Python Básico

Marcos Roberto Ribeiro

Formação Inicial e Continuada

INSTITUTO FEDERAL
Minas Gerais

+IFMG
Campus Bambuí

Marcos Roberto Ribeiro

Python Básico

1ª Edição

Belo Horizonte

Instituto Federal de Minas Gerais

2022

Pró-reitor de Extensão	Carlos Bernardes Rosa Júnior
Diretor de Programas de Extensão	Niltom Vieira Junior
Coordenação do curso	Marcos Roberto Ribeiro
Arte gráfica	Ângela Bacon
Diagramação	Eduardo dos Santos Oliveira

FICHA CATALOGRÁFICA
Dados Internacionais de Catalogação na Publicação (CIP)

R484p Ribeiro, Marcos Roberto.

Python básico [recurso eletrônico] / Marcos Roberto Ribeiro. – Belo Horizonte : Instituto Federal de Minas Gerais, 2022.

E-book, no formato PDF.
Material didático para Formação Inicial e Continuada.
ISBN 978-65-5876-046-7

1. Formação inicial e continuada. 2. Controle de fluxo. 3. Modularização e tratamento de exceções. I. Título.

CDD 005.13

Catalogação: Douglas Bernardes de Castro - CRB-6/2802

Índice para catálogo sistemático:
Linguagens de programação: Processamento de dados: 005.13

2022
Direitos exclusivos cedidos ao
Instituto Federal de Minas Gerais
Avenida Mário Werneck, 2590,
CEP: 30575-180, Buritis, Belo Horizonte – MG,
Telefone: (31) 2513-5157

Sobre o material

Este curso é autoexplicativo e não possui tutoria. O material didático, incluindo suas videoaulas, foi projetado para que você consiga evoluir de forma autônoma e suficiente.

Caso opte por imprimir este *e-book*, você não perderá a possibilidade de acessar os materiais multimídia e complementares. Os *links* podem ser acessados usando o seu celular, por meio do glossário de Códigos QR disponível no fim deste livro.

Embora o material passe por revisão, somos gratos em receber suas sugestões para possíveis correções (erros ortográficos, conceituais, *links* inativos etc.). A sua participação é muito importante para a nossa constante melhoria. Acesse, a qualquer momento, o Formulário “Sugestões para Correção do Material Didático” clicando nesse *link* ou acessando o QR *Code* a seguir:

Formulário de Sugestões

Para saber mais sobre a Plataforma +IFMG acesse

http://mais.ifmg.edu.br

Palavra do autor

Aprender a programar não é uma tarefa trivial, principalmente, devido à abundância de conceitos envolvidos. Por outro lado, as linguagens de programação são ferramentas essenciais para o desenvolvimento de softwares. Uma das linguagens de programação mais usadas no mundo é a linguagem Python. A linguagem Python é uma linguagem aberta, multiplataforma e utilizada em uma grande variedade de aplicações. O presente curso visa apresentar os conceitos básicos da linguagem Python. Devido à grande popularidade do Python, seu aprendizado por parte dos desenvolvedores pode ser um grande diferencial.

Bons estudos!

Marcos Roberto Ribeiro

Apresentação do curso

Este curso está dividido em quatro semanas, cujos objetivos de cada uma são apresentados, sucintamente, a seguir.

SEMANA 1	- Conhecer os tipos básicos de dados e funcionamento de variáveis; - Desenvolver códigos com entrada e saída de dados; - Entender o funcionamento de expressões aritméticas, relacionais, lógicas e suas combinações.
SEMANA 2	- Entender o funcionamento do fluxo de execução do código, das estruturas de decisão e das estruturas de repetição; - Combinar estruturas de decisão e de repetição para resolver problemas; - Desenvolver códigos com estruturas de decisão e de repetição
SEMANA 3	- Entender o funcionamento da tratamento de exceções; - Conhecer o conceito de modularização e decomposição de problemas; - Construir códigos modularizados e com tratamento de exceção.
SEMANA 4	- Conhecer os diferentes tipos de coleções de dados da linguagem Python; - Empregar as coleções de dados mais adequadas na resolução de problemas; - Escrever códigos usado coleções de dados.

Carga horária: 40 horas.

Estudo proposto: 2h por dia em cinco dias por semana (10 horas semanais).

Apresentação dos Ícones

Os ícones são elementos gráficos para facilitar os estudos, fique atento quando eles aparecem no texto. Veja aqui o seu significado:

Atenção: indica pontos de maior importância no texto.

Dica do professor: novas informações ou curiosidades relacionadas ao tema em estudo.

Atividade: sugestão de tarefas e atividades para o desenvolvimento da aprendizagem.

Mídia digital: sugestão de recursos audiovisuais para enriquecer a aprendizagem.

Sumário

Objetivos
- Conhecer os tipos básicos de dados e funcionamento de variáveis;
- Desenvolver códigos com entrada e saída de dados;
- Entender o funcionamento de expressões aritméticas, relacionais, lógicas e suas combinações.

Mídia digital: Antes de iniciar os estudos, vá até a sala virtual e assista ao vídeo “Apresentação do curso”.

1.1 Introdução

As linguagens de programação são ferramentas essenciais para o desenvolvimento de softwares. Uma das linguagens de programação mais usadas no mundo é a linguagem Python[1] (TIOBE, 2021; PYPL, 2021). Devido a grande popularidade do Python, seu aprendizado por parte dos desenvolvedores pode ser um grande diferencial no mercado de trabalho. Além disso, desde sua criação no final da década de 1980, a linguagem passou por uma constante evolução até chegar à versão 3 em 2008 (WIKIPÉDIA, 2021b). A linguagem Python é uma linguagem aberta, multiplataforma e utilizada uma grande variedade de aplicações (MENEZES, 2019; RAMALHO, 2015).

1.1.1 Lógica de programação

Ao mesmo tempo que desenvolvemos códigos utilizando uma linguagem Python, precisamos entender a lógica de programação por trás desses códigos. No estudo da lógica de programação, normalmente, utilizamos algum tipo de pseudocódigo para escrever algoritmos. Os algoritmos são sequências finitas de instruções para resolver determinados problemas. Apesar de não parecer, podemos descrever na forma de algoritmos várias tarefas do nosso, como, por exemplo, fazer uma receita culinária ou analisar se um aluno foi aprovado. Basicamente, um algoritmo recebe uma entrada, efetua um processamento e retorna um resultado. No caso de uma receita culinária, os ingredientes são a entrada, a execução da receita é o processamento, e o prato pronto é o resultado.

Os algoritmos podem ser representados de diversas maneiras, como descrição narrativa, fluxogramas ou códigos Python. Assim, para escrever um algoritmo corretamente é preciso entender a sintaxe e a semântica da linguagem. A sintaxe define as regras de escrita, ou seja, se determinada instrução está escrita corretamente. Por outro lado, a semântica é o significado da instrução.

1 https://www.python.org/

Na descrição narrativa os algoritmos são escritos com o mínimo de formalidade usando a linguagem natural (GOMES; BARROS, 2015). Apesar de não haver grande formalidade, a descrição narrativa deve ser uma sequência de frases curtas, claras e objetivas. É recomendável usar apenas um verbo por frase. A Figura 1 apresenta um exemplo de algoritmo na forma de descrição narrativa para verificar a aprovação de um aluno considerando a médias de duas notas. Contudo, a descrição narrativa não é muito usada na prática. Isso acontece porque a linguagem natural pode gerar ambiguidade e sua execução por computador é muito mais complexa do que a execução de instruções em linguagens formais.

1) Somar as duas notas do aluno
2) Calcular a nota final dividindo a soma por dois
3) Se a nota final for maior que 60, o aluno foi aprovado
4) Senão, o aluno foi reprovado

Figura 1 – Algoritmo para verificar aprovação de aluno em descrição narrativa
Fonte: Elaborado pelo Autor.

Os fluxogramas usam símbolos específicos para as ações dos algoritmos e setas para conectar esses símbolos. A Figura 2 mostra os símbolos usados em fluxogramas bem como seus respectivos significados.

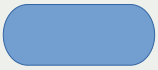
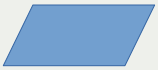
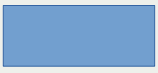
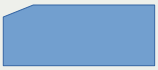
Símbolo	Significado
	Terminador (início e fim do algoritmo)
	Leitura (entrada de dados)
	Processo (processamento de dados)
	Escrita (saída de dados)
	Decisão (desvio do fluxo de execução)
	Conector (junção de fluxos de execução)

Figura 2 – Símbolos usados em fluxogramas
Fonte: Elaborado pelo Autor.

A Figura 3 apresenta novamente o algoritmo para verificar a aprovação de um aluno através de um fluxograma. Podemos observar que no fluxograma o algoritmo requer mais rigor formal do que na descrição narrativa. Após o *início*, nos paralelogramos, ocorre a entrada de dados das notas do aluno (*N1* e *N2*). Em seguida, no retângulo, a média (*M*) é calculada com a expressão $M \leftarrow (N1 + N2)/2$. O próximo passo é o losango que verifica se a média está acima de *60*. Em caso afirmativo, a mensagem *Aprovado* é apresentado. Caso contrário, a mensagem *Reprovado* é mostrada. Os dois fluxos de execução se encontrar no conector e o algoritmo é finalizado.

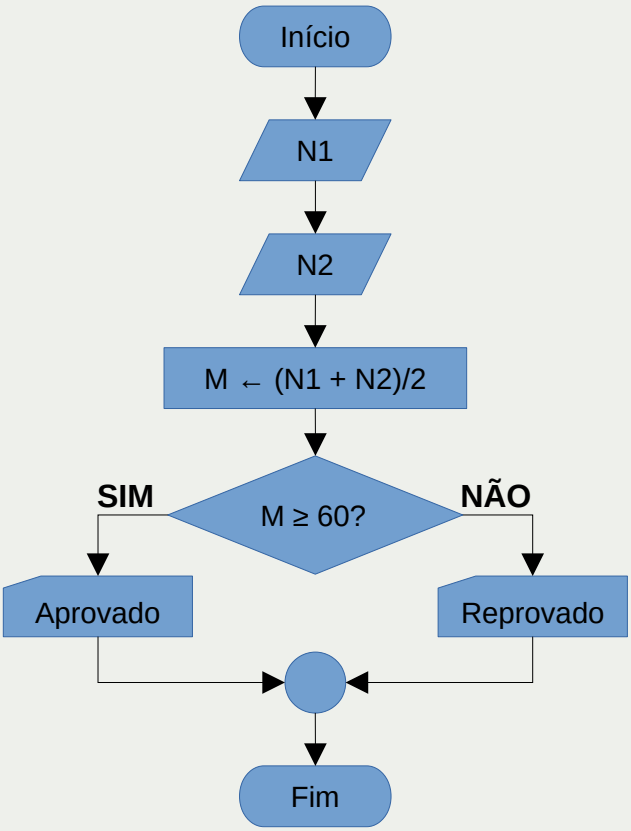


Figura 3 – Algoritmo para verificar aprovação de aluno em fluxograma
Fonte: Elaborado pelo Autor.

A Figura 4 apresenta o código para verificar a aprovação de aluno na linguagem Python. Observe que as linhas de código são numeradas para facilitar a identificação de instruções ou possíveis erros. Por exemplo, na linha 5 temos a instrução **print('Aprovado')** que escreve a mensagem de aprovado na tela. Ao longo do curso, aprenderemos outras instruções da linguagem Python.

```
1 n1 = float(input('Informe nota 1: '))
2 n2 = float(input('Informe nota 2: '))
3 m = (n1 + n2) / 2
4 if m >= 60:
5     print('Aprovado')
6 else:
7     print('Reprovado')
```

Figura 4 – Código Python para verificar aprovação de aluno
Fonte: Elaborado pelo Autor.

1.1.2 A ferramenta Spyder

Na maioria das distribuições do sistema operacional Linux, o ambiente Python já vem instalado por padrão. Para outros sistemas operacionais, como Windows e MacOS, é preciso baixar e instalar esse ambiente. Contudo, no presente curso, utilizaremos a ferramenta Spyder[2] que já conta com o ambiente Python. No Linux, a instalação da ferramenta está disponível nos sistemas de pacotes (TAGLIAFERRI, 2018). No caso do

2 http://www.spyder-ide.org

Windows e MacOS, é recomendável utilizar a instalação *standalone* disponível no site https://docs.spyder-ide.org/current/installation.html. A versão padrão do ambiente Python usada pelo Spyder é a versão 3.

O Spyder é uma ferramenta livre e multiplataforma para desenvolvimento Python. A vantagem de utilizar essa ferramenta são os diversos recursos disponíveis na mesma como editor de código, console, explorador de variáveis, depurador e ajuda. O editor de código possui um explorador de código e autocomplemento. O console permite a execução de comandos Python. Com o explorador de variáveis é possível visualizar e modificar o conteúdo das variáveis. O depurador auxilia na execução passo a passo do código e o sistema de ajuda fornece documentação de comandos da linguagem.

1.2 Tipos de dados

Os códigos em Python são usados para resolver problemas do mundo real e, ao serem executados, precisam trabalhar com diversos tipos de dados. A Figura 5 apresenta os tipos de dados básicos da linguagem Python (PSF, 2021). Os tipos de dados são usados principalmente para representar varáveis e literais. Os Literais são os dados informados literalmente no código. As variáveis são explicadas na próxima seção.

Tipo	Descrição
int	Números inteiros
float	Números fracionários
bool	Valores lógicos como **True** (verdadeiro) ou **False** (falso)
str	Valores textuais

Figura 5 – Tipos básicos de dados em Python
Fonte: Elaborado pelo Autor.

1.3 Variáveis

Uma variável representa uma posição de memória no computador cujo conteúdo pode ser lido e alterado durante a execução do código (BORGES, 2010). Em Python, deferente de outras linguagens de programação, não precisamos declarar as variáveis. As variáveis são criadas automaticamente quando ocorre uma atribuição de valor com o operador de igualdade (**=**). A atribuição de valores a variáveis pode ser realizada com literais, outras variáveis, resultados de expressões e valores retornados por funções.

A Figura 6 mostra um exemplo de código com os quatro tipos básicos de dados. No caso dos tipos textuais, é importante colocar o valor a ser atribuído entre aspas. Aqui cabe uma observação sobre a função **print()**. Podemos colocar quantos literais ou variáveis desejarmos separados por vírgula. A função irá escrever todos na tela. No caso das variáveis, a função escreve seu valor na tela (PILGRIM, 2009).

```
1 n1 = 10
2 n2 = 2.5
3 nome = 'José'
4 teste = True
5 print(n1, n2, nome, teste)
```

Figura 6 – Atribuição de valores a variáveis
Fonte: Elaborado pelo Autor.

Para que não ocorram erros no código, o nome de uma variável devem obedecer às seguintes regras:

- Deve obrigatoriamente começar com uma letra;
- Não deve possuir espaços em branco;
- Não pode conter símbolos especiais, exceto o sublinhado **(_)**;
- Não deve ser uma palavra reservada (uma palavra da já existente na linguagem, como **print**, por exemplo).

Dica do Professor: A linguagem Python, assim como diversas outras linguagens, é *case sensitive*. Nessas linguagens, a mudança de maiúscula para minúscula, ou vice-versa, modifica o nome da variável (ou outro elemento). Assim, os nomes **teste** e **Teste** são considerados distintos.

1.4 Depuração de código

Alguns recursos interessantes da ferramenta Spyder são os pontos de parada, a execução passo a passo e o explorador de variáveis. Tais recursos são muito úteis para fazer a depuração do código e encontrar possíveis erros de funcionamento.

Os pontos de paradas podem ser ativados com um clique duplo sobre o número de cada linha. Aparece um pequeno círculo vermelho antes do número da linha, se o ponto de parada foi ativado. Assim, quando o código é executado no modo depurar, o Spyder interrompe a execução no ponto de parada ativado.

A execução em modo depurar é feita pelo menu *Depurar / Depurar* ou CTRL-F5 (pelo teclado). O explorador de variáveis é o painel no lado esquerdo. Quando o código for executado, você consegue ver o valor de cada variável. A execução passo a passo, no menu *Depurar / Executar linha* ou CTRL+F10, permite executar uma linha do código de cada vez. A utilização dos recursos explicados de forma conjunta pode ser muito útil para encontrar possíveis erros no funcionamento dos códigos.

A Figura 7 mostra um exemplo de depuração de código feito no Spyder. Podemos ver que foi marcado um ponto de parada na linha 5. Os valores e tipos das variáveis podem ser vistos no explorador.

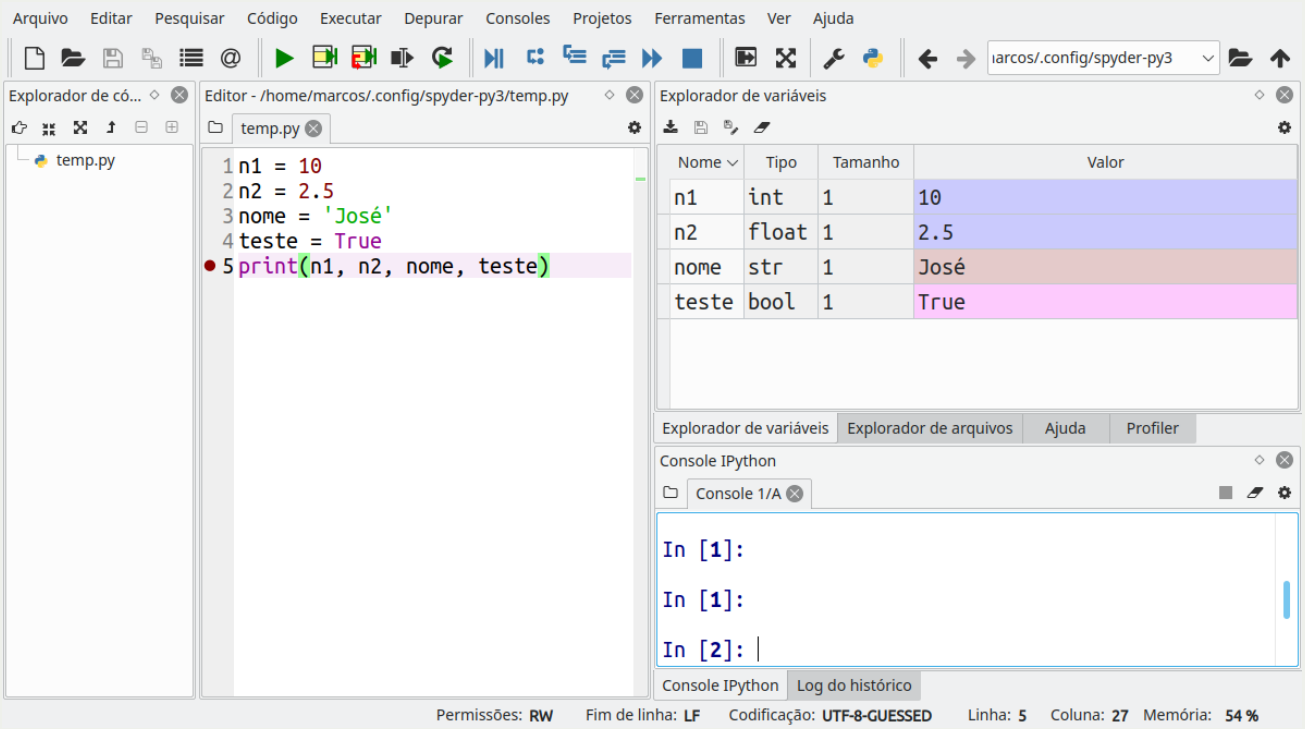


Figura 7 – Depuração de código no Spyder
Fonte: Elaborado pelo Autor.

1.5 Legibilidade

Um dos principais cuidados na escrita de códigos é a ter uma boa legibilidade. Um código escrito por você deve ser facilmente entendido por outra pessoa que vá estudá-lo ou usá-lo (ou por você mesmo no futuro). Alguns fatores que influenciam diretamente o entendimento de códigos são nomes sugestivos, endentação e comentários.

Os nomes sugestivos para variáveis e outras estruturas são importantes para que você identifique de forma rápida a referida variável ou estrutura. A endentação diz respeito aos espaços em branco a esquerda para alinhar e organizar os blocos de instruções. No caso do Python, a endentação é essencial porque é por meio dela que colocamos blocos de códigos dentro de outras instruções. Já os comentários textos não executáveis com a finalidade de explicar o código (SWEIGART, 2021). Os comentários começam com o caractere **#.**

```
1 # -*- coding: utf-8 -*-
2
3 n1 = 10        # Número inteiro
4 n2 = 2.5       # Número fracionário
5 nome = 'José' # Textual (sempre entre aspas)
6 teste = True  # Variável lógica
```

Figura 8 – Código com comentários
Fonte: Elaborado pelo Autor.

A Figura 8 um exemplo de código com comentários em Python. Nas linhas 3 a 6, temos comentários comuns explicando as atribuições. Já na linha 1, temos um comentário especial usado para especificar a codificação de caracteres usada no arquivo. Nesse

exemplo e nos demais ao longo do livro usaremos sempre a codificação UTF8. A linguagem Python diversos outros comentários especiais usados para outras funções.

1.6 Entrada e saída de dados

Quando um algoritmo recebe dados do usuário, dizemos que ocorre uma entrada de dados. De forma análoga, quando o algoritmo exibe mensagens para o usuário, acontece uma saída de dados. Em Python, a entrada de dados é efetuada com a instrução **input()**. Dentro dos parênteses colocamos um texto para ser mostrado ao usuário antes da entrada dos dados. Normalmente, usamos uma variável para receber a resposta digitada. A função **input()** sempre retorna um valor textual digitado pelo usuário. Se precisarmos de outro tipo de dado, temos que realizar uma conversão[3].

No caso da saída de dados, temos a instrução **print()**. Essa instrução recebe as informações a serem mostradas para o usuário separadas por vírgula. Se usarmos uma variável, será mostrado o valor dessa variável. Em diversas situações é importante compor mensagens adequadas para o usuário combinando literais e variáveis. A Figura 9 exibe um código usando as funções **input()** e **print()**. Na linha 5, usamos a instrução **int()** para converter texto retornado pelo **input()** para um número inteiro.

```
1 print('Bem vindo!')
2 # Nome do usuário
3 nome = input('Informe seu nome: ')
4 # Idade convertida para inteiro
5 idade = int(input('Informe sua idade: '))
6 print() # Escreve uma linha em branco
7 # Escreve mensagem usando as variáveis
8 print('Olá,', nome, 'sua idade é', idade)
9 print('Até mais!')
```

Figura 9 – Código usando **input()** e **print()**
Fonte: Elaborado pelo Autor.

A instrução **print()**, por padrão, separa os elementos recebidos com um espaço em branco e, no final, escreve o caractere de nova linha (**'\n'**). Esse comportamento pode ser modificado através dos parâmetros **sep** (texto para separar os elementos) e **end** (texto a ser colocado no final da lina). Por exemplo, se quisermos que os elementos fiquem colados e a saída não passe para a próxima linha, podemos usar uma instrução no seguinte formato **print(..., sep='', end='')**. As reticências (...) devem ser substituídas pelos elementos a serem escritos na tela.

1.7 Operadores

Assim como a maioria das linguagens de programação, a linguagem Python possui operadores aritméticos, relacionais e lógicos (PSF, 2021). Com esses operadores é possível criar expressões que também podem ser combinadas, principalmente, para a realização de testes.

3 Podem ocorrer erros nessa conversão, mas trataremos desse detalhe.

1.7.1 Operadores aritméticos

A realização de cálculos matemáticos está entre as principais tarefas feitas por algoritmos. Para executarmos tais cálculos, devemos utilizar os chamados operadores aritméticos. A Figura 10 mostra os operadores aritméticos disponíveis na linguagem Python. A ordem precedência dos operadores aritméticos é a mesma ordem precedência da matemática. Também podem ser usados parênteses para alterar a ordem de precedência. A Figura 11 mostra um exemplo simples com expressões matemáticas usando os operadores aritméticos.

Operador	Descrição
+	Adição
-	Subtração
*	Multiplicação
/	Divisão
//	Divisão inteira
%	Resto de divisão
**	Potenciação

Figura 10 – Operadores aritméticos do Python
Fonte: Elaborado pelo Autor.

```
 1 ni = 10 // 3
 2 print('10 // 3 =', ni)
 3 ni = 10 % 3
 4 print('10 % 3 =', ni)
 5 nr = 2.34 * 3.58
 6 print('2.34 * 3.58 =', nr)
 7 nr = 2.99 / 4.1
 8 print('2.99 / 4.1 =', nr)
 9 nr = (10.5 - 7.8) * (3.2 + 200.43)
10 print('(10.5 - 7.8) * (3.2 + 200.43) =', nr)
```

Figura 11 – Código usando operadores aritméticos
Fonte: Elaborado pelo Autor.

Com o conteúdo visto até o momento, podemos fazer um código para simular uma calculadora simples. A ideia é obter dois números com o usuário e mostrar os resultados das possíveis operações aritméticas entre esses números. A Figura 12 mostra uma possível solução.

```
 1 print('Informe dois números')
 2 n1 = float(input('Primeiro número: '))
 3 n2 = float(input('Segundo número: '))
 4 r = n1 + n2
 5 print(n1, '+', n2, '=', r)
 6 r = n1 - n2
 7 print(n1, '-', n2, '=', r)
 8 r = n1 * n2
 9 print(n1, '*', n2, '=', r)
10 r = n1 / n2
11 print(n1, '/', n2, '=', r)
```

Figura 12 – Código para simular calculadora simples
Fonte: Elaborado pelo Autor.

O operador **+** pode ser usado também para concatenar variáveis e literais textuais. No entanto, não é possível concatenar diretamente um elemento textual e um elemento numérico. Nesse caso, é necessário converter o valor numérico para textual antes da concatenação. O código da Figura 13 demonstra como isso pode ser feito.

```
1 print('Informe os dados')
2 nome = input('Nome: ')
3 sobrenome = input('Sobrenome: ')
4 idade = int(input('Idade: '))
5 mensagem = nome + ' ' + sobrenome + ', ' + str(idade)
6 print(mensagem)
```

Figura 13 – Código para simular calculadora simples
Fonte: Elaborado pelo Autor.

Em Python, é comum utilizar os operadores aritméticos compostos. Eles são equivalentes à aplicação do operador seguido de uma atribuição. A Figura 14 apresenta os operadores compostos, suas formas de utilização e equivalência. Por exemplo, se temos uma variável **x** valendo **10** e usamos a expressão **x += 2**, estamos somando **2** à variável **x**, ou seja, é o mesmo que usar a expressão **x = x + 2**.

Operador	Forma de utilização	Equivalência
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
//=	x //= y	x = x // y
%=	x %= y	x = x % y
**=	x **= y	x = x ** y

Figura 14 – Operadores compostos
Fonte: Elaborado pelo Autor.

1.7.2 Operadores relacionais

Os operadores relacionais são usados para comparar dois valores. Os valores a serem comparados podem ser literais, variáveis ou expressões matemáticas. O resultado

dessa comparação é um valor lógico **True** (verdadeiro) ou **False** (falso). A Figura 15 apresenta os operadores relacionais disponíveis na linguagem Python.

Operador	Descrição
==	Igual
!=	Diferente
<	Menor
<=	Menor ou igual
>	Maior
>=	Maior ou igual

Figura 15 – Operadores relacionais
Fonte: Elaborado pelo Autor.

O código da Figura 16 implementa um comparador de valores que possibilita testar todos os operadores relacionais para dois números informados. Execute o referido código no Spyder e observe o resultado das comparações para diferentes valores.

```
1 n1 = int(input('Informe um número: '))
2 n2 = int(input('Informe outro número: '))
3 print(n1, '==', n2, '->', n1 == n2)
4 print(n1, '!=', n2, '->', n1 != n2)
5 print(n1, '<',  n2, '->', n1 < n2)
6 print(n1, '<=', n2, '->', n1 <= n2)
7 print(n1, '>',  n2, '->', n1 > n2)
8 print(n1, '>=', n2, '->', n1 >= n2)
```

Figura 16 – Código comparador de valores
Fonte: Elaborado pelo Autor.

1.7.3 Operadores lógicos

Os operadores lógicos são equivalentes aos operadores da Lógica Proposicional. Em Python, temos o operador unário **not** e os operadores binários **and** e **or**. A Figura 17 mostra o funcionamento do operador **not**. Esse operador é usado para obter o oposto de um valor lógico.

Expressão	Resultado
`not True`	`False`
`not False`	`True`

Figura 17 – Operador lógico **not**
Fonte: Elaborado pelo Autor.

A Figura 18 descreve o funcionamento do operador lógico **and**. Podemos observar que esse operador retorna **True** apenas se os dois operandos forem **True**. Ou seja, se um dos operandos for **False**, o resultado do operador também será **False**.

Expressão	Resultado
`True and True`	`True`
`True and False`	`False`
`False and True`	`False`
`False and False`	`False`

Figura 18 – Operador lógico **and**
Fonte: Elaborado pelo Autor.

A Figura 19 exibe o funcionamento do operador lógico **or**. Esse operador retorna **False** somente se os dois operandos forem **False**. Como o resultado de um operador relacionado é um valor lógico, em muitas situações, os operadores lógicos são usados para combinar as comparações relacionais.

Expressão	Resultado
`True or True`	`True`
`True or False`	`True`
`False or True`	`True`
`False or False`	`False`

Figura 19 – Operador lógico **or**
Fonte: Elaborado pelo Autor.

O código da Figura 20 é um exemplo de utilização dos operadores lógicos. Utilize o Spyder para executar esse código algumas vezes informando valores diferentes para as variáveis **n1** e **n2**. Observe funcionamento do código e tente calcular sozinho os valores das variáveis **p**, **q** e **r**.

```
1 n1 = int(input('Informe um número: '))
2 n2 = int(input('Informe outro número: '))
3 p = (n1 > n2)
4 q = (n1 != n2)
5 r = not (p or q) and (not p)
6 print('p =', p)
7 print('q =', q)
8 print('r =', r)
```

Figura 20 – Código com expressões lógicas e relacionais
Fonte: Elaborado pelo Autor.

1.8 Utilizando funções

O Python possui uma biblioteca padrão com funções que podem ser usadas diretamente no código. Além disso, podemos importar diversas outras bibliotecas que podem ser utilizadas para as mais variadas tarefas (BORGES, 2010). A Figura 21 exibe algumas funções da biblioteca padrão. Além da biblioteca padrão, abordaremos funções relacionadas ao tipo textual e também a biblioteca de funções matemáticas.

Função	Funcionamento
abs(n)	Retorna o valor absoluto de **n**
chr(n)	Retorna o caractere representado pelo número **n**
ord(c)	Retorna o código correspondente ao caractere **c**
round(n, d)	Arredonda **n** considerando **d** casas decimais
type(o)	retorna o tipo de **o**

Figura 21 – Algumas funções da biblioteca padrão do Python
Fonte: Elaborado pelo Autor.

1.8.1 Manipulação de dados textuais

Na linguagem Python, o tipo de dado textual (**str**) possui funções relacionadas que auxiliam na manipulação de dados desse tipo (DOWNEY, 2015). Por serem funções associadas ao tipo **str**, é preciso usá-las com o dado desse tipo. Por exemplo, se tempos uma variável **x** do tipo **str**, podem chamar uma função **lower()** com a instrução **x.lower()**. A Figura 23 contém alguns exemplos de funções de manipulação de texto. O código da Figura 26 demostra a utilização de algumas dessas funções.

Função	Funcionamento
s.find(subtexto, ini, fim)	Procura **subtexto** em **s**, começando da posição **ini** até a posição **fim**. Retorna **−1**, se o **subtexto** não for encontrado.
s.format(x_1 , ..., x_n)	Retorna **s** com os parâmetros x_1, ..., x_n incorporados e formatados.
s.lower()	Retorna o **s** com todas as letras minúsculas.
s.replace(antigo, novo, n)	Retorna o **s** substituindo **antigo** por **novo**, nas **n** primeiras ocorrências.

Figura 22 – Algumas funções associadas ao tipo **str**
Fonte: Elaborado pelo Autor.

```
 1 nome_completo = input('Informe seu nome completo: ')
 2 sobrenome = input('Informe seu sobrenome: ')
 3
 4 pos = nome_completo.find(sobrenome)
 5
 6 if pos != -1:
 7     print('Seu sobrenome começa na posição ', pos)
 8 else:
 9     print('Sobrenome não encontrado')
10
11 n = float(input('Informe um número: '))
12 print('{n:.8f}'.format(n=n))
```

Figura 23 – Código com manipulação de dados textuais
Fonte: Elaborado pelo Autor.

1.8.2 Funções matemáticas

Além das funções da biblioteca padrão da linguagem, podemos importar bibliotecas adicionais. Uma dessas bibliotecas é a **math** contendo funções matemáticas (CORRÊA, 2020). A importação de bibliotecas adicionais é realizada com a instrução **import**, normalmente, incluída no início do código. A Figura 24 apresenta algumas das funções disponíveis na biblioteca **math**. Já o código da Figura 25 demostra como tais funções podem ser utilizadas.

Função	Funcionamento
ceil(x)	Retorna o teto de **x**
floor(x)	Retorna o piso de **x**
trunc(x)	Retorna a parte inteira de **x**
exp(x)	Retorna $\mathbf{e}^{\mathbf{x}}$
log(x, b)	Retorna o logaritmo de **x** em uma base **b**. Se a base não for especificada, retorna o logaritmo natural de **x**
sqrt(x)	Retorna a raiz quadrada de **x**
pi	Retorna o valor de π

Figura 24 – Algumas funções da biblioteca **math**
Fonte: Elaborado pelo Autor.

```
1 import math
2
3 n = float(input('Informe um número: '))
4 x = n * math.pi
5 print('x = n * pi = ', n)
6 print('Teto de x =', math.ceil(x))
7 print('Piso de x =', math.floor(x))
8 print('Log de x na base 10 =', math.log(x, 10))
9 print('Raiz de x =', math.sqrt(x))
```

Figura 25 – Código com funções matemáticas
Fonte: Elaborado pelo Autor.

Atividade: Resolva os exercícios a seguir. As respostas estão na próxima página, mas é importante que você tente resolver por conta própria e use as respostas apenas para conferência.

1.9 Exercícios

Escreva os códigos em Python para resolver os problemas a seguir:

A) Calcular a área de um triângulo a partir de sua base e altura.

B) Escreva um código que receba um número de segundos e converta este número em horas, minutos e segundos. Escreva também um código que faça o contrário.

C) Considere a fórmula para cálculo de juros simples, J = (C × I × T) / 100, onde J, C, I e T correspondem a juros, capital, taxa e tempo, respectivamente. Construa um código que solicite ao usuário os valores de C, I e T e calcule J.

As respostas estão na próxima página, mas é importante que você tente resolver os problemas e use as respostas apenas para conferência.

1.10 Respostas dos exercícios

Escreva códigos em Python para resolver os problemas a seguir:

A) Calcular a área de um triângulo a partir de sua base e altura.

```
print('Informe os dados do triângulo')
base = float(input('Base: '))
altura = float(input('Altura: '))
area = base * altura / 2
print('A área do triângulo é', area)
```

B) Escreva um código que receba um número de segundos e converta este número em horas, minutos e segundos. Escreva também um código que faça o contrário.

```
print('Segundos para H:M:S')
total_segundos = int(input('Informe número total de segundos: '))
minutos = total_segundos // 60
segundos = total_segundos % 60
horas = minutos // 60
minutos %= 60
print(horas, ':', minutos, ':', segundos)
```

```
print('H:M:S para Segundos')
print('Informe os dados')
horas = int(input('Horas: '))
minutos = int(input('Minutos: '))
segundos = int(input('Segundos: '))
total_segundos = horas * 60 * 60 + minutos * 60 + segundos
print('O total de segundos é', total_segundos)
```

C) Considere a fórmula para cálculo de juros simples, J = (C × I × T) / 100, onde J, C, I e T correspondem a juros, capital, taxa e tempo, respectivamente. Construa um código que solicite ao usuário os valores de C, I e T e calcule J.

```
print('Informe os dados para o cálculo dos juros')
capital = float(input('Capital: '))
taxa = float(input('Taxa de juros: '))
tempo = float(input('Tempo: '))
juros = capital * taxa * tempo / 100
print('O valor dos juros é', juros)
```

1.11 Revisão

Antes de prosseguirmos para a próxima o próximo conteúdo, é importante que você estude, revise o conteúdo e realize pesquisas sobre os conceitos apresentados para ampliar seus conhecimentos.

Mídia digital: Antes de avançarmos nos estudos, vá até a sala virtual e assista ao vídeo “Revisão da Primeira Semana” para recapitular tudo que aprendemos.

Nos encontramos na próxima semana.

Bons estudos!

Objetivos
- Entender o funcionamento do fluxo de execução do código, das estruturas de decisão e das estruturas de repetição;
- Combinar estruturas de decisão e de repetição para resolver problemas;
- Desenvolver códigos com estruturas de decisão e de repetição.

2.1 Introdução

Os códigos que desenvolvemos até o momento possui um fluxo de execução direto de forma que uma linha é executada após a outra sem nenhum tipo de desvio. Entretanto, para resolver diversos tipos de problemas precisamos ter um comportamento diferente no fluxo de execução. Nesses casos, pode ser necessário usar as estruturas de decisão e de repetição. Nas próximas seções estudaremos esses tipos de estruturas na linguagem Python.

2.2 Estruturas de decisão

Para resolver diversos tipos de problemas, precisamos usar as estruturas de decisão (também conhecidas como desvios condicionais ou estruturas condicionais). As estruturas de decisão são testes efetuados para decidir se uma determinada ação deve ser realizada (BORGES, 2010).

2.2.1 Estrutura de decisão simples

A estrutura de decisão mais simples utiliza uma única instrução **if**, seguida por uma expressão lógica e pelo bloco de instruções a ser executado se o valor da expressão lógica for verdadeiro (CEDER, 2018). Considere, por exemplo, o problema de verificar se um aluno foi aprovado. Isso é feito testando se a nota do aluno é maior ou igual a 60. A Figura 26 mostra o código para resolver esse problema.

```
1 nota = float(input('Informe a nota: '))
2 if nota >= 60:
3     print('Aprovado')
4 print('Boas férias')
```

Figura 26 – Estrutura de decisão simples para testar aprovação de aluno
Fonte: Elaborado pelo Autor.

A endentação deve ser feita corretamente para especificar quais instruções estão dentro da estrutura condicional. Utilizamos quatro espaços para endentar um bloco de instruções. Observe que, depois da expressão lógica seguida por dois pontos (**:**), começa o bloco de instruções do **if**. Esse bloco deve ser obrigatoriamente endentado. No código, a mensagem “Aprovado” é escrita na

tela somente quando a nota é maior ou igual a 60. Por outro lado, a mensagem “Boas férias” é mostrada incondicionalmente, pois não está dentro do desvio condicional.

2.2.2 Estrutura de decisão composta

Na estrutura de decisão simples, executamos alguma ação somente se a expressão lógica testada for verdadeira. Por outro lado, em certas situações, também precisamos realizar alguma medida se o teste for falso. Nesse caso, precisamos utilizar uma estrutura de decisão composta acrescentando a instrução **else** após o bloco de código da instrução **if**. A instrução **else** é seguida por outro bloco de código que será executado quando o teste for falso.

Como exemplo, vamos reconsiderar o problema de aprovação do aluno e escrever uma mensagem quando o mesmo for reprovado. A Figura 27 mostra o novo código contendo a mensagem “Reprovado” quando o aluno tem nota menor que 60.

```
1 n = float(input('Informe a nota: '))
2 if n >= 60:
3     print('Aprovado')
4 else:
5     print('Reprovado')
6 print('Boas férias')
```

Figura 27 – Código com estrutura de decisão composta
Fonte: Elaborado pelo Autor.

2.2.3 Estruturas de decisão aninhadas

Na estrutura de decisão simples, ocorre um único teste e o fluxo de execução do algoritmo pode seguir por um ou dois caminhos. Em determinadas situações, precisamos de algoritmos com vários testes e, por consequência, vários fluxos de execução. Para fazer isso, temos que inserir uma estrutura de decisão dentro de outra estrutura de decisão. Nesse caso, dizemos que as estruturas de decisão estão aninhadas.

Como exemplo, vamos considerar mais uma vez o problema de aprovação de aluno, mas, agora, vamos tratar as seguintes situações:

- Se o aluno tiver nota maior ou igual a 60, será aprovado;
- Se a nota for menor que 40, ao aluno é reprovado;
- Por fim, se a nota for maior ou igual a 40 e menor do que sessenta o aluno está de recuperação.

A Figura 28 apresenta o código para resolver o problema considerando as novas situações. Observe que, dentro do primeiro **else**, quando o aluno não é aprovado, ocorre um segundo teste para verificar se o aluno foi reprovado ou está de recuperação. O código pode ser escrito de outras formas, mudando a ordem dos testes, e obter o mesmo resultado.

```
1 n = float(input('Informe a nota: '))
2 if n >= 60:
3     print('Aprovado')
4 else:
5     if n >= 40:
6         print('Reavaliação')
7     else:
8         print('Reprovado')
9 print('Boas férias')
```

Figura 28 – Código com estruturas de decisão aninhadas
Fonte: Elaborado pelo Autor.

Observe que, no código da Figura 28, tivemos que endentar mais o **if** mais interno. Caso tenhamos muitas estruturas de decisão aninhadas, a legibilidade pode ficar prejudicada. Assim, outra maneira de usar as estruturas de decisão aninhadas é de forma consecutiva, como mostrado na Figura 29. A instrução **elif** funciona como uma junção do **else** com o **if**. Essa instrução é especialmente útil quando temos muitos testes consecutivos a serem feitos.

```
1 n = float(input('Informe a nota: '))
2 if n >= 60:
3     print('Aprovado')
4 elif n >= 40:
5     print('Reavaliação')
6 else:
7     print('Reprovado')
8 print('Boas férias')
```

Figura 29 – Código com estruturas de decisão consecutivas usando **elif**
Fonte: Elaborado pelo Autor.

2.2.4 Resolvendo problemas com estruturas de decisão

Como as estruturas de decisão são muito usadas, vamos resolver alguns problemas utilizando essas estruturas antes de avançarmos para o próximo conteúdo.

Ano bissexto

Como primeiro problema, criaremos um código para verificar se um ano é bissexto ou não. Um ano é bissexto se é múltiplo de 400, ou então se é múltiplo de quatro, mas não é múltiplo de 100. Por exemplo, 2012 (múltiplo de 4, mas não múltiplo de 100) é bissexto, 1900 (múltiplo de quatro e de 100) não é bissexto, 2000 (múltiplo de 400 é bissexto). A Figura 30 mostra o código para resolver esse problema.

```
1 ano = int(input('Informe o ano: '))
2 if (ano % 400 == 0) or (ano % 4 == 0 and ano % 100 != 0):
3     print('Ano bissexto')
4 else:
5     print('Ano não bissexto')
```

Figura 30 – Código para detectar ano bissexto
Fonte: Elaborado pelo Autor.

Observe que temos um teste que combina expressões relacionais com operadores lógicos na linha 2. As expressões relacionais são usadas para testar se a variável **ano** é ou não múltipla de

certos números usando o operador de resto de divisão (**%**). Note que os parênteses na expressão após o **or** são obrigatórios porque temos que saber se o ano é múltiplo de 4 e não é múltiplo de 100 ao mesmo tempo. Poderíamos construir o código sem os operadores lógicos, mas precisaríamos de mais linhas com mais estruturas de decisão.

Tipos de triângulos

Agora construiremos um código para classificar triângulos. Antes de mais nada, temos que testar se os três lados fornecidos pelo usuário são válidos para formar um triângulo. Isso é verdade apenas se nenhum lado for maior que a soma dos outros dois. Depois disso, temos que classificar o triângulo em isósceles (dois lados iguais), equilátero (três lados iguais) ou escaleno (três lados diferentes).

A Figura 31 apresenta o código classificador de triângulos. Assim como no problema anterior, combinamos expressões relacionais com operadores lógicos nas estruturas de decisão. Os parênteses delimitando as expressões relacionais (linhas 5, 8 e 10) poderiam ser omitidos, mas seu uso melhora a legibilidade do código.

```
 1 print('Informes os 3 lados do triângulo:')
 2 a = float(input('Lado 1: '))
 3 b = float(input('Lado 2: '))
 4 c = float(input('Lado 3: '))
 5 if (a > b + c) or (b > a + c) or (c > a + b):
 6     print('Triângulo inválido')
 7 else:
 8     if (a == b) and (b == c):
 9         print('Triângulo equilátero')
10     elif (a == b) or (b == c) or (a == c):
11         print('Triângulo isósceles')
12     else:
13         print('Triângulo escaleno')
```

Figura 31 – Código de um classificador de triângulos
Fonte: Elaborado pelo Autor.

Equações de segundo grau

Por fim, consideraremos a solução de equações de segundo grau no formato $Ax^2 + Bx + C = 0$. O algoritmo para resolver esse problema deve fazer o seguinte:

1) Ler os termos A, B e C;

2) Garantir que temos uma equação de segundo grau testando se A é diferente de zero;

3) Se for uma equação de segundo grau, calculamos o delta ($\Delta = B^2 - 4 \times A \times C$);

4) Após o cálculo, temos três situações para o delta:

- Se o delta for menor que zero, a equação não possui raízes;
- Se o delta for igual a zero, então a equação possui uma única raiz;
- Por fim, se o delta é maior que zero temos duas raízes $X_1 = \frac{-B+\sqrt{\Delta}}{2 \times A}$ e $X_2 = \frac{-B-\sqrt{\Delta}}{2 \times A}$.

A Figura 32 apresenta o código para resolver equações de segundo grau.

```
import math
print('Informe os termos da equação Ax² + Bx +C')
a = float(input('A: '))
b = float(input('B: '))
c = float(input('C: '))
if a == 0:
    print('Não é uma equação de segundo grau')
else:
    delta = b**2 - 4 * a * c
    if delta < 0:
        print('A equação não tem raízes')
    elif (delta == 0):
        x1 = b * (-1) / 2 * a
        print('A equação possui a raiz:', x1, '')
    else:
        raiz_delta = math.sqrt(delta)
        x1 = (b * (-1) + raiz_delta)/ 2 * a
        x2 = (b * (-1) - raiz_delta)/ 2 * a
        print('A equação possui duas raízes:')
        print('x1 =', x1)
        print('x2 =', x2)
```

Figura 32 – Código para resolver equações de segundo grau
Fonte: Elaborado pelo Autor.

Atividade: Resolva os exercícios a seguir. As respostas estão na próxima página, mas é importante que você tente resolver por conta própria e use as respostas apenas para conferência.

2.2.1 Exercícios

Escreva os códigos em Python para resolver os problemas a seguir:

A) Receber três números e informar o maior deles.

B) Testar se um número é ímpar ou par, sem usar o operador %.

C) Calcular o imposto de renda de um salário considerando as seguintes alíquotas:

- Até R$ 1.903,98: isento;
- De R$ 1.903,99 até R$ 2.826,65: 7,5%;
- De R$ 2.826,66 até R$ 3.751,05: 15%;
- De R$ 3.751,06 até R$ 4.664,68: 22,5%;
- Acima de R$ 4.664,68: 27,5%.

As respostas estão na próxima página, mas é importante que você tente resolver os problemas e use as respostas apenas para conferência.

2.2.2 Respostas dos exercícios

Escreva os códigos em Python para resolver os problemas a seguir:

A) Receber três números e informar o maior deles.

```
print('Informe três números')
a = int(input('A: '))
b = int(input('B: '))
c = int(input('C: '))
if (a > b) and (a > c):
    print('O maior número é ', a)
elif b > c:
    print('O maior número é ', b)
else:
    print('O maior número é ', c)
```

B) Testar se um número é ímpar ou par, sem usar o operador %.

```
n = int(input('Informe um número: '))
if n / 2 * 2 - n == 0:
    print('O número é par!')
else:
    print('O número é ímpar')
```

C) Calcular o imposto de renda de um salário considerando as seguintes alíquotas:

- Até R$ 1.903,98: isento;
- De R$ 1.903,99 até R$ 2.826,65: 7,5%;
- De R$ 2.826,66 até R$ 3.751,05: 15%;
- De R$ 3.751,06 até R$ 4.664,68: 22,5%;
- Acima de R$ 4.664,68: 27,5%.

```
salario = float(input('Informe o salário: '))
if salario <= 1903.98:
    imposto = 0
elif salario <= 2826.65:
    imposto = salario * 7.5 / 100
elif salario <= 3751.05:
    imposto = salario * 15 / 100
elif salario <= 4664.68:
    imposto = salario * 22.5 / 100
else:
    imposto = salario * 27.5 / 100
print('O imposto de renda  é', imposto)
```

2.3 Estruturas de repetição

As estruturas de repetição possibilitam executar repetidamente blocos de instruções por um número definido de vezes ao até que uma dada condição seja atendida.

2.3.1 Laço de repetição while

O laço de repetição **while** é indicado quando não sabe o número de repetições a serem executadas. Por exemplo, a listagem dos números pares menores do que um número informado pelo usuário. Nesse problema, não é possível saber o número de repetições, pois não tem como prever o número que o usuário informará.

No laço **while**, as repetições ocorrem enquanto uma determinada condição for verdadeira (CORRÊA, 2020). Assim, é importante garantir que a condição de parada realmente aconteça e o laço não fique repetindo indefinidamente. Vamos considerar novamente o problema da listagem dos números pares. Nesse caso, a condição de parada pode ser quando chegarmos a um número maior do que o número informado pelo usuário.

A Figura 33 mostra o código para listagem de números pares. Usamos a variável **atual** para armazenar o número atual, começando pelo zero. O laço **while** usa a condição de parada **atual < n**, para verificar se o número atual ainda é menor que **n** informado pelo usuário.

```
1 atual = 0
2 n = int(input('Informe um número: '))
3 while atual < n:
4     print(atual)
5     atual += 2
```

Figura 33 – Listagem de números pares
Fonte: Elaborado pelo Autor.

A estrutura de repetição **while** possui uma condição que é testada antes mesmo de executar a primeira repetição. Enquanto essa condição for verdadeira, as repetições continuam acontecendo. Se o usuário informar zero, por exemplo, não acontece nenhuma repetição. Dentro do laço, somamos mais dois ao número atual para chegarmos ao próximo número par.

2.3.2 Laço de repetição for

O laço de repetição **for** é indicado quando se sabe o número de repetições a serem feitas. Basicamente, o laço **for** percorre de forma automática os elementos de uma estrutura de lista. A maneira mais comum de utilizar o laço **for** é com a função **range()**. Essa função recebe um número inteiro **n** e gera um intervalo de números de 0 até **n – 1** (CORRÊA, 2020).

Para exemplificar o uso do laço **for**, vamos considerar o problema de somar 10 números informados pelo usuário. A Figura 34 mostra o código para resolver esse problema. Observe que, no laço, for usamos a função **range(10)** e a variável **cont** para percorrer os números do intervalo gerado pela função **range()**. Assim, na primeira repetição, a variável **cont** vale **0**, na segunda, vale **1**, e assim por diante até assumir o valor **9**.

```
soma = 0
for cont in range(10):
    n = float(input('Informe um número: '))
    soma += n
print(soma)
```

Figura 34 – Soma de 10 números usando o laço **for**
Fonte: Elaborado pelo Autor.

É possível notar, no código para soma dos 10 números, que a única função da variável **cont** é controlar as repetições do laço **for**. Quando temos uma variável que não é usada em nenhuma outra parte do código, podemos substituir essa variável pela variável anônima **_** (sublinhado).

Outro detalhe importante é a função **range()**. Além do fim do intervalo, podemos definir o início e a periodicidade dos números. Se usarmos, por exemplo, a instrução **range(2, 6)**, será retornado o intervalo de números "**2, 3, 4, 5**", ou seja, o primeiro número é o início e o segundo número é o fim do intervalo. Lembrando que o número do fim não é incluído no intervalo. Quando incluímos um terceiro número, definimos a periodicidade dos números. Como exemplo, se usarmos a instrução **range(3, 19, 4)** teremos a sequência "**3, 7, 11, 15**". A sequência começa em **3**, e os demais números são obtidos somando **4** ao número atual, até atingir o fim do intervalo. Além disso, no lugar dos números, podemos usar qualquer variável ou expressão que retorne um número inteiro. Também podemos obter intervalos em ordem decrescente, por exemplo, a instrução **range(5,0,-1)** retorna a sequência "**5, 4, 3, 2, 1**".

2.3.3 Interrupção e continuação de laços de repetição

O laço de repetição **while** precisa testar a condição de parada antes mesmo da primeira repetição. Entretanto, em diversos momentos, precisamos executar a primeira repetição antes de testar a condição de parada. Nesse caso, podemos criar um laço com a instrução **while True** e utilizar a instrução **break** para finalizar o laço de repetição.

Como exemplo, vamos considerar a soma de uma quantidade indeterminada de números informados pelo usuário. Devemos parar de somar apenas quando o usuário informar o número **0** (zero). A Figura 35 apresenta o código para resolver esse problema. É importante tomar um certo cuidado com laços **while True**, temos que garantir que a instrução **break** será executada em algum momento e o laço não fique repetido indefinidamente.

```
soma = 0
while True:
    n = float(input('Informe um número: '))
    if n == 0:
        break
    soma = soma + n
print('Soma dos números:', soma)
```

Figura 35 – Soma indefinida de números
Fonte: Elaborado pelo Autor.

Vamos considerar agora uma modificação no problema de somar números. Além do que já foi mencionado, suponha que os números negativos não devam ser somados. Diante disso, podemos usar a instrução **continue** para ignorar os números negativos e *pular* para a próxima repetição do laço (CEDER, 2018). O código da Figura 36 mostra a solução para a modificação do

problema. As instruções **break** e **continue** podem ser usadas tanto no laço **while** quanto no laço **for**.

```
1 soma = 0
2 while True:
3     n = float(input('Informe um número: '))
4     if n < 0:
5         continue
6     if n == 0:
7         break
8     soma = soma + n
9 print('Soma dos números:', soma)
```

Figura 36 – Soma indefinida de números (exceto negativos)
Fonte: Elaborado pelo Autor.

2.3.4 Resolvendo problemas com estruturas de repetição

Assim como as estruturas de decisão, as estruturas de repetição também são extensamente usadas para resolver diversas categorias de problemas. Portanto, demostraremos como usá-las para resolver alguns problemas e praticar o uso das estruturas antes de avançarmos para o próximo conteúdo.

Cálculo de MDC

Um problema interessante para ser resolvido com laço de repetição é o cálculo de máximo divisor comum (MDC) com a técnica de Euclides (WIKIPÉDIA, 2021a). Lembrando que o MDC de números é o maior número que os divide sem deixar resto. Basicamente, a técnica de Euclides consiste nos seguintes passos:

- Dividimos o maior número pelo menor e verificamos se o resto da divisão é zero;
- Em caso afirmativo, o menor número é o MDC;
- Caso contrário, substituímos o maior número pelo menor, o menor número pelo resto da divisão e repetimos o processo.

Repare que a repetição do processo no terceiro ponto caracteriza uma estrutura de repetição. Como exemplos, demonstraremos como calcular o MDC de 144 e 56. O processo é o seguinte:

- Começamos dividindo 144 por 56. Como o resto da divisão é 32, vamos considerar os números 56 e 32 e repetir o processo;
- Dividimos 56 por 32 e temos 24 como resto. Agora, consideramos 32 e 32 e dividimos novamente;
- Na divisão de 32 por 24, chegamos a 8 como resto. Assim, a próxima divisão será 24 por 8;
- Por fim, dividimos 24 por 8 e temos zero como resto. Logo, o MDC é o número 8.

O laço de repetição mais adequado para implementar o algoritmo de Euclides é o **while**, pois não tem como prever quantas divisões precisam ser feitas. A Figura 37 mostra o código do algoritmo de Euclides.

A condição de parada do laço é **True**. Todavia, na linha 11, testamos se o resto da última divisão for zero, interrompemos o laço com a instrução **break** (na linha 12). Além do laço de repetição, usamos uma estrutura de decisão na linha 5 para testar se os números são válidos (positivos maiores que zero). O **print()** da linha 10 não é necessário, ele foi incluído apenas para mostrar as divisões do processo.

```
 1 print('Informe dois números')
 2 n1 = int(input('N1: '))
 3 n2 = int(input('N2: '))
 4
 5 if n1 < 1 or n2 < 1:
 6     print('Números inválidos para MDC')
 7 else:
 8     while True:
 9         resto = n1 % n2
10         print (n1, '/' , n2, '-> resto:', resto)
11         if resto == 0:
12             break
13         n1 = n2
14         n2 = resto
15     print('O MDC é ', n2)
```

Figura 37 – Soma indefinida de números (exceto negativos)
Fonte: Elaborado pelo Autor.

Combinações de elementos

Agora, vamos considerar o problema de gerar as combinações de dois elementos a partir de um conjunto de números naturais com **n** elementos, ou seja, um conjunto A = {1, 2, 3, ..., n}. Antes de gerar as combinações, o código deve perguntar o número de elementos do conjunto ao usuário. A Figura 38 mostra o código para resolver o problema descrito.

```
 1 n = int(input('Informe o número de elementos do conjunto: '))
 2
 3 print('Elementos:', end=' ')
 4 for cont in range(1, n+1):
 5     print(cont, end=' ')
 6
 7 print('\nCombinações:', end='')
 8 for cont in range(1, n+1):
 9     for cont2 in range(1, n+1):
10         print('(', cont, ', ', cont2, ')', sep='', end=' ')
```

Figura 38 – Combinações com dois elementos de um conjunto
Fonte: Elaborado pelo Autor.

Após pegar o número de elementos informado pelo usuário (linha 1), escrevemos todos os elementos do conjunto na tela (linhas 3 a 5). No laço de repetição usamos a instrução **range(1, n+1)** para termos a sequência de elemento de **1** até **n**. Repare também que usamos o parâmetro **end=' '**, na função **print()** para evitar a quebra de linha e os elementos ficarem um após o outro.

Nas linhas 7 a 10, escrevemos as combinações dos elementos. Usamos dois laços de repetição aninhados para que cada elemento seja combinado com os demais (inclusive, com ele mesmo). O primeiro elemento é a variável **cont** e o segundo elemento é a variável **cont2**. A função **print()** dentro do laço mais interno escreve cada uma das combinações. Foram usados os

parâmetros **sep=''** e **end=' '** para que os elementos fiquem juntos e as combinações fiquem separadas.

Subconjuntos

Por fim, vamos considerar o problema de gerar os subconjuntos de dois elementos a partir de um conjunto com **n** números naturais. A princípio, esse problema pode ser parecido com o anterior. No entanto, pela definição matemática, temos que considerar os seguintes pontos:

- Os subconjuntos não podem ter elementos repetidos;
- A ordem dos elementos não altera o conjunto;
- Não devemos ter subconjuntos repetidos.

A Figura 39 mostra o código que resolve esse problema. Veja que esse código é muito parecido com a solução para as combinações. A diferença está no laço mais interno, o intervalo de elementos desse laço começar com o primeiro elemento maior do que **cont**. Isso garante que não combinaremos um elemento com ele mesmo nem com seus antecessores.

```
 1 n = int(input('Informe o número de elementos do conjunto: '))
 2
 3 print('Elementos:', end=' ')
 4 for cont in range(1, n+1):
 5     print(cont, end=' ')
 6
 7 print('\nSubconjuntos:', end='')
 8 for cont in range(1, n+1):
 9     for cont2 in range(cont+1, n+1):
10         print('{', cont, ', ', cont2, '}', sep='', end=' ')
```

Figura 39 – Subconjuntos com dois elementos de um conjunto
Fonte: Elaborado pelo Autor.

Atividade: Resolva os exercícios a seguir. As respostas estão na próxima página, mas é importante que você tente resolver por conta própria e use as respostas apenas para conferência.

2.3.1 Exercícios

Escreva os códigos em Python para resolver os problemas a seguir:

A) Leia uma quantidade indeterminada de números. A cada número informado, o usuário deve informar se deseja continuar ou parar. Ao final, o código deve retornar o maior e o menor número recebido.

B) Calcule o fatorial de um número. O fatorial de um número n, representado por n!, é calculado como $n! = n \times (n - 1) \times ... \times 2 \times 1$. Sendo que $1! = 0! = 1$.

C) Simular uma calculadora simples. O código deve solicitar ao usuário a operação desejada (soma, multiplicação, divisão, subtração ou potência) ou então sair. Quando o usuário escolher uma operação, o código deve solicitar dois números, realizar a operação sobre estes números e exibir o resultado. O código deve sempre solicitar uma nova operação até que o usuário escolha sair.

2.3.2 Respostas dos exercícios

Escreva os códigos em Python para resolver os problemas a seguir:

A) Leia uma quantidade indeterminada de números. A cada número informado, o usuário deve informar se deseja continuar ou parar. Ao final, o código deve retornar o maior e o menor número recebido.

```
maior = float('-inf') # -∞, qualquer número pode ser maior
menor = float('inf')  # +∞, qualquer número pode ser menor
while True:
    n = float(input('Informe um número: '))
    if n > maior:
        maior = n
    if n < menor:
        menor = n
    resp = input('Deseja continuar? (S/N)')
    if resp.lower() == 'n':
        break
print('Maior número informado:', maior)
print('Menor número informado:', menor)
```

B) Calcule o fatorial de um número. O fatorial de um número n, representado por n!, é calculado como n! = n × (n − 1) × ... × 2 × 1. Sendo que 1! = 0! = 1.

```
n = int(input('Informe um número: '))
fat = 1
for cont in range(n, 1, -1):
    fat *= cont
print('O fatorial de', n, 'é', fat)
```

C) Simular uma calculadora simples. O código deve solicitar ao usuário a operação desejada (soma, subtração, multiplicação ou divisão) ou então sair. Quando o usuário escolher uma operação, o código deve solicitar dois números, realizar a operação sobre estes números e exibir o resultado. O código deve sempre solicitar uma nova operação até que o usuário escolha sair.

```
while True:
    print('Calculadora (+, -, /, *)')
    print('s: sair')
    resp = input('Informe a operação desejada (s para sair): ')
    if resp == 's':
        break
    print('Informe dois números:')
    n1 = float(input('N1: '))
    n2 = float(input('N1: '))
    if resp == '+':
        r = n1 + n2
    elif resp == '-':
        r = n1 - n2
    elif resp == '*':
        r = n1 * n2
    elif resp == '/':
        r = n1 / n2
    else:
        print('Operação inválida!')
        continue
    print(n1, resp, n2, '=', r)
```

2.4 Revisão

Antes de prosseguirmos para a próxima o próximo conteúdo, é importante que você estude, revise o conteúdo e realize pesquisas sobre os conceitos apresentados para ampliar seus conhecimentos.

Mídia digital: Antes de avançarmos os estudos, vá até a sala virtual e assista ao vídeo “Revisão da Segunda Semana” para recapitular tudo que aprendemos.

Nos encontramos na próxima semana.

Bons estudos!

Objetivos
- Entender o funcionamento da tratamento de exceções;
- Conhecer o conceito de modularização e decomposição de problemas;
- Construir códigos modularizados e com tratamento de exceção.

3.1 Introdução

Mesmo os códigos escritos corretamente podem se deparar com erros durante a execução. Além disso, dependendo do problema a ser resolvido, podemos ter códigos muito extensos que precisam ser organizados adequadamente para facilitar o entendimento e a manutenção. Nas próximas seções, vamos abordar tanto o tratamento de exceções para lidar com erros inesperados, quanto com a modularização para organizar melhor os códigos.

3.2 Tratamento de exceções

Considere uma instrução **n = float(input('Informe um número'))**. Estamos pedindo ao usuário para digitar um número. Todavia, se for digitado qualquer texto que não possa ser convertido, para número ocorrerá um erro em tempo de execução. Assim, ainda que o código esteja correto, podem ocorrer situações que causam erros. Esses erros são chamados de exceções e podem ser contornados usando a instrução **try** da linguagem Python (PSF, 2021).

O código da Figura 40 é um exemplo simples de tratamento de exceção. Dentro da cláusula **try**, colocamos as instruções de código que deveriam ser executadas normalmente. Se ocorrer algum erro, o fluxo de execução é direcionado para a cláusula **except** (CEDER, 2018). Execute o código e informe números corretos e incorretos. Você poderá ver que, ao ocorrer um erro, será executada a instrução da cláusula **except**.

```
1 try:
2     print('Informe dois números')
3     n1 = float(input('n1: '))
4     n2 = float(input('n2: '))
5     r = n1 / n2
6     print(n1, '/', n2, '=', r)
7 except:
8     print('Ocorreu um erro.')
```

Figura 40 – Tratamento simples de exceção
Fonte: Elaborado pelo Autor.

No código anterior, se ocorrer um erro, a execução será finalizada na linha 8. Assim, caso o usuário quiser tentar novamente, será preciso executar o código novamente. Uma solução para esse problema é colocar o tratamento de exceção dentro de um laço de repetição. Dessa maneira, se ocorrer um erro, o usuário poderá tentar novamente sem ter que executar todo o código mais uma vez. Essa solução é mostrada na Figura 41. Como foi usado um laço **while True**, temos que

incluir a instrução **break** no final da cláusula **try** para, na ausência de erros, finalizar o laço. Por outro lado, se ocorrer algum erro, a cláusula **except** é disparada e, depois de sua execução, o laço continua as repetições.

```
 1 import math
 2 while True:
 3     try:
 4         print('Informe dois números')
 5         n1 = float(input('n1: '))
 6         n2 = float(input('n2: '))
 7         r = n1 / n2
 8         break
 9     except:
10         print('Ocorreu um erro! Tente novamente.')
11 print(n1, '/', n2, '=', r)
```

Figura 41 – Tratamento de exceção com repetição
Fonte: Elaborado pelo Autor.

Em nosso código, além do erro de digitação pelo usuário, pode acontecer o erro de divisão por zero na linha 7. A instrução **except** da linha 9 captura todos os tipos de erros. Se for necessário tratar erros diferentes, temos que colocar uma cláusula **except** para cada tipo de erro. Para descobrir as classes dos erros, podemos usar a função **type()** como mostrado no código da Figura 42.

```
 1 while True:
 2     try:
 3         print('Informe dois números')
 4         n1 = float(input('n1: '))
 5         n2 = float(input('n2: '))
 6         r = n1 / n2
 7         break
 8     except Exception as e:
 9         print('Ocorreu o seguinte erro:', type(e))
10 print(n1, '/', n2, '=', r)
```

Figura 42 – Descobrindo classes de erros
Fonte: Elaborado pelo Autor.

Se for digitado um número incorretamente, temos a exceção **ValueError**. Se o segundo número for zero, temos a exceção **ZeroDivisionError**. O código da Figura 43 mostra como fazer o tratamento de exceção específico para cada uma dessas classes de erro.

```
while True:
    try:
        print('Informe dois números')
        n1 = float(input('n1: '))
        n2 = float(input('n2: '))
        r = n1 / n2
        break
    except ValueError as e:
        print(e)
        print('Número inválidos! Tente novamente.')
    except ZeroDivisionError as e:
        print(e)
        print('Divisão por zero! Tente novamente.')
print(n1, '/', n2, '=', r)
```

Figura 43 – Tratamento de exceções específicas
Fonte: Elaborado pelo Autor.

A instrução **try** possui também as cláusulas **else** e **finally**. A cláusula **else** pode ser usada para realizar alguma ação quando não ocorrer erros. Já a cláusula **finally** é executa incondicionalmente, ocorrendo erros ou não. A cláusula **finally** é útil para executar ações de limpeza como fechamento de arquivos.

3.3 Modularização

A modularização consiste em dividir um algoritmo em partes menores chamadas sub-rotinas ou funções com o intuito de facilitar o desenvolvimento e a manutenção de código (BORGES, 2010). Considerando um problema grande a ser resolvido, as funções representam pequenas partes do problema maior com menor complexidade. Além disso, as funções podem ser trabalhadas de forma independente e a localização de erros se torna mais fácil.

Um código deve ser desenvolvido de forma modular sempre que possível. Assim, se um mesmo trecho de código é executado em pontos diferentes do programa, podemos criar uma função para executar esse código uma única vez e chamar a função quando necessário. As principais vantagens da modularização são as seguintes:

- Evita a reescrita desnecessária de códigos similares;
- Melhora legibilidade do código;
- Permite desenvolvimento em equipe (cada programador cuida de um trecho do código);
- As funções podem ser testadas isoladamente para verificação de erros;
- A manutenção se torna mais fácil, apenas algumas partes podem precisar de alteração.

3.3.1 Funções

Funções são trechos de código que executam determinada tarefa ao serem chamados e depois retornam o controle para o ponto em que foram chamadas (CEDER, 2018). Apesar de ainda não termos criados nossas próprias funções, já escrevemos diversos códigos utilizando funções prontas como **input()** e **print()**. Quando chamamos uma função devemos informar os parâmetros

necessários. Isto significa que se vamos usar uma função **teste()** que recebe um parâmetro **int** e outro **str**, então devemos usar a instrução **teste(a, b)**, onde **a** é do tipo **int** e **b** é do tipo **str**.

A criação de novas funções é feita com a instrução **def**, seguida pelo nome da função e seus parâmetros entre parênteses e dois pontos (**:**). A linha com a instrução **def** é chada de declaração ou cabeçalho da função. Tanto o nome da função e seus parâmetros devem obedecer às mesmas regras dos nomes de variáveis. Após a declaração, vem o chamado corpo da função com suas instruções endentadas. No corpo da função, quando a função precisar tiver que retornar algum valor, é usada a instrução **return**.

O código da Figura 44 contém a função **saudacao()** que não possui parâmetros e não retorna valores. É importante incluir uma linha em branco entre o corpo da função e a próxima instrução para manter uma boa legibilidade no código. É importante frisar que as instruções **print()** da função não caracterizam retorno de valor. Quando uma função retorna um valor, podemos atribuir ser resultado a uma variável. Esse é o caso da função **input()**, por exemplo.

Na linha 6 do código, acontece chamada à função. Essa chamada direciona o fluxo de execução do código para a primeira instrução da função (linha 2). Após a execução de todas as linhas da função, o fluxo de execução retorna para a linha onde ocorreu a chamada. Além disso, as funções são executadas somente quando são chamadas. No código da Figura 44, por exemplo, a primeira linha a ser executada é a linha 6.

```
1 def saudacao():
2     print('*******************************')
3     print('*          BEM VINDO          *')
4     print('*******************************')
5
6 saudacao()
```

Figura 44 – Função **saudacao()**
Fonte: Elaborado pelo Autor.

No código da Figura 44 criamos apenas uma função. Porém, para a grande maioria dos problemas precisar escrever códigos com várias funções. Além disso, podemos criar uma função pode chamar outras funções. Uma função pode declarar variáveis para serem utilizadas apenas internamente. Estas variáveis, assim como os parâmetros da função, são chamadas de variáveis locais enquanto as variáveis de fora da função são as variáveis globais. É importante que as variáveis sejam locais sempre que possível.

3.3.2 Passagem de parâmetros e retorno de resultado

No código anterior, podemos ver que a função **saudacao()** não recebe nenhum parâmetro e nem dados do usuário. Isso faz com que essa função sempre realize as mesmas ações. Na maioria das vezes, precisamos criar funções parametrizáveis que realizam ações específicas conforme os parâmetros recebidos. Na prática, os parâmetros são como variáveis já inicializadas recebidas pelas funções.

A Figura 45 mostra um código com a função **cubo()**. A função recebe como parâmetro o número **num** e calcula o cubo do mesmo. Contudo, essa função ainda pode ser melhorada. Observe que não foi usada a instrução **return** para retornar o resultado do cálculo. A função está simplesmente escrevendo na tela. Assim, se modificarmos a função para retornar o resultado,

teremos uma função mais útil. Isso porque podemos usar a função em qualquer lugar, sem ter que escrever na tela e usar seu resultado da maneira que for mais apropriada.

```
1 def calcula_cubo(num):
2     cubo = num * num * num
3     print(num, 'ao cubo é', cubo)
4
5 n = float(input('Informe um número: '))
6 calcula_cubo(n)
```

Figura 45 – Função **cubo()** sem retorno
Fonte: Elaborado pelo Autor.

```
1 def cubo(num):
2     return num * num * num
3
4 n = float(input('Informe um número: '))
5 print(n, 'ao cubo é', cubo(n))
6 print(n, 'elevado a nona é', cubo(cubo(n)))
```

Figura 46 – Função **cubo()** com retorno
Fonte: Elaborado pelo Autor.

A Figura 46 mostra a modificação da função **cubo()** com o retorno do resultado. Repare que podemos usar diretamente a função dentro do **print()**. Além disso, podemos usar o resultado da função em qualquer expressão numérica, como na instrução **cubo(cubo(n))** da linha 6.

3.3.3 Nomes de parâmetros e parâmetros opcionais

Na chamada de funções é possível, passar parâmetros em ordem diferente daquela especificada da declaração, desde que existam atribuições aos nomes dos parâmetros. Também podemos criar funções com parâmetros opcionais. Os parâmetros opcionais devem ser os últimos e devem ter um valor padrão já atribuído. Assim, na chamada da função, se o parâmetro opcional não for passado, será usado o valor padrão.

```
1 def juros(capital, taxa, tempo=12):
2     return (capital * taxa * tempo) / 100
3
4 print('Cálculo de juros')
5 cap = float(input('Capital: '))
6 tax = float(input('Taxa: '))
7 tem = input('Tempo (deixe em branco para o padrão de 12): ')
8 if tem == '':
9     jur = juros(cap, tax)
10 else:
11     tem = float(tem)
12     jur = juros(taxa=tax, capital=cap, tempo=tem)
13 print('O valor dos juros é', jur)
```

Figura 47 – Código com Nomes de parâmetros e parâmetros opcionais
Fonte: Elaborado pelo Autor.

A Figura 47 exibe um código com parâmetros opcionais e chamada de função usando os nomes dos parâmetros. Na função **juros()**, temos o parâmetro **tempo** como opcional, seu valor padrão é **12**. Na linha 9, é feita uma chamada de função sem usar os nomes de parâmetros. Nesse

caso temos que manter os parâmetros na ordem correta. Primeiro, o parâmetro **capital** e, depois, o parâmetro **taxa**. Nessa linha o parâmetro opcional **tempo** não foi usado. Já na linha 12, ocorre uma chamada de função usando os nomes dos parâmetros. Observe que eles não estão na mesma ordem da declaração, mas, como usamos os nomes, isso não é um problema.

3.3.4 Recursão

A recursão ocorre quando uma função chama a si mesma. Na prática, é possível usar laços de repetição para substituir recursões. Contudo, em muitas situações, funções recursivas podem ser mais intuitivas do que laços de repetição. Uma observação importante é que precisamos ter cuidado com a condição de parada, assim como fazemos nos laços de repetição. Caso contrário, podemos cair em uma recursão infinita que a função faz chamadas a ela mesma indefinidamente.

```
1 def fat(num):
2     if num <= 1:
3         return 1
4     return num * fat(num - 1)
5
6 n = int(input('Informe um número: '))
7 print('O fatorial de', n, 'é', fat(n))
```

Figura 48 – Função recursiva **fat()**
Fonte: Elaborado pelo Autor.

O código da Figura 48 mostra apresenta a função recursiva **fat()**. O **if** da linha 2 faz o teste da condição de parada. Quando o parâmetro de entrada for menor ou igual a um, seu fatorial será um. Na linha 5, ocorre a chamada recursiva, usando a equivalência **n! = n * (n-1)!**.

3.3.5 Módulos e bibliotecas

Dependendo da quantidade de código, pode ser interessante a criar módulos para agrupar as funções correlacionadas. Normalmente, os módulos possuem apenas definições de funções ou outras estruturas e o código principal controla o fluxo de execução do código importando os módulos e chamando suas funções.

Para exemplificar a criação de módulos, consideraremos o problema de calcular o cubo e o fatorial de um número. A Figura 49 mostra o módulo com as funções que realizam esses cálculos.

```
1 def cubo(num):
2     return num * num * num
3
4 def fat(num):
5     if num <= 1:
6         return 1
7     return num * fat(num - 1)
```

Figura 49 – Módulo **mat.py**
Fonte: Elaborado pelo Autor.

Na Figura 50 temos o código do módulo principal que controla o fluxo de execução e efetua as chamadas as funções do módulo **mat.py**. É importante que ambos módulos sejam salvos na mesma pasta ou diretório. No código, especificamos quais funções deveriam ser importados

usando a instrução **from ... import ...**. Poderíamos fazer de outra maneira, fazendo somente a importação do módulo **mat** e chamando as funções no formado **mat.cubo(...)** e **mat.fat(...)**.

```
1 from mat import cubo, fat
2
3 n = int(input('Informe um número: '))
4 print(n, 'ao cubo é', cubo(n))
5 print('O fatorial de', n, 'é', fat(n))
```

Figura 50 – Módulo **principal.py**
Fonte: Elaborado pelo Autor.

A linguagem Python é uma linguagem interpretada, ou seja, não é preciso compilar o código-fonte para gerar arquivos executáveis. No caso do Linux, podemos criar scripts executáveis usando o comentário especial **#!/usr/bin/env python3** na primeira linha do módulo principal. Além disso, temos que dar permissão de execução ao arquivo. No caso do Windows, podemos associar a abertura de arquivos **.py** ao programa **pythonw.exe** disponível na pasta de instalação do Spyder. Assim, os scripts podem ser executados diretamente através do gerenciador de arquivos ou linha de comando. A Figura 51 mostra um script executável juntando os dois módulos do exemplo anterior.

```
1 #!/usr/bin/env python3
2
3 # -*- coding: utf-8 -*-
4
5 def cubo(num):
6     return num * num * num
7
8 n = int(input('Informe um número: '))
9 print(n, 'ao cubo é', cubo(n))
```

Figura 51 – Exemplo de script executável
Fonte: Elaborado pelo Autor.

3.4 Decomposição de problemas

A principal vantagem da modularização é a possibilidade de usarmos decomposição de problemas para resolver problemas mais complexos. Assim, podemos quebrar um problema maior em pequenas partes. Cada uma dessas partes pode ser resolvida com uma função específica. Ao final, juntamos as funções para solucionar o problema inicial.

Para exemplificar a decomposição de problemas construiremos uma calculadora de expressões considerando os seguintes pontos:

- A calculadora consiste em um console onde o usuário digita comandos;
- O usuário pode digitar expressões aritméticas para serem calculadas, ver o histórico de expressões ou sair;
- Os comandos **h** e **s** são usados respectivamente para histórico e sair;
- Deve ser realizado tratamento de exceção no cálculo da expressão digitada para garantir que a mesma não possuir erros;

- O histórico deve guardar apenas as expressões sem erros.

A descrição da calculadora pode parecer muito complexa, mas vamos decompô-la em problemas menores para facilitar a implementação. Primeiro construiremos uma função que recebe o texto da expressão e calcula o resultado. Para facilitar um pouco mais a nossa vida, usaremos a função **eval()** do Python. Essa função é capaz de executar um texto como se fosse os códigos em Python. No caso de uma expressão aritmética, a função **eval()** retorna o resultado dessa expressão. Entretanto, se a expressão for possuir algum erro de sintaxe, ocorrerá um erro. Portanto, temos que fazer o tratamento de exceções ao executarmos a função **eval()**.

```
1 def calcula(expr):
2     try:
3         return eval(expr)
4     except:
5         print('Expressão inválida!')
6         return None
```

Figura 52 – Função **calcula()** da calculadora de expressões
Fonte: Elaborado pelo Autor.

A Figura 52 exibe o código da função **calcula()**. Usamos a instrução **try** para fazer o tratamento de exceção. No caso de algum erro, mostramos a mensagem de expressão inválida e retornamos o valor **None**. Esse valor é um valor nulo que podemos testar ao receber o resultado da função.

```
1 def historico(expr, res):
2     global HIST
3     if res is not None:
4         HIST += '\n\n'+ expr
5         HIST += '\n' + str(res)
```

Figura 53 – Função **historico()** da calculadora de expressões
Fonte: Elaborado pelo Autor.

Agora criaremos uma função para atualizar o histórico da calculadora. Seu código é mostrado na Figura 53. A instrução **global HIST** da linha 2 é usada para podermos alterar a variável global **HIST**. Tal variável será declarada posteriormente para armazenar o histórico. Nesse problema estamos adotando maiúsculas para variáveis globais e minúsculas para variáveis locais. A função **historico()** recebe os parâmetros **expr** (texto da expressão calculada) e **res** (resultado do cálculo). Na linha 3, fazemos um teste para verificar se a expressão é válida. O teste **res is not None** é usado para verificar se o resultado da expressão (**res**) é diferente de **None**. Dentro do **if** apenas adicionamos a expressão e seu resultado ao histórico. Utilizamos o **'\n'** para separar as linhas do histórico.

Depois de construirmos as duas funções auxiliares, vamos escrever a função principal da calculadora que deverá gerenciar o fluxo de execução e chamar as funções auxiliares quando necessário. A Figura 54 mostra o código dessa função.

```
 1 def principal():
 2     while True:
 3         print('Informe a expressão matemática')
 4         print('(h para histórico, s para sair)')
 5         expr = input()
 6         if expr.lower() == 's':
 7             break
 8         if expr.lower() == 'h':
 9             print(HIST, '\n')
10         else:
11             res = calcula(expr)
12             historico(expr, res)
13             print(res, '\n')
```

Figura 54 – Função **principal()** da calculadora de expressões
Fonte: Elaborado pelo Autor.

Temos um laço de repetição para que o usuário possa digitar quantas expressões desejar. Na linha 6, testamos se o usuário informou o comando **s** e interrompemos o laço de repetição. No caso do comando **h**, o teste é feito na linha 8. A linha 9 apenas mostra o histórico de cálculos guardado na variável **HIST**. Repare que não precisamos da instrução **global HIST** nessa função porque estamos apenas lendo o conteúdo da variável global.

Se o usuário não informar os comandos **s** ou **h**, partimos para o cálculo da expressão na linha 11. O resultado da expressão é guardado na variável **res**. Em seguida, na linha 12, chamamos a função de atualizar o histórico. Por fim, na linha 13, mostramos o resultado do cálculo da expressão.

As funções desenvolvidas até agora não são suficientes para que a calculadora funcione. Temos que declarar a variável global **HIST** e chamar a função **principal()**. A Figura 55 exibe o código completo da calculadora de expressões. Os comentários das linhas 1 e 2 permitem que o código seja executado na forma de script. A variável global **HIST** é inicializada com um texto vazio na linha 4. O **if** da linha 33 utiliza a variável especial **__name__** do Python para verificar se o módulo foi executado como um script. Quando isso acontece o conteúdo dessa variável é **'__main__'**.

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

HIST = ''

def calcula(expr):
    try:
        return eval(expr)
    except:
        print('Expressão inválida!')
        return None

def historico(expr, res):
    global HIST
    if res is not None:
        HIST += '\n\n'+ expr
        HIST += '\n' + str(res)

def principal():
    while True:
        print('Informe a expressão matemática')
        print('(h para histórico, s para sair)')
        expr = input()
        if expr.lower() == 's':
            break
        if expr.lower() == 'h':
            print(HIST, '\n')
        else:
            res = calcula(expr)
            historico(expr, res)
            print(res, '\n')

if __name__ == '__main__':
    principal()
```

Figura 55 – Código completo da calculadora de expressões
Fonte: Elaborado pelo Autor.

Atividade: Resolva os exercícios a seguir. As respostas estão na próxima página, mas é importante que você tente resolver por conta própria e use as respostas apenas para conferência.

3.5 Exercícios

Escreva os códigos em Python para resolver os problemas a seguir:

A) Crie a função **input_int(mensagem)** semelhante à função **input()**. A função deve receber uma mensagem, exibi-la ao usuário e garantir que o número digitado seja válido. Enquanto o usuário digitar um número inválido, a função deve informar o erro e solicitar e digitação novamente. Dica: utilize a instrução **try**.

B) Implemente um módulo com as funções:

- **ano_bissexto(ano)**, retorna **True** se o **ano** for bissexto e **False**, em caso negativo;
- **dias_mes(ano, mes)**: retorna a quantidade de dias do mês, deve usar a função **ano_bissexto()**.

Construa um script que receba uma data do usuário (mês e ano) e mostre o resultado das funções implementadas.

C) Escreva um módulo com funções para calcular o máximo divisor comum (MDC) e o mínimo múltiplo comum (MMC) de dois números. Para o MDC, você deve adaptar o algoritmo de Euclides já estudado. No caso do MMC, pode ser usada a fórmula **$MMC(n_1, n_2) = n_1 * n_2 / MDC(n_1, n_2)$.** Implemente também um script com chamadas a essas funções sobre números digitados pelo usuário.

3.6 Respostas dos exercícios

Escreva os códigos em Python para resolver os problemas a seguir:

A) Crie a função **input_int(mensagem)** semelhante à função **input()**. A função deve receber uma mensagem, exibi-la ao usuário e garantir que o número digitado seja válido. Enquanto o usuário digitar um número inválido, a função deve informar o erro e solicitar e digitação novamente. Dica: utilize a instrução **try**.

```
1 def input_int(mensagem):
2     while True:
3         try:
4             return int(input(mensagem))
5         except:
6             print('Número inválido! Tente novamente.')
```

B) Implemente um módulo com as funções:

- **ano_bissexto(ano)**, retorna **True** se o **ano** for bissexto e **False**, em caso negativo;
- **dias_mes(ano, mes)**: retorna a quantidade de dias do mês, deve usar a função **ano_bissexto()**.

Construa um script que receba uma data do usuário (mês e ano) e mostre o resultado das funções implementadas.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

def ano_bissexto(ano):
    if (ano % 400 == 0):
        return True
    elif (ano % 4 == 0) and not (ano % 100 == 0):
        return True
    else:
        return False

def dias_mes(ano, mes):
    if mes == 1 or mes == 3 or mes == 5 or mes == 7:
        return 31
    elif mes == 8 or mes == 10 or mes == 12:
        return 31
    elif mes == 4 or mes == 6 or mes == 9 or mes == 11:
        return 30
    elif mes == 2:
        if ano_bissexto(ano):
            return 29
        else:
            return 28
    else:
        return -1

def principal():
    print('Informe a data')
    ano = int(input('Ano: '))
    mes = int(input('Mês: '))

    print('Ano bissexto:', ano_bissexto(ano))
    print('Dias do mês:', dias_mes(ano, mes))

if __name__ == '__main__':
    principal()
```

C) Escreva um módulo com funções para calcular o máximo divisor comum (MDC) e o mínimo múltiplo comum (MMC) de dois números. Para o MDC, você deve adaptar o algoritmo de Euclides já estudado. No caso do MMC, pode ser usada a fórmula **$MMC(n_1, n_2) = n_1 * n_2 / MDC(n_1, n_2)$.** Implemente também um script com chamadas a essas funções sobre números digitados pelo usuário.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

def mdc(n1, n2):
    while True:
        resto = n1 % n2
        if resto == 0:
            return n2
        n1 = n2
        n2 = resto

def mmc(n1, n2):
    return n1 * n2 // mdc(n1, n2)

def principal():
    print('Informe dois números inteiros:')
    n1 = int(input())
    n2 = int(input())
    print('MDC:', mdc(n1, n2))
    print('MMC:', mmc(n1, n2))

if __name__ == '__main__':
    principal()
```

3.7 Revisão

Antes de prosseguirmos para a próxima o próximo conteúdo, é importante que você estude, revise o conteúdo e realize pesquisas sobre os conceitos apresentados para ampliar seus conhecimentos.

Mídia digital: Antes de avançarmos os estudos, vá até a sala virtual e assista ao vídeo “Revisão da Terceira Semana” para recapitular tudo que aprendemos.

Nos encontramos na próxima semana.

Bons estudos!

Objetivos
- Conhecer os diferentes tipos de coleções de dados da linguagem Python;
- Empregar as coleções de dados mais adequadas na resolução de problemas;
- Escrever códigos usado coleções de dados.

4.1 Introdução

Os tipos de dados básicos permitem armazenar um único valor de um tipo de dado. Entretanto, em muitas situações, precisamos de estruturas de dados mais complexas, chamadas de coleções de dados, para agrupar diversos elementos de dados. A vantagem dessas estruturas está na facilidade de acesso e manipulação desses elementos de dados. Os principais tipos de coleções de dados são listas, tuplas, dicionários e conjuntos (PSF, 2021).

Para ilustrarmos a necessidade do uso de coleções de dados, vamos considerar o código da Figura 56. O código recebe o preço de 10 produtos e calcula o preço médio dos mesmos. Agora imagine que fosse necessário listar os produtos com preço acima da média. Como fazer isto de uma forma eficiente?

```
1 soma = 0
2 print('Informe o preço dos produtos')
3 for cont in range(10):
4     mensagem = 'Produto ' + str(cont+1) + ': '
5     preco = float(input(mensagem))
6     soma += preco
7 media = soma / 10
8 print('O preço médio é', media)
```

Figura 56 – Cálculo do preço médio de produtos
Fonte: Elaborado pelo Autor.

O problema das preços acima da média é que usamos os preços dos produtos para calcular a média e depois precisamos novamente desses preços. Uma solução seria declarar 10 variáveis uma para cada produto. Porém, imagine uma lista de 50 produtos ou mais, a declaração de variáveis individuais não é viável na prática.

Outra solução seria pedir para o usuário para informar os preços, realizar o cálculo da média e solicitar os preços novamente. Essa solução também não é eficiente, pois o usuário precisa digitar a lista de preços duas vezes. Além da sobrecarga do usuário, pode ocorrer uma digitação errada na segunda vez. A melhor solução para o problema é a utilizar uma coleção de dados para guardar os preços de todos os produtos e, depois do cálculo da média, revisitar a coleção para buscar aqueles produtos com preço acima da média.

4.2 Listas

As listas são estruturas dinâmicas e sequenciais de elementos (CORRÊA, 2020). Por ser dinâmica, podemos alterar a lista com a inclusão ou remoção de elementos. Além disso, por sua característica sequencial, dizemos que os elementos são indexados. Assim, podemos ler ou modificar os elementos da lista usando seu índice. O índice de um elemento é a sua posição dentro da lista, com a numeração começando em 0 (zero). A Figura 57 mostra uma representação de lista de números com suas respectivas posições. O elemento **10**, por exemplo, está na posição **4**.

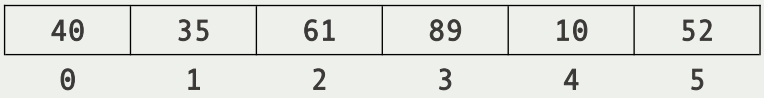


Figura 57 – Representação de lista de números
Fonte: Elaborado pelo Autor.

Em Python, a declaração de listas é feita colocando os elementos da lista separados por vírgula dentro de colchetes, por exemplo, **minha_lista = [1, 2, 3, 4, 5]**. Também podemos criar listas vazias simplesmente abrindo e fechando colchetes, por exemplo, **lista_vazia = []**. Além disso, as listas possuem diversas funções para facilitar sua manipulação. A Figura 58 apresenta algumas dessas funções.

Função	Funcionamento
l.append(x)	Adiciona o elemento **x** no final da lista **l**
l.insert(p, x)	Insere o elemento **x** na posição **p** da lista **l**
l.pop(p)	Remove e retorna o elemento da posição **p** de **l** (se **p** não for informado, a última posição é considerada)
l.clear()	Remove todos os elementos da lista **l**

Figura 58 – Algumas funções de listas
Fonte: Elaborado pelo Autor.

4.2.1 Resolvendo o problema dos preços acima da média com listas

O código da Figura 59 mostra uma possível solução para o problema dos produtos com preço acima da média utilizando listas. Na linha 2, criamos uma lista vazia para guardar os preços dos produtos. Dentro do primeiro laço de repetição, na linha 8, os preços são adicionados no final da lista com a função **append()**. O Segundo laço de repetição percorre os preços da lista e mostra apenas aqueles acima da média.

```
soma = 0
lista_precos = []
print('Informe o preço dos produtos')
for cont in range(10):
    mensagem = 'Produto ' + str(cont+1) + ': '
    preco = float(input(mensagem))
    soma += preco
    lista_precos.append(preco)
media = soma / 10
print('O preço médio é', media)
print('Os preços acima da média são:')
for preco in lista_precos:
    if preco > media:
        print(preco)
```

Figura 59 – Utilização de listas para resolver o problema dos preços acima da média
Fonte: Elaborado pelo Autor.

O código anterior mostra apenas os preços acima da média, mas não exibe os produtos com esses preços. Para resolver essa questão, no segundo laço, temos que percorrer as posições da lista usando o range. Essa solução é apresentada na Figura 60. Como estamos varrendo a lista pelos índices, usamos **lista_precos[cont]** para acessar o preço na posição **cont**. O código usa a função **range()** para gerar o intervalo de índices da lista. Uma alternativa é usar a função **enumerate()** que numera os elementos da lista e os retorna junto com seus índices. Essa solução alternativa é mostrada na Figura 61.

```
soma = 0
lista_precos = []
print('Informe o preço dos produtos')
for cont in range(10):
    mensagem = 'Produto ' + str(cont+1) + ': '
    preco = float(input(mensagem))
    soma += preco
    lista_precos.append(preco)
media = soma / 10
print('O preço médio é', media)
print('Os produtos com preço acima da média são:')
for cont in range(10):
    if lista_precos[cont] > media:
        print('Produto', cont+1)
        print('Preço: ', lista_precos[cont])
```

Figura 60 – Solução mostrando os produtos com preços acima da média
Fonte: Elaborado pelo Autor.

```
soma = 0
lista_precos = []
print('Informe o preço dos produtos')
for cont in range(10):
    mensagem = 'Produto ' + str(cont+1) + ': '
    preco = float(input(mensagem))
    soma += preco
    lista_precos.append(preco)
media = soma / 10
print('O preço médio é', media)
print('Os produtos com preço acima da média são:')
for cont, preco in enumerate(lista_precos):
    if preco > media:
        print('Produto', cont+1)
        print('Preço: ', preco)
```

Figura 61 – Solução mostrando os produtos com preços acima da média usando **enumerate()**
Fonte: Elaborado pelo Autor.

4.2.2 Matrizes

A Figura 62 exibe um código para fazer a cotação de preços de um produto com fornecedores e listar aqueles fornecedores com preço abaixo da média. O código realiza a cotação de um único produto com 5 fornecedores. Suponha agora que fosse necessário realizar a cotação de 5 produtos com 3 fornecedores e, em seguida, calcular o preço médio por produto e o preço médio por fornecedor. Uma solução seria criar uma lista para cada produto, mas essa solução não é muito adequada porque o número de produtos pode mudar. Nesse caso, seria interessante utilizar de matrizes.

```
soma = 0
lista_precos = []
print('Informe o preço do produto de cada fornecedor')
for cont in range(5):
    mensagem = 'Fornecedor ' + str(cont+1) + ': '
    preco = float(input(mensagem))
    soma = soma + preco
    lista_precos.append(preco)
media = soma / 5
print('O preço médio é', media)
print('Os fornecedores com preço abaixo da média são:')
for cont in range(5):
    if lista_precos[cont] < media:
        print('Fornecedor', cont, ', preço: ', lista_precos[cont])
```

Figura 62 – Solução mostrando os produtos com preços acima da média
Fonte: Elaborado pelo Autor.

Uma matriz é uma estrutura de dados com múltiplas dimensões. As matrizes mais simples são as matrizes bidimensionais que podem ser vistas como tabelas e seus elementos podem ser referenciados através de uma linha e uma coluna. Como exemplo, uma matriz 3x4 (3 linhas por 4 colunas) é representada na Figura 63. Nessa matriz, a posição [0,1] (linha 0, coluna 1) possui o elemento 89. Em Python, podemos implementar matrizes na forma de listas de listas.

		Colunas			
		0	1	2	3
Linhas	0	61	89	10	52
	1	30	23	17	28
	2	65	72	81	58

Figura 63 – Representação de matriz bidimensional
Fonte: Elaborado pelo Autor.

```
 1 NUM_PRODUTOS = 5
 2 NUM_FORNECEDORES = 3
 3
 4 mat_precos = []
 5 for cont_prod in range(NUM_PRODUTOS):
 6     print('Produto ' + str(cont_prod+1))
 7     print('Informe o preço de cada fornecedor')
 8     linha = []
 9     for cont_forn in range(NUM_FORNECEDORES):
10         mensagem = 'Fornecedor ' + str(cont_forn+1) + ': '
11         preco = float(input(mensagem))
12         linha.append(preco)
13     mat_precos.append(linha)
14
15 print('\nPreço médio dos produtos')
16 for cont_prod, linha in enumerate(mat_precos):
17     soma = 0
18     for preco in linha:
19         soma = soma + preco
20     media = soma / NUM_FORNECEDORES
21     print('Produto', cont_prod+1, ':', media)
22
23 print('\nPreço médio dos fornecedores')
24 for cont_forn in range(NUM_FORNECEDORES):
25     soma = 0
26     for cont_prod in range(NUM_PRODUTOS):
27         soma = soma + mat_precos[cont_prod][cont_forn]
28     media = soma / NUM_PRODUTOS
29     print('Fornecedor', cont_forn+1, ':', media)
```

Figura 64 – Preço médio de produtos e de fornecedores
Fonte: Elaborado pelo Autor.

O código da Figura 64 apresenta a solução do problema de preço médio de produtos e de fornecedores utilizando um matriz 5x3. Cada linha da matriz representa um produto e cada coluna representa um fornecedor. Os dois laços aninhados recebem todos os preços. A lista **linha** serve para pegar os preços dos fornecedores. Depois, essa lista é adicionada à lista **mat_precos** para compor a matriz.

Os preços médios dos produtos são calculados no segundo grupo de laços aninhados. O laço externo percorre as linhas (produtos), enquanto o laço interno percorre os preços de cada coluna (fornecedor). Como a matriz é representada por listas de linhas, não conseguimos percorrer diretamente coluna por coluna da matriz. Assim, no terceiro grupo de laços aninhados, percorremos as colunas e linhas usando a função **range()**. Na linha 27, usamos a notação

mat_precos[cont_prod][cont_forn] (**matriz[linha][coluna]**) para acessar o preço do for produto **cont_prod** (linha) do fornecedor **cont_forn** (coluna).

4.2.3 Inicialização e seleção de elementos

Em algumas situações, precisamos inicializar listas com certos valores ou com um determinado número de posições. Nesses casos, podemos fazer isso com a função **range()** ou com o operador *. A Figura 65 mostra alguns exemplos de inicialização de listas. A função **list()** converte o intervalo gerado pelo **range(10)** para o formato de lista.

```
1 # Lista inicializada com números de 0 a 9
2 lista = list(range(10))
3 print(lista)
4
5 # Lista de 10 posições com valores 0
6 lista = [0]*10
7 print(lista)
```

Figura 65 – Exemplos de inicialização de listas com **range()** e *
Fonte: Elaborado pelo Autor.

Outra maneira de inicializar listas é utilizando a compressão. A compressão consiste em escrever um código entre colchetes que gera uma lista de valores. Também podemos ler um texto e quebrá-lo em lista. Nesse caso, podemos quebrar o texto com a função **split()** e usar a função **len()** para obter o tamanho da lista. A Figura 66 exemplifica a compressão de listas e a criação de listas a partir de texto.

```
1 # Lista inicializada com números de 0 a 9
2 lista = [n for n in range(9)]
3 print(lista)
4 # Leitura de string e com split
5 texto = input('Informe números (separados com espaços): ')
6 lista = [int(x) for x in texto.split()]
7 print(lista)
8 print(len(lista))
```

Figura 66 – Exemplo de compressão e lista a partir de texto
Fonte: Elaborado pelo Autor.

Dica do Professor: A função **len()** é uma função da biblioteca padrão da linguagem Python que retorna a quantidade de elementos de listas ou de outros tipos de dados como texto, por exemplo.

Além da seleção de um único elemento pelo seu índice, as listas permitem diversos outros tipos de seleções. A utilização de índices negativos faz a seleção dos elementos a partir do final da lista. O índice **-1** representa o último elemento, **-2** é o penúltimo elemento, e assim por diante. Também podemos selecionar sub-listas, ou seja, pedaços da lista. No caso das sub-listas usamos a notação de colchetes após a lista indicando a posição inicial e final da sub-lista. A Figura 67 mostra alguns exemplos de sub-listas a partir de uma lista **l = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]**.

Para verificarmos se um elemento existe dentro de uma lista temos que varrer todos os seus elementos. Uma maneira de fazer isso escrevendo de forma mais concisa é utilizar o operador **in**. Basicamente, a instrução **x in l**, retorna **True** se o elemento **x** existir na lista **l**. De forma análoga, podemos, a expressão **x not in l**, retorna **True**, se o elemento **x** não estiver na lista **l**.

Notação	Resultado	Explicação
l[:]	**[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]**	Copia a lista
l[1:]	**[1, 2, 3, 4, 5, 6, 7, 8, 9]**	Todos os elementos a partir do segundo
l[:-1]	**[0, 1, 2, 3, 4, 5, 6, 7, 8]**	Todos os elementos até o penúltimo
l[3:7]	**[3, 4, 5, 6]**	Do quarto ao sétimo elemento

Figura 67 – Seleção de sub-listas de uma lista **l = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]**
Fonte: Elaborado pelo Autor.

4.2.4 Implementando o jogo da forca com listas

Para praticarmos o conteúdo sobre listas, vamos implementar o jogo da forca em Python. O jogo será de dois jogadores. Um jogador informa a palavra e o outro tenta descobrir. O jogo deve permitir apenas palavras sem espaços e sem caracteres especiais. Assim, vamos começar criando a função **valida_palavra()** para verificar se a palavra informada é válida. O Código da função é mostrado na Figura 68. A função percorre todas as letras da palavra e verifica se as mesmas são caracteres de **'A'** a **'Z'**. Se alguma letra inválida for encontrada, e função retorna **False**. Se a palavra for válida a função retorna **True**.

```
1 def valida_palavra(palavra):
2     for letra in palavra:
3         if letra < 'A' or letra > 'Z':
4             return False
5     return True
```

Figura 68 – Função **valida_palavra()** para o jogo da forca
Fonte: Elaborado pelo Autor.

Vamos criar também a função **novo_jogo()** para pedir a palavra ao jogador. A Figura 69 apresenta o código dessa função. Usamos um laço de repetição para garantir que o jogador digite uma palavra correta. Enquanto a palavra não for válida, a função pede para o jogador digitar novamente. Na linha 5 recebemos a palavra digitada pelo usuário e fazemos alguns tratamentos no texto. A função **upper()** é usada para converter o texto para maiúsculo e a função **strip()** remove possíveis espaços em branco no início e final do texto.

```
1 def novo_jogo():
2     while True:
3         print('Informe a palavra para seu adversário')
4         print('Não use espaços ou caracteres especiais')
5         palavra = input().upper().strip()
6         if valida_palavra(palavra):
7             return palavra
8         else:
9             print('Palavra inválida! Tente novamente.')
```

Figura 69 – Função **novo_jogo()** para o jogo da forca
Fonte: Elaborado pelo Autor.

Depois que a palavra for informada, precisamos de mais três informações para continuar o jogo. Vamos usar a lista **letras_tela** com o mesmo número de caracteres da palavra do jogo para mostrar as letas já descobertas. Inicialmente, todos os caracteres dessa lista serão **'_'**. Por exemplo, se a palavra do jogo for **'PAPEL'** teremos **letras_tela = ['_', '_', '_', '_', '_']**. A cada letra que o jogador acertar, trocamos o caractere **'_'** correspondente pela letra acertada. Se a primeira letra digitada for **'A'**, por exemplo, teremos **letras_tela = ['_', 'A', '_', '_', '_']**.

```
1 def tem_letra(palavra, letra, letras_tela):
2     encontrada = False
3     for cont, carac in enumerate(palavra):
4         if carac == letra:
5             encontrada = True
6             letras_tela[cont] = letra
7     return encontrada
```

Figura 70 – Função **tem_letra()** para o jogo da forca
Fonte: Elaborado pelo Autor.

O código da Figura 70 apresenta a função **tem_letra()** para controlar a troca de letras da maneira que foi explicado. A função inicializa a variável **encontrada** como **False** supondo que a **letra** não será encontrada na **palavra**. O laço de repetição varre todos os caracteres da **palavra**. O encontrar um caractere igual a **letra**, a função coloca a letra na posição (**cont**) correspondente da lista **letras_tela**. Por fim, a função retorna **True**, se a **letra** for encontrada na **palavra**.

Nós também precisamos de uma função para mostrar o estado atual do jogo (letras certas, letras digitadas e erros). A Figura 71 exibe o código contendo essa função. O **print()** da linha 2 escreve 100 linhas em branco na tela para que o jogador não veja a palavra digitada no início do jogo. Na linha 3, usamos a função **join()** para mostra as letras já acertadas na tela (**letras_tela**). A função **join()** permite juntar elementos textuais de uma lista usando um delimitador. Por exemplos, se temos **d=';'** e **l = ['a', 'b', 'c', 'd']** a instrução **d.join(l)** retorna o texto **'a;b;d;d'**. No nosso caso, usamos o texto vazio (**''**) como delimitador, então, como resultado, temos as letras de **letras_tela** todas juntas.

```
1 def mostra_jogo(letras_tela, digitadas, erros):
2     print('\n'*100)
3     print(''.join(letras_tela))
4     print('Letras digitadas:', digitadas)
5     print('Erros:', erros)
```

Figura 71 – Função **mostra_jogo()** para o jogo da forca
Fonte: Elaborado pelo Autor.

```
1 def valida_letra(letra, digitadas):
2     if len(letra) != 1:
3         return False
4     if letra < 'A' or letra > 'Z' or letra in digitadas:
5         return False
6     return True
```

Figura 72 – Função **valida_letra()** para o jogo da forca
Fonte: Elaborado pelo Autor.

Assim como validamos a palavra digitada no início do jogo, temos que validar as letras digitadas pelo jogador adversário durante o jogo. A Figura 72 mostra o código contendo a função **valida_letra()** para realizar essa tarefa. A função retorna **True** se a letra válida e **False** em caso

negativo. Na linha 2, testamos se o jogador digitou uma única letra. Na linha 4, verificamos se a letra é válida (de **'A'** a **'Z'**) e se já foi digitada.

```
 1 def principal():
 2     palavra = novo_jogo()
 3     letras_tela = ['_'] * len(palavra)
 4     erros = 0
 5     digitadas = ''
 6     while True:
 7         mostra_jogo(letras_tela, digitadas, erros)
 8         letra = input('Informe uma letra: ').upper().strip()
 9         if not valida_letra(letra, digitadas):
10             continue
11         digitadas += letra
12         if not tem_letra(palavra, letra, letras_tela):
13             erros += 1
14             if erros == 5:
15                 print('Você perdeu!')
16                 break
17         if '_' not in letras_tela:
18             print('Você acertou!')
19             break
20
22 if __name__ == '__main__':
23     principal()
```

Figura 73 – Função **principal()** para o jogo da forca
Fonte: Elaborado pelo Autor.

Por fim, apresentamos a função **principal()** na Figura 73. Na linha 2, pegamos a palavra a ser descoberta no jogo. A seguir, inicializamos os dados do jogo. A variável **letras_tela** recebe a lista de caracteres **'_'** que serão substituídos a media que o jogador acertar as letras. O número inicial de erros é zero e a variável **digitadas**, inicializada com texto vazio, representa as letras já digitadas.

O laço de repetição é usado para que novas letras sejam solicitadas até que o jogador tenha cinco erros ou acerte a palavra. Dentro do laço, mostrados o estado do jogo com a função **mostra_jogo()** e pegamos a letra digitada. Novamente, usamos **upper()** e **strip()** para remover espaços e deixar a letra maiúscula. Em seguida, usamos a função **valida_letra()** para verificar se a letra é válida. Em caso negativo, usamos a instrução **continue** para ignorar as demais instruções do laço e passar para a próxima repetição. Assim, o usuário terá que digitar a letra novamente.

Se a letra for válida, adicionamos a mesma na variável **digitadas** (na linha 11) e fazemos mais dois testes. O teste da linha 12 verifica se a letra não está na palavra do jogo. Nessa situação, adicionamos um erro para o jogador (na linha 13) e verificamos se o limite de erros foi atingido (na linha 14). Quando o número de erros chegar a cinco, o jogador perde, o **break** da linha 16 finaliza o laço e o jogo termina.

O **if** da linha 17 verifica se o jogador já acertou todas as letras. Isso acontece quando não temos mais o caractere **'_'** na lista **letras_tela**. Assim, o jogador vence e o jogo termina.

4.3 Tuplas

As tuplas são estruturas usadas para agrupar uma quantidade fixa de elementos. A especificação de tuplas é feita escrevendo seus elementos separados por vírgula. Se a tupla tiver um único elemento, é preciso incluir uma vírgula após esse elemento. Normalmente, por uma questão de legibilidade, colocamos essa lista de elementos entre parênteses. As tuplas são recomendadas para agrupar quantidades fixas e pequenas de elementos.

Uma aplicação interessante para tuplas é o agrupamento de dia, mês e ano em uma data. A Figura 74 ilustra essa aplicação. Observe que as tuplas de data são criadas com ano, mês e dia, nessa ordem. Isso possibilita comparar as tuplas diretamente como foi feito na linha 13. A comparação é feita considerando o primeiro elemento, depois o segundo e assim por diante.

```
 1 print('Informe as datas')
 2 print('Primeira data')
 3 dia = int(input('Dia: '))
 4 mes = int(input('Mês: '))
 5 ano = int(input('Ano: '))
 6 data1 = (ano, mes, dia)
 7 print('Segunda data')
 8 dia = int(input('Dia: '))
 9 mes = int(input('Mês: '))
10 ano = int(input('Ano: '))
11 data2 = (ano, mes, dia)
12 recente = data1
13 if data2 > data1:
14     recente = data2
15 print('Data mais recente:', recente)
```

Figura 74 – Utilização de tuplas para representar datas
Fonte: Elaborado pelo Autor.

Cada elemento de uma tupla possui um índice, começando pelo zero (DOWNEY, 2015). Assim, podemos usar a instrução **t[n]** para acessar o elemento da tupla **t** na posição **n-1**. Os índices dos elementos das tuplas representando datas podem vistos na Figura 75. No entanto, é importante frisar que as tuplas são tipos imutáveis. Assim, podemos ler seus elementos, mas não podemos alterá-los.

0	1	2
ano	mes	dia

Figura 75 – Índices dos elementos de uma tupla representando data
Fonte: Elaborado pelo Autor.

Outra peculiaridade das tuplas é a possibilidade de atribuir o valor dos elementos de uma tupla a um conjunto de variáveis em uma única instrução no formato **a, ..., x = t**. Tal instrução é valida desde que o número de variáveis seja igual ao número de elementos da tupla.

Outra aplicação interessante de tuplas é no problema dos preços acima da média visto anteriormente. Além do preço, seria interessante guardar o nome do produto para mostrar quando necessário. Podemos podemos fazer isso agrupando esses dados em

uma tupla. A Figura 76 demonstra como podemos implementar o código. Nas linhas 6 e 7 pegamos o nome e o preço do produto, respectivamente. Em seguida, juntamos os dados em uma tupla (linha 8). Na linha 10, adicionamos a tupla à lista de produtos. No segundo laço de repetição percorremos os produtos da lista e extraímos seu nome e preço. O teste da linha 16 é usado para verificar se o preço do produto está acima da média.

```
 1 soma = 0
 2 lista_produtos = []
 3 print('Informe o dados dos produtos')
 4 for cont in range(10):
 5     print('\nProduto ' , cont+1)
 6     nome = input('Nome: ')
 7     preco = float(input('Preço: '))
 8     produto = (nome, preco)
 9     soma += preco
10     lista_produtos.append(produto)
11 media = soma / 10
12 print('O preço médio é', media)
13 print('Os produtos com preço acima da média são:')
14 for produto in lista_produtos:
15     nome, preco = produto
16     if preco > media:
17         print('Produto:', nome)
18         print('Preço:', preco)
```

Figura 76 – Produtos acima da média usando tuplas
Fonte: Elaborado pelo Autor.

4.4 Conjuntos

Os conjuntos são coleções não ordenadas de elementos (CORRÊA, 2020). Eles devem ser utilizados quando a existência de um elemento na coleção é mais importante do que a ordem do elemento ou do que a quantidade de vezes que o elemento aparece. Na verdade, os conjuntos em Python, assim como na matemática, não possuem elementos repetidos.

Podemos criar um conjunto com elementos separados por vírgula entre parênteses ou usar o construtor **set()** (TAGLIAFERRI, 2018). No caso do construtor, temos que passar uma coleção de elementos, como uma lista, por exemplo. Conjuntos vazios devem ser criados com o construtor **set()** sem sem nenhum parâmetro. Para exemplificar as operações sobre conjuntos, vamos considerar os conjuntos **s1 = {1, 2, 3, 4}**, **s2 = {4, 5, 6}** e **s3 = {4, 5}**. A Figura 77 mostra algumas operações sobre esses conjuntos na linguagem Python e a notação matemática correspondente. Observe que as operações mostradas podem ser feitas com operadores ou funções.

Notação Matemática	Código Python		Resultado
	Operadores	Funções	
$s_1 \cap s_2$	s1 & s2	s1.intersect(s2)	{4}
$s_1 \cup s_2$	s1 | s2	s1.union(s3)	{1, 2, 3, 4, 5, 6}
$s_1 - s_2$	s1 - s2	s1.difference(s2)	{1, 2, 3}
$s_3 \subseteq s_2$	s3 <= s2	s3.issubset(s2)	True
$s_1 \supseteq s_3$	s1 >= s3	s1.issuperset(s3)	False

Figura 77 – Operações sobre conjuntos de dados
Fonte: Elaborado pelo Autor.

Além das operações entre conjuntos podemos verificar se um elemento existe no conjunto como operador **in**. Também podemos adicionar e remover elementos com as operações **add()** e **remove()**, respectivamente.

Para fixar os conteúdos já apresentados, vamos agora desenvolver um jogo de bingo. Inicialmente, vamos importar função **shuffle()** da biblioteca **random** e declarar o total de números como mostrado na Figura 78. Se você preferir, pode usar uma quantidade diferente de números para o jogo. Em seguida, criamos uma função que gera uma lista embaralhada de números. O código dessa função é mostrado na Figura 79. Na linha 2, criamos a lista com os números de 1 a 80 e, na linha 3, usamos a função **shuffle()** para embaralhar essa lista.

```
1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 from random import shuffle
5
6 TOTAL_NUM = 80
```

Figura 78 – Código inicial do jogo de bingo
Fonte: Elaborado pelo Autor.

```
1 def gera_numeros():
2     lista_num = list(range(1, TOTAL_NUM-1))
3     shuffle(lista_num)
4     return lista_num
```

Figura 79 – Função **gera_numeros()** do jogo de bingo
Fonte: Elaborado pelo Autor.

Cada jogador deverá possuir uma cartela com os seus números. O código da Figura 80 demostra como gerar uma cartela de um jogador. Nós usamos a função **gera_numeros()** para gerar os números embaralhados e tomar 30% desses números para compor a cartela. Observe que, na linha 4, a cartela é gerada na forma de conjunto porque a ordem dos números não importa mais.

```
1 def gera_cartela():
2     lista_num = gera_numeros()
3     cartela_num = int(TOTAL_NUM*0.3)
4     cartela_set = set(lista_num[:cartela_num])
5     return cartela_set
```

Figura 80 – Função **gera_cartela()** do jogo de bingo
Fonte: Elaborado pelo Autor.

A partir da função que gera a cartela, podemos criar uma função para gerar o lista de jogadores como mostrado na Figura 81. A função **gera_jogadores()** recebe o número de jogadores e, para cada jogador, obtém seu nome e sua cartela de números. Na linha 7, criamos uma tupla composta pelo nome e pelo conjunto de números do jogador (cartela). Em seguida, adicionamos a tupla à lista de jogadores.

```
1 def gera_jogadores(num_jogadores):
2     lista_jogadores = []
3     for cont in range(num_jogadores):
4         print('\n\nJogador', cont+1)
5         nome = input('Nome do jogador: ')
6         cartela = gera_cartela()
7         jogador = (nome, cartela)
8         lista_jogadores.append(jogador)
9     return lista_jogadores
```

Figura 81 – Função **gera_jogadores()** do jogo de bingo
Fonte: Elaborado pelo Autor.

Durante o jogo, vamos sortear os números e removê-los das cartelas dos jogadores. Assim, o jogador que ficar sem números será o vencedor. O código da Figura 82 apresenta a função para remover o número sorteado das cartelas dos jogadores. Basicamente, percorremos a lista de jogadores, e removemos o número de suas cartelas. Como a lista de jogadores é uma lista de tuplas, podemos fazer com que o laço pegue os dois elementos da tupla ao mesmo tempo. Usamos a variável anônima (_) para descartar o nome do jogador, pois esse dado não será necessário. O **if** da linha 3 verifica se a cartela realmente possui o número antes de removê-lo.

```
1 def remove_numero(lista_jogadores, numero):
2     for _, cartela in lista_jogadores:
3         if numero in cartela:
4             cartela.remove(numero)
```

Figura 82 – Função **remove_numero()** do jogo de bingo
Fonte: Elaborado pelo Autor.

Para mostrar os números sorteados aos jogadores, vamos manter uma lista com os mesmos. Além disso, a cada número sorteado, temos que mostrar o estado atual do jogo. A Figura 83 mostra o código responsável por essa tarefa. A função mostra os números já sorteados e os números de cada jogador.

```
1 def mostra_jogo(sorteados, lista_jogadores):
2     print('*****************************')
3     print('*           BINGO           *')
4     print('*****************************')
5     print('\nNúmeros sorteados:', sorteados, '\n')
6     for jogador in lista_jogadores:
7         nome, cartela = jogador
8         print(nome, ': ', cartela, sep='')
```

Figura 83 – Função **mostra_jogo()** do jogo de bingo
Fonte: Elaborado pelo Autor.

Outra tarefa importante é verificar se algum jogador já venceu o jogo. Como mencionamos, quando os jogadores que ficarem sem números serão os vencedores. O código da Figura 84 mostra a função que verifica se já temos jogadores vencedores. Na linha 2, a função cria um

conjunto vazio para adicionar os vencedores. Em seguida, o laço de repetição varre a lista de jogadores. A linha 4 testa se o jogador está com a cartela vazia. Em caso afirmativo, o nome desse jogador é adicionado ao conjunto de vencedores. Ao final, a função retorna o conjunto com os nomes dos vencedores. No início do jogo, quando nenhum jogador tem cartela vazia, a função retorna um conjunto vazio de vencedores.

```
1 def busca_vencedores(lista_jogadores):
2     vencedores = set()
3     for nome, cartela in lista_jogadores:
4         if len(cartela) == 0:
5             vencedores.add(nome)
6     return vencedores
```

Figura 84 – Função **busca_vencedores()** do jogo de bingo
Fonte: Elaborado pelo Autor.

```
1 def principal():
2     print('Iniciando o bingo')
3     num_jog = int(input('Informe o número de jogadores: '))
4     lista_jogadores = gera_jogadores(num_jog)
5     lista_numeros = gera_numeros()
6     sorteados = set()
7     while True:
8         mostra_jogo(sorteados, lista_jogadores)
9         print('Pressione ENTER para sortear um número')
10        input()
11        num = lista_numeros.pop()
12        sorteados.add(num)
13        remove_numero(lista_jogadores, num)
14        vencedores = busca_vencedores(lista_jogadores)
15        if len(vencedores) > 0:
16            mostra_jogo(sorteados, lista_jogadores)
17            print('\nBingo!')
18            print('Vencedor(res):', vencedores)
19            break
20
21 if __name__ == '__main__':
22     principal()
```

Figura 85 – Função **principal()** do jogo de bingo
Fonte: Elaborado pelo Autor.

Por fim, a Figura 85 mostra a função principal do jogo. Primeiro, criamos a lista de jogadores, geramos a lista com todos os números embaralhados e inicializamos o conjunto de números sorteados. Na linha 7, começamos o laço que se repete a cada número sorteado.

Dentro do laço, mostramos o estado atual do jogo e sorteamos um número na lina 11. Como a lista de números já está embaralhada, podemos simplesmente retirar o último elemento da lista. Em seguida, adicionamos o número sorteado ao conjunto **sorteados** e chamamos a função **remove_numero()** para remover o número sorteado da cartela dos jogadores.

Na linha 14, chamamos a função **busca_vencedores** para e, na linha 15, verificamos se já existe algum vencedor. Quando existir algum vencedor, exibimos o estado do jogo mais uma vez , mostramos os vencedores e interrompemos o laço de repetição.

4.5 Dicionários

Um dicionário é um tipo especial de coleção que faz mapeamento ou associação de chave-valor (CORRÊA, 2020). Isso significa que, para cada chave do dicionário, existe um valor associado. A chave deve ser um tipo de dado imutável, mas o valor pode ser qualquer tipo de dado. Os tipos imutáveis são os tipos de dados básicos e outros dados que não podem sofres modificações como, por exemplo, tuplas compostas de dados imutáveis.

A maneira mais comum de criar dicionários com lista de chaves e valores (chave:valor) entre chaves. Um dicionário vazio pode ser criado com **{}** (abre e fecha chaves). Considerando um dicionário **d** e uma chave **c**, podemos consultar o valor para a chave **c** como comando **d[c]**. Já a adição ou alteração do valor pode ser feita com a instrução **d[c] = v**, onde **v** o valor a ser atribuído à chave. A Figura 86 mostra um exemplo simples de código que cria e manipula dicionários.

```
1 dic_vazio = {}
2 dic_letras = {1:'A', 2:'B', 3:'C'}
3 dic_letras[4] = 'E'
4 dic_letras[4] = 'D'
5 print(dic_vazio, dic_letras)
6 for cont in range(1, 5):
7     print(cont, ':', dic_letras[cont])
```

Figura 86 – Exemplo simples com dicionários
Fonte: Elaborado pelo Autor.

Além das manipulações usando chaves e colchetes, os dicionários possuem diversas funções que facilitam seu tratamento. A Figura 87 exibe algumas dessas funções e e seu funcionamento.

Função	Funcionamento
d.clear()	Remove todos os elementos do dicionário **d**
d.copy()	Retorna uma cópia de **d**
d.items()	Retorna as chaves-valores de **d** no formato de tuplas
d.keys()	Retorna uma lista com as chaves de **d**
d.popitem()	Retorna uma tupla (chave, valor) qualquer(se **d** estiver vazio, ocorre um erro)
d.update(d2)	Atualiza os elementos de **d** utilizando os elementos de **d2**
d.values()	Retorna uma lista com os valores de **d**

Figura 87 – Algumas funções de dicionários
Fonte: Elaborado pelo Autor.

Para fixarmos os conteúdos estudados até o momento, vamos implementar um script para simular um caixa de supermercado. Basicamente, o script permitirá o cadastro e listagem de produtos e suas vendas. Vamos começar com a declaração de algumas variáveis globais que serão usadas no cadastro de produtos como descrito na Figura 88.

```
1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 DESCRICAO = 'DESCRIÇÃO'
5 ESTOQUE = 'ESTOQUE'
6 PRECO = 'PREÇO'
```

Figura 88 – Variáveis globais do caixa de supermercado
Fonte: Elaborado pelo Autor.

A primeira função que criaremos será usada para cadastrar novos produtos ou atualizar produtos existentes. Vamos armazenar cada produto como um dicionário contendo chaves para sua descrição, estoque e preço. Vamos usar também um dicionário de produto onde a chave de cada produto será seu código. Dessa forma, teremos um dicionário de dicionários.

```
1 def cad_produto(dic_produtos):
2     print('Cadastrar/atualizar produto')
3     codigo = int(input('Código do produto: '))
4     if codigo in dic_produtos:
5         produto = dic_produtos[codigo]
6         print('\nProduto existente:\n', produto)
7     else:
8         print('\nProduto não encontrado')
9         descricao = input('Informe a descrição: ')
10        produto = {DESCRICAO: descricao}
11        dic_produtos[codigo] = produto
12    produto[ESTOQUE] = float(input('Estoque: '))
13    produto[PRECO] = float(input('Preço: '))
```

Figura 89 – Função **cad_produto()** do caixa de supermercado
Fonte: Elaborado pelo Autor.

A Figura 89 mostra o código da função responsável pelo cadastro de produtos no dicionário **dic_produtos**. Após solicitar o código ao usuário, a função verifica se o produto já existe (linha 4). Em caso afirmativo, pegamos o produto no dicionário (linha 5) e mostramos na tela. Se o produto não existir, criamos um dicionário para o mesmo contendo a descrição informada pelo usuário (linha 10) e inserimos o produto no dicionário de produtos (linha 11). Nas linhas 12 e 13, atualizamos (para produtos existentes) ou acrescentamos (para novos produtos) o estoque e o preço.

Vamos criar também uma função para listar os produtos do dicionário **dic_produtos()** como apresentado na Figura 90. Se o dicionário estiver vazio (**if** da linha 2), exibimos uma mensagem informativa e saímos da função com o **return** da linha 4. Se o dicionário contiver produtos, o laço de repetição percorre o dicionário e escreve todos os produtos na tela. Usamos um **input()** na linha 7 para que o usuário possa ver a lista de produtos e, depois, pressionar a tecla ENTER. Isso é necessário porque teremos outras funções para escrever na tela. Se não fizermos essa "parada" com o **input()**, outras informações podem aparecer depois da lista de produtos e prejudicarem a visualização dos produtos pelo usuário.

```
1 def listar_produtos(dic_produtos):
2     if len(dic_produtos) == 0:
3         print('\nNenhum produto encontrado!')
4         return
5     for codigo, produto in dic_produtos.items():
6         print('CÓDIGO:', codigo, produto)
7     input('Pressine ENTER para voltar')
```

Figura 90 – Função **listar_produtos()** do caixa de supermercado
Fonte: Elaborado pelo Autor.

Com os produtos cadastrados, o sistema deverá permitir a venda dos mesmos. A Figura 91 exibe o código da função **venda()** que realiza essa tarefa. A função recebe como parâmetros o dicionário de produtos e a lista de vendas. A lista de vendas é usada para salvar as vendas efetivadas. A função começa solicitado o código do produto a ser vendido. Na linha 4, a função verificar se o código é inválido (produto inexistente) e, nesse caso, finaliza sem realizar a venda (linha 6).

```
1 def venda(dic_produtos, lista_vendas):
2     print('\nVenda de produto')
3     codigo = int(input('Código do produto: '))
4     if codigo not in dic_produtos:
5         print('Produto não encontrado!')
6         return
7     produto = dic_produtos[codigo]
8     print('\nProduto:\n', produto)
9     quant = float(input('Quantidade vendida: '))
10    if quant > produto[ESTOQUE]:
11        print('\nEstoque insuficiente!')
12        return
13    print('Total da venda:', produto[PRECO] * quant)
14    venda = (codigo, quant, produto[PRECO])
15    produto[ESTOQUE] -= quant
16    lista_vendas.append(venda)
```

Figura 91 – Função **venda()** do caixa de supermercado
Fonte: Elaborado pelo Autor.

Se o código do produto for válido, a função busca esse produto no dicionário **dic_produtos** e solicita a quantidade a ser vendida ao usuário. Em seguida, na linha 10, a função verifica se a quantidade é maior do que o estoque do produto. Em caso afirmativo, a venda não é efetivada. Por fim, se o código do produto e a quantidade vendida forem válidos, função prossegue mostrando as informações da venda e atualizando o dados. A atualização dos dados consiste em fazer a retirada do estoque (linha 15) e adicionar a venda à lista de vendas.

Nosso caixa de supermercado vai precisar também de um relatório de vendas para listar todas as vendas já realizadas. A Figura 92 apresenta o código para emitir tal relatório. Primeiro, a função informa que nenhuma venda foi encontrada, se a **lista_vendas** estiver vazia (linha 2). Por outro lado, se houverem vendas, a função inicializa a variável **total_geral** e percorre todas as vendas da lista.

Para cada venda a função, exibe as informações do produto, quantidade vendida, preço e total. Além disso, o total da venda é somado ao total geral que, por sua vez, é exibido no final do relatório.

```
 1 def relatorio_vendas(dic_produtos, lista_vendas):
 2     if len(lista_vendas) == 0:
 3         print('Nenhuma venda encontrada!')
 4         return
 5     total_geral = 0
 6     for codigo, quant, preco in lista_vendas:
 7         produto = dic_produtos[codigo]
 8         print('CÓDIGO:', codigo, end=' ')
 9         print('| PRODUTO:', produto[DESCRICAO], end=' ')
10         print('| QUANTIDADE:', quant, end=' ')
11         print('| PREÇO:', preco, end=' ')
12         total = quant * preco
13         print('| TOTAL:', total)
14         total_geral += total
15     print('TOTAL GERAL:', total_geral)
16     input('Pressine ENTER para voltar')
```

Figura 92 – Função **relatorio_vendas()** do caixa de supermercado
Fonte: Elaborado pelo Autor.

Para utilizar todas essas funções, teremos que mostrar um menu de opções para o usuário e pegar a opção escolhida. A função da Figura 93 escreve o menu descrito na tela e captura a escolha do usuário.

```
 1 def menu():
 2     print('\n*****************************')
 3     print('*           CAIXA           *')
 4     print('*****************************')
 5     print('(C)adastrar/atualizar produto')
 6     print('(L)istar produtos             ')
 7     print('(V)ender produto              ')
 8     print('(R)elatório de vendas         ')
 9     print('(S)air                        ')
10     print('*****************************')
11     escolha = input('Informe sua opção: ').upper()
12     return escolha
```

Figura 93 – Função **menu()** do caixa de supermercado
Fonte: Elaborado pelo Autor.

Finalmente, podemos construir a função principal que controlará o fluxo de execução do código. Essa função é mostrada na Figura 94. No início, a função cria a lista de vendas (**lista_vendas**) e o dicionário de produtos (**dic_produtos**). Em seguida, começa o laço de repetição que é executado até que o usuário decida sair. Dentro do laço de repetição, chamamos a função **menu()** para obtermos a opção desejada pelo usuário. Em seguida, as demais funções são chamadas de acordo com a escolha do usuário. Se o usuário escolher sair, o **break** da linha 15 é executado, interrompendo o laço de repetição e encerrando a execução do código.

```
def principal():
    lista_vendas = []
    dic_produtos = {}
    while True:
        escolha = menu()
        if escolha == 'C':
            cad_produto(dic_produtos)
        elif escolha == 'L':
            listar_produtos(dic_produtos)
        elif escolha == 'V':
            venda(dic_produtos, lista_vendas)
        elif escolha == 'R':
            relatorio_vendas(dic_produtos, lista_vendas)
        elif escolha == 'S':
            break

if __name__ == '__main__':
    principal()
```

Figura 94 – Função **principal()** do caixa de supermercado
Fonte: Elaborado pelo Autor.

Atividade: Resolva os exercícios a seguir. As respostas estão na próxima página, mas é importante que você tente resolver por conta própria e use as respostas apenas para conferência.

4.6 Exercícios

Escreva os códigos em Python para resolver os problemas a seguir:

A) Criar uma função para resolver equações de segundo grau retornando as raízes dentro de uma tupla. Crie também um função principal para receber os termos A, B e C da equação e mostrar o resultado para o usuário.

B) Construir um script capaz de calcular a velocidade média de uma viagem a partir das velocidades de cada trecho. O usuário deve informar o número de trechos e, em seguida, informar a distância e velocidade de cada trecho. O cálculo da velocidade média é feito somando o produto da distância pela velocidade de cada trecho e dividindo essa soma pela soma das distâncias. Após o cálculo da velocidade média, o programa deve os trechos com velocidade acima da média.

C) Escrever um código para receber uma lista de números. Percorrer a lista e utilizar um dicionário para sumarizar a quantidade de vezes que cada número aparece na lista.

4.7 Respostas dos exercícios

Escreva os códigos em Python para resolver os problemas a seguir:

A) Criar uma função para resolver equações de segundo grau retornando as raízes dentro de uma tupla. Crie também um função principal para receber os termos A, B e C da equação e mostrar o resultado para o usuário.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import math

def raizes_equacao(a, b, c):
    delta = b**2 - 4 * a * c
    if a == 0 or delta < 0:
        return ()
    elif (delta == 0):
        x1 = b * (-1) / 2 * a
        return (x1,)
    else:
        raiz_delta = math.sqrt(delta)
        x1 = (b * (-1) + raiz_delta)/ 2 * a
        x2 = (b * (-1) - raiz_delta)/ 2 * a
        return (x1, x2)

def principal():
    print('Informe os termos da equação Ax² + Bx + C')
    a = float(input('A: '))
    b = float(input('B: '))
    c = float(input('C: '))
    raizes = raizes_equacao(a, b, c)
    if len(raizes) == 0:
        print('\nA equação não é uma equação de segundo grau')
    elif len(raizes) == 1:
        print('\nA raiz da equação é x =', raizes[0])
    else:
        print('\nAs raizes da equação são ', end='')
        print('x1 =', raizes[0], ' e x2 =', raizes[1])

if __name__ == '__main__':
    principal()
```

B) Construir um script capaz de calcular a velocidade média de uma viagem a partir das velocidades de cada trecho. O usuário deve informar o número de trechos e, em seguida, informar a distância e velocidade de cada trecho. O cálculo da velocidade média é feito somando o produto da distância pela velocidade de cada trecho e dividindo essa soma pela soma das distâncias. Após o cálculo da velocidade média, o programa deve os trechos com velocidade acima da média.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

num_trechos = int(input('Informe o número de trechos: '))

lista_trechos = []
soma_dist = 0
soma_veloc_dist = 0
for cont in range(num_trechos):
    print('------------------------------')
    print('Trecho ', cont+1)
    dist = float(input('Distância: '))
    soma_dist += dist
    veloc = float(input('Velocidade: '))
    trecho = (dist, veloc)
    lista_trechos.append(trecho)
    soma_veloc_dist += veloc*dist

veloc_med = soma_veloc_dist/soma_dist
print('\nVelocidade média:', veloc_med)

print('\nTrechos com velocidade acima da média:')
for cont in range(num_trechos):
    dist, veloc = lista_trechos[cont]
    if veloc > veloc_med:
        print('Trecho ', cont+1 , ', distância =', dist,
              ', velocidade =', veloc)
```

C) Escrever um código para receber uma lista de números. Percorrer a lista e utilizar um dicionário para sumarizar a quantidade de vezes que cada número aparece na lista.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

lista_num = []
while True:
    n = int(input('Informe um número (0 para parar):'))
    if n == 0:
        break
    lista_num.append(n)

dic_num = {}
for n in lista_num:
    if n in dic_num:
        dic_num[n] += 1
    else:
        dic_num[n] = 1

print('\nInformações sobre os números')
for n, quant in dic_num.items():
    print('Número', n, ':', quant)
```

4.8 Revisão

Antes de finalizarmos o curso, é importante que você estude, revise o conteúdo e realize pesquisas sobre os conceitos apresentados para ampliar seus conhecimentos.

Mídia digital: Antes de finalizarmos, vá até a sala virtual e assista ao vídeo “Revisão da Quarta Semana” para recapitular tudo que aprendemos.

Bons estudos!

Finalizando o curso

A prática é uma atividade fundamental para um bom aprendizado da lógica de programação e de qualquer linguagem de programação. Uma boa tarefa prática interessante é revisitar os problemas e exercícios estudados e tentar resolvê-los por conta própria. Além disso, você pode se aprofundar mais em diversos conteúdos usando as referencias citadas no livro.

Recomendamos também que você sempre busque pelo constante aprimoramento profissional. Você pode encontrar diversos conteúdos interessantes na Plataforma +IFMG (https://mais.ifmg.edu.br).

Atividade final

Atividade: Para concluir o curso e gerar o seu certificado, vá até a sala virtual e responda ao Questionário “Avaliação final”. Este teste é constituído por 10 perguntas de múltipla escolha, que se baseiam em todo o conteúdo estudado.

Referências

BORGES, L. E. **Python para Desenvolvedores**. 2. ed. Rio de Janeiro: Edição do Autor, 2010. Disponível em: https://ricardoduarte.github.io/python-para-desenvolvedores/

CEDER, N. The **Quick Python Book**. 3. ed. Shelter Island: Manning Publications, 2018. Disponível em: https://livebook.manning.com/book/the-quick-python-book-third-edition/

CORRÊA, E. **Meu primeiro livro de Python**. 2. ed. Rio de Janeiro: Edubd, 2020. Disponível em: https://github.com/edubd/meu_primeiro_livro_de_python

DOWNEY, A. B. T**hink Python**: How to Think Like a Computer Scientist. 2. ed. Needham: Green Tea Press, 2015. Disponível em: https://greenteapress.com/wp/think-python-2e/

GOMES, B. E. G.; BARROS, T. M. **Fundamentos de Lógica e Algoritmos**. Natal: IFRN Editora, 2015. Disponível em: http://proedu.rnp.br/handle/123456789/1381

MENEZES, N. N. C. **Introdução à programação com Python**: algoritmos e lógica de programação para iniciantes. 3. ed. São Paulo: Novatec, 2019.

PILGRIM, M. **Dive Into Python 3**. New York: Apress, 2009. Disponível em: https://diveintopython3.net/

PYPL. **PYPL**: PopularitY of Programming Language. 2021. Disponível em: https://pypl.github.io/PYPL.html

PYTHON SOFTWARE FOUNDATION (PSF). **Python 3.10.1 documentation**. 2021. Disponível em: https://docs.python.org/

RAMALHO, L. **Python fluente**: programação clara, concisa e eficaz. São Paulo: Novatec, 2015.

SWEIGART, A. Beyond the basic stuff with python: best practices for writing clean code. San Francisco: No Starch Press, 2021. Disponível em: https://inventwithpython.com/beyond/

TAGLIAFERRI, L. **How To Code in Python 3**. New York: DigitalOcean, 2018. Disponível em: https://assets.digitalocean.com/books/python/how-to-code-in-python.pdf

TIOBE. **TIOBE Index for December 2021**. 2021. Disponível em: https://www.tiobe.com/tiobe-index/

WIKIPÉDIA. **Algoritmo de Euclides**. 2021. Disponível em: https://pt.wikipedia.org/wiki/Python

WIKIPÉDIA. **Python**. 2021. Disponível em: https://pt.wikipedia.org/wiki/Python

Currículo do autor

Marcos Roberto Ribeiro

O autor possui graduação em Ciência da Computação pelo Centro Universitário de Formiga (2005), mestrado em Ciência da Computação pela Universidade Federal de Uberlândia (2008) e doutorado em Ciência da Computação pela Universidade Federal de Uberlândia (2018). Atua como professor em disciplinas de cursos técnicos e superiores do Instituto Federal Minas Gerais - Campus Bambuí desde 2010. As disciplinas ministradas estão relacionadas principalmente com Programação, Banco de Dados e Desenvolvimento de Sistemas.

Currículo Lattes: http://lattes.cnpq.br/8439091552425995

Glossário de códigos QR (Quick Response)

Mídia digital
Apresentação do curso

Mídia digital
Revisão da primeira semana

Mídia digital
Revisão da segunda semana

Mídia digital
Revisão da terceira semana

Mídia digital
Revisão da quarta semana

Feito por (professor-autor)	Data	Revisão de *layout*	Data	Versão
Marcos Roberto Ribeiro	20/01/2022	Desginado pela proex	11/01/2022	1.0

Plataforma +IFMG

Formação Inicial e Continuada EaD

A Pró-Reitoria de Extensão (Proex), desde o ano de 2020, concentrou seus esforços na criação do Programa +IFMG. Esta iniciativa consiste em uma plataforma de cursos *online*, cujo objetivo, além de multiplicar o conhecimento institucional em Educação à Distância (EaD), é aumentar a abrangência social do IFMG, incentivando a qualificação profissional. Assim, o programa contribui para o IFMG cumprir seu papel na oferta de uma educação pública, de qualidade e cada vez mais acessível.

Para essa realização, a Proex constituiu uma equipe multidisciplinar, contando com especialistas em educação, *web design*, *design* instrucional, programação, revisão de texto, locução, produção e edição de vídeos e muito mais. Além disso, contamos com o apoio sinérgico de diversos setores institucionais e também com a imprescindível contribuição de muitos servidores (professores e técnico-administrativos) que trabalharam como autores dos materiais didáticos, compartilhando conhecimento em suas áreas de atuação.

A fim de assegurar a mais alta qualidade na produção destes cursos, a Proex adquiriu estúdios de EaD, equipados com câmeras de vídeo, microfones, sistemas de iluminação e isolação acústica, para todos os 18 *campi* do IFMG.

Somando à nossa plataforma de cursos *online*, o Programa +IFMG disponibilizará também, para toda a comunidade, uma Rádio *Web* Educativa, um aplicativo móvel para Android e iOS, um canal no Youtube com a finalidade de promover a divulgação cultural e científica e cursos preparatórios para nosso processo seletivo, bem como para o Enem, considerando os saberes contemplados por todos os nossos cursos.

Parafraseando Freire, acreditamos que a educação muda as pessoas e estas, por sua vez, transformam o mundo. Foi assim que o +IFMG foi criado.

O +IFMG significa um IFMG cada vez mais perto de você!

Professor Carlos Bernardes Rosa Jr.
Pró-Reitor de Extensão do IFMG

Características deste livro:
Formato: A4
Tipologia: Arial e Capriola.
E-book:
1ª. Edição
Formato digital