        else:
            return y
```

Esta é apenas uma das maneiras de escrever uma função que realiza esta tarefa. Mai adiante, veremos outras maneiras de escrever funções diferentes, mas que realizam exatamente a mesma tarefa. ■

Exercício 2.29 E se desejarmos encontrar o máximo de três valores numéricos? Será que a função fica muito diferente?

Solução. Novamente, existe mais de um jeito de escrever uma função deste tipo. Uma das maneiras é adaptar um pouco a ideia que utilizamos no exercício anterior. Podemos chamar a função de `maximoDe3`. Precisamos informar ao Python que a função receberá três parâmetros, que podemos chamar de `x`, `y` e `z`.

```
def maximoDe3(x, y, z):
    if x >= y and x >= z:
        return x
    elif y >= x and y >= z:
        return y
    else:
        return z
```

Neste código, testamos se `x` é maior ou igual a `y` e também maior ou igual a `z`. Se o teste for verdadeiro, a função deve retornar o valor de `x`. Se o teste for falso, então já sabemos que o maior dos três valores é `y` ou `z` (o `x` já está fora da jogada). Se `y` for maior ou igual a `x` e maior ou igual a `z`, então a função deve retornar `y`. Por fim, se ambos os testes forem falsos, então sabemos que o maior dos três valores é seguramente `z`. Nesse ponto do código, podemos simplesmente retornar `z` sem precisar realizar nenhum teste adicional. ■

Ausência de parâmetros ou de retorno

Diferentemente do que costuma acontecer quando usamos funções matemáticas, o Python permite definir funções sem parâmetros e também funções que não fornecem um valor de retorno. Vamos ver separadamente cada uma das duas situações.

O seguinte código mostra uma função que pergunta à usuária seu nome e retorna o nome lido. Perceba que a função não recebe nenhum parâmetro, afinal ela não precisa de nenhum valor externo para realizar sua tarefa. Isso fica claro quando olhamos para a linha "`def ler_nome():`" e percebemos os parênteses sem nenhum argumento entre eles.

```
def ler_nome():
    nome = input('Por favor, digite seu nome: ')
    return nome

nome_usuaria = ler_nome()
print('Olá, {}. Sejam bem-vinda(o).'.format(nome_usuaria))
```

Note que a chamada de uma função sempre requer o uso de parênteses, mesmo que a função não receba parâmetros. Isto fica claro quando olhamos para a linha "`nome_usuaria = ler_nome()`".

Um exemplo de função que não retorna nenhum valor é mostrado a seguir. A função `saudar` recebe um nome e imprime na tela uma saudação à usuária, mas não utiliza a instrução `return` para retornar nenhum valor. De fato, seu funcionamento não produz nenhum tipo de resultado que possa ser retornado.

```
def ler_nome():
    nome = input('Por favor, digite seu nome: ')
    return nome

def saudar(nome):
    print('Olá, {}. Sejam bem-vinda(o).'.format(nome))

nome_usuaria = ler_nome()
saudar(nome_usuaria)
```

Obs

É importante observar que o programa principal ficou menor quando incluimos a função "`saudar`". Esta observação reforça a ideia de que funções tendem a simplificar o código e deixá-lo mais legível. O programa principal agora se resume a duas linhas de código, que podem ser facilmente interpretadas pela pessoa que lê o programa.

Modificando o valor de um parâmetro

Como vimos, os parâmetros recebem cópias dos valores que são fornecidos na chamada da função. Por causa deste fato, se o código da função alterar o valor de um parâmetro, isso não causará nenhuma alteração nas variáveis usadas na chamada da função. Vamos ver um exemplo para esclarecer melhor essa afirmação.

■ **Exemplo 2.100** O código na Figura 2.13 contém uma função que retorna o maior de dois valores fornecidos como parâmetros. Vamos entender o funcionamento da função `maximo`. Se o valor do parâmetro `x` for maior ou igual ao valor do parâmetro `y`, então a função retorna o valor de `x`. Caso contrário, a função faz com que `x` receba o valor armazenado em `y` e, depois, retorna este valor. Os valores das variáveis `a` e `b` no programa principal nos permitem afirmar que o valor retornado será o do parâmetro `y`, pois `y` > `x`.

É interessante ver que, após a execução da função, os valores das variáveis `a` e `b` no programa principal não foram alteraods, apesar da instrução `x = y` na linha 3. Isto pode ser visto ao executarmos o programa e percebermos que os comandos `print(a)` e `print(b)` imprimem os valores 9 e 15, nesta ordem. ■

Este mesmo comportamento acontece não apenas quando estamos lidando com parâmetros do tipo `int` (inteiro), mas também com parâmetros dos tipos `float`, `str` e `bool`. Quando lidamos com parâmetros mais complexos (`list`, `set`, `dict`), o Python funciona de maneira diferente: mudanças feitas nestes parâmetros são refletidas nas variáveis fornecidas no momento da chamada da função. O Exemplo 2.101 demonstra este comportamento.

■ **Exemplo 2.101** Na Figura 2.14, a função `removerZeros` foi projetada para receber como parâmetro uma lista. Enquanto a lista contiver algum valor igual a zero, a função remove um destes valores iguais a zero. Ao final da execução da função, a lista terá todo seu conteúdo original, exceto pelos valores iguais a zero que existiam nela.

Figura 2.13: Exemplo de modificação do valor de um parâmetro.

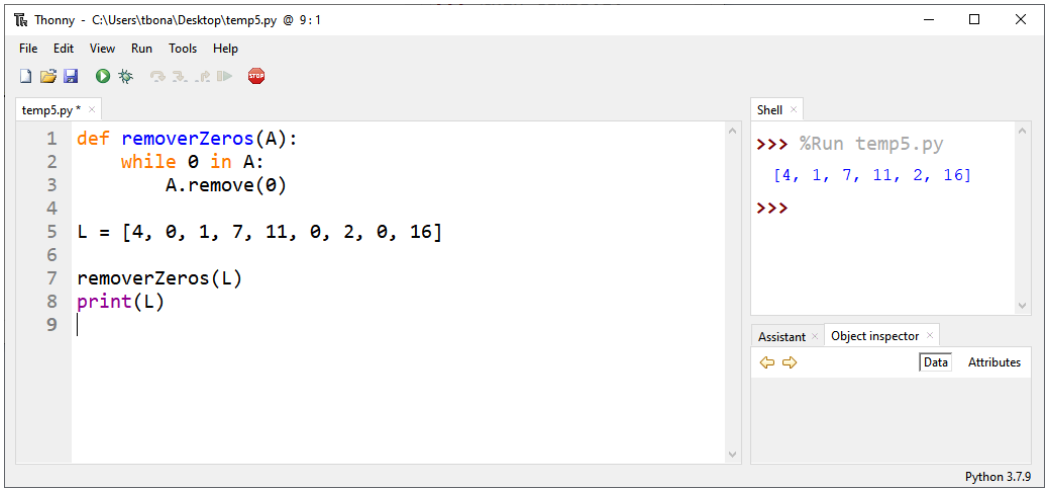

Figura 2.14: Exemplo de modificação do valor de um parâmetro do tipo `list`.

Ao final do programa principal, o conteúdo da lista `L` é impresso na tela, o que deixa claro que s lista foi realmente modificada: todos os zeros foram removidos dela. ■

Funções como subprogramas

Cada uma das funções que vimos funciona como um pequeno programa (por esta razão, funções também são chamadas de **subprogramas** ou **subrotinas**). O significado disso é: da mesma forma que eu posso executar um programa várias vezes, talvez em momentos diferentes do dia e possivelmente utilizando dados diferentes a cada execução, um programa pode executar várias vezes alguma função, também em diferentes momentos e possivelmente utilizando dados diferentes a cada execução.

É possível, inclusive, copiar a função `media` para outro programa e utilizá-la sem haver qualquer necessidade de modificação na definição da função. Trechos de código que são executados mais de uma vez em um mesmo programa são candidatos naturais a se tornarem uma ou mais funções. Em vez de escrevermos mais de uma vez o mesmo trecho de código, podemos criar uma função e simplesmente chamá-la sempre que necessário.

Uma consequência do uso de uma mesma função em vários pontos do código é que, se a função contiver um erro, este erro só precisará ser corrigido uma vez. O outro lado da moeda também é verdade: se cometermos um erro na escrita de uma função que é chamada várias vezes,

o erro acontecerá em vários pontos do programa. Portanto, aqui vale a máxima: "com grandes poderes vêm grandes responsabilidades".

Exercício 2.30 Será que você consegue escrever uma versão mais simples da função `maximoDe3`, desenvolvida no exercício 2.29? Uma sugestão é fazer uso da função `maximo` que retorna o maior de dois valores numéricos.

Solução. Antes de mais nada, vamos nos convencer que a seguinte propriedade é verdadeira: o máximo de três valores numéricos, x, y e z, pode ser calculado por meio do uso da função `maximo`, que retorna o maior de dois valores numéricos. De fato, se calcularmos o máximo de x e y (vamos chamar este máximo de t), então podemos calcular o máximo entre t e z. Este valor final é o máximo dos três valores.

Com isso em mente, podemos escrever o seguinte programa:

```
def maximo(x, y):
    if x >= y:
        return x
    else:
        return y

def maximoDe3(x, y, z):
    t = maximo(x, y)
    return maximo(t, z)

a = 6
b = 9
c = 2
print(maximoDe3(a,b,c))
```

Observe que a função `maximoDe3` primeor calcula o máximo entre `x` e `y`. Em seguida, ela calcula o valor entre este máximo e `z`. Ao final da execução do programa, o valor impresso na tela será 9. Este exemplo ilustra o uso oportunista de uma função que já estava pronta (`maximo`) para facilitar a escrita de uma nova função (`maximoDe3`). ■

Escopo

Quando falamos sobre parâmetros, dissemos que eles são variáveis que existem somente dentro da função onde aparecem. Essa é uma ideia importante, que tem muito a ver com o que chamamos de **escopo**.

Nós dizemos que o **escopo de uma função** é todo o código da função, desde a linha que contém o `def` até a última linha de código que pertence à função. A importância de falar sobre o escopo de uma função é que isso nos permite entender a região do programa onde as variáveis da função existem e são acessíveis (isto é, podem ter seus valores lidos ou modificados).

Por exemplo, quando criamos uma variável dentro de uma função, a variável só existe desde o momento em que ela é utilizada pela primeira vez (o momento de sua criação), até o final do escopo da função. Esta região do código é o **escopo da variável**.

Algo semelhante vale para os parâmetros da função: eles existem desde o momento em que a função começa a executar (a chamada da função) até o momento em que ela termina. Ou

seja, o escopo dos parâmetros coincide com o escopo da função. A figura 2.15 ilustra estes conceitos.

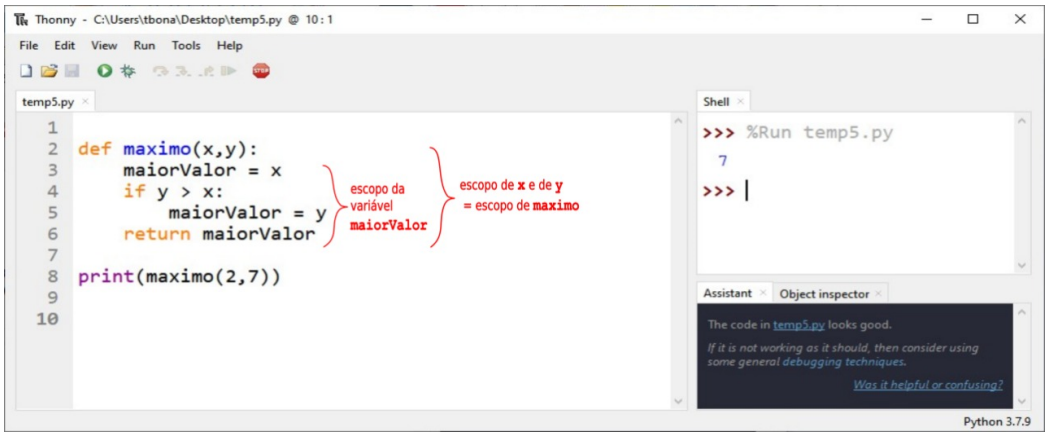

Figura 2.15: Escopo das variáveis de uma função.

Um parâmetro ou uma variável de uma função só pode ser acessado(a) a partir de uma linha de código que esteja dentro de seu escopo. Isto quer dizer que, por exemplo, os parâmetros ou variáveis de uma função não podem ser acessados a partir de outras funções, nem a partir do programa principal. Se tentarmos escrever `print(maiorValor)` na linha 9 do programa da figura 2.15, teremos um erro, pois esta instrução estaria fora do escopo da variável `maiorValor`.

O programa principal tem seu próprio escopo também. As variáveis definidas no programa principal existem desde o primeiro momento em que são utilizadas até o final do programa. Isto significa que uma variável não pode ser utilizada antes de ser criada. Por outro lado, o escopo do programa principal é especial e pode ser acessado a partir de qualquer função. Dessa forma, todas as variáveis do programa principal podem ser acessadas a partir de qualquer função, desde que a chamada da função aconteça dentro do escopo da variável – isto é, depois que a variável já foi criada. A figura 2.16 ilustra esta situação.

Figura 2.16: Escopo das variáveis do programa principal.

Como a chamada da função `saudacao` na linha 6 está dentro do escopo da variável `nome`, a função é capaz de acessar o valor da variável. Se a chamada acontecesse na linha 4 do código, ela seria inválida, pois estaria fora do escopo da variável `nome`.

Esta forma de organizar o acesso aos dados do programa complementa bem o que falamos sobre parâmetros e valores de retorno. Como cada variável está acessível apenas dentro de seu escopo, utilizamos os parâmetros e os valores de retorno para enviar dados de uma parte do programa para outra.

Variáveis com o mesmo nome

O Python permite que duas variáveis tenham o mesmo nome, desde que estejam em escopos independentes. Por exemplo, é possível que duas funções tenham parâmetros com o mesmo nome, sem que haja nenhum conflito. O código a seguir mostra duas funções que possuem parâmetros com os mesmos nomes.

```
def maximo(x, y):
    maiorValor = x
    if y > x:
    maiorValor = y
    return maiorValor

def media(x, y):
    return (x + y)/2
```

A partir do contexto, o Python sabe a qual variável `x` estamos nos referindo cada vez que `x` aparece no código das duas funções. Esta é uma boa notícia, já que não precisamos inventar um nome diferente para cada parâmetro de nossas funções!

Uma situação parecida, mas diferente, é quando uma função utiliza um nome de variável que existe no programa principal. Como o escopo do programa principal é acessível a partir do código das funções, o Python precisa cuidar para saber se estamos nos referindo à variável do programa principal ou a uma variável criada dentro da função. Vamos ver dois exemplos.

■ **Exemplo 2.102** No código a seguir, temos uma função chamada `bomDia`, que recebe um parâmetro `nome` e imprime uma mensagem educada. Para isso, a função utiliza uma variável chamada `mensagem`.

```
def bomDia(nome):
    mensagem = 'Bom dia, ' + nome
    print(mensagem)

mensagem = 'Digite seu nome, por gentileza:'
usuario = input(mensagem)
bomDia(usuario)
```

O programa principal também possui uma variável chamada `mensagem`. Isso é um problema? A resposta é não. Como vimos antes, o programa principal não consegue acessar a variável `mensagem` que existe na função. Mas, a função `bomDia` é capaz de acessar a variável do programa principal. O que acontece neste exemplo é que a instrução "`mensagem = 'Bom dia, ' + nome`" é interpretada pelo Python como sendo a criação de uma variável dentro da função. A partir deste ponto, toda vez que usamos `mensagem` dentro dafunção, o Python entende que estamos falando variável que foi criada no escopo da função. ■

■ **Exemplo 2.103** No código abaixo, temos uma situação um pouco diferente: a função `boaTarde` apenas utiliza o valor da variável `usuario` do programa principal, isto é, ela consulta o valor armazenado na variável. Algo diferente acontece nesse exemplo: o Python compreende que não existe uma variável `usuario` criada dentro da função, e que o programador deseja acessar o valor da variável que existe no problema principal.

```
def boaTarde():
    mensagem = 'Boa tarde, ' + usuario
```

```
        print(mensagem)

    mensagem = 'Digite seu nome, por gentileza:'
    usuario = input(mensagem)
    boaTarde()
```

Dessa forma, o programa imprime a frase "`Bom dia, João`" ou "`Bom dia, Ana`", dependendo do nome que for fornecido por meio do teclado. ■

Exercício 2.31 Escreva um programa que imprime todos os anos bissextos do século 21.

Solução. Para este exercício, é conveniente usarmos o código que fizemos no Exemplo 2.83. Uma ideia é natural é utilizar um laço de repetição para enumerar todos os anos do século 21: $2000, 2001, \ldots, 2099$.

Para cada ano, desejamos utilizar o código do Exercício 2.83 para verificar se trata-se de um ano bissexto ou não. Que tal criarmos uma função que recebe um ano e retorna um valor lógico (verdadeiro/falso) dependendo se o ano é bissexto (valor `True`) ou não (valor `False`).

```
def bissexto(ano):
    if ano % 4 == 0:
        if ano % 100 == 0 and not ano % 400 == 0:
            return False
        else:
            return True
    else:
        return False

for x in range(2000,2100):
    if bissexto(x) == True:
        print('O ano {} é bissexto.'.format(x))
```

Note que ajustamos um pouco o código do Exercício 2.83. Em vez de imprimir uma mensagem na tela, dizendo se o ano é bissexto, apenas retornamos um valor lógico. O código que chama a função `bissexto`, dentro do laço de repetição `for` do programa principal, é responsável pela impressão, caso o valor retornado pela função seja igual a `True`. ■

2.10 – Bibliotecas de programação

O Python é uma linguagem de programação muito popular mundialmente. Esta é uma das grandes qualidades do Python nos dias de hoje, pois, além de existir muito conteúdo online sobre a linguagem, existe também um grande número de usuários que desenvolvem programas em Python.

Com o intuito de compartilhar seus programas com a comunidade mundial de programadores Python, muitos usuários da linguagem produzem o que chamamos de **bibliotecas** para o Python. Uma biblioteca é nada mais do que um conjunto de funções que estão organizadas sob um mesmo nome, o nome da biblioteca. Cada biblioteca costuma conter funções relacionadas a um determinado tipo de tarefa. Por exemplo, podemos ter uma biblioteca para gerar números aleatórios, ou uma biblioteca para manipular imagens, e assim por diante.

Quando decidimos utilizar uma biblioteca, precisamos informar ao Python o nome dela, utilizando a instrução `import`, conforme o código a seguir.

```
import nome_da_biblioteca as nb
```

A partir da execução desta linha de código, as funções da biblioteca passam a estar disponíveis para uso em nosso programa. Isso é muito útil porque podemos nos beneficiar de código produzido por outras pessoas, muitas vezes pessoas que são especialistas em algum assunto, como manipulação de imagens, som, ou análise de dados.

A parte "`as nb`" do código acima simplesmente informa ao Python que iremos nos referir às funções da biblioteca por meio do apelido "`nb`", em vez do nome completo "`nome_da_biblioteca`". A escolha deste apelido é nossa, e deve seguir as mesmas regras para a nomenclatura de variáveis.

Existem literalmente mais de 100 mil bibliotecas para o Python, disponíveis online. A seguir, veremos exemplos de uso de três bibliotecas.

2.10.1 – A biblioteca `random`

A biblioteca `random` é voltada para a geração de valores aleatórios e operações relacionadas. A geração de números aleatórios é muito útil nas áreas de Estatística, Física, e Ciência de Dados, por exemplo. Números aleatórios são usados também para testar programas (simulando diferentes situações e diferentes valores dos dados utilizados por um programa) ou mesmo para o que chamamos de **arte generativa**.

Para começar a utilizar a biblioteca, vamos importá-la para nosso programa:

```
import random as rd
```

A biblioteca `random` possui diversas funções, algumas bastante avançadas. Vamos ver alguns exemplos de funções e criar alguns programas.

A função `randint` da biblioteca `random` recebe como parâmetros dois valores inteiros. Estes dois números inteiros são interpretados como um intervalo. A função retorna um número inteiro dentro deste intervalo. Por exemplo, o seguinte código armazena na variável `x` um número inteiro gerador aleatoriamente dentro do intervalo [3,10]. Ao final, o valor da variável `x` é impresso na tela.

```
import random as rd

x = rd.randint(3,10)
print(x)
```

No código acima, utilizamos `rd.randint` porque é necessário informar ao Python que estamos acessando a função `randint` que pertence à biblioteca `random` (que, neste código, foi apelidada de `rd`). Isso quer dizer que podemos ter difeentes bibliotecas definindo funções diferentes, mas com o mesmo nome. O Python nunca irá se confundir, pois vamos sempre utilizar o nome (ou apelido) da biblioteca antes de chamar alguma de suas funções.

Execite o código algumas vezes, para perceber que, cada vez que o programa é executado,

temos um número que é provavelmente diferente daquele obtido na execução anterior (embora existam repetições algumas vezes).

Outra função útil da biblioteca `random` é a `choice`. Ela recebe um único parâmetro, que deve ser um objeto iterável do Python, tal como uma string, uma lista, um conjunto, ou um objeto do tipo `range`. Considere o seguinte código:

```
import random as rd

L = [4, 1, 8, 7, 12, 3]
x = rd.choice(L)
print(x)
```

A instrução `x = rd.choice(L)` atribui à variável `x` um dos valores da lista `L`. Assim como no código anterior, é interessante executar o código algumas vezes para perceber que, ao longo de várias execuções, diferentes elementos da lista `L` são impressos na tela.

Obs Tanto a função `randint` quanto a função `choice` fazem a escolha do valor a ser retornado de maneira uniforme. Isto significa que, cada vez que a função `randint` é executada com parâmetros `a` e `b`, cada um dos números no intervalo [`a`,`b`] tem a mesma chance de ser retornado pela função. Da mesma forma, quando chamamos a função `choice(L)`, cada um dos elementos da lista `L` tem a mesma chance de ser retornado pela função.

Exercício 2.32 Escreva um programa que gera um jogo aleatório da Mega-Sena.

Solução. Na Mega-Sena, cada jogo é composto por exatamente seis números distintos, escolhidos entre os números inteiros de 1 até 60, incluindo possivelmente os extremos, 1 e 60.

Podemos usar a função `randint` para gerar valores inteiros no intervalo [1,60]. Combinando esta ideia com um laço de repetição, podemos obter seis valores inteiros para formar um jogo completo. No entanto, esta abordagem possui um problema: é possível que o mesmo número seja gerado duas ou mais vezes neste processo. Considere o seguinte código:

```
import random as rd

jogo = []
for i in range(6):
    numero = rd.randint(1,60)
    jogo.append(numero)

print(jogo)
```

A variável `jogo` é uma lista inicialmente vazia. Em cada iteração do laço de repetição, geramos um número inteiro entre 1 e 60, e incluimos este número na lista. Após seis iterações, temos uma lista de seis números no intervalo desejado. Mas, e se houver repetições? O que fazer?

Muitas vezes, o programa irá executar corretamente. Mas, em algumas das execuções, inevitavelmente veremos números repetidos, o que significa dizer que nem sempre o programa gera realmente um jogo válido da Mega-Sena.

No Exemplo 2.87 e no Exercício 2.26, utilizamos o recurso de criar um conjunto a partir de uma lista, com o objetivo de remover elementos duplicados. Podemos fazer o mesmo aqui: em vez de armazenar os elementos selecionados em uma lista, vamos armazená-los em um conjunto (`set`). Assim, se gerarmos um número repetido, a inserção deste número no conjunto não terá nenhum efeito e, portanto, o tamanho do conjunto não aumentará. O código a seguir adota esta ideia.

```
import random as rd

jogo = set() # Cria um conjunto vazio

while len(jogo) < 6:
    numero = rd.randint(1,60)
    jogo.add(numero)

print(sorted(jogo))
```

Além de definir a variável `jogo` como um conjunto, este código utiliza um laço de repetição diferente. Como não sabemos se haverá números repetidos, nem quantos números repetidos serão obtidos, não podemos afirmar de antemão quantas iterações do laço devem ser feitas. Esta é uma situação em que o laço `while` pode ajudar bastante. Em vez de executar exatamente seis iterações, vamos executar quantas iterações forem necessárias até que o conjunto contenha seis elementos (que, garantidamente, serão distintos).

Ao final, o código imprime os elementos do conjunto, utilizando antes a função `sorted` para apresentá-los na tela em ordem crescente. ■

Na solução do Exercício 2.32, utilizamos a linha de código "`numero = rd.randint(1,60)`" para gerar um número inteiro entre 1 e 60. Uma forma equivalente de realizar esta mesma tarefa é:

```
    numero = rd.choice(range(1,61))
```

De fato, na linha de código acima, estamos selecionando aleatoriamente um número da sequência de inteiros iniciada em 1 e concluída em 60 (lembre-se: o `range` não inclui o número que define o final do intervalo, ou seja, o 61 está fora do `range`).

Como falamos anteriormente, a função `choice` da biblioteca `random` funciona com qualquer tipo de objeto iterável do Python. Vejamos mais um exemplo, desta vez com listas de strings.

Antes do exemplo, vamos discutir mais uma função da biblioteca `random`: `shuffle`. O verbo `shuffle`, em inglês, significa **embaralhar**. È exatamente isso que esta função faz: recebe uma lista como parâmetro e modifica permuta os elementos da lista aleatoriamente, isto é, embaralha a lista. O código a seguir mostra um exemplo de uso de `shuffle`.

```
import random as rd

L = [6, 3, 'abc', 100]
rd.shuffle(L)
print(L)
```

Cada vez que o código é executado, uma permutação aleatória dos elementos da lista é realizada e esta versão permutada da lista é impressa. É possível que a permutação obtida seja

exatamente igual ao conteúdo original da lista. Porém, assim como as outras funções que vimos, a permutação obtida pela função `shuffle` é selecionada de maneira uniforme dentre todas as possíveis permutações dos elementos de `L`.

■ **Exemplo 2.104** Suponha que você tem um grupo de alunos e deseja formar um time de futsal. Um time completo de futsal é composto por cinco membros, sendo um deles o goleiro. Imagine que você tem uma lista de alunos que jogam na linha e uma lista de alunos que jogam no gol, isto é, como goleiros. Vamos escrever um programa que monta um time aleatoriamente a partir destas duas listas.

Considere as seguintes listas de alunos:

```
Linha = ['Dé', 'Sá', 'Adeodato', 'Miguel', 'Teo', 'Buiu', 'Ron']
Gol = ['Gê', 'Branco', 'Caio']
```

Precisamos escolher exatamente quatro jogadores da lista `Linha` e exatamente um jogador da lista `Gol`. Com base nestas duas listas, o restante do programa pode ser escrito com inspiração nas ideias do Exercício 2.32. O código a seguir realizar a tarefa desejada.

```
import random as rd

Linha = ['Dé', 'Sá', 'Adeodato', 'Miguel', 'Teo', 'Buiu', 'Edu']
Gol = ['Gê', 'Branco', 'Caio']

# Embaralhar lista de jogadores da linha
rd.shuffle(Linha)

# Escolher quatro jogadores da linha
Time = Linha[:4]

# Escolher goleiro
Time.append(rd.choice(Gol))

print(Time)
```

É interessante notar que este código não possui nenhum laço de repetição `for` ou `while`. Fizemos uso da operação `shuffle` para embaralhar a lista `Linha`. Este embaralhamento simplifica nossa vida, pois agora podemos simplesmente tomar quaisquer quatros elementos da lista permutada (os quatro primeiros, ou os quatro últimos, por exemplo) para compor os jogadores da linha. Para finalizar, usamos `choice` para selecionar um goleiro. ■

2.10.2 – A biblioteca `math`

A biblioteca `math` reune um conjunto de funções matemáticas. Daremos destaque neste material a algumas funções desta biblioteca que podem ser bastante úteis para o desenvolvimento de aplicações que lidam com ângulos, funções trigonométricas, logaritmos, além de outras funções e constantes clássicas da matemática.

O comando para utilizar a biblioteca é:

```
import math
```

Não vamos usar nenhum apelido para esta biblioteca, pois seu nome já é curto o suficiente.

A função `ceil` da biblioteca `math` recebe como parâmetro um valor numérico do tipo `float` e retorna o menor número inteiro maior que ou igual a ele. Por exemplo, o seguinte código armazena na variável `x` o menor número inteiro maior ou igual a 3.45. Ao final, o valor armazenado em `x` é exibido na tela.

```
import math
x = math.ceil(3.45)
print(x)
```

A função `floor` da biblioteca `math` recebe como parâmetro um valor numérico do tipo `float` e retorna o maior número inteiro menor que ou igual a ele. Por exemplo, o seguinte código armazena na variável `x` o maior número inteiro menor ou igual a 6.4. Ao final, o valor armazenado em `x` é exibido na tela.

```
import math
x = math.floor(6.4)
print(x)
```

A função `gdc` recebe dois números inteiros e retorna o máximo divisor comum entre eles. Um código utilizando essa função para obter o MDC entre 13 e 4 pode ser visto a seguir:

```
import math
x = math.gdc(12, 4)
print(x)
```

A função `log` recebe dois números inteiros – o logaritmando e a base – e retorna o logaritmo. Um código utilizando essa função para obter o logaritmo de 8 na base 2 pode ser visto a seguir:

```
import math
x = math.log(8, 2)
print(x)
```

A função `pow` recebe dois parâmetros – uma base e um expoente – e retorna o resultado da exponenciação da base pelo expoente fornecidos. Um código utilizando essa função para calcular a potência 2^5 pode ser visto a seguir:

```
import math
x = math.pow(2,5)
print(x)
```

A função `sqrt` recebe um número e retorna o valor da sua raiz quadrada. Um código utilizando essa função para obter a raiz quadrada de 25 pode ser visto a seguir:

```
import math
x = math.sqrt(25)
print(x)
```

Exercício 2.33 Faça um programa para calcular a distância Euclidiana entre dois pontos no espaço bidimensional.

Solução. Para dois pontos $P = (p_x, p_y)$ e $Q = (q_x, q_y)$, a distância Euclidiana entre eles é dada pela equação

$$d = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}. \tag{2.1}$$

Para calcular a distância entre dois pontos no espaço 2D, nosso programa precisará obter os valores de p_x, p_y, q_y e q_y.

Em nosso programa, vamos escrever uma função para receber 4 parâmetros, que serão as coordenadas `x` e `y` de cada um dos pontos e retornar o valor da distância Euclidiana. Para deixar a escrita da função um pouco mais enxuta, vamos criar uma segunda função, que retornará o resultado do quadrado de uma dada expressão. No programa principal, vamos obter os valores das coordenadas dos dois pontos e exibir o resultado obtido.

```
import math

def quadrado(x):
    return math.pow(x, 2)

def dist2D(px, py, qx, qy):
    return math.sqrt(quadrado(px-qx) + quadrado(py - qy))

Px = input("Digite a coordenada x do ponto P: ")
Py = input("Digite a coordenada y do ponto P: ")
Qx = input("Digite a coordenada x do ponto Q: ")
Qy = input("Digite a coordenada y do ponto Q: ")

print(dist2D(float(Px), float(Py), float(Qx), float(Qy)))
```

■

Vamos ver a seguir algumas funções trigonométricas disponibilizadas pela biblioteca `math`.

A função `sin` recebe um número e retorna o valor do seu seno, em radianos. Um código utilizando essa função para obter o seno de 90 graus pode ser visto a seguir:

```
import math
x = math.sin(3.1415/2) #90 graus, em radianos, é Pi/2
print(x)
```

Seria bem interessante se não precisássemos fazer essa conta para converter de graus para radianos. Felizmente, a biblioteca `math` disponibiliza duas funções para conversão de graus para radianos e vice-versa.

A função para converter de graus para radianos é `radians`. Ela recebe o valor de um ângulo em graus e o transforma em radianos. Com esta função, o código que calcula o seno de 90 graus pode ser melhorado para:

```
import math
x = math.sin(math.radians(90))
print(x)
```

Este código, inclusive, produz um resultado mais preciso, se comparado com o código anterior.

A função para converter de radianos para graus é `degrees`. Ela recebe o valor de um ângulo em radianos e o transforma em graus. Um código utilizando esta função para converter o ângulo $\frac{3\Pi}{2}$ para graus pode ser visto abaixo:

```
import math
x = math.degrees(3*3.1415/2)
print(x)
```

Para aumentar a precisão numérica do resultado obtido pelo código acima, vamos utilizar uma constante disponibilizada pela biblioteca: a constante Π. Na biblioteca `math`, essa constante é escrita como `math.pi`.

Com esta constante, o código que transforma o ângulo $\frac{3\Pi}{2}$ para graus pode ser melhorado para:

```
import math
x = math.degrees(3*math.pi/2)
print(x)
```

Dando continuidade às funções trigonométricas, falta estudarmos as funções `cos` e `tan`. A função `cos` calcula o cosseno de um ângulo em radianos. Um código utilizando essa função para obter o cosseno de 180 graus pode ser visto a seguir:

```
import math
x = math.cos(math.pi) #180 graus, em radianos, é Pi
print(x)
```

A função `tan` calcula tangente de um ângulo em radianos. Um código utilizando essa função para obter a tangente de 48 graus pode ser visto a seguir:

```
import math
x = math.tan(math.radians(48))
print(x)
```

As funções trigonométricas inversas – arco seno, arco cosseno e arco tangente –, também estão disponíveis na biblioteca `math`.

A função `asin` retorna o arco seno de um valor. O resultado produzido é dado em radianos. O valor do parâmetro precisa estar dentro do domínio da função arco seno: [-1, 1]. O resultado produzido está no intervalo entre $\frac{-\Pi}{2}$ e $\frac{\Pi}{2}$.

Um código utilizando essa função para obter o arco seno de 0.5 pode ser visto a seguir:

```
import math
x = math.asin(.5)
print(x)
```

A função `acos` retorna o arco cosseno de um valor. O resultado produzido é dado em radianos. O valor do parâmetro precisa estar dentro do domínio da função arco cosseno: [-1, 1]. O resultado produzido está no intervalo entre 0 e Π.

Um código utilizando essa função para obter o arco cosseno de 0.5 pode ser visto a seguir:

```
import math
x = math.acos(.5)
print(x)
```

A função `atan` retorna o arco tangente de um valor. O resultado produzido é dado em radianos. O resultado produzido está no intervalo entre $\frac{-\Pi}{2}$ e $\frac{\Pi}{2}$.

Um código utilizando essa função para obter o arco tangente de 67 pode ser visto a seguir:

```
import math
x = math.atan(67)
print(x)
```

2.10.3 – A biblioteca `datetime`

O Exercício 2.19 fez uso desta biblioteca para utilizar uma função que detecta a hora do dia. Vamos aproveitar para conhecer um pouco mais sobre essa biblioteca.

Para começar a usar, o comando é:

```
import datetime as dt
```

Perceba que, para esta biblioteca, escolhemos usar um apelido. Será mais prático se referir a ela somente por "dt".

A função `datetime.today` retorna o dia atual. O código a seguir mostra como obter a data atual utilizando essa função. Ao final, o programa exibe a data atual na tela:

```
import datetime as dt
x = dt.datetime.today()
print(x)
```

A data retornada pela função contém ano, mês, dia, hora, minuto, segundo e microssegundos. Podemos aproveitar toda essa informação para extrair partes que nos interessam. Por exemplo, o código abaixo extrai somente a informação sobre a hora:

```
import datetime as dt
x = dt.datetime.today().hour
print(x)
```

A função `datetime.now` tem comportamento parecido com o da função `datetime.today`. Ela também retorna a hora atual do sistema. O código a seguir mostra como obter a data atual

utilizando essa função. Ao final, o programa exibe a data atual na tela:

```
import datetime as dt
x = dt.datetime.now()
print(x)
```

A data retornada pela função também contém ano, mês, dia, hora, minuto, segundo e microssegundos. Para aproveitar essa informação para extrair partes que nos interessam, a notação muda um pouco, se comparada com a função `datetime.today`. O código abaixo mostra como extrair somente a informação sobre o ano:

```
import datetime as dt
x = dt.datetime.now()
print(x.year)
```

A função `datetime` permite definir uma data específica. O código abaixo mostra como armazenar a data de 17 de maio de 2020:

```
import datetime as dt
x = dt.datetime(2020, 5, 17)
print(x)
```

Exercício 2.34 Faça um programa que obtenha o ano de nascimento de uma pessoa e calcule a idade dela.

Solução. Obtendo a informação do ano de nascimento, vamos calcular a idade pela diferença entre o ano de nascimento e o ano atual. O ano atual será obtido automaticamente, pelo função `datetime.today()`. O código abaixo apresenta uma solução para o problema proposto:

```
import datetime as dt
anoNascimento = input("Informe o ano do seu nascimento: ")
anoAtual = dt.datetime.today().year
idade = anoAtual- int(anoNascimento)
print(idade)
```

■