Python Avançado

Marcos Roberto Ribeiro

Formação Inicial e Continuada

+ IFMG
Campus Bambuí

Marcos Roberto Ribeiro

Python Avançado

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

189p.: il. color.

E-book, no formato PDF.
Material didático para Formação Inicial e Continuada.
ISBN 978-65-5876-044-3

1. Linguagem *Open-Source*. 2. Controle de fluxo. 3. Modularização. 4. Algoritmos. I. Título.

CDD 005.13

Catalogação: Rejane Valéria Santos - CRB-6/2907

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 conceitos avançados 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 básico da linguagem de programação Python; - Desenvolver códigos com entrada e saída de dados, estruturas de decisão e repetição; - Entender e desenvolver códigos com tratamento de exceções, modularização e coleções de dados.
SEMANA 2	- Conhecer os conceitos relacionados a programação orientada a objetos (POO); - Desenvolver e entender códigos com POO;
SEMANA 3	- Conhecer o básico da manipulação de arquivos; - Conhecer as diferenças entre arquivos de texto simples, arquivos CSV e arquivos JSON; - Desenvolver códigos com persistência de dados em arquivos.
SEMANA 4	- Conhecer os conceitos básicos de interface gráfica; - Entender o funcionamento de eventos relacionados com componentes; - Desenvolver códigos com interfaces gráficas.

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

Semana 1 - A Linguagem Python.....15
1.1 Introdução.....15
1.2 Entrada e saída de dados.....18
1.3 Operadores e expressões.....18
1.3.1 Operadores aritméticos.....19
1.3.2 Operadores relacionais.....20
1.3.3 Operadores lógicos.....21
1.4 Utilizando funções.....22
1.4.1 Manipulação de dados textuais.....22
1.4.2 Funções matemáticas.....23
1.5 Estruturas de decisão.....24
1.5.1 Estrutura de decisão simples.....24
1.5.2 Estrutura de decisão composta.....25
1.5.3 Estruturas de decisão aninhadas.....25
1.5.4 Resolvendo problemas com estruturas de decisão.....26
1.6 Estruturas de repetição.....27
1.6.1 Laço de repetição while.....27
1.6.2 Laço de repetição for.....28
1.6.3 Interrupção e continuação de laços de repetição.....29
1.6.4 Resolvendo problemas com estruturas de repetição.....30
1.7 Tratamento de exceções.....31
1.8 Modularização.....33
1.8.1 Funções.....34
1.8.2 Passagem de parâmetros e retorno de resultado.....35
1.8.3 Nomes de parâmetros e parâmetros opcionais.....35
1.8.4 Recursão.....36
1.8.5 Módulos e bibliotecas.....37
1.9 Decomposição de problemas.....38
1.10 Coleções.....40

1.10.1 Listas.....41
1.10.2 Tuplas.....44
1.10.3 Conjuntos.....46
1.10.4 Dicionários.....47
1.11 Exercícios.....48
1.12 Respostas dos exercícios.....49
1.13 Revisão.....52
Semana 2 - Programação orientada a objetos.....55
2.1 Introdução.....55
2.2 Classes e objetos.....55
2.3 Encapsulamento.....56
2.4 Atributos e métodos de classe.....58
2.5 Agregação.....59
2.6 Herança.....60
2.7 Polimorfismo.....63
2.8 Resolvendo problemas com POO.....63
2.8.1 Jogo da velha.....63
2.8.2 Simulador de caixa de supermercado.....69
2.9 Exercícios.....79
2.10 Respostas dos exercícios.....82
2.11 Revisão.....87
Semana 3 - Arquivos.....89
3.1 Introdução.....89
3.2 Arquivos de texto simples.....89
3.3 Jogo da forca com arquivo.....91
3.4 Arquivos CSV.....96
3.5 Arquivos JSON.....102
3.6 Exercícios.....109
3.7 Respostas dos exercícios.....112
3.8 Revisão.....119
Semana 4 - Interfaces gráficas.....121

4.1 Introdução.....121
4.2 Criação manual de interfaces.....121
4.3 Qt Designer.....122
4.4 Usando interfaces do Qt Designer em códigos Python.....124
4.5 Jogo da vela com interface gráfica.....128
4.6 Controle de gastos de veículos.....133
4.7 Exercícios.....143
4.8 Respostas dos exercícios.....146
4.9 Revisão.....156
Finalizando o curso.....159
Atividade final.....159
Referências.....161
Currículo do autor.....162

Objetivos
- Conhecer os básico da linguagem de programação Python;
- Desenvolver códigos com entrada e saída de dados, estruturas de decisão e repetição;
- Entender e desenvolver códigos com tratamento de exceções, modularização e coleções de dados.

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).

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 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 https://www.python.org/

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

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.

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 1 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 1 – Tipos básicos de dados em Python
Fonte: Elaborado pelo Autor.

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 2 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 2 – 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.

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 3 – Código com comentários
Fonte: Elaborado pelo Autor.

A Figura 3 mostra um exemplo de código com comentários em Python. Após cada atribuição, temos comentários comuns descrevendo o tipo de dado. Já na primeira linha,

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.

1.2 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 4 exibe um código usando as funções **input()** e **print()**. A instrução **int()** é usada 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 4 – 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.

1.3 Operadores e expressões

Assim como a maioria das linguagens de programação, a linguagem Python possui operadores aritméticos, relacionais e lógicos (PSF, 2021). Com esses operadores é

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

possível criar expressões que também podem ser combinadas, principalmente, para a realização de testes.

1.3.1 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 5 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 6 mostra um exemplo simples com expressões matemáticas usando os operadores aritméticos.

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

Figura 5 – 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 6 – 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 7 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 7 – 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 8 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 8 – 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. 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**.

1.3.2 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 9 apresenta os operadores relacionais disponíveis na linguagem Python.

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

Figura 9 – Operadores relacionais
Fonte: Elaborado pelo Autor.

O código da Figura 10 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 10 – Código comparador de valores
Fonte: Elaborado pelo Autor.

1.3.3 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 11 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 11 – Operador lógico **not**
Fonte: Elaborado pelo Autor.

A Figura 12 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 12 – Operador lógico **and**
Fonte: Elaborado pelo Autor.

A Figura 13 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 13 – Operador lógico **or**
Fonte: Elaborado pelo Autor.

O código da Figura 14 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 14 – Código com expressões lógicas e relacionais
Fonte: Elaborado pelo Autor.

1.4 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 15 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 15 – Algumas funções da biblioteca padrão do Python
Fonte: Elaborado pelo Autor.

1.4.1 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 17 contém alguns exemplos de funções de manipulação de texto. O código da Figura 20 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 16 – 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 17 – Código com manipulação de dados textuais
Fonte: Elaborado pelo Autor.

1.4.2 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 18 apresenta algumas das funções disponíveis na biblioteca **math**. Já o código da Figura 19 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 e^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 18 – 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 19 – Código com funções matemáticas
Fonte: Elaborado pelo Autor.

1.5 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).

1.5.1 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 20 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 20 – 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.

1.5.2 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 21 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 21 – Código com estrutura de decisão composta
Fonte: Elaborado pelo Autor.

1.5.3 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 22 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 22 – Código com estruturas de decisão aninhadas
Fonte: Elaborado pelo Autor.

Observe que, no código da Figura 22, 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 23. Nesse caso, incluímos um **if** imediatamente após o **else** para fazer o segundo teste. Também é possível juntar o **else** com o **if** formando a instrução **elif**, como é mostrado no código da Figura 24. Essa contração 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  else if n >= 40:
5      print('Reavaliação')
6  else:
7      print('Reprovado')
8  print('boas férias')
```

Figura 23 – Código com estruturas de decisão consecutivas
Fonte: Elaborado pelo Autor.

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

1.5.4 Resolvendo problemas com estruturas de decisão

Para exemplificar o uso de estruturas de decisão, 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 25 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 25 – Código para resolver equações de segundo grau
Fonte: Elaborado pelo Autor.

1.6 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.

1.6.1 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 26 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 26 – 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.

1.6.2 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 27 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**.

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

Figura 27 – 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**".

1.6.3 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 28 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.

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

Figura 28 – 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 29 mostra a solução para a modificação do problema. As instruções **break** e **continue** podem ser usadas tanto no laço **while** quanto no laço **for**.

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

Figura 29 – Soma indefinida de números (exceto negativos)
Fonte: Elaborado pelo Autor.

1.6.4 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. 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 30 mostra o código do algoritmo de Euclides.

```
print('Informe dois números')
n1 = int(input('N1: '))
n2 = int(input('N2: '))

if n1 < 1 or n2 < 1:
    print('Números inválidos para MDC')
else:
    while True:
        resto = n1 % n2
        print (n1, '/' , n2, '-> resto:', resto)
        if resto == 0:
            break
        n1 = n2
        n2 = resto
    print('O MDC é ', n2)
```

Figura 30 – Soma indefinida de números (exceto negativos)
Fonte: Elaborado pelo Autor.

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

1.7 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 31 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**.

```
try:
    print('Informe dois números')
    n1 = float(input('n1: '))
    n2 = float(input('n2: '))
    r = n1 / n2
    print(n1, '/', n2, '=', r)
except:
    print('Ocorreu um erro.')
```

Figura 31 – Tratamento simples de exceção
Fonte: Elaborado pelo Autor.

No código anterior, se ocorrer um erro, a execução será finalizada. 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 32. 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 32 – 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 instrução **r = n1 / n2**. A instrução **except** 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 33.

```
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 33 – 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 34 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 34 – 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.

1.8 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.

1.8.1 Funções

Funções são trechos de código que executam determinada tarefa ao serem chamados e depois retornam o controle para o ponto em que foram chamados (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 35 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 última linha do código, acontece chamada à função. Essa chamada direciona o fluxo de execução do código para a primeira instrução da função. 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 35, por exemplo, a primeira instrução a ser executada é **saudacao()**.

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

Figura 35 – Função **saudacao()**
Fonte: Elaborado pelo Autor.

No código da Figura 35 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.

1.8.2 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 36 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 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 36 – 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 37 – Função **cubo()** com retorno
Fonte: Elaborado pelo Autor.

A Figura 37 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))**.

1.8.3 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 38 – Código com Nomes de parâmetros e parâmetros opcionais
Fonte: Elaborado pelo Autor.

A Figura 38 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 instrução **jur = juros(cap, tax)**, 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 instrução **jur = juros(taxa=tax, capital=cap, tempo=tem)**, 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.

1.8.4 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 39 – Função recursiva **fat()**
Fonte: Elaborado pelo Autor.

O código da Figura 39 mostra apresenta a função recursiva **fat()**. O **if** faz o teste da condição de parada. Quando o parâmetro de entrada for menor ou igual a um, seu fatorial será um. Na última linha da função, ocorre a chamada recursiva, usando a equivalência **n! = n * (n-1)!**.

1.8.5 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 40 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 40 – Módulo **mat.py**
Fonte: Elaborado pelo Autor.

Na Figura 41 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 41 – 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 42 mostra um script executável juntando os dois módulos do exemplo anterior.

```
#!/usr/bin/env python3

# -*- coding: utf-8 -*-

def cubo(num):
    return num * num * num

n = int(input('Informe um número: '))
print(n, 'ao cubo é', cubo(n))
```

Figura 42 – Exemplo de script executável
Fonte: Elaborado pelo Autor.

1.9 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()**.

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

Figura 43 – Função **calcula()** da calculadora de expressões
Fonte: Elaborado pelo Autor.

A Figura 43 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 44 – 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 44. A instrução **global HIST** 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). O **if** faz 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 45 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 45 – Função **principal()** da calculadora de expressões
Fonte: Elaborado pelo Autor.

Temos um laço de repetição para que o usuário possa digitar quantas expressões desejar. A expressão **expr.lower() == 's'** testa se o usuário informou o comando **s** e interrompe o laço de repetição. No caso do comando **h**, o teste é feito com a expressão **expr.lower() == 'h'** e apenas mostramos 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. O resultado da expressão é guardado na variável **res**. Em seguida, chamamos a função de atualizar o histórico. Por fim, 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 46 exibe o código completo da calculadora de expressões. Os comentários das duas primeiras linhas permitem que o código seja executado na forma de script. A variável global **HIST** é inicializada com um texto vazio. A instrução **if __name__ == '__main__'** 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__'**.

```
1  #!/usr/bin/python3
2  # -*- coding: utf-8 -*-
3
4  HIST = ''
5
6  def calcula(expr):
7      try:
8          return eval(expr)
9      except:
10         print('Expressão inválida!')
11         return None
12
13 def historico(expr, res):
14     global HIST
15     if res is not None:
16         HIST += '\n\n'+ expr
17         HIST += '\n' + str(res)
18
19 def principal():
20     while True:
21         print('Informe a expressão matemática')
22         print('(h para histórico, s para sair)')
23         expr = input()
24         if expr.lower() == 's':
25             break
26         if expr.lower() == 'h':
27             print(HIST, '\n')
28         else:
29             res = calcula(expr)
30             historico(expr, res)
31             print(res, '\n')
32
33 if __name__ == '__main__':
34     principal()
```

Figura 46 – Código completo da calculadora de expressões
Fonte: Elaborado pelo Autor.

1.10 Coleções

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 47. 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 47 – 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.

1.10.1 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 48 mostra uma representação de lista de números com suas respectivas posições. O elemento **10**, por exemplo, está na posição **4**.

40	35	61	89	10	52
0	1	2	3	4	5

Figura 48 – 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 49 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 49 – Algumas funções de listas
Fonte: Elaborado pelo Autor.

O código da Figura 50 mostra uma possível solução para o problema dos produtos com preço acima da média utilizando listas. Criamos uma lista vazia para guardar os preços dos produtos (**lista_precos**). Dentro do primeiro laço de repetição, 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 50 – 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 51. 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 52.

```
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 51 – 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 52 – Solução mostrando os produtos com preços acima da média usando **enumerate()**
Fonte: Elaborado pelo Autor.

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 53 mostra alguns exemplos de inicialização de listas. A função **list()** converte o intervalo gerado pelo **range(10)** para o formato de lista.

```
# Lista inicializada com números de 0 a 9
lista = list(range(10))
print(lista)

# Lista de 10 posições com valores 0
lista = [0]*10
print(lista)
```

Figura 53 – 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 54 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 54 – 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 55 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 55 – Seleção de sub-listas de uma lista **l = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]**
Fonte: Elaborado pelo Autor.

1.10.2 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 56 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 no **if**. 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 56 – 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 57.

0	1	2
ano	mes	dia

Figura 57 – 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 58 demonstra como podemos implementar o código. Dentro do laço **for**, pegamos o nome e o preço do produto, respectivamente. Em seguida, juntamos os dados em uma tupla e adicionamos essa tupla à lista de produtos. No segundo laço de repetição percorremos os produtos da lista e extraímos seu nome e preço. O **if** é usado para verificar se o preço do produto está acima da média.

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

Figura 58 – Produtos acima da média usando tuplas
Fonte: Elaborado pelo Autor.

1.10.3 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 um alista, 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 59 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 59 – 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.

1.10.4 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 sofre 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 60 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 60 – 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 61 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 61 – Algumas funções de dicionários
Fonte: Elaborado pelo Autor.

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

1.11 Exercícios

Escreva os códigos em Python modularizados e com tratamento de exceções para resolver os problemas a seguir:

A) Desenvolver um algoritmo que construa uma lista de tuplas com nome e salário de funcionários. Em seguida, o código deve recalcular os salários considerando os seguintes aumentos:

- 20% para salários de até R$2.000,00;
- 15% para salários entre R$2.000,00 e R$5.000,00;
- 5% para salários maiores que R$5.000,00;

Por fim, o código deve exibir o total de aumento (total dos salários novos menos o total de salários antigos e os nomes dos funcionários com salários menores que R$2.000,00.

B) Construir um simulador de urna eletrônica. Inicialmente, o simulador deve permitir o cadastro de candidatos. O usuário pode cadastrar quantos candidatos desejar. O cadastro de um candidato envolve um número e seu nome. O número deve ser armazenado em formato textual, mas deve possuir exatamente dois dígitos numéricos. A função **isdigit()** do tipo textual pode ser usada para verificar se o texto é um número válido. Por fim, os candidatos cadastrados devem ser mantidos em um dicionário (número: nome).

Após o cadastro de candidatos, o simulador deve iniciar a votação. O simulador deve permitir uma quantidade indeterminada de votos. Para votar, o usuário deve informar o número do candidato. O sistema deve mostrar o nome do candidato para o usuário confirmar. Números inválidos devem ser computados como votos nulos. Já o texto vazio deve ser contabilizado como voto em branco. A sumarização dos votos deve ser feita usando um dicionário. Ao término da votação, o simulador deve mostrar o total e a porcentagem de votos de cada candidato, nulos e brancos.

1.12 Respostas dos exercícios

Escreva os códigos em Python modularizados e com tratamento de exceções para resolver os problemas a seguir:

A) Desenvolver um algoritmo que construa uma lista de tuplas com nome e salário de funcionários. Em seguida, o código deve recalcular os salários considerando os seguintes aumentos:

- 20% para salários de até R$2.000,00;
- 15% para salários entre R$2.000,00 e R$5.000,00;
- 5% para salários maiores que R$5.000,00;

Por fim, o código deve exibir o total de aumento (total dos salários novos menos o total de salários antigos e os nomes dos funcionários com salários menores que R$2.000,00.

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

def le_salario():
    while True:
        try:
            return float(input('Salário: '))
        except:
            print('Valor inválido! Informe novamente')


def le_funcionario():
    nome = input('Nome: ')
    salario = le_salario()
    return (nome, salario)

def aumenta_salarios(lista_funcionarios):
    total = 0
    for cont, funcionario in enumerate(lista_funcionarios):
        nome, salario = funcionario
        if salario <= 2000:
            salario *= 1.2
        elif salario <= 5000:
            salario *=1.15
        else:
            salario *=1.05
        total += salario
        lista_funcionarios[cont] = (nome, salario)
    return total

def principal():
    lista_funcionarios = []
    total_antigo = 0
    while True:
        funcionario = le_funcionario()
        lista_funcionarios.append(funcionario)
```

```
37          total_antigo += funcionario[1]
38          resp = input('Continuar (S/N): ').lower().strip()
39          if resp != 's':
40              break
41      total_novo = aumenta_salarios(lista_funcionarios)
42      total_aumento = total_novo - total_antigo
43      print('Total de aumento:', total_aumento)
44      print('Funcionários com salário abaixo de R$2.000,00:')
45      for nome, salario in lista_funcionarios:
46          if salario < 2000:
47              print(nome)
48
49  if __name__ == '__main__':
50      principal()
```

B) Construir um simulador de urna eletrônica. Inicialmente, o simulador deve permitir o cadastro de candidatos. O usuário pode cadastrar quantos candidatos desejar. O cadastro de um candidato envolve um número e seu nome. O número deve ser armazenado em formato textual, mas deve possuir exatamente dois dígitos numéricos. A função **isdigit()** do tipo textual pode ser usada para verificar se o texto é um número válido. Por fim, os candidatos cadastrados devem ser mantidos em um dicionário (número: nome).

Após o cadastro de candidatos, o simulador deve iniciar a votação. O simulador deve permitir uma quantidade indeterminada de votos. Para votar, o usuário deve informar o número do candidato. O sistema deve mostrar o nome do candidato para o usuário confirmar. Números inválidos devem ser computados como votos nulos. Já o texto vazio deve ser contabilizado como voto em branco. A sumarização dos votos deve ser feita usando um dicionário. Ao término da votação, o simulador deve mostrar o total e a porcentagem de votos de cada candidato, nulos e brancos.

```
1   #!/usr/bin/python3
2   # -*- coding: utf-8 -*-
3
4   def lista_cand(dict_cand):
5       print('\n'*100)
6       print('Candidatos:')
7       for numero, nome in dict_cand.items():
8           print(numero, '-', nome)
9
10  def adiciona_cand(dict_cand):
11      lista_cand(dict_cand)
12      print('\nInforme os dados do novo candidato')
13      num = input('Número: ')
14      nome = input('Nome: ')
15      if len(num) != 2 or not num.isdigit() or num in dict_cand:
16          print('Cadastro inválido!')
17      else:
18        dict_cand[num] = nome
19
20  def pega_voto(dict_cand, dict_votos):
21      while True:
22          lista_cand(dict_cand)
23          print('\nInforme seu voto')
```

```
24          voto = input('Número do candidato: ').strip()
25          if voto == '':
26              voto = 'Branco'
27          elif voto in dict_cand:
28              voto = voto + ' - ' + dict_cand[voto]
29          else:
30              voto = 'Nulo'
31          print('Voto:', voto)
32          resp = input('Confirma (S/N): ').lower().strip()
33          if resp == 's':
34              soma_voto(dict_votos, voto)
35              break
36
37  def soma_voto(dict_voto, voto):
38      if voto in dict_voto:
39          dict_voto[voto] += 1
40      else:
41          dict_voto[voto] = 1
42
43  def principal():
44      dict_cand = {}
45      while True:
46          adiciona_cand(dict_cand)
47          resp = input('Continuar cadastros (S/N): ').lower().strip()
48          if resp == 'n':
49              break
50      dict_voto = {}
51      total = 0
52      while True:
53          pega_voto(dict_cand, dict_voto)
54          total += 1
55          resp = input('Encerrar votação (S/N): ').lower().strip()
56          if resp == 's':
57              break
58      print('Resultado da eleição:')
59      for voto in dict_voto:
60          porcent = round(dict_voto[voto] / total * 100, 2)
61          print(voto, ': ', dict_voto[voto],
62                ' (', porcent, '%', ')', sep='')
63
64  if __name__ == '__main__':
65      principal()
```

1.13 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
- Conhecer os conceitos relacionados a programação orientada a objetos (POO);
- Desenvolver e entender códigos com POO;

2.1 Introdução

Como vimos no capítulo anterior, a modularização facilita consideravelmente desenvolvimento e manutenção de códigos. Entretanto, podemos ter mais vantagens ainda com a utilização da programação orientada a objetos (POO), especialmente na facilidade de reutilização de códigos. Na POO, os programas são compostos por objetos que, por sua vez, possuem atributos e métodos. Os atributos são como variáveis inerentes ao objetos para representar suas características. Já os métodos são funções específicas dos objetos (CEDER, 2018).

Na linguagem Python, todas as variáveis são objetos. Como exemplo, se tivermos uma variável **nome** do tipo **str**, a instrução **nome.lower()** chama o método **lower()** do objeto **nome**. Como sabemos, esse método é uma função do objeto **nome** que retorna um **str** com todas as letras minúsculas. De agora em diante, vamos chamar de funções somente as funções isoladas e de métodos as funções associadas a objetos.

Uma das vantagens da POO é a possibilidade de modelar entidades do mundo real como objetos para resolver problemas. Como exemplo, considere o desenvolvimento de um programa para um banco. Possíveis objetos são clientes, contas, agências etc. Um objeto conta pode ter atributos como número e saldo e métodos como depositar, sacar e transferir.

2.2 Classes e objetos

Na POO, antes de criarmos um objeto, temos que definir uma classe. A classe é uma espécie de estrutura para a criação de objetos. Assim, os objetos de uma mesma classe possuem os mesmos atributos e métodos. Porém, cada objeto pode ter valores diferentes para seus atributos (CORRÊA, 2020). Por exemplo, podemos ter os objetos **conta1** e **conta2** da classe **Conta**. Porém, a **conta1** pode ter um número 1111 e a **conta2** pode ter o número 2222.

A definição de classes é feita com a instrução **class** seguida pelo nome da classe e dois pontos. Logo abaixo, definimos os demais componentes da classe. Por convenção, os nomes de classes devem começar com um letra maiúscula. A Figura 62 mostra como podemos criar uma classe para representar contas bancárias.

Quando criamos objetos de uma classe dizemos que esse objeto é uma instância da classe. O método **__init__()** (com dois sublinhados no início e no final) é chamado de

construtor. Devemos usar esse nome porque é uma definição da linguagem Python. O construtor é chamado automaticamente quando um objeto é criado, como **conta1 = Conta(1111)**. Na maioria das vezes, o construtor é usado para inicializar os atributos do objeto. A instrução **conta1 = Conta(1111)**, por exemplo, criamos o objeto **conta1** com número **1111**.

```
1  class Conta:
2
3      def __init__(self, numero):
4          self._numero = numero
5          self._saldo = 0.0
6
7      def depositar(self, valor):
8          self._saldo += valor
9
10     def saldo(self):
11         return self._saldo
12
13 conta1 = Conta(1111)
14 print('Conta 1:', conta1.saldo())
15 conta2 = Conta(2222)
16 conta2.depositar(500.0)
17 print('Conta 2:', conta2.saldo())
```

Figura 62 – Primeira definição da classe **Conta**
Fonte: Elaborado pelo Autor.

O parâmetro **self** nos métodos das classes representa a instância do objeto. Assim, quando a instrução **conta1.saldo()** é executada, o parâmetro **self** do método **saldo()** será o objeto **conta1**. Observe que o código mostrará na tela os saldos das contas chamando o mesmo método **saldo()**. No entanto, o parâmetro **self** do método será a **conta1** na primeira chamada e **conta2** na segunda chamada.

A chamada ao método **__init__()** ocorre de forma automática quando criamos os objetos. Dessa forma, os parâmetros incluídos nos parênteses após a classe são repassados para o método **__init__()**. Dentro do método, inicializamos os atributos **_numero** e **_saldo** do objeto. Em Python, normalmente, colocamos um sublinhado no início do nome de atributos de objetos.

2.3 Encapsulamento

Considere que precisemos sacar dinheiro de uma conta. Inicialmente, podemos pensar em alterar diretamente o atributo **_saldo**. Por exemplo, se quisermos sacar 1000 do objeto **conta1**, usamos a instrução **conta1._saldo -= 1000**. Essa instrução não verifica se a conta tem saldo suficiente, o que pode levar a saldo negativo na conta.

Mesmo sendo possível, acessar diretamente os atributos de um objeto, isso não é uma boa prática de POO. O ideal é que os atributos sejam acessados e manipulados apenas por métodos ou propriedades. Essa técnica é chamada de encapsulamento (DOWNEY, 2015). Como exemplo vamos criar o método **sacar()** para a classe **Conta** como mostrado na Figura Figura 63. Colocamos o comentário **# ...** para não ter que reescrever o código já existente. Na definição do método, consultamos o saldo da conta

antes de efetivar o saque. Isso poderia ser feito manualmente, mas a definição do método facilita o saque de qualquer objeto do tipo conta.

```
class Conta:
    # ...

    def sacar(self, valor):
        if self._saldo >= valor:
            self._saldo -= valor
            return True
        return False

conta1 = Conta(1111)
print('Conta 1:', conta1.saldo())
conta2 = Conta(2222)
conta2.depositar(500.0)
conta2.sacar(1000.00)
print('Conta 2:', conta2.saldo())
```

Figura 63 – Método **sacar()** da classe **Conta**
Fonte: Elaborado pelo Autor.

Outra funcionalidade interessante para a classe **Conta** é a transferência de valores entre contas. Tal funcionalidade pode ser implementada por meio do método **transferir()** exibido na Figura 64. Note que, apesar de transferir um valor da conta atual (**self**) para outra conta (**destino**), o referido método não viola o encasulamento. Isso porque usamos o método **depositar()** para levar o valor para a conta de destino.

```
class Conta:
    # ...

    def transferir(self, destino, valor):
        if self.sacar(valor):
            destino.depositar(valor)
            return True
        return False

conta1 = Conta(1111)
conta2 = Conta(2222)
conta2.depositar(500.0)
print('Saldo atual')
print('Conta 1:', conta1.saldo())
print('Conta 2:', conta2.saldo())
conta2.transferir(conta1, 200.00)
print('Saldo após transferência')
print('Conta 1:', conta1.saldo())
print('Conta 2:', conta2.saldo())
```

Figura 64 – Método **transferir()** da classe **Conta**
Fonte: Elaborado pelo Autor.

2.4 Atributos e métodos de classe

Os atributos inicializados no método construtor são específicos para as instâncias da classe, ou seja, cada objeto tem seus próprios atributos. Imagine agora que temos que

contabilizar a quantidade de contas criadas. Isso pode ser feito usando atributos de classe e métodos de classe.

Os atributos de classe devem ser incluídos dentro da classe, de fora dos métodos que já criados. Os métodos de classe devem possuir o decorador **@classmethod** e terem **cls** como primeiro parâmetro. Diferentemente do parâmetro **self** que representa a instância da classe o parâmetro **cls** representa a própria classe. A Figura 65 mostra como podemos fazer tais alterações na classe **Conta**.

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

class Conta:

    _quant = 0

    @classmethod
    def adiciona_conta(cls):
        cls._quant+=1

    @classmethod
    def quantidade(cls):
        return cls._quant

    def __init__(self, numero):
        self.adiciona_conta()
        self._numero = numero
        self._saldo = 0.0
    # ...

print('Contas criadas:', Conta.quantidade())
conta1 = Conta(1111)
print('Contas criadas:', Conta.quantidade())
conta2 = Conta(2222)
print('Contas criadas:', Conta.quantidade())
# ...
```

Figura 65 – Classe **Conta** com atributo de classe e método de classe
Fonte: Elaborado pelo Autor.

O atributo de classe **_quant** é um contador de instâncias inicializado com o (zero) na classe. O método de classe **adiciona_conta()** é responsável por incrementar o contador. Esse método é chamado pelo método construtor da classe. Assim, sempre que um objeto é instanciado, o contador é incrementado automaticamente.

2.5 Agregação

Além dos atributos e funcionalidades já implementados para as contas, pode ser necessário ter que lidar com mais informações. Como exemplo podemos ter que cadastrar nome e endereço do cliente da conta. Porém, esses dados não são da conta e se dos clientes vinculados às contas. Assim, podemos criar uma classe cliente para o cadastro de clientes e, depois de cadastrado, associamos uma conta a esse cliente. Nesse caso, estamos fazendo uma agregação. A agregação ocorre quando um objeto possui outro

objeto como atributo. O Código da Figura 66 mostra como podemos implementar esses detalhes.

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

class Cliente:

    def __init__(self, nome, endereco):
        self._nome = nome
        self._endereco = endereco

    def imprime(self):
        print('Cliente:', self._nome,
              '\nEndereço: ', self._endereco)


class Conta:
    # ...

    def __init__(self, numero, cliente):
        self.adiciona_conta()
        self._numero = numero
        self._cliente = cliente
        self._saldo = 0.0

    # ...
    def imprime(self):
        print('Conta:', str(self._numero),
              '\nSaldo: ', str(self._saldo))
        self._cliente.imprime()

cliente = Cliente('Ana', 'Rua Um')
conta1 = Conta(1111, cliente)
cliente = Cliente('Loja X', 'Praça central')
conta2 = Conta(2222, cliente)
conta1.depositar(500.0)
conta1.transferir(conta2, 200.00)
conta1.imprime()
print()
conta2.imprime()
```

Figura 66 – Classes **Conta** e **Cliente**
Fonte: Elaborado pelo Autor.

2.6 Herança

No código da Figura 66, podemos observar que temos dois tipos de clientes. A **conta1** é de uma pessoa física e a **conta2** é de uma pessoa jurídica (empresa). Para o banco, é importante diferenciar esses dois tipos de clientes. Um alternativa seria incluir mais um atributo na classe **Cliente** para guardar o seu tipo. A Figura 67 mostra como essa alternativa é implementada.

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

class Cliente:

    def __init__(self, nome, endereco, tipo):
        self._tipo = tipo
        self._nome = nome
        self._endereco = endereco

    def imprime(self):
        print('Cliente:', self._nome, '(' + self._tipo + ')',
              '\nEndereço: ', self._endereco)
    # ...

cliente = Cliente('Ana', 'Rua Um', 'PF')
conta1 = Conta(1111, cliente)
cliente = Cliente('Loja X', 'Praça central', 'PJ')
conta2 = Conta(2222, cliente)
conta1.depositar(500.0)
conta1.transferir(conta2, 200.00)
conta1.imprime()
print()
conta2.imprime()
```

Figura 67 – Classe **Cliente** com tipo
Fonte: Elaborado pelo Autor.

Nessa alteração podemos ver uma das vantagens da POO. Nós alteramos apenas a classe **Cliente** e a classe **Conta** continuou funcionando sem nenhuma necessidade de modificação. O problema de incluirmos apenas o tipo na classe **Cliente** é que podem surgir outros atributos específicos para cada tipo. Como exemplo, temos CPF e data de nascimento para pessoas físicas e CNPJ para pessoas jurídicas. Assim, podemos observar que todos os tipos de clientes possuem atributos em comum (nome e endereço) e alguns clientes possuem atributos específicos (atributos de pessoas físicas e de pessoas jurídicas). Uma solução interessante para essa situação é utilizar Herança.

Na POO, a herança consiste em criar classes mais específicas a partir de uma classe mais genérica (SWEIGART, 2021). No nosso exemplo, a classe mais genérica seria **Cliente** e as classes mais específicas seriam a pessoa física e a pessoa jurídica. A Figura 68 mostra como podemos implementar o conceito de herança em nosso código. Observe que na definição das classes **ClientePF** e **ClientePJ** passamos a classe **Cliente** como parâmetro. Isso faz com que as novas classes herdem todas as características da classe **Cliente**. A classe **Cliente** é chamada de classe pai ou superclasse e as classes **ClientePF** e **ClientePJ** são chamadas de classes filhas ou subclasses.

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

class Cliente:

    def __init__(self, nome, endereco):
        self._nome = nome
        self._endereco = endereco

    def imprime(self):
        print('Cliente:', self._nome, '(', type(self), ')',
              '\nEndereço: ', self._endereco)


class ClientePF(Cliente):

    def __init__(self, nome, endereco, cpf, nascimento):
        super().__init__(nome, endereco)
        self._cpf = cpf
        self._nascimento = nascimento

class ClientePJ(Cliente):

    def __init__(self, nome, endereco, cnpj):
        super().__init__(nome, endereco)
        self._cnpj = cnpj
    # ...

cliente = ClientePF('Ana', 'Rua Um', 'XXX.XXX.XXX-ZZ', '1995-05-20')
conta1 = Conta(1111, cliente)
cliente = ClientePJ('Loja X', 'Praça Central', 'XX.XXX.XXX/YYYY-ZZ')
conta2 = Conta(2222, cliente)
conta1.depositar(500.0)
conta1.transferir(conta2, 200.00)
conta1.imprime()
print()
conta2.imprime()
```

Figura 68 – Herança da classe **Cliente**
Fonte: Elaborado pelo Autor.

As classes filhas podem ter novos atributos e métodos. Além disso, elas podem sobrescrever métodos da classe pai. Em nosso exemplo, sobrescrevemos o método **__init__()** porque precisamos passar novos parâmetros para a criação das classes filhas. Além do nome e endereço, a **ClassePF** recebe o CPF e a data de nascimento na criação. Já a classe **ClientePJ** recebe, junto com o nome e endereço, o CNPJ da empresa.

Dentro do construtor das classes filhas usamos a instrução **super()** para referenciar a classe pai. Dessa maneira, a instrução **super().__init__(nome, endereco)** chama o construtor da classe pai para inicializar os atributos comuns e, depois, os demais atributos são inicializados.

Nós modificamos também o método **imprime()** da classe pai para mostrar o tipo do cliente com a instrução **super(self)**. Mesmo sendo um método da classe pai, no momento da chamada, **self** será o objeto instanciado. Assim, a instrução **super(self)** retornará

<class '__main__.ClientePF'> para a classe **ClientePF** e **<class '__main__.ClientePJ'>** para a classe **ClientePJ**. Uma maneira de exibir mensagens mais bem formatadas é mover o método **imprime()** da classe pai para as classes filhas como descrito no código da Figura 69.

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

class Cliente:

    def __init__(self, nome, endereco):
        self._nome = nome
        self._endereco = endereco

class ClientePF(Cliente):

    def __init__(self, nome, endereco, cpf, nascimento):
        super().__init__(nome, endereco)
        self._cpf = cpf
        self._nascimento = nascimento

    def imprime(self):
        print(self._nome, '\nCPF:' + self._cpf,
              '\nNascimento: ' + self._nascimento)

class ClientePJ(Cliente):

    def __init__(self, nome, endereco, cnpj):
        super().__init__(nome, endereco)
        self._cnpj = cnpj

    def imprime(self):
        print(self._nome, '\nCNPJ: ' + self._cnpj)
    # ...

cliente = ClientePF('Ana', 'Rua Um', 'XXX.XXX.XXX-ZZ', '1995-05-20')
conta1 = Conta(1111, cliente)
cliente = ClientePJ('Loja X', 'Praça Central', 'XX.XXX.XXX/YYYY-ZZ')
conta2 = Conta(2222, cliente)
conta1.depositar(500.0)
conta1.transferir(conta2, 200.00)
conta1.imprime()
print()
conta2.imprime()
```

Figura 69 – Método **imprime()** nas classes filhas
Fonte: Elaborado pelo Autor.

No código anterior, movemos o método **imprime()** da classe pai para as classes filhas. Isso foi feito intencionalmente para que tenhamos apenas clientes como pessoas físicas e pessoas jurídicas. Assim, se for criado um cliente usando a classe pai, a chamada ao método **imprime()** gerará um erro[4].

4 O mesmo efeito pode ser obtido com classes abstratas que estão fora do escopo desse livro.

2.7 Polimorfismo

Considere agora que o banco, além da conta corrente, possui uma conta investimento. Esse novo tipo de conta possui as mesmas características da conta corrente, mas os depósitos rendem juros de 1% que são incorporados no saldo da conta. Nesse caso, podemos criar uma classe filha e sobrescrever o método de depósito para implementar a conta investimento como mostrado na Figura 68.

```
# ...
class ContaInvestimento(Conta):

    def depositar(self, valor):
        self._saldo += valor * 1.01


cliente = ClientePF('Ana', 'Rua Um', 'XXX.XXX.XXX-ZZ', '1995-05-20')
conta1 = ContaInvestimento(1111, cliente)
cliente = ClientePJ('Loja X', 'Praça Central', 'XX.XXX.XXX/YYYY-ZZ')
conta2 = Conta(2222, cliente)
conta1.depositar(500.0)
conta1.transferir(conta2, 200.00)
conta1.imprime()
print()
conta2.imprime()
```

Figura 70 – Classe **ContaInvestimento**
Fonte: Elaborado pelo Autor.

Agora, o objeto **conta1** é da classe **ContaInvestimento**. Portanto, temos um método com mesmo nome na classe pai e na classe filha. No entanto, ao ser executado o Python determina dinamicamente qual método deve ser chamado. Essa capacidade de determinar se deve ser chamado o método da classe pai ou da classe filha é chamada de polimorfismo.

2.8 Resolvendo problemas com POO

Para melhorarmos nossos conhecimentos, vamos resolver alguns problemas usando a abstração da POO. Primeiro, vamos implementar um jogo da velha e, em seguida, um simulador de caixa de supermercado.

2.8.1 Jogo da velha

O jogo da velha consiste em um tabuleiro 3x3 (3 linhas por 3 colunas) com dois jogadores quem marcam alternadamente as posições do tabuleiro. Normalmente, um jogador usa a letra **X** e outro usa **O**. O jogador que conseguir marcar três posições em linha reta vence o jogo. Para começar o nosso projeto vamos criar uma classe para representar o tabuleiro do jogo. Mais especificamente, dentro dessa classe, vamos incluir uma lista de listas com espaços em branco, como mostrado na Figura 71.

```
class Tabuleiro:

    def __init__(self):
        # Inicializa todas as posições com ' '
        self._posicoes = [
                [' ', ' ', ' '],
                [' ', ' ', ' '],
                [' ', ' ', ' ']]

    def imprime(self):
        # Imprime letras das colunas
        print('\n |A|B|C')
        for cont, linha in enumerate(self._posicoes):
            # Imprime número e posições da linha
            print('--------')
            print(cont+1, '|' + '|'.join(linha), sep='')
```

Figura 71 – Classe **Tabuleiro**
Fonte: Elaborado pelo Autor.

O atributo **_posicoes** é a lista de listas para representar o tabuleiro. Essa estrutura também pode ser vista como uma matriz de três linhas por três colunas. Nesse caso, podemos acessar a posição da linha 0 e coluna 2 com a instrução **self._posicoes[0][2]**, por exemplo.

O método **imprime()** que escreve o tabuleiro na tela como apresentado na Figura 72. Primeiro, escrevemos o cabeçalho das colunas e, depois, escrevermos linha por linha. No caso das linhas, usamos o método **join()** para concatenar as colunas de cada linha usando o caractere **|** (pipe). A variável **cont** é incrementada com 1 porque, internamente, as posições das listas começam em 0 (zero).

```
  |A|B|C
--------
 1| | |
--------
 2| | |
--------
 3| | |
```

Figura 72 – Tabuleiro do jogo mostrado na tela
Fonte: Elaborado pelo Autor.

Durante o jogo, os jogadores farão as jogadas informando a linha e a coluna da posição que desejam marcar. Por exemplo, a jogada **2B** indica que o jogador quer marcar a posição da linha **2** e coluna **B**. Portanto, precisamos também de um método que faça as jogadas no tabuleiro. O código da Figura 73 demostra como isso pode ser feito.

```
class Tabuleiro:
    # ...

    def jogada(self, posicao, simbolo):
        # Tratamento de exceções para erros de digitação
        try:
            # A linha é o primeiro caractere digitado
            linha = int(posicao[0]) - 1
            # A coluna é o segunda caractere (letra)
            letra = posicao[1].upper()
            # Converte letra para número
            coluna = ord(letra) - ord('A')
            # Verifica se a posição está vazia
            if self._posicoes[linha][coluna] == ' ':
                # Marca a posição com o simbolo do jogador
                self._posicoes[linha][coluna] = simbolo
                return True
        except:
            # Em caso de erro, nenhuma posição é marcada
            pass
        return False
```

Figura 73 – Método **jogada()** da classe **Tabuleiro**
Fonte: Elaborado pelo Autor.

O método recebe com o parâmetros a posição digitada e o símbolo do jogador. O tratamento de exceção é usado para para lidar com possíveis erros de digitação. Para obter a linha pegamos o primeiro caractere da posição digitada (**posicao[0]**) convertemos para inteiro e subtraímos 1. Essa subtração é necessária porque o jogador digita as linhas de 1 a 3, mas na representação por lista as linhas vão de 0 a 2.

No caso da coluna, pegamos o segundo caractere (**posicao[1]**) passamos para maiúscula e usamos a função **ord()** para converter para um valor numérico. Dessa forma, após as manipulações descritas, verificamos se a posição informada (**self._posicoes[linha][coluna]**) está vazia. Em caso afirmativo, marcamos tal posição com o símbolo do jogador e retornamos **True**.

Por outro lado, se houver qualquer erro de digitação ou posição inválida, a execução é redirecionada para o **except**. No **except**, usamos a instrução **pass** para não realizar nenhuma ação. Isso faz com que a execução saia do tratamento de erro e o método retorna **False**.

O jogo da velha pode finalizado quando um jogador vence ou quando não há mais jogadas possíveis e o jogo termina empatado. Dessa forma, precisamos de maneiras para testar essas duas situações. Primeiro, vamos tratar o caso de empate. Para isso, podemos criar um método que verifique se ainda restam jogados possíveis no tabuleiro.

A Figura 74 apresenta o código do método **tem_jogada()** que retorna verdadeiro se ainda houverem posições vagas no tabuleiro. O método percorre linha por linha da matriz que representa o tabuleiro. Em cada linha, usamos o operador **in** para verificar se a linha tem um espaço em branco. Em caso afirmativo, o método retorna **True**. Contudo, se todas as linhas forem percorridas e o espaço vazio não for encontrado, o método retorna **False**.

```python
class Tabuleiro:
    # ...

    def tem_jogada(self):
        # Varre o tabuleiro procurando por posições vazias
        for linha in self._posicoes:
            if ' ' in linha:
                return True
        return False
```

Figura 74 – Método **tem_jogada()** da classe **Tabuleiro**
Fonte: Elaborado pelo Autor.

```python
class Tabuleiro:
    # ...

    def todas_linhas(self):
        # Retorna todas as linhas possiveis em formato de tuplas
        # Linhas, colunas, diagonal e transversal
        todas = []
        # Linhas
        for linha in self._posicoes:
            todas.append(tuple(linha))
        # Colunas
        for cont in range(3):
            coluna = [self._posicoes[0][cont],
                      self._posicoes[1][cont],
                      self._posicoes[2][cont]]
            todas.append(tuple(coluna))
        # Diagonal e transversal
        diagonal = []
        transversal = []
        for cont in range(3):
            diagonal.append(self._posicoes[cont][cont])
            transversal.append(self._posicoes[2 - cont][cont])
        todas.append(tuple(diagonal))
        todas.append(tuple(transversal))
        return todas
```

Figura 75 – Método **todas_linhas()** da classe **Tabuleiro**
Fonte: Elaborado pelo Autor.

A Figura 75 exibe o método **todas_linhas()** da classe **Tabuleiro**. Esse método retorna todas as linhas retas possíveis do tabuleiro, incluindo linhas, colunas, diagonal e transversal. Posteriormente, podemos usar esse método para testar se um jogador é o vencedor. Isso acontece se uma das linhas retornadas possuir todas as posições iguais ao símbolo do jogador.

O método utiliza a lista **todas** para guardar todas as linhas retas em formato de tupla. Primeiro, temos o laço de repetição que pega as linhas horizontais. Depois, outro laço de repetição para pegar as colunas. As colunas não podem ser pegas diretamente como as linhas. Então, usamos a instrução **self._posicoes[linha][cont]** variando o contador **cont** de 0 a 2 para montar as linhas retas referente às colunas. Por fim, o último laço de repetição monta as linhas da diagonal e da transversal.

Agora, depois de definirmos a classe **Tabuleiro**, podemos criar a classe **Velha** que usa o tabuleiro para implementar o jogo. A XXX mostra essa classe com os métodos construtor e **imprime()**. O construtor **__init__()** cria o tabuleiro do jogo e sorteia o jogador que vai começar. O sorteio é feito usando a função **random()** da biblioteca **random**. Essa função retorna um número aleatório maior ou igual a zero e menor que um. Se o número for maior ou igual a 0.5 o jogador **X** começa. Caso contrário, o jogador **O** inicia o jogo.

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

from random import random


class Tabuleiro:
    # ...


class Velha:

    def __init__(self):
        # Inicializa tabuleiro
        self._tabuleiro = Tabuleiro()
        # Sorteia jogador
        if random() >= 0.5:
            self._jogador = 'X'
        else:
            self._jogador = 'O'

    def imprime(self):
        # Mostra o estado do jogo
        print('\n'*50)
        print('Jogo da velha\n')
        self._tabuleiro.imprime()
```

Figura 76 – Classe **Velha** com os métodos construtor e **imprime()**
Fonte: Elaborado pelo Autor.

Durante o jogo, precisaremos alternar entre os jogadores **X** e **O** e pegar suas jogadas. Isso será feito como método **troca_jogador()** mostrado na Figura 77.

```
# ...
class Velha:
    # ...

    def troca_jogador(self):
        # Troca o jogador
        if self._jogador == 'X':
            self._jogador = 'O'
        else:
            self._jogador = 'X'
```

Figura 77 – Método **troca_jogador()** da classe **Velha**
Fonte: Elaborado pelo Autor.

```
# ...
class Velha:
    # ...

    def pega_jogada(self):
        # Laço infinito para o caso de jogadas inválidas
        while True:
            # Mostra o tabuleiro
            self.imprime()
            # Mostra o jogador que está jogando
            print('\nJogador', self._jogador)
            # Pega linha e coluna da jogada
            posicao = input('Informe a jogada: ')
            # Tenta executar a jogada no tabuleiro
            if self._tabuleiro.jogada(posicao, self._jogador):
                # Se a jogada for válida, interrompe o laço
                break
```

Figura 78 – Método **pega_jogada()** da classe **Velha**
Fonte: Elaborado pelo Autor.

Outra tarefa importante da classe **Velha** será pegar a jogada de um jogador e verificar se a mesma é válida. O método **pega_jogada()** mostrado na Figura 78 é responsável por essa tarefa. O método possui um laço de repetição que é executado até que o jogador informe uma jogada válida. Dentro do laço, mostramos o estado atual do jogo com o método **imprime()**, informamos qual jogador deve jogar e pegamos sua jogada.

O método **jogada()** do objeto **_tabuleiro** é utilizado para tentar executar a jogada informada. Se for uma jogada válida, o método marca a posição no tabuleiro e retorna **True**, fazendo com que o laço de repetição seja interrompido.

```
# ...
class Velha:
    # ...

    def eh_vencedor(self, jogador):
        # Testa se um jogador é vencedor
        linhas = self._tabuleiro.todas_linhas()
        # O Jogador é vencedor ter tiver uma linha com 3 posições
        if tuple([jogador]*3) in linhas:
            return True
        return False
```

Figura 79 – Método **eh_vencedor()** da classe **Velha**
Fonte: Elaborado pelo Autor.

A cada jogada realizada, precisamos testar se algum jogador já venceu o jogo. A Figura 79 contém o método **eh_vencedor()** para testar se um jogador venceu. Primeiro, pegamos todas as linhas possíveis com o método **todas_linhas()** do objeto **_tabuleiro**. Depois, testamos se, nessas linhas, existe uma com três posições do jogador.

A instrução **tuple([jogador]*3)** é cria uma lista com três posições com o símbolo **jogador** e converte para tupla. Essa tupla representa uma linha com três posições do jogador. O **if** testa se tal tupla existe é uma das linhas do tabuleiro. Em caso afirmativo, o jogador conseguiu marcar três posições em linha reta e é o vencedor do jogo.

```
# ...
class Velha:
    # ...

    def jogar(self):
        # Repete enquanto hoverem jogadas possíveis
        while self._tabuleiro.tem_jogada():
            # Motra o estado do jogo
            self.imprime()
            # Pega a jogada
            self.pega_jogada()
            # Testa se o jogador venceu
            if self.eh_vencedor(self._jogador):
                self.imprime()
                print('\nFim de jogo!')
                print('Vitória do jogador', self._jogador)
                # Finaliza se tiver um vencedor
                return
            # Troca de jogador
            self.troca_jogador()
        # Se terminarem as jogadas, o jogo fica empatado
        self.imprime()
        print('\nJogo empatado!')


if __name__ == '__main__':
    jogo = Velha()
    jogo.jogar()
```

Figura 80 – Método **jogar()** da classe **Velha**
Fonte: Elaborado pelo Autor.

Para finalizar, precisamos de um método para conduzir a execução do jogo. A Figura 80 mostra o método **jogar()** para realizar essa tarefa. O método possui um laço de repetição que é executado enquanto houverem jogadas no tabuleiro. Dentro do laço, mostramos o estado do jogo, pegamos a jogada, testamos se o jogador atual venceu e trocamos de jogador.

No caso de um jogador vencer, mostramos o estado do jogo novamente e informamos a vitória. Quando não há mais posições vazias no tabuleiro, o laço de repetição termina e informamos que o jogo terminou empatado.

2.8.2 Simulador de caixa de supermercado

O primeiro passo para criarmos o nosso simulador de caixa de supermercado é definir uma classe para cadastrar os produtos do supermercado. Basicamente, os produtos devem ter uma descrição e um preço. A Figura 81 mostra a classe **Produto** com seu construtor.

```
class Produto:

    def __init__(self, descricao, preco):
        self._descricao = descricao
        self._preco = preco
```

Figura 81 – Classe **Produto** com construtor e método **altera()**
Fonte: Elaborado pelo Autor.

Além do método construtor, a linguagem Python possui outros métodos com nomes iniciados e terminados com dois sublinhados. Esse métodos são chamados de métodos especiais ou métodos mágicos e são usados para certas funções especiais. Um método especial interessante é o **__str__()**. Com ele é possível definir um formato textual para o objeto da classe. Assim, é possível imprimir esse formato usando a função **print()**, por exemplo.

Para a nossa classe **Produto**, vamos definir o método **__str__()** para retornar a descrição e o preço do produto em formato textual. A Figura 82 mostra o código usado para implementar o método. Como o valor retornado deve ser do tipo **str**, usamos o método **format()** para formatar corretamente os atributos e retornar no formato correto.

```
class Produto:
    # ...

    def __str__(self):
        return '{d} (${p:0.2f})'.format(
                d=self._descricao,
                p=self._preco)
```

Figura 82 – Classe **Produto** com método **__str__()**
Fonte: Elaborado pelo Autor.

O nosso simulador precisará cadastrar os produtos para, posteriormente, vendê-los. No cadastro, além do nome e preço, teremos que guardar o estoque do produto. Por outro lado, nas vendas, precisaremos da quantidade vendida e do total (preço vezes quantidade) de cada produto. Dessa forma, usaremos a classe **Produto** como base e criaremos classes filhas com as peculiaridades do cadastro no estoque e da venda. A Figura 83 mostra a classe **ProdutoEstoque** que, além da descrição e preço, possui o estoque do produto.

```
class ProdutoEstoque(Produto):

    def __init__(self, descricao, preco):
        super().__init__(descricao, preco)
        self._estoque = 0.0
```

Figura 83 – Classe **ProdutoEstoque**
Fonte: Elaborado pelo Autor.

Na classe **ProdutoEstoque**, a descrição e o preço dos produtos podem ser alterados pelo usuário, devemos ter métodos para ler e modificar os atributos **_descricao** e **_preco**. Em algumas linguagens de programação, é necessário definir os atributos como privados e criar métodos públicos para acessá-los. Em Python, podemos usar o decorador **@property** para definirmos o método de leitura e o decorador **@metodo.setter** para

definir o método de gravação. A Figura 84 mostra como criar tais métodos usando decoradores.

```
class ProdutoEstoque(Produto):
    # ...

    @property
    def preco(self):
        return self._preco

    @preco.setter
    def preco(self, preco):
        self._preco = preco

    @property
    def descricao(self):
        return self._descricao

    @descricao.setter
    def descricao(self, descricao):
        self._descricao = descricao

    def entrada(self, quantidade):
        self._estoque += quantidade

    def saida(self, quantidade):
        if quantidade <= self._estoque:
            self._estoque -= quantidade
            return True
        return False
```

Figura 84 – Classe **ProdutoEstoque** com métodos usando decoradores
Fonte: Elaborado pelo Autor.

Observe que os métodos de leitura e de gravação possuem o mesmo nome, mas têm parâmetros e decoradores diferentes. Se tivermos um objeto **p** da classe **ProdutoEstoque**, podemos usar a instrução **p.preco** como se fosse um atributo e ler ou alterar o preço do produto. Contudo, ao fazermos isso, o Python usa os métodos decorados para as operações efetivar as operações.

No caso do estoque, não é permitida a alteração direta. Então, criamos um método para dar entrada no estoque e outro para dar saída. O método da saída testa se temos estoque suficiente do produto antes de efetivar a saída. Se a saída for efetivada, o método retorna **True**. Senão, o método retorna **False**.

Para finalizar a classe **ProdutoEstoque**, vamos implementar o método **__str__()** para estender sua versão da classe pai e incluir o estoque na representação textual do produto. A Figura 85 demostra como o método é implementado.

```
class ProdutoEstoque(Produto):
    # ...

    def __str__(self):
        texto = super().__str__()
        texto += ', Estoque: {e:0.3f}'.format(e=self._estoque)
        return texto
```

Figura 85 – Método **__str__()** da classe **ProdutoEstoque**
Fonte: Elaborado pelo Autor.

Agora vamos criar a classe de produtos para as vendas. Nessa classe, além da descrição e preço do produto, teremos a quantidade vendida. A Figura 86 apresenta a implementação dessa classe. Como os dados dos produtos não podem ser alterados depois da venda, não incluímos métodos para alteração dos atributos. O método total() é usado para calcular o total do produto (preço vezes a quantidade). Não incluímos métodos de leitura dos atributos porque usaremos apenas a representação textual do produto para listar os produtos já vendidos.

```
class ProdutoVenda(Produto):

    def __init__(self, descricao, preco, quantidade):
        super().__init__(descricao, preco)
        self._quantidade = quantidade

    @property
    def total(self):
        return self._quantidade * self._preco

    def __str__(self):
        texto = super().__str__()
        texto += ', Qtde: {q:0.3f}'.format(q=self._quantidade)
        texto += ', Total: ${t:0.2f}'.format(t=self.total)
        return texto
```

Figura 86 – Classe **ProdutoVenda**
Fonte: Elaborado pelo Autor.

O nosso projeto vai precisar também de uma classe para agrupar os produtos de uma mesma venda. Para isso, vamos usar a classe **Venda** da Figura 87. Basicamente, a classe tem como atributos uma lista de produtos e o total da venda. Os produtos são adicionados à venda por meio do método **adiciona_produto()**. Quando um produto é adicionado, o total da venda é automaticamente atualizado.

O método **numero_produtos()** é usado para consultar a quantidade de produtos presentes na venda. A classe **Venda** implementa também o método **__str__()** para retornar sua representação textual. Nesse método listamos os produtos da venda e seu total geral. O método **total()** é usado para retornar o total geral da venda.

```
class Venda:

    def __init__(self):
        self._produtos = []
        self._total = 0.0

    @property
    def total(self):
        return self._total

    @property
    def numero_produtos(self):
        return len(self._produtos)

    def adiciona_produto(self, produto):
        self._produtos.append(produto)
        self._total += produto.total

    def __str__(self):
        texto = '\n' + '-'*50
        texto += '\nProdutos:'
        for produto in self._produtos:
            texto += '\n' + str(produto)
        texto += '\n' + '-'*50
        texto += 'Total da venda: ${t:0.2f}'.format(t=self._total)
        texto += '\n' + '-'*50
        return texto
```

Figura 87 – Classe **Venda**
Fonte: Elaborado pelo Autor.

O nosso simulador fará diversas iterações com o usuário e, em alguns momentos, precisará receber valores numéricos. Para evitarmos, erros em tempo de execução como a digitação de valores inválidos, vamos criar uma função para receber valores numéricos e outra para confirmar perguntas com o usuário. Essas funções são exibidas na Figura 88.

```
def pergunta(mensagem, tipo=int):
    while True:
        try:
            resp = input(mensagem)
            return tipo(resp)
        except:
            print('Valor inválido! Informe novamente.')


def confirma(mensagem, resposta):
    texto = input(mensagem).strip()
    if texto.lower() == resposta.lower():
        return True
    return False
```

Figura 88 – Função **pergunta()** com tratamento de exceção
Fonte: Elaborado pelo Autor.

A função **pergunta()** recebe uma mensagem e o tipo de dado (**int** por padrão) que deve ser retornado. Usando o tratamento de exceção, a função exibe a mensagem para o

usuário, pega sua resposta e tenta retorná-la convertendo para o tipo desejado. Se ocorrer algum erro, a instrução **except** é disparada, informando uma mensagem de valor inválido. O laço de repetição garante que o processo se repete até que o usuário informa um valor válido.

A função **confirma()** exibe uma mensagem para o usuário e verifica se o texto informado é igual a resposta esperada. Em caso afirmativo, a função retorna **True**. Senão, ela retorna **False**. Essa função poderá ser usada para confirmar preguntas do tipo sim ou não com o usuário.

Finalmente, podemos agora implementar a classe Caixa para simular o caixa de supermercado. A Figura 89 mostra a referida classe com o método construtor é o método **menu()**. O método construtor inicializa o dicionário **_produtos** para manter o cadastro de produtos e a lista **_vendas** para armazenar as vendas realizadas. O dicionário **_produtos** será usado para guardar objetos da classe **ProdutoEstoque** e as chaves serão códigos numéricos. O método **menu()** apresenta as possíveis operações que o caixa pode realizar.

```
# ...
class Caixa:

    def __init__(self):
        self._produtos = {}
        self._vendas = []

    @classmethod
    def menu(cls):
        print()
        print('**********************************')
        print('*           CAIXA                *')
        print('**********************************')
        print('(C) Cadastrar/atualizar produto ')
        print('(E) Entrada de estoque          ')
        print('(V) Vender                      ')
        print('(R) Relatório de vendas         ')
        print('(S) Sair                        ')
        print('**********************************')
        escolha = input('Informe sua opção: ').upper()
        return escolha
```

Figura 89 – Classe **Caixa** com método construtor e método **menu()**
Fonte: Elaborado pelo Autor.

A primeira operação que deve ser feita no caixa é o cadastro de produtos. Para isso precisamos de métodos para buscar produtos já cadastrados, incluir novos produtos e atualizar dados de produtos existentes. O código da Figura 90 mostra os métodos usados para implementas essas funções.

O método **busca_produtos()** é usado buscar produtos a partir de seus códigos numéricos. Se o dicionário de produtos estiver vazio, o método retorna **None**. Caso contrário, a função lista os produtos já cadastrados e pergunta o código ao usuário. Se o código informado for encontrado no dicionário, a função retorna o produto associado ao código. Se o produto não for encontrado a função retorna **None**.

O método **dados_produto()** serve para pegar a descrição e preço do produto com o usuário. Os dados retornados pela função são usados para cadastrar novos produtos ou alterar produtos existentes.

```
# ...
class Caixa:
    # ...

    def busca_produto(self):
        if len(self._produtos) == 0:
            print('Nenhum produto cadastrado!')
            return None
        print('\nProdutos:')
        for cod, produto in self._produtos.items():
            print(cod, ':', produto)
        codigo = pergunta('Código do produto: ')
        if codigo in self._produtos:
            return self._produtos[codigo]
        print('Produto não encontrado!')
        return None

    @classmethod
    def dados_produto(cls):
        print('\nInforme os dados')
        descricao = input('Descrição: ')
        preco = pergunta('Preço: ', float)
        return descricao, preco

    def cadastro_produto(self):
        produto = self.busca_produto()
        if produto is not None:
            print('Produto cadastrado:', produto)
            if confirma('Alterar? (S/N) ', 'S'):
                descricao, preco = self.dados_produto()
                produto.descricao = descricao
                produto.preco = preco
        else:
            if confirma('Incluir? (S/N) ', 'S'):
                descricao, preco = self.dados_produto()
                produto = ProdutoEstoque(descricao, preco)
                codigo = len(self._produtos)
                self._produtos[codigo] = produto
```

Figura 90 – Métodos **busca_produtos()**, **dados_produtos()** e **cadastro_produtos()** da classe **Caixa**
Fonte: Elaborado pelo Autor.

O método **cadastro_produto()**, inicialmente, chama a função **busca_produto()** e armazena o resultado na variável **produto**. Se a variável for diferente de **None**, significa que um produto foi retornado. O método mostra o produto e pergunta se o usuário deseja fazer alterações. Se o usuário decidir por alterar, os dados retornados por **dados_produtos()** são usados para atualizar a descrição e preço do produto.

No caso da variável produto ser **None**, ou o produto não foi encontrado ou o dicionário de produtos está vazio. Assim, a função pergunta se o usuário deseja incluir um novo produto. Em caso afirmativo, o método pega os dados do novo produto, cria o objeto

da classe **ProdutoEstoque** e armazena no dicionário. O código usado é a quantidade atual de produtos no dicionário obtida com a função **len()**.

Conforme foi explicado sobre a classe **ProdutoEstoque**, o estoque de um produto não pode ser alterado diretamente. Portanto, após o cadastro de um produto. Usuário deve fazer a entrada de estoque. A Figura 91 mostra o código do método **entrada_estoque()** responsável pela realização dessa tarefa.

```
# ...
class Caixa:
    # ...

    def entrada_estoque(self):
        produto = self.busca_produto()
        if produto is not None:
            quantidade = pergunta('Quantidade de entrada: ', float)
            produto.entrada(quantidade)
```

Figura 91 – Método **entrada_estoque()** da classe **Caixa**
Fonte: Elaborado pelo Autor.

Após o cadastro e entrada de estoque, é possível realizar vendas com o método **venda()** apresentado na Figura 92. Inicialmente, criamos um objeto **venda** da classe **Venda**. Depois, criamos um laço de repetição para pegar os produtos da venda.

Dentro do laço, usamos o resultado do método **busca_produto()** para adicionar o produto à venda. Produtos inválidos (**None**) que sejam retornados não são incluídos. Além disso, usamos o método **saida()** do objeto **produto** para garantir que temos estoque suficiente para a quantidade vendida. Quando o usuário parar de incluir mais produtos, o laço de repetição é interrompido. Com o fim do laço de repetição, testamos se o número de produtos é maior que zero e adicionamos a venda à lista de vendas.

O código da Figura 92 contém também o método **relatorio_vendas()** para mostrar todas as vendas realizadas na tela. Se não houver nenhuma venda, o método apenas exibe um aviso de nenhuma venda encontrada. No caso de existir pelo menos uma venda, o método percorre a lista de vendas e exibe cada uma delas. Enquanto percorre a lista, o método atualiza o total geral de vendas para ser exibido no final do relatório.

```
# ...
class Caixa:
    # ...

    def venda(self):
        print('Venda')
        venda = Venda()
        while True:
            produto = self.busca_produto()
            if produto is not None:
                quantidade = pergunta('Quantidade vendida: ', float)
                if produto.saida(quantidade):
                    produto_venda = ProdutoVenda(produto.descricao,
                                                 produto.preco,
                                                 quantidade)
                    venda.adiciona_produto(produto_venda)
            print(venda)
            if confirma('Adicionar mais produtos? (S/N) ', 'N'):
                break
        if venda.numero_produtos > 0:
            self._vendas.append(venda)

    def relatorio_vendas(self):
        if len(self._vendas) == 0:
            print('Nenhuma venda encontrada!')
            return
        total_geral = 0
        for cont, venda in enumerate(self._vendas):
            print('\nVenda', cont)
            print(venda)
            total_geral += venda.total
        print('TOTAL GERAL:', total_geral)
        input('Pressine ENTER para voltar')
```

Figura 92 – Métodos **venda()** e **relatorio_vendas()** da classe **Caixa**
Fonte: Elaborado pelo Autor.

Por fim, criamos o método **iniciar()** mostrado na Figura 93 para executar o simulador de caixa. O método possui um laço de repetição que é interrompido somente quando o usuário desejar sair. Dentro do laço, exibimos o menu de opções e chamamos o método correspondente à opção selecionada.

```
# ...
class Caixa:
    # ...

    def iniciar(self):
        while True:
            escolha = self.menu()
            if escolha == 'C':
                self.cadastro_produto()
            elif escolha == 'E':
                self.entrada_estoque()
            elif escolha == 'V':
                self.venda()
            elif escolha == 'R':
                self.relatorio_vendas()
            elif escolha == 'S':
                break


if __name__ == '__main__':
    Caixa().iniciar()
```

Figura 93 – Métodos **iniciar()** da classe **Caixa**
Fonte: Elaborado pelo Autor.

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

2.9 Exercícios

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

A) Jogo de bingo:

- Crie uma classe **Jogador** que deve guardar o nome do jogador, os números do jogador e os números marcados durante o jogo. Utilize conjuntos para guardar os números. O nome do jogador poderá ser consultado por um propriedade;

- A classe **Jogador** deve ter o método **marca()** para marcar cada número que for sorteado e o método **faltantes()** para retornar o conjunto de números faltantes. Implemente também o método **imprime()** para mostrar o nome do jogador e seus números ainda não marcados;

- Crie a classe **Bingo()** que deve ser inicializada com a quantidade de números do jogo. No construtor da classe use a quantidade de números para gerar uma lista com todos esses números. Por exemplo, se quantidade de números for 50, a lista deve conter os números de 1 a 50. Utilize a função **shuffle()** da biblioteca **random** para embaralhar a lista de números. Além da lista de números, a classe deve ter outros atributos para guardar as listas de números já sorteados, jogadores e vencedores;

- Implemente o método **adiciona_jogador()** na classe **Bingo** para adicionar os jogadores. O método deve receber o nome do jogador e gerar seus números. Crie um cópia da lista de números do jogo, embaralhe essa cópia e pegue 30% de seus números para ser a lista de números do jogador. De posse do nome e dos números do jogador crie um objeto da classe **Jogador** e o adicione na lista de jogadores;

- A classe **Bingo** deve ter também os métodos **imprime()** e **sorteia()**. O primeiro deve mostrar os números já sorteados, os nomes e números faltantes de cada jogador. O segundo deve retirar um número da lista com todos os números (pode ser o último), adicionar esse número aos sorteados. Além disso, para cada número sorteado, será preciso marcar esse número para todos os jogadores e verificar se algum jogador já venceu o jogo. Um jogador vence o jogo quando não possui mais números para serem marcados;

- O principal método da classe Bingo será o método **jogar()**. Esse método deve verificar se existem jogadores cadastrados e iniciar um laço de repetição para sortear os números. A cada repetição do laço, o método deve mostrar o estado do jogo e sortear um número. É interessante incluir um **input()** para que os números sejam sorteadas sempre que o usuário pressionar ENTER. O laço de repetição deve continuar até que o jogo tenha algum vencedor. Por fim, após a conclusão do laço, o método deve mostrar os vencedores;

- Após a conclusão das classes, implemente um código para criar o objeto da classe **Bingo**, adicionar os jogadores e executar o método **jogar()**.

B) Jogo da forca:

- O jogo a ser implementado consiste em um jogo da forca de dois jogadores. O primeiro jogador insere a palavra no jogo e o segundo jogador tenta descobrir qual a palavra;

- Comece criando a classe **Palavra** para representar a palavra do jogo, se construtor deve receber a palavra e uma dica (opcional). A partir da palavra recebida no construtor, a classe deve criar um atributo **_tela** contendo a representação da palavra a ser mostrada na tela. Inicialmente, esse atributo é composto por um sublinhado para cada letra da palavra. A medida que o jogador for acertando as letras, os sublinhados serão substituídos pelas letras acertadas;

- A classe **Palavra** deve possuir um método **tela()** para mostrar o seu atributo tela e a dica (se houver). Inclua também o método **completada()** para informar ser a palavra já foi completada (todas as letras foram acertadas);

- Durante o jogo, precisaremos do método **tem_letra()** da classe palavra para verificar se um determinada letra exite na palavra. Esse método deve compara a letra recebida como parâmetro com todos as demais letras da palavra. Se a letra recebida for encontrada, o sublinhado do atribulo **_tela** na mesma posição deve ser substituído pela letra. O método **tem_letra()** deve retornar **True** se a letra for encontrada na palavra;

- A classe palavra deve possuir também um método de classe para validar a palavra a ser usada durante o jogo. Esse método deve retornar **True**, se todos os caracteres da palavra são letras entre A e Z;

- Além da classe palavra, cria a classe **Forca** para representar o jogo. O construtor da classe deve receber um objeto da classe **Palavra**, armazená-lo como atributo e criar também os atributos **_erros** e **_digitadas**. O atributo **_erros** deve ser inicializado com zero e servirá para contar os erros do jogador. O atributo **_digitadas** deve armazenar todas as letras digitadas durante o jogo;

- A classe **Forca** deve possuir também os métodos **mostrar()** e **eh_letra_valida()**. O método **mostrar()** deve exibir o estado atual do jogo composta pela representação da palavra na tela, as letras já digitadas e os erros. Nesse método é interessante usar uma instrução **print('\n'*100)** no início para limpar a tela e o segundo jogador não ver a palavra inserida pelo primeiro jogador. O método **eh_letra_valida()** retorna **True** se a entrada do usuário for uma letra válida. Isso é verdade se a entrada for uma única letra, essa letra estiver entre A e Z e não tiver sido digitada anteriormente;

- Por fim, a classe **Forca** deve ter um método **jogar()** contendo um laço de repetição para controlar o fluxo de execução do jogo. A cada repetição, devemos mostrar o estado do jogo e pegar a letra digitada pelo jogador. Essa letra deve ser validada usando o método **eh_letra_valida()**, as letras inválidas podem ser desconsideradas. Quando a letra for válida, devemos adicioná-la às letras digitadas e verificar se ela existe na palavra. Se não existir na palavra, contabilizamos o erro e o jogador perde ao atingir cinco erros. O último passo no laço de repetição é verificar se a palavra foi completada para finalizar o jogo com a vitória do jogador.

- Para finalizar a implementação do jogo, pega a palavra e a dica como primeiro jogador. Se for uma palavra válida, crie um objeto **Palavra**, crie o jogo **Forca** e chame o método **jogar()**.

2.10 Respostas dos exercícios

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

A) Jogo de bingo

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

from random import shuffle

class Jogador:

    def __init__(self, nome, numeros):
        # Nome do jogador
        self._nome = nome
        # Conjunto de números já marcados
        self._marcados = set()
        # Conjunto com todos os números do jogador
        self._numeros = set(numeros)

    @property
    def nome(self):
        return self._nome

    def marca(self, numero):
        # Testa se o jogador possui o número
        if numero in self._numeros:
            # Adiciona o número aos marcados
            self._marcados.add(numero)

    def faltantes(self):
        # Todos os números menos os números marcados
        return self._numeros - self._marcados

    def imprime(self):
        # Lista com números faltantes
        faltantes = list(self.faltantes())
        # Ordena a lista
        faltantes.sort()
        # Converte números para str
        faltantes_str = [str(n) for n in faltantes]
        # Imprime nome e números faltantes do jogador
        print(self._nome + ':', ' '.join(faltantes_str))

class Bingo:

    def __init__(self, total_numeros):
        # Cria lista de todos os números do jogo
        lista_numeros = list(range(1, total_numeros-1))
        # Lista de números sorteados
        self._sorteados = []
        # Lista de jogadores
        self._jogadores = []
```

```
49          # Lista de vencedores
50          self._vencedores = []
51          # Embaralha a lista de números
52          shuffle(lista_numeros)
53          # Armazena a lista de números como atributo
54          self._numeros = lista_numeros
55
56      def adiciona_jogador(self, nome):
57          # Copia a lista de números
58          lista_numeros = self._numeros[:]
59          # Embaralha a lista de números
60          shuffle(lista_numeros)
61          # Calcula a quantidade de números do jogador (30% to total)
62          quantidade_numeros = int(len(lista_numeros)*0.3)
63          # Pega apenas a quantidade de números do jogador
64          numeros = lista_numeros[:quantidade_numeros]
65          # Cria o jogador
66          jogador = Jogador(nome, numeros)
67          # Adiciona o jogador à lista de jogadores
68          self._jogadores.append(jogador)
69
70      def imprime(self):
71          # Mostra o estado do jogo
72          print('\n'*50)
73          print('BINGO\n')
74          # Mostra números sorteados
75          sorteados = [str(n) for n in self._sorteados]
76          print('Sorteados:', ' '.join(sorteados), '\n')
77          # Mostra cada jogador
78          for jogador in self._jogadores:
79              jogador.imprime()
80
81      def sorteia(self):
82          # O número sorteado é o último da lista (já embaralhada)
83          numero = self._numeros.pop()
84          # Adiciona o número aos sorteados
85          self._sorteados.append(numero)
86          # Percorre a lista d ejogadores
87          for jogador in self._jogadores:
88              # Marca o número para o jogador
89              jogador.marca(numero)
90              # Testa se o jogador já marcou todos os números
91              if len(jogador.faltantes()) == 0:
92                  # Adicionar o jogador à lista de vencedores
93                  self._vencedores.append(jogador)
94
95      def jogar(self):
96          # Testa se existem jogadores
97          if len(self._jogadores) > 0:
98              # Repete até que hajam vencedores
99              while not self._vencedores:
100                 # Mostra o estado do jogo
101                 self.imprime()
102                 # input() apenas para o usuário pressionar ENTER
103                 input('Pressione ENTER para sortear um número.')
```

```
104                 # Sorteia um número
105                 self.sorteia()
106             # Mostra o resultado final do jogo
107             self.imprime()
108             # Mostra os vencedores
109             print('\nVencedor(res):')
110             for jogador in self._vencedores:
111                 print(jogador.nome)
112 
113 
114 if __name__ == '__main__':
115     # Inicia o jogo com 50 números
116     bingo = Bingo(50)
117     print('Iniciando o jogo')
118     print('Informe os nomes dos jogadores (vazio para parar)')
119     # Laço para adicionar os jogadores
120     cont = 1
121     while True:
122         # Pega o nome do jogador
123         nome = input('Jogador '+str(cont) +': ')
124         cont += 1
125         # Se o nome estiver em branco, interrompe o laço
126         if nome.strip() == '':
127             break
128         # Adiciona o jogador ao jogo
129         bingo.adiciona_jogador(nome)
130     # Inicia o jogo
131     bingo.jogar()
```

B) Jogo da forca

```
 1 #!/usr/bin/env python3
 2 # -*- coding: utf-8 -*-
 3 
 4 
 5 class Palavra:
 6 
 7     def __init__(self, palavra, dica=''):
 8         # Guarda a palavra
 9         self._palavra = palavra
10         # Guarda a dica
11         self._dica = dica
12         # Um sublinhado para cada letra para mostra na tela
13         self._tela = ['_'] * len(palavra)
14 
15     @property
16     def tela(self):
17         # Retorna a representação da palavra na tela
18         mensagem = ''.join(self._tela)
19         if len(self._dica) > 0:
20             mensagem += '\n\nDICA: ' + self._dica
21         return mensagem
22 
23     def tem_letra(self, letra):
24         # Supões que não tem a letra
```

```
25          encontrada = False
26          # Percorre os caracteres da palavra
27          for cont, carac in enumerate(self._palavra):
28              # Verifica se a letra é igual ao caractere atual
29              if carac == letra:
30                  # Letra encontrada
31                  encontrada = True
32                  # Mostra a letra na representação da tela
33                  self._tela[cont] = letra
34          return encontrada
35
36      @property
37      def completada(self):
38          # A palavra é completada quando não tem mais sublinhados
39          return not ('_' in self._tela)
40
41      @classmethod
42      def eh_valida(cls, palavra):
43          # Para cada letra da palavra
44          for letra in palavra:
45              # Verifica se a letra está entre A e Z
46              if letra < 'A' or letra > 'Z':
47                  # Se não estiver, retorna False
48                  return False
49          # Retorna True, se a palavra for válida
50          return True
51
52
53  class Forca:
54
55      def __init__(self, palavra):
56          # Palavra do Forca
57          self._palavra = palavra
58          # Erros do jogador
59          self._erros = 0
60          # Letras já digitadas
61          self._digitadas = ''
62
63      def mostrar(self):
64          # 100 linhas em branco para efeito de limpar tela
65          print('\n'*100)
66          # Representação de tela da palavra
67          print(self._palavra.tela)
68          # Letras digitadas e erros
69          print('Letras digitadas:', self._digitadas)
70          print('Erros:', self._erros)
71
72      def eh_letra_valida(self, letra):
73          # A letra deve ser um único caracter
74          if len(letra) != 1:
75              return False
76          # Dever ser entre A e Z e não ter sido digitada antes
77          if letra < 'A' or letra > 'Z' or letra in self._digitadas:
78              return False
79          return True
```

```
 80
 81      def jogar(self):
 82          # Laço que repete até o Forca terminar
 83          while True:
 84              # Mostra o estado do Forca
 85              self.mostrar()
 86              # Pega uma letra
 87              letra = input('Informe uma letra: ').upper().strip()
 88              # Se a letra não for válida, pega novamente
 89              if not self.eh_letra_valida(letra):
 90                  continue
 91              # Adiciona letra às digitadas
 92              self._digitadas += letra
 93              # Verifica se a letra não existe na palavra
 94              if not self._palavra.tem_letra(letra):
 95                  # Contabiliza o erro
 96                  self._erros += 1
 97                  # Testa se já tem 5 erros
 98                  if self._erros == 5:
 99                      # O jogador perde e o Forca termina
100                      self.mostrar()
101                      print('Você perdeu!')
102                      break
103              # Verifica se a palavra foi compeltada
104              if self._palavra.completada:
105                  # O jogador vernce e o Forca termina
106                  self.mostrar()
107                  print('Você acertou!')
108                  break
109
110
111  if __name__ == '__main__':
112      # Laço para pegar uma palavra válida
113      while True:
114          print('Informe a palavra para seu adversário')
115          print('Não use espaços ou caracteres especiais')
116          # Pega a palavra
117          palavra_str = input().upper().strip()
118          # Testa se a palavra é valida
119          if Palavra.eh_valida(palavra_str):
120              # Se for, interrompe o laço
121              break
122          else:
123              # Senão, solicita a palavra novamente
124              print('Palavra inválida! Tente novamente.')
125      # Pega a dica (opcional)
126      dica = input('Infome uma dica (ou deixe em branco): ').strip()
127      # Cria o objeto palavra
128      palavra = Palavra(palavra_str, dica)
129      # Cria e inicia o jogo
130      forca = Forca(palavra)
131      forca.jogar()
```

2.11 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 Segunda Semana” para recapitular tudo que aprendemos.

Nos encontramos na próxima semana.

Bons estudos!

Objetivos
- Conhecer o básico da manipulação de arquivos;
- Conhecer as diferenças entre arquivos de texto simples, arquivos CSV e arquivos JSON;
- Desenvolver códigos com persistência de dados em arquivos.

3.1 Introdução

Os códigos que escrevemos até o momento não salvam as informações coletadas durante a execução. Portanto, todos os dados inseridos são perdidos quando finalizamos a execução. Isso acaba impactando negativamente na utilidade de diversos códigos. Uma das maneiras de persistirmos dados é por meio de arquivos. A linguagem Python possui diversas bibliotecas e funções para manipulação de arquivos (CEDER, 2018). O presente capítulo abordará algumas dessas possibilidades.

3.2 Arquivos de texto simples

O tipo de arquivo mais simples de ser manipulado em Python é o arquivo texto simples onde os dados são gravados e lidos diretamente no formato **str** (CORRÊA, 2020). Antes de mais nada, temos que abrir o arquivo usando a função **open()** e associá-lo a uma variável. Os principais parâmetros da função **open()** são o nome do arquivo e o modo de abertura. O nome do arquivo é um **str** com o caminho completo para o arquivo ou, quando estiver no diretório atual, apenas o nome.

Caractere	Significado
`'r'`	Abre para leitura (padrão)
`'w'`	Abre para escrita de novo arquivo (todo o conteúdo é apagado)
`'x'`	Abre para criação exclusiva, falhando caso o arquivo exista
`'a'`	Abre para escrita anexando o conteúdo no final do arquivo
`'b'`	Modo binário
`'t'`	Modo texto (padrão)
`'+'`	Abre para atualização (leitura e escrita)

Figura 94 – Modos de abertura de arquivos
Fonte: Elaborado pelo Autor.

O modo de abertura é uma combinação válida dos caracteres mostrados na Figura 94. O modo padrão de abertura é arquivos de texto para leitura (**'rt'**). Quando a execução da função **open()** não encontre erros, seu resultado é um objeto associado ao arquivo aberto que pode ler ou gravar dados. É importante que, após as operações de leitura e escrita o arquivo seja fechado usando o método **close()**.

Para começarmos a trabalhar com arquivos vamos criar um código simples que pega contatos com o usuário e os guarda em arquivo. Precisamos abrir o arquivo no modo **'w'** para escrita e gravar os dados usando o método **write()** como mostrado na Figura 95. Observe que, ao gravarmos o contato no arquivo, acrescentamos **'\n'** para que cada contato fique em um linha. Isso facilita o processo de leitura posteriormente.

```
print('Informe os dados de cada contato para gravação')
print('Deixe em branco para parar')
arq = open('contatos.txt', 'w')
while True:
    contato = input('Contato: ')
    if contato == '':
        break
    arq.write(contato + '\n')
arq.close()
```

Figura 95 – Escrevendo dados em arquivo de texto
Fonte: Elaborado pelo Autor.

O método **write()** é usado quando queremos escrever linhas individuais ou partes do arquivo de cada vez. O objeto de arquivo também possui o método **writelines()** que permite escrever uma lista de **str** de uma só vez. Normalmente, o **writelines()** é mais usado quando já temos uma lista de dados e desejamos guardá-la em arquivo.

Quanto à leitura, primeiro, temos que abrir o arquivo no modo de leitura. Em seguida, podemos usar os métodos **read()**, **readline()** ou **readlines()** para ler os dados do arquivo. O código da Figura 96 demostra como tais métodos podem ser usados.

```
print('Contatos cadastrados:')

arq = open('contatos.txt')
linhas = arq.readlines()
for contato in linhas:
    print(contato.strip())
arq.close()
print('-'*50)

arq = open('contatos.txt')
while True:
    contato = arq.readline()
    if contato == '':
        break
    print(contato.strip())
arq.close()
print('-'*50)

arq = open('contatos.txt')
print(arq.read().strip())
arq.close()
```

Figura 96 – Lendo dados de arquivo de texto
Fonte: Elaborado pelo Autor.

O método **readlines()** retorna uma lista com todas as linhas do arquivo. Já o método **readline()** pode ser usado para ler uma linha de cada vez. O método **read()** retorna um

único **str** com todo o conteúdo do arquivo. Quando estamos no final do arquivo, os métodos **read()** e **redline()** retornam '' (**str** vazio) e o método **readlines()** retorna **[]** (lista vazia). Portanto, a decisão de qual método usar vai depender do funcionamento do código.

No código, podemos notar também o uso do método **strip()**. Na gravação dos dados incluímos o **'\n'** para forçar a gravação de um contato por linha. Quando lemos os dados, lemos também o **'\n'**. Usamos o **strip()** para remover esse caractere e escrever corretamente o contato na tela.

O código de leitura vai funcionar corretamente apenas se o arquivo já existir. Caso contrário, acontecerá o erro **FileNotFoundError**. Para resolver esse problema, podemos fazer tratamento de exceções ou testar se o arquivo existe usando a função **isfile()** da biblioteca **os.path**. A Figura 97 demostra como a função **isfile()** pode ser usada.

```
import os

if not os.path.isfile('contatos.txt'):
    print('Arquivo não encontrado')
else:
    print('Contatos cadastrados:')
    arq = open('contatos.txt')
    linhas = arq.readlines()
    for contato in linhas:
        print(contato.strip())
    arq.close()
```

Figura 97 – Verificando se o arquivo existe
Fonte: Elaborado pelo Autor.

3.3 Jogo da forca com arquivo

Para demonstrar de forma mais prática como podemos utilizar arquivos, vamos realizar uma melhoria no jogo da forca construído como exercício no capítulo anterior. O jogo original solicita uma palavra de um jogador para que outro jogador tente descobre qual foi é a palavra. Nossa melhoria envolverá a criação de um arquivo com as palavras do jogo. Assim, depois do cadastro de algumas palavras, os jogadores podem simplesmente jogar sem ter que inserir palavras.

A melhoria no jogo será feita com a classe Arquivo mostrada na Figura 98. No construtor, inicializamos o nome do arquivo e um dicionário para guardar as palavras e as dicas. Usamos a função **isfile()** para verificar se o arquivo existe, antes de abri-lo. Na abertura do arquivo usamos a instrução **with open(...) as arq**. Usando essa instrução, o arquivo é fechado automaticamente e não precisamos chamar o método **close()**. Após a abertura do arquivo, usamos um laço **for** para percorrer suas linhas. Essa é outra maneira de ler os dados do arquivos sem usar os métodos **read()**, **readline()** ou **readlines()**. A medida que lemos cada linha, extraímos a palavra e sua dica. No arquivo, incluímos a palavra e sua dica em uma mesma linha separadas por **':'** (dois pontos).

O método **sorteia_palavra()** soteia uma palavra do dicionário para ser usada durante o jogo. Primeiro, extraímos a lista de palavras usando o método **keys()** do dicionário. Usamos a função **shuffler()** para embaralhar a lista e pegamos a primeira

palavra da lista embaralhada. A partir da palavra, pegamos sua dica no dicionário e o método retorna uma tupla com a palavra e a dica.

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

from random import shuffle
from os.path import isfile


class Palavra:
    # ...


class Forca:
    # ...


class Arquivo:

    def __init__(self, nome_arquivo):
        # Nome do arquivo
        self._nome_arquivo = nome_arquivo
        # Dicionário (palavra:dica)
        self._dic_palavras = {}
        # Checa se o arquivo existe
        if isfile(self._nome_arquivo):
            # Abre o arquivo para leitura
            with open(self._nome_arquivo) as arq:
                # Percore as linhas do arquivo
                for linha in arq:
                    # Separa a palavra da dica
                    palavra, dica = tuple(linha.strip().split(':'))
                    # Adiciona no dicionário
                    self._dic_palavras[palavra] = dica

    def sorteia_palavra(self):
        # Extrai a lista de palavras do dicionário
        lista_palavras = list(self._dic_palavras.keys())
        # Embaralha a lista de palavras
        shuffle(lista_palavras)
        # Pega a primeira palavra da lista embaralhada
        palavra = lista_palavras[0]
        # Pega a dica associada à palavra
        dica = self._dic_palavras[palavra]
        # Retorna a palavra e sua dica
        return (palavra, dica)
```

Figura 98 – Classe **Arquivo** com métodos **__init__()** e **sorteia_palavra()**
Fonte: Elaborado pelo Autor.

Antes de iniciarmos o jogo, precisamos saber se já existe alguma palavra cadastrada no arquivo. Vamos implementar o método especial **__len__()** para que possamos usar a função **len()** sobre um objeto da classe **Arquivo** e saber quantas palavras estão cadastradas. A Figura 99 mostra como o método especial é implementado.

Basicamente, a quantidade de palavras cadastradas será a quantidade de palavras contidas no dicionário **_dic_palavras**.

```
# ...
class Arquivo:
    # ...

    def __len__(self):
        # A função len sobre o objeto será o tamanho o dicionário
        return len(self._dic_palavras)

    def lista_palavras(self):
        print('-'*50)
        print('Lista de palavras')
        print('-'*50)
        # Verifica se o dicionário está vazio
        if len(self._dic_palavras) == 0:
            print('Nenhuma palavra cadastrada!')
        else:
            # Extrai a lista de palavras do dicionário
            lista_palavras = list(self._dic_palavras.keys())
            # Ordena as palavras
            lista_palavras.sort()
            # Lista as palavras com suas dicas
            for palavra in lista_palavras:
                dica = self._dic_palavras[palavra]
                print(palavra, ' (', dica, ')', sep='')
        print('-'*50)
```

Figura 99 – Métodos **__len__()** e **lista_palavras()** da classe **Arquivo**
Fonte: Elaborado pelo Autor.

O código da Figura 99 apresenta também o método **lista_palavras()** usado para mostrar todas as palavras já cadastradas. Esse método faz uso da função **len()** para testar se o dicionário de palavras está vazio. Nesse caso, o método apenas exibe a mensagem de nenhuma palavra cadastrada. Por outro lado, se houverem palavras cadastradas, o método extrai a lista de palavras do dicionário, ordena essa lista e mostra as palavras com suas respectivas dicas.

A classe **Arquivo** permitirá também a inclusão e exclusão de palavras. Tais operações são implementadas através dos métodos **incluir()** e **excluir()** mostrado na Figura 100. O método **incluir()** recebe a palavra e a dica do usuário, testa se a palavra é válida e, se não existir, a insere no dicionário. No caso da dica, usamos o método **replace()** para remover o caractere **':'** porque ele é usado como separador nas linhas arquivo e, havendo mais de um separador, teremos erro na leitura do arquivo. O método **excluir()** recebe a palavra a ser excluída do usuário, verifica se a mesma existe no dicionário e a remove usando o método **pop()** do dicionário.

O método **salvar()** grava as palavras do dicionário em arquivo. No arquivo, escrevemos em cada linha um **str** com a palavra e a dica, separadas por **':'** (dois pontos). Portanto, no método, percorremos o dicionário, montamos a linha usando a palavra e dica e gravamos essa linha no arquivo.

```
# ...
class Arquivo:
    # ...

    def incluir(self):
        print('Inclusão de palavra')
        # Recebe a palavra e a dica do usuário
        palavra = input('Palavra: ').strip().upper()
        # Remove ':' da dica para evitar problemas no arquivo
        dica = input('Dica: ').replace(':', ' ', -1)
        # Verifica se a palavra é valida
        if not Palavra.eh_valida(palavra):
            print('Palavra inválida!')
        # Verifica se a palavra já existe no dicionário
        elif palavra in self._dic_palavras:
            print('Palavra já cadastrada!')
        else:
            # Guarda a palavra no dicionário
            self._dic_palavras[palavra] = dica

    def excluir(self):
        # Recebe a palavra do usuário
        palavra = input('Palavra a ser excluída: ')
        palavra = palavra.strip().upper()
        # Verifica se a palavra não existe no dicionário
        if palavra not in self._dic_palavras:
            print('Palavra não encontrada')
        else:
            # Remove a palavra do dicionário
            self._dic_palavras.pop(palavra)

    def salvar(self):
        # Abre o arquivo para escrita
        with open(self._nome_arquivo, 'w') as arq:
            # Para cada palavra e dica
            for palavra, dica in self._dic_palavras.items():
                #  Constrói a linha a ser gravada no arquivo
                linha = palavra + ':' + dica + '\n'
                # Grava linha no arquivo
                arq.write(linha)
```

Figura 100 – Métodos **incluir()**, **excluir()** e **salvar()** da classe **Arquivo**
Fonte: Elaborado pelo Autor.

Por fim, criamos o método **cadastro()** mostrado na Figura 101. Esse método exibe as opções (incluir, excluir, salvar e sair) e chama os métodos adequados. A Figura 101 apresenta também a função **principal()** para criar o objeto **arq_palavras** da classe **Arquivo** e entrar no laço com o menu de opções. Se a opção de jogar for selecionada, a função verifica se existem palavras cadastradas. Em caso afirmativo, sorteamos uma palavra com o método **sorteia_palavra()** do objeto **arq_palavras** e iniciamos o jogo. Se o usuário selecionar a opção de cadastro de palavras, chamamos o método **cadastro()** do objeto **arq_palavras**.

```
# ...
class Arquivo:
    # ...

    def cadastro(self):
        while True:
            self.lista_palavras()
            print('Informe a opção desejada:')
            resp = input('(I)ncluir (E)xcluir (S)alvar Sai(r) ')
            resp = resp.lower().strip()
            if resp == 'r':
                break
            elif resp == 'i':
                self.incluir()
            elif resp == 'e':
                self.excluir()
            elif resp == 's':
                self.salvar()


def principal():
    # Cria objeto associado ao arquivo de palavras
    arq_palavras = Arquivo('palavras.txt')
    while True:
        print('-'*50)
        print('Jogo da forca')
        print('-'*50)
        print('(J)ogar')
        print('(C)adastro de palavras')
        print('(S)air')
        resp = input('Informe a opção desejada: ').strip().lower()
        if resp == 'j':
            if len(arq_palavras) == 0:
                print('Não há palavras cadastradas!')
            else:
                # Sorteia uma palavra
                palavra, dica = arq_palavras.sorteia_palavra()
                # Cria e inicia o jogo Forca
                forca = Forca(Palavra(palavra, dica))
                forca.jogar()
                # Espere ENTER ao terminar o jogo
                input()
        elif resp == 'c':
            arq_palavras.cadastro()
        elif resp == 's':
            break


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

Figura 101 – Método **cadastro()** da classe **Arquivo** e função **principal()**
Fonte: Elaborado pelo Autor.

3.4 Arquivos CSV

No código anterior usamos um arquivo de texto simples para armazenar as palavras do jogo da forca. Como precisávamos guardar a palavra e a dica, guardamos ambas informações em uma única linha do arquivo usando **':'** como separador. Em várias situações práticas, precisamos guardar não duas, mas várias informações por linha. Muitas vezes esses dados são armazenados em dados CSV. A sigla vem da expressão inglesa *comma separeted values* que significa valores separados por vírgula (podem ser usados delimitadores diferentes da vírgula).

A manipulação de arquivos CSV manualmente pode se tornar uma tarefa relativamente complexa. Entretanto, a linguagem Python possui a biblioteca **csv** com diversas funcionalidades prontas para lidar com arquivos CSV. Para ilustrar a utilização de referida biblioteca vamos implementar uma agenda de contatos com dados armazenados em arquivo CSV.

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

import csv
from os.path import isfile


# Escreve linha na tela
def linha():
    print('-'*50)


class Contato:

    # Campos dos contatos
    campos = ['Nome', 'Telefone', 'Aniversário']

    def __init__(self, valores):
        # Inicializa dicionário para os campos
        self._dic = {}
        # Precorre lista de campos
        for cont, campo in enumerate(self.campos):
            # Atribui o valor correspondente ao campo
            self._dic[campo] = valores[cont]
```

Figura 102 – Projeto de agenda de contatos com a classe **Contato**
Fonte: Elaborado pelo Autor.

A Figura 102 mostra o código inicial do projeto. Vamos precisar das bibliotecas **csv** e **os.path**. Criamos uma função **linha()** para imprimir linhas durante a apresentação de dados e a classe **Contato**. A lista **campos** da classe **Contato** representa quais dados serão guardados. O construtor da classe, recebe uma lista de valores, cria um dicionário com os campos como chaves para os valores recebidos.

Depois de criados, os contatos poderão ser alterados pelo usuário. Para isso vamos criar o método **alterar()** para a classe **Contato** como mostrado na Figura 103. Basicamente, percorremos todos os campos, mostramos seus valores atuais e solicitamos

novos valores para o usuário. Se o valor informado for diferente de vazio, atualizamos o campo com esse novo valor. Na mesma figura é apresentado também o método **valores()** que retorna a lista de valores para os campos do contato.

```
# ...
class Contato:
    # ...

    def alterar(self):
        linha()
        print('Alteração de contato')
        linha()
        # Percorre campos
        for campo, valor in self._dic.items():
            # Mostra o campo como valor atual
            print(campo + ' (' + valor + ')', sep='')
            # Pega novo valor
            novo_valor = input('Novo valor: ').strip()
            # Verifica se o valor é diferente de ''
            if novo_valor != '':
                # Atribui novo valor ao campo
                self._dic[campo] = novo_valor

    @property
    def valores(self):
        # Retorna lista com valores dos campos
        lista_valores = []
        for campo in self.campos:
            lista_valores.append(self._dic[campo])
        return lista_valores
```

Figura 103 – Métodos **alterar()** e **valores()** da classe **Contato**
Fonte: Elaborado pelo Autor.

Os contatos deverão ser impressos na tela e também serão comparados para ordenar a lista de contatos futuramente. Assim precismos implementar os métodos especiais **__str__()** e **__lt__()** conforme descrito no código da Figura 104. O método **__str__()** concatena os campos e seus valores. Já o método **__lt__()** retorna o resultado da comparação do objeto atual com outro objeto. A comparação é feita com as tuplas de valores dos contatos.

```
# ...
class Contato:
    # ...

    def __str__(self):
        # Str com camo e valores
        lista_cv = [campo + ': ' + self._dic[campo]
                    for campo in self._dic]
        return '\n'.join(lista_cv)

    def __lt__(self, other):
        # Comparação < (necessário para ordenar a lista de contatos)
        return tuple(self.valores) < tuple(other.valores)
```

Figura 104 – Métodos **__str__()** e **__lt__()** da classe **Contato**
Fonte: Elaborado pelo Autor.

Os novos contatos poderão ser criados por meio do método **novo()** apresentado na Figura 105. Esse método obtém os valores dos campos com o usuário e, caso os dados sejam válidos, retorna um objeto da classe **Contato** inicializado com os valores informados. O único dado obrigatório é o nome do contato. Se o usuário informar um um nome vazio, o contato não será criado.

```
# ...

class Contato:
    # ...

    @classmethod
    def novo(cls):
        linha()
        print('Novo contato')
        linha()
        # Lista de valores
        lista_valores = []
        for campo in cls.campos:
            # Obtém o valor de cada campo
            valor = input(campo + ': ').strip()
            # Adiciona à lista de valores
            lista_valores.append(valor)
        # Verifica se o nome ficou vazio
        if len(lista_valores[0]) == 0:
            print('Contato inválido, o nome é obrigatório!')
            # Não cria contato com nome vazio
            return None
        else:
            # Cria o contato
            return Contato(cls.campos, lista_valores)
```

Figura 105 – Método **novo()** da classe **Contato**
Fonte: Elaborado pelo Autor.

Após a definição da classe **Contato**, vamos criar a classe **Arquivo** que manipulará a lista de contatos e o arquivo CSV. A Figura 106 mostra a classe **Arquivo** com seu construtor e os método **listar()** e **buscar()**. O construtor verifica se o arquivo existe e, em caso afirmativo, faz a leitura dos contatos. Observe que criamos um objeto leitor da classe

reader da biblioteca **csv**. O leitor faz o tratamento automático dos valores delimitados no arquivo CSV. Após a criação do leitor, usamos a instrução **next(leitor)** para ignorar a primeira linha do arquivo (cabeçalho). Em seguida, percorremos as demais linhas, criamos os contatos e adicionamos a lista.

O método **listar()** é usado para escrever a lista de contatos na tela. Se a lista de contatos estiver vazia, o método apenas mostra uma mensagem informando que não há contatos cadastrados. Se houverem contatados na lista, o método ordena a lista e usa um laço de repetição para escrever cada um dos contatos na tela.

O método **buscar()** recebe um número inteiro e verificar se o mesmo é um aposição válida na lista de contatos. Se sim, o método retorna o contato da posição. Senão, o método retorna **None**.

```
# ...

class Arquivo:

    def __init__(self, nome_arquivo):
        # Nome do arquivo e lista de contatos
        self._nome_arquivo = nome_arquivo
        self._lista_contatos = []
        # Checa se o arquivo existe
        if isfile(self._nome_arquivo):
            # Abre o arquivo para leitura
            with open(self._nome_arquivo) as arq:
                # Criar leitor de CSV
                leitor = csv.reader(arq)
                # Ignora linha de cabeçalho
                next(leitor)
                # Percore as linhas usando o leitor
                for linha in leitor:
                    # Cria contato e adiciona na lista
                    contato = Contato(linha)
                    self._lista_contatos.append(contato)

    def listar(self):
        # Verifica se a lista de contatos está vazia
        if len(self._lista_contatos) == 0:
            print('Nenhum contato cadastrado!')
            linha()
        else:
            # Ordena a lista de contatos
            self._lista_contatos.sort()
            # Percorre a lista de contatos
            for cont, contato in enumerate(self._lista_contatos):
                print('Código:', cont)
                print(str(contato))
                linha()

    def buscar(self, codigo):
        # Busca contato pelo código
        if 0 <= codigo and codigo < len(self._lista_contatos):
            return self._lista_contatos[codigo]
        return None
```

Figura 106 – Classe **Arquivo** com construtor, métodos **listar()** e **buscar()**
Fonte: Elaborado pelo Autor.

Concluindo a classe Arquivo, criamos também os métodos **incluir()**, **excluir()** e **salvar()** conforme mostrado na Figura 107. O método incluir usa o método **novo()** da classe **Contato** para criar um contato com as informações recebidas pelo usuário. Se o contato for válido (diferente de **None**), o método o inclui na lista de contatos.

O método **excluir()** recebe um código que representa a posição do contato dentro da lista. Se essa posição for válida, o contato é removido da lista usando o método **pop()**. O método **salvar()** abre o arquivo CSV em modo de escrita e cria um escritor usando a classe **csv.writer**. Primeiro, usamos o escritor para gravar o cabeçalho do arquivo. Depois, percorremos a lista de contados para salvar cada um dos contatos no arquivo.

```python
# ...

class Arquivo:
    # ...

    def incluir(self):
        # Cria novo contato
        contato = Contato.novo()
        # Testa se o contato é válido
        if contato is not None:
            # Incluir na lista de contatos
            self._lista_contatos.append(contato)

    def excluir(self, codigo):
        # Verifica se o código do contato existe
        if 0 <= codigo and codigo < len(self._lista_contatos):
            # Remove da lista de contatos
            self._lista_contatos.pop(codigo)

    def salvar(self):
        # Abre o arquivo para escrita
        with open(self._nome_arquivo, 'w') as arq:
            # Cria escritor CSV
            escritor = csv.writer(arq)
            # Escreve cabeçalho
            escritor.writerow(Contato.campos)
            # Percorre lista de contatos
            for contato in self._lista_contatos:
                # Escreve cada contato
                escritor.writerow(contato.valores)
```

Figura 107 – Métodos **incluir()**, **excluir()** e **salvar()** da classe **Arquivo**
Fonte: Elaborado pelo Autor.

```python
# ...

class Agenda:

    def __init__(self, nome_arquivo):
        # Cria objeto associado ao arquivo
        self._arq = Arquivo(nome_arquivo)

    def menu(self):
        # Menu que lista e recebe a opção do usuário
        linha()
        print('Agenda de contatos')
        linha()
        self._arq.listar()
        print('(I)ncluir | (E)xcluir | (A)lterar | (S)alvar | Sai(r)')
        return input('Informe a opção desejada: ').strip().lower()
```

Figura 108 – Classe **Agenda** com seu construtor e método **menu()**
Fonte: Elaborado pelo Autor.

A última classe do nosso projeto é a classe **Agenda** responsável por controlar a interação como usuário. A Figura 108 mostra a classe, se construtor e o método **menu()**. O construtor recebe o nome do arquivo e inicializa o atributo **_arq** como um objeto da classe

Arquivo. O método **menu()** exibe a lista de contatos e o menu de opções para o usuário. O retorno do método é a opção do menu escolhida pelo usuário.

```
# ...

class Agenda:
    # ...

    def executar(self):
        while True:
            resp = self.menu()
            if resp == 'i':
                self._arq.incluir()
            elif resp == 'e':
                codigo = int(input('Código do contato: '))
                self._arq.excluir(codigo)
            elif resp == 'a':
                codigo = int(input('Código do contato: '))
                # Busca contato
                contato = self._arq.buscar(codigo)
                # Testa se o contato foi encontrado
                if contato is not None:
                    contato.alterar()
            elif resp == 's':
                self._arq.salvar()
            elif resp == 'r':
                break


if __name__ == '__main__':
    agenda = Agenda('contatos.csv')
    agenda.executar()
```

Figura 109 – Método **executar()** da classe Agenda e **if** de execução
Fonte: Elaborado pelo Autor.

A Figura 109 mostra o método **executar()** da classe **Agenda** e o **if** de execução do arquivo. O método **executar()** começa chamando o método **menu()** para obter a opção escolhida pelo usuário. Em seguida, de acordo com a resposta do usuário, os demais métodos são executados para incluir, excluir, alterar ou salvar.

Em nosso projeto, não foi necessário, mas a biblioteca **csv** possui diversas opções relacionadas ao formato dos arquivos como definir o delimitador de campos, conversão de valores numéricos dentre outras. Recomendamos que você leia a documentação da biblioteca para conhecer melhor essas opções.

3.5 Arquivos JSON

O *Javascript Object Notation (JSON)* é um formato de dados muito utilizando, principalmente, na integração de sistemas. O JSON é um formato compacto e aberto que utiliza texto legível, no formato atributo-valor criando em 2000. Na prática, os dados no formato JSON são muito parecidos com o resultado de um **print()** de dicionários e listas.

Vamos agora criar um projeto utilizando arquivos no formato JSON. O projeto é um sistema de gerenciamento de árvore genealógica. Basicamente, o sistema permitirá o cadastro de pessoas, bem como a atribuição de seus pais. Após o cadastro de algumas pessoas, o sistema poderá imprimir a árvore genealógica contendo os ancestrais de um indivíduo.

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

import json
from os.path import isfile

PAI = 'Pai'
MAE = 'Mãe'


# Escreve linha na tela
def linha():
    print('-'*50)


class Arvore:

    def __init__(self, nome_arquivo):
        # Arquivo e árvore de pessoas
        self._nome_arquivo = nome_arquivo
        self._arvore = {}
        # Checa se o arquivo existe
        if isfile(self._nome_arquivo):
            # Abre arquivo e lê usando JSON
            with open(self._nome_arquivo) as arq:
                self._arvore = json.load(arq)
```

Figura 110 – Classe **Arvore** e código inicial do projeto de árvore genealógica
Fonte: Elaborado pelo Autor.

A Figura 110 apresenta o código inicial do projeto com as importações, variáveis globais, função **linha()** e classe **Arvore** com seu construtor. As importações contemplam a biblioteca **json** para manipular o conteúdo dos arquivos e a biblioteca **os.path** para verificar a existência de arquivos. As variáveis globais **PAI** e **MAE** serão usadas como chaves de dicionário para indicar o pai e a mãe de um indivíduo. A classe **Arvore** conterá o código principal do projeto, seu construtor recebe o nome do arquivo JSON com os dados já armazenados, lê esse arquivo e armazena seu conteúdo no dicionário **_arvore**.

O dicionário **_arvore** terá como chaves os nomes das pessoas e como valor dicionários contento o pai e a mãe de cada pessoa. Como exemplo, considere uma pessoa A que tem como pai B e mãe C, por sua vez, B tem como pai D. Essa árvore genealógica é representada pelo dicionário mostrado na Figura 111.

```
{'A': {'Pai': 'B', 'Mãe': 'C'},
 'B': {'Pai': 'D', 'Mãe': ''},
 'C': {'Pai': '', 'Mãe': ''},
 'D': {'Pai': '', 'Mãe': ''}}
```

Figura 111 – Exemplo de árvore genealógica representada usando dicionário
Fonte: Elaborado pelo Autor.

Depois da árvore carregada no dicionário precisamos de métodos para incluir outras pessoas e listar as pessoas já cadastradas. A Figura 112 mostra a implementação desses métodos.

```
# ...

class Arvore:
    # ...

    def incluir(self, nome):
        # Verifica se o nome é válido
        if nome == '':
            print('Nome inválido!')
            input()
        elif nome in self._arvore:
            print('Pessoa já cadastrada ou nome inválido!')
            input()
        else:
            # Inclui nome na árvore (sem pai e mãe)
            self._arvore[nome] = {PAI: '', MAE: ''}

    def listar(self):
        # Testa se há pessoas na árvore
        if len(self._arvore) == 0:
            print('Nenhuma pessoa cadastrada!')
            linha()
        else:
            # Obtém e ordena lista de nomes
            lista_nomes = list(self._arvore.keys())
            lista_nomes.sort()
            # Para cada nome
            for nome in lista_nomes:
                # Obtém dados do nome
                dados = self._arvore[nome]
                # Cria lista de dados (campo: valor)
                lista_dados = []
                for campo, valor in dados.items():
                    if valor != '':
                        lista_dados.append(campo + ': ' + str(valor))
                # Junta dados em str
                dados_str = ''
                if len(lista_dados) > 0:
                    dados_str = '(' + ', '.join(lista_dados) + ')'
                # Imprime nome e dados
                print(nome, dados_str)
                linha()
```

Figura 112 – Métodos **incluir()** e **listar()** da classe **Arvore**
Fonte: Elaborado pelo Autor.

O método **incluir()** recebe o nome da pessoa e verifica e o mesmo é válido. Os nomes vazios ou já cadastrados são considerados inválidos. Os nomes válidos são incluídos como chaves no dicionário e o valor é um dicionário com o pai e a mãe que, inicialmente são vazios. Posteriormente, criaremos métodos para cadastrar os pais das pessoas.

O método **listar()** verifica se o dicionário está vazio e mostra uma mensagem que nenhuma pessoa está cadastrada. Se houverem pessoas cadastradas, o método cria uma lista ordenada com as mesmas. Em seguida, o método percorre a lista e obtém os dados (de pai e mãe) de cada pessoa e exibe a informação da pessoa juntamente com seus pais. O dado de pai ou mãe que não estiver cadastrado, não é exibido.

```
# ...

class Arvore:
    # ...

    def buscar(self, nome):
        # Verifica se nome existe e o retorna
        if nome in self._arvore:
            return self._arvore[nome]
        # Retorna '' se o nome não existir
        return ''

    def alterar_pais(self, nome):
        # Busca a pessoa na árvore
        pessoa = self.buscar(nome)
        if pessoa != '':
            # Obtém nome dos pais
            nome_pai = input('Pai: ').strip()
            nome_mae = input('Mãe: ').strip()
            # Altera nome dos pais (se existirem na árvore ou for '')
            if nome_pai in self._arvore or nome_pai == '':
                pessoa[PAI] = nome_pai
            if nome_mae in self._arvore or nome_mae == '':
                pessoa[MAE] = nome_mae

    def excluir(self, nome):
        # Busca a pessoa
        pessoa = self.buscar(nome)
        # Testa se a pessoa foi encontrada
        if pessoa != '':
            # Verifica se a pessoa é pai ou mãe de alguém
            for outro, dados in self._arvore.items():
                if dados[PAI] == nome:
                    print(nome, 'é pai de', outro)
                    input()
                    return
                if dados[MAE] == nome:
                    print(nome, 'é mãe de', outro)
                    input()
                    return
            self._arvore.pop(nome)
```

Figura 113 – Métodos **buscar()**, **alterar_pais()** e **excluir()** da classe **Arvore**
Fonte: Elaborado pelo Autor.

Os métodos **buscar()**, **alterar_pais()** e **excluir()** da classe **Arvore** são apresentados na Figura 113. O método **buscar()** recebe o nome da pessoa e retorna os dados da mesma. Se a pessoa não existir o método retorna **''** (**str** vazio). O método **alterar_pais()** recebe o nome da pessoa e, se a mesma existir, procede com a alteração de seus pais. O nome do pais ou da mão deve ser previamente cadastrado ou ser vazio, ou a alteração não será efetivada.

O método **excluir()** recebe o nome e busca a pessoa no dicionário. Se a pessoa existir, o método varre o dicionário para verificar se a pessoa é pai ou mãe de alguém. Em

caso afirmativo, essa informação é apagada porque o nome da pessoa será excluído. Por fim, a entrada com a pessoa é removida do dicionário.

A Figura 114 mostra os métodos **salvar()** e **ancestrais()** da classe **Arvore**. O método **salvar()** abre o arquivo em modo de escrita e grava o dicionário com a função **dumps()** da biblioteca **json**.

```
# ...

class Arvore:
    # ...

    def salvar(self):
        # Salva arquivo usando JSON
        with open(self._nome_arquivo, 'w') as arq:
            json.dump(self._arvore, arq, indent=2, ensure_ascii=False)

    def ancestrais(self, nome, nivel=0):
        # Busca pessoa
        pessoa = self.buscar(nome)
        if pessoa != '':
            # Imprime o nome da pessoa
            print('  '*nivel, nome, sep='')
            # Busca resursiva pelos ancestrais
            self.ancestrais(pessoa[PAI], nivel+1)
            self.ancestrais(pessoa[MAE], nivel+1)
```

Figura 114 – Métodos **salvar()** e **ancestrais()** da classe **Arvore**
Fonte: Elaborado pelo Autor.

O método **ancestrais()** utiliza recursão para mostrar todos os ancestrais de uma pessoa. O método tem o parâmetro opcional **nivel**, inicializado com 0 (zero), para escrever espaços proporcionalmente a cada geração de ancestrais anterior. Assim, os pais da pessoa aparecem com dois espaços de endentação, os avós com quatro espaços e assim por diante.

```
# ...

class Arvore:
    # ...

    def menu(self):
        print('\n'*5)
        linha()
        print('Ávore Genealógica')
        linha()
        self.listar()
        print('(+) | (-) | (P)ais | (A)ncestrais | (S)alvar | Sai(r)')
        return input('Informe a opção desejada: ').strip().lower()
```

Figura 115 – Método **menu()** da classe **Arvore**
Fonte: Elaborado pelo Autor.

A Figura 115 exibe o método **menu()** da classe **Árvore**. Esse método é responsável por exibir o menu de opções para o usuário e obter a resposta digitada.

```
# ...

    def executar(self):
        while True:
            resp = self.menu()
            if resp in ['-', '+', 'p', 'a', 'd']:
                nome = input('Informe o nome: ')
                if resp == '+':
                    self.incluir(nome)
                elif resp == '-':
                    self.excluir(nome)
                elif resp == 'p':
                    self.alterar_pais(nome)
                elif resp == 'a':
                    linha()
                    print('Árvore Genealógica:')
                    linha()
                    self.ancestrais(nome)
                    linha()
                    input()
                elif resp == 'd':
                    self.descendentes(nome)
                    input()
            elif resp == 's':
                self.salvar()
            elif resp == 'r':
                break


if __name__ == '__main__':
    arvore = Arvore('arvore.json')
    arvore.executar()
```

Figura 116 – Método **executar()** da classe **Arvore** e **if** de execução do arquivo
Fonte: Elaborado pelo Autor.

Para finalizar, a Figura 116 exibe o método **executar()** da classe **Arvore** e o **if** de execução do arquivo. O **if** de execução do arquivo cria um objeto da classe **Arvore** e chama se método **executar()**. O método, por sua vez, possui um laço de repetição para exibir o menu de opções e executar a opção desejada pelo usuário, até que a a opção de sair seja escolhida. Execute o código do projeto, faça alguns cadastro e observe o conteúdo do arquivo JSON.

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

3.6 Exercícios

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

A) Folha de pagamento

- Desenvolva um sistema de folha de pagamento com cadastro de funcionários e dependentes. Vamos guardar as informações de cada funcionário em um dicionário. Portanto, comece declarando variáveis textuais para serem as chaves desse dicionário. As informações a serem guardadas serão CPF, nome, tempo de serviço, lista de dependentes e salário;

- Crie a classe **Folha** com um construtor que recebe o nome do arquivo de dados. No construtor, inicialize os atributos **_nome_arquivo** (nome do arquivo de dados) e **_funcionarios** (lista de funcionários, inicialmente vazia). Ainda no construtor, leia o arquivo (no formado JSON) e carregue seu conteúdo para a lista **_funcionarios**;

- Implemente o método **buscar()** da classe para buscar funcionários pelo CPF. O método deve retornar a posição do funcionário dentro da lista **_funcionarios** ou -1 (menos um), se o CPF não existir;

- Escreva o método de classe **calcula_salario()** para calcular o salário de cada funcionário. O salário base dos funcionários é de R$2.000,00. O salário final de cada funcionário será o salário base acrescido de 1% para cada ano de tempo de serviço mais R$150,00 para cada dependente;

- Desenvolva o método de classe **pega_dados()** que será usado para obter os dados de um funcionário. De posse dos dados, esse método deve calcular o salário e retornar um dicionário com as informações obtidas;

- Crie o método **incluir()** que recebe o CPF de um funcionário, verifica se o mesmo é válido e faz a inclusão do funcionário. O CPF é válido se não for vazio (diferente de '') e não estiver previamente cadastrado. O método **buscar()** pode ser usado para verificar se um CPF já foi cadastrado. No caso de CPF válido, o método **pega_dados()** pode ser usado para obter os demais dados;

- Implemente o método de classe **mostar_funcionario()** para receber o dicionário de dados de um funcionário e mostrar suas informações na tela;

- Escreva o método **alterar()** que recebe um CPF e, caso o mesmo esteja cadastrado, altera os dados do funcionário existente. O método **buscar()** pode ser usado para verificar se o CPF já está cadastrado. A alteração consiste em mostrar os dados atuais do funcionário, obter os novos dados usando o método **pega_dados()** e mostrar os dados

após a alteração. O método **mostrar_funcionario()** pode ser usado para exibir os dados antigos e novos;

- Desenvolva os métodos **excluir()** e **listar()**. o Método **excluir()** recebe um CPF e excluir o funcionário associado ao CPF. O método **listar()** deve percorrer a lista de funcionários, mostrando as informações de cada um e o total de salários de todos os funcionários;

- Crie também os métodos **salvar()** e **menu()**. O método **salvar()** deve salvar a lista **_funcionarios** no arquivo JSON **_nome_arquivo**. O método **menu()** deve exibir o menu de opções e obter a opção desejada pelo usuário. As opções do sistema são incluir, excluir, alterar, salvar e sair.

- Para finalizar a classe **Folha**, implemente o método **executar()** contendo um laço de repetição que lista os funcionários e executa a opção desejada pelo usuário, até que a opção sair seja selecionada;

- Por fim, na execução do arquivo, crie um objeto da classe **Folha** e chame o método **executar()**.

B) Campeonato de futebol

- Desenvolva um sistema para controlar campeonatos de futebol. Os nomes dos times devem ser salvos em um arquivo de texto simples e os dados dos jogos devem ser salvos em um arquivo CSV;

- Crie a classe **Campeonato** com um construtor que receba os nomes dos arquivos, inicialize os atributos **_arquivo_times**, **_arquivo_jogos**, **_dic_times** e **_dic_jogos**. Os atributos **_dic_times** e **_dic_jogos** são dicionários para armazenar, respectivamente, os times e os jogos do campeonato. O dicionário de times tem como chave o nome do time o como valor os pontos que o time tem no campeonato. Já o dicionário de jogos tem como chave a tupla (mandante, visitante) e como valor a tupla (gols_mandante, gols_visitante);

- Implemente o método **soma_pontos()** para somar pontos aos times de acordo com o resultado de um jogo. O método recebe como parâmetro os nomes dos times mandante e visitante bem como os gols desses times no jogo. O time vencedor ganha três pontos. Em caso de empate, ambos times ganham um ponto. Os pontos ganhos devem ser somados ao valor do time no dicionário **_dic_times**;

- Escreva os métodos **incluir_time()** e **incluir_jogo()**. O método **incluir_time()** recebe um nome de time e, caso não exista, insere no dicionário **_dic_times** sem nenhum ponto inicialmente. O método **incluir_jogo()** recebe os nomes dos times mandante e visitante e seus gols no jogo. O resultado do jogo é incluído apenas se os times existirem no dicionário **_dic_times**. Nesse caso, o jogo é incluído no dicionário **_dic_jogos** e os pontos do jogo são contabilizados para os times;

- Desenvolva os métodos **listar_times()** e **listar_jogos()** para listar, respectivamente, os times e jogos já cadastrados;

- Crie os métodos **salvar()** e **menu()**. O método **salvar()** deve armazenar em arquivo os times e jogos do campeonato. Já o método **menu()** deve exibir o menu de opções para o usuário e obter a resposta selecionada. As opções são incluir time, incluir, jogo, listar jogos, salvar e sair;

- Implemente também o método **executar()** com um laço de repetição para listar os jogos, exibir o menu de opções e executar a opção selecionada pelo usuário. O laço de repetição é interrompido quando o usuário escolhe a opção sair;

- Para finalizar, na execução do código, crie um objeto da classe **Campeonato** e chame o método **executar()**.

3.7 Respostas dos exercícios

A) Folha de pagamento

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

import json
from os.path import isfile

SALARIO_BASE = 2000
CPF = 'CPF'
NOME = 'Nome'
TEMPO = 'Tempo'
DEPENDENTES = 'Dependentes'
SALARIO = 'Salário'


# Escreve linha na tela
def linha():
    print('-'*50)


class Folha:

    def __init__(self, nome_arquivo):
        # Arquivo e árvore de pessoas
        self._nome_arquivo = nome_arquivo
        self._funcionarios = []
        # Checa se o arquivo existe
        if isfile(self._nome_arquivo):
            # Abre arquivo e lê usando JSON
            with open(self._nome_arquivo) as arq:
                self._funcionarios = json.load(arq)

    def buscar(self, cpf):
        # Procura por CPF na lista de funcionarios
        for cont, func in enumerate(self._funcionarios):
            if cpf == func[CPF]:
                # Retorna posição dpo CPF encontrado
                return cont
        return -1

    @classmethod
    def calcula_salario(cls, tempo, dependentes):
        # Soma bônus de tempo de serviço (1% por ano)
        salario = SALARIO_BASE * (1 + tempo * 0.01)
        # Soma bônus de dependentes (5% por dependente)
        salario += len(dependentes) * 150
        return salario

    @classmethod
    def pega_dados(cls):
        # Pega nome e tempo de serviço
```

```
51          print('Informe os dados')
52          nome = input('Nome: ').strip()
53          tempo = int(input('Tempo de serviço: ').strip())
54          # Pega a lista de dependentes
55          lista_dependentes = []
56          while True:
57              cont = str(len(lista_dependentes) + 1)
58              mensagem = 'Dependente ' + cont + \
59                  ' (vazio para interromper): '
60              dependente = input(mensagem).strip()
61              if dependente == '':
62                  break
63              lista_dependentes.append(dependente)
64          # Cálcula salário
65          salario = cls.calcula_salario(tempo, lista_dependentes)
66          return {
67                  NOME: nome,
68                  TEMPO: tempo,
69                  DEPENDENTES: lista_dependentes,
70                  SALARIO: salario
71                  }
72
73      def incluir(self, cpf):
74          # Verifica se o CPF é válido
75          if cpf == '':
76              print('CPF inválido!')
77              input()
78          # Verifica se o CPF já existe
79          elif self.buscar(cpf) != -1:
80              print('CPF já cadastrado!')
81              input()
82          else:
83              # Pega os dados do novo funcionário
84              dados_dic = self.pega_dados()
85              # Acrescenta CPF
86              dados_dic[CPF] = cpf
87              # Adiciona na lista de funcionários
88              self._funcionarios.append(dados_dic)
89
90      def mostra_funcionario(cls, funcionario):
91          print('Funcionário:', funcionario[NOME],
92                '(' + funcionario[CPF] + ')', '|',
93                'Tempo de serviço:', funcionario[TEMPO])
94          if len(funcionario[DEPENDENTES]) > 0:
95              print('Dependentes:')
96              for dep in funcionario[DEPENDENTES]:
97                  print('-', dep)
98          print('Salário: {s:.2f}'.format(s=funcionario[SALARIO]))
99
100     def alterar(self, cpf):
101         posicao = self.buscar(cpf)
102         if posicao == -1:
103             print('CPF não encontrado!')
104             input()
105         else:
```

```
106                 print('Dados atuais: ')
107                 linha()
108                 self.mostra_funcionario(self._funcionarios[posicao])
109                 # Pega novos dados e substitui
110                 dados_dic = self.pega_dados()
111                 dados_dic[CPF] = cpf
112                 self._funcionarios[posicao] = dados_dic
113                 linha()
114                 print('Dados alterados:')
115                 self.mostra_funcionario(dados_dic)
116                 input()
117
118         def listar(self):
119             if len(self._funcionarios) == 0:
120                 print('Nenhum funcionário cadastrado!')
121                 linha()
122             else:
123                 # Ordena funcionários pelo nome
124                 self._funcionarios.sort(key=lambda d: d[NOME])
125                 total_salarios = 0
126                 # Percore lista de funcionários e mostra cada um
127                 for func in self._funcionarios:
128                     total_salarios += func[SALARIO]
129                     self.mostra_funcionario(func)
130                     linha()
131                 # Mostra o total de salários
132                 print('Total: {s:.2f}'.format(s=total_salarios))
133                 linha()
134
135         def excluir(self, cpf):
136             # Procura por CPF para escluir
137             posicao = self.buscar(cpf)
138             # Se existir remove
139             if posicao != -1:
140                 self._funcionarios.pop(posicao)
141
142         def salvar(self):
143             # Salva arquivo usando JSON
144             with open(self._nome_arquivo, 'w') as arq:
145                 json.dump(self._funcionarios, arq, indent=2,
146                           ensure_ascii=False)
147
148         def menu(self):
149             print('\n'*5)
150             linha()
151             print('Folha de pagamento')
152             linha()
153             self.listar()
154             print('(+) | (-) | (A)lterar | (S)alvar | Sai(r)')
155             return input('Informe a opção desejada: ').strip().lower()
156
157         def executar(self):
158             while True:
159                 resp = self.menu()
160                 if resp in ['-', '+', 'a']:
```

```
161                     cpf = input('Informe o CPF: ')
162                     if resp == '+':
163                         self.incluir(cpf)
164                     elif resp == '-':
165                         self.excluir(cpf)
166                     elif resp == 'a':
167                         self.alterar(cpf)
168                 elif resp == 's':
169                     self.salvar()
170                 elif resp == 'r':
171                     break
172 
173 
174 if __name__ == '__main__':
175     folha = Folha('folha.json')
176     folha.executar()
```

B) Campeonato de futebol

```
 1 #!/usr/bin/env python3
 2 # -*- coding: utf-8 -*-
 3 
 4 import csv
 5 from os.path import isfile
 6 
 7 
 8 # Escreve linha na tela
 9 def linha():
10     print('-'*50)
11 
12 
13 class Campeonato:
14 
15     def __init__(self, arquivo_times, arquivo_jogos):
16         # Nomes dos arquivos
17         self._arquivo_times = arquivo_times
18         self._arquivo_jogos = arquivo_jogos
19         # Dicionários de times e jogos
20         self._dic_times = {}
21         self._dic_jogos = {}
22         # Leitura dos arquivos
23         self.le_arquivos()
24 
25     def soma_pontos(self, time_m, time_v, gols_m, gols_v):
26         # Testa se número de gols do mandante é maior que visitante
27         if gols_m > gols_v:
28             # Mais três pontos para o mandante
29             self._dic_times[time_m] += 3
30         # Testa se número de gols do visitante é maior que mandante
31         elif gols_v > gols_m:
32             # Mais três pontos para o visitante
33             self._dic_times[time_v] += 3
34         else:
35             # Em caso de empante, mais um ponto para cada
36             self._dic_times[time_m] += 1
37             self._dic_times[time_v] += 1
```

```
38
39      def incluir_time(self, time):
40          # Verifica se o time não existe
41          if time not in self._dic_times:
42              # Inclui time
43              self._dic_times[time] = 0
44
45      def incluir_jogo(self, time_m, time_v, gols_m, gols_v):
46          # Verifica se os dois times exitem
47          if time_m in self._dic_times and time_v in self._dic_times:
48              # Adiciona o jogo
49              self._dic_jogos[(time_m, time_v)] = (gols_m, gols_v)
50              # Soma os pontos
51              self.soma_pontos(time_m, time_v, gols_m, gols_v)
52
53      def le_arquivos(self):
54          # Checa se o arquivo existe
55          if isfile(self._arquivo_times):
56              with open(self._arquivo_times) as arq:
57                  # Lê um time por linha
58                  for linha in arq:
59                      time = linha.strip()
60                      # Time inicia sem pontos no campeonato
61                      self._dic_times[time] = 0
62          if isfile(self._arquivo_jogos):
63              with open(self._arquivo_jogos) as arq:
64                  # Arquivo de jogos no formato CSV
65                  leitor = csv.reader(arq)
66                  # Ignora cabeçalho
67                  next(leitor)
68                  # Percorre linhas do arquivi
69                  for linha in leitor:
70                      # Obtém valores da linha
71                      # Times mandante e visitante com seus gols
72                      time_m, time_v, gols_m, gols_v = tuple(linha)
73                      # Incluir jogo no campeonato
74                      self.incluir_jogo(time_m, time_v,
75                                        gols_m, gols_v)
76
77      def listar_times(self):
78          # Verifica se não há times cadastrados
79          if len(self._dic_times) == 0:
80              print('Nenhum time cadastrado!')
81              linha()
82          else:
83              # Obtém lista de times
84              lista_times = list(self._dic_times.items())
85              # Ordena times decrescentemente pelo número de pontos
86              lista_times.sort(key=lambda t: t[1], reverse=True)
87              # Mostra cada time com seus pontos
88              for time, pontos in lista_times:
89                  print(time, ':', pontos)
90
91      def listar_jogos(self):
92          # Verifica se não existem jogos
```

```
 93            if len(self._dic_jogos) == 0:
 94                print('Nenhum jogo cadastrado!')
 95                linha()
 96            else:
 97                linha()
 98                print('Jogos do campeonato')
 99                # Percorre dicionário de jogos
100                for jogo, placar in self._dic_jogos.items():
101                    time_m, time_v = jogo
102                    gols_m, gols_v = placar
103                    print(time_m, gols_m, 'x', gols_v, time_v)
104                linha()
105            input()
106
107        def salvar(self):
108            # Salva arquivo de times
109            with open(self._arquivo_times, 'w') as arq:
110                for time in self._dic_times.keys():
111                    arq.write(time + '\n')
112            # Salva arquivo de jogos
113            with open(self._arquivo_jogos, 'w') as arq:
114                escritor = csv.writer(arq)
115                escritor.writerow(['Mandante', 'Visitante',
116                                   'Gols M', 'Gols V'])
117                for jogo, placar in self._dic_jogos.items():
118                    escritor.writerow(list(jogo) + list(placar))
119
120        def menu(self):
121            linha()
122            print('Campeonato')
123            linha()
124            self.listar_times()
125            print('+(T)ime', '+(J)ogo', '(L)istar jogos',
126                  '(S)alvar', ' Sai(r)', sep=' | ')
127            return input('Informe a opção desejada: ').strip().lower()
128
129        def executar(self):
130            while True:
131                resp = self.menu()
132                if resp == 't':
133                    time = input('Informe o time: ').strip()
134                    self.incluir_time(time)
135                elif resp == 'j':
136                    print('Informe os dados do jogo')
137                    time_m = input('Mandante: ').strip()
138                    time_v = input('Visitante: ').strip()
139                    gols_m = int(input('Gols mandante: ').strip())
140                    gols_v = int(input('Gols visitante: ').strip())
141                    self.incluir_jogo(time_m, time_v, gols_m, gols_v)
142                elif resp == 'l':
143                    self.listar_jogos()
144                elif resp == 's':
145                    self.salvar()
146                elif resp == 'r':
147                    break
```

```
148
149
150  if __name__ == '__main__':
151      compeonato = Campeonato('times.txt', 'jogos.csv')
152      compeonato.executar()
```

3.8 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 Terceira Semana” para recapitular tudo que aprendemos.

Nos encontramos na próxima semana.

Bons estudos!

Objetivos
- Conhecer os conceitos básicos de interface gráfica;
- Entender o funcionamento de eventos relacionados com componentes;
- Desenvolver códigos com interfaces gráficas.

4.1 Introdução

Até o momento, desenvolvemos apenas programas de linha de comando usando uma comunicação textual com o usuário. Além dos programas de linha de comando, é possível desenvolver aplicações com interfaces gráficas. As aplicações com interfaces gráficas possuem diversos componentes visuais para capturar ou mostrar informações ao usuário. Alguns exemplos de componentes são botões, caixas de texto e listas de seleção (BORGES, 2010). Existem várias bibliotecas disponíveis para criação de interface gráfica. Nesse livro, focaremos na criação de interfaces em Python usando a biblioteca PyQt[5].

4.2 Criação manual de interfaces

Uma das maneiras de criar interfaces é de forma manual especificando detalhadamente como cada componente deve ser desenhado. A Figura 117 apresenta um exemplo de criação de interface manual contendo uma janela e um rótulo com texto. Ao executar esse código será exibida a janela mostrada na Figura 118.

```
from PyQt5 import QtWidgets
from PyQt5.QtWidgets import QApplication, QMainWindow

# Cria uma aplicação com a bibliteca Qt
app = QApplication([])
# Cria uma janela principal
win = QMainWindow()
# Definie posição (0, 0) e dimensões da janela (500x300)
win.setGeometry(0, 0, 500, 300)
# Definie o título da janela
win.setWindowTitle('Exemplo com Qt')
# Cria um componente de rótulo
label = QtWidgets.QLabel(win)
# Define o texto do componente
label.setText('Olá, Mundo!')
# Posiciona o componente na posição 100, 100
label.move(100, 100)
# Mostra a janela
win.show()
# Executa a aplicação
app.exec_()
```

Figura 117 – Criação de janela manualmente com a biblioteca PyQt
Fonte: Elaborado pelo Autor.

5 https://www.riverbankcomputing.com/software/pyqt/

Figura 118 – Janela criada com a biblioteca Qt
Fonte: Elaborado pelo Autor.

4.3 Qt Designer

A criação de interfaces de forma manual é uma tarefa exaustiva. Dessa forma, foram desenvolvidas ferramentas para auxiliar nessa tarefa. No caso da biblioteca Qt, uma alternativa interessante é o Qt Designer. O Qt Designer é uma ferramenta livre e disponível para vários sistemas operacionais. No Linux, ele pode ser instalado por meio do pacote **qttools5-dev-tools**. Para Windows e Mac, existem instalações disponíveis para download[6]. A Figura 119 exibe a tela principal do Qt Designer.

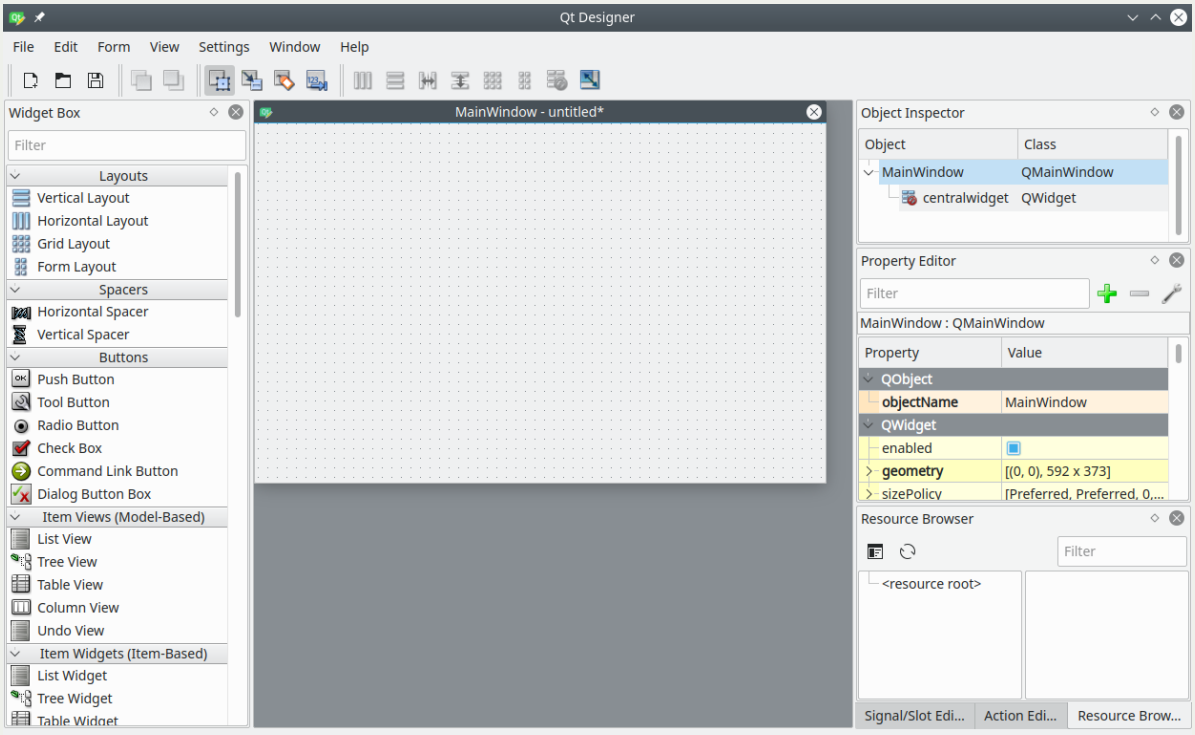

Figura 119 – Tela principal do Qt Designer
Fonte: Elaborado pelo Autor.

6 https://build-system.fman.io/qt-designer-download

No centro da janela fica a interface. Do lado esquerdo temos o painel de componentes que podem ser inseridos na interface. Para inserir um componente, basta arrastá-lo e soltar sobre a interface. O painel da direita contém o *Object Inspector* e o *Property Editor*. O *Object Inspector* exibe a hierarquia de componentes da interface. Já o *Property Editor* exibe as propriedades do componente selecionado.

Quando abrimos o Qt Designer pela primeira vez ou quando clicamos do botão *New*, a ferramenta exibe tela de seleção de tipo de interface mostrada na Figura 120. Para nossos exemplos, vamos escolher a opção *Main Window*. Essa opção cria uma janela de interface com uma barra de menu e uma barra de status que podem ser removidas caso não sejam necessárias. Para removê-las você pode clicar com o botão direto do mouse e escolher a opção *Remove Menu Bar* ou *Remove Status Bar*.

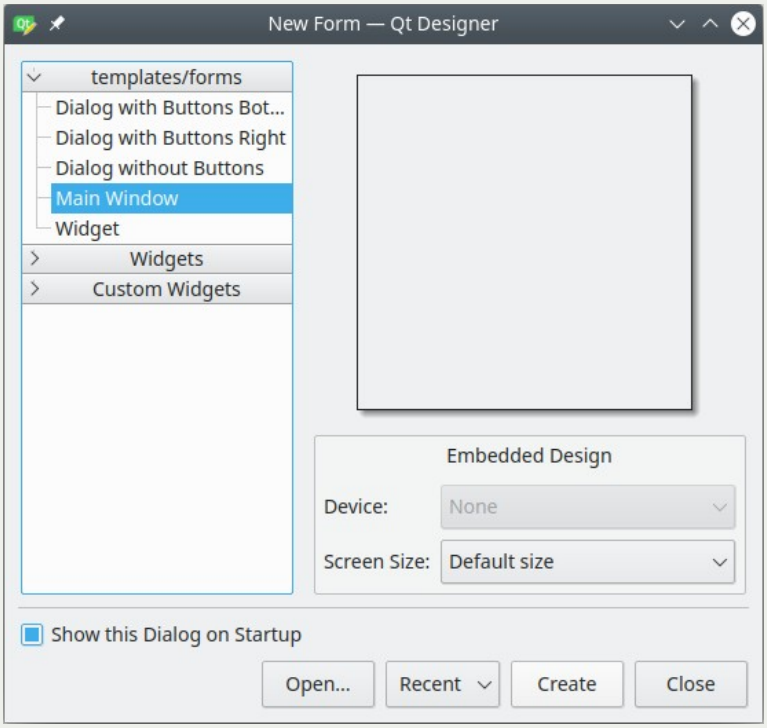

Figura 120 – Tela *New Form* do Qt Designer
Fonte: Elaborado pelo Autor.

A biblioteca Qt possui uma grande quantidade de componentes. Contudo, muito raramente todos serão usados em um único projeto. Os principais componentes para construção de interfaces são os seguintes:

- **PushButton** (botão para receber cliques);

- **RadioButton** (componente usado em grupo para o usuário escolher uma única opção);

- **CheckBox** (caixa de marcação para o usuário marcar ou desmarcar);

- **ListWidget** (lista de item para o usuário selecionar);

- **TableWidget** (tabela com linhas e colunas);

- **GroupBox** (agrupamento de componentes);

- **ComboBox** (caixa de seleção de itens);

- **LineEdit** (caixa de texto);

- **TextEdit** (caixa de texto com múltiplas linhas);

- **SpinBox** (caixa de texto para entrada de números);

- **Label** (rótulo para exibição de texto).

4.4 Usando interfaces do Qt Designer em códigos Python

As interfaces criadas no Qt Designer são salvas em arquivos com extensão **ui**. Esses arquivos armazenam todos os detalhes da interface no formato XML. Assim, podemos criar as interfaces no Qt Designer e desenvolver códigos que carregam os aquivos **ui** durante a execução. No código, podemos associar os eventos dos componentes à métodos ou funções que serão automaticamente executados quando o evento ocorrer. Os eventos são as possíveis interações do usuário com a interface como o clique do mouse ou o pressionamento de uma tecla.

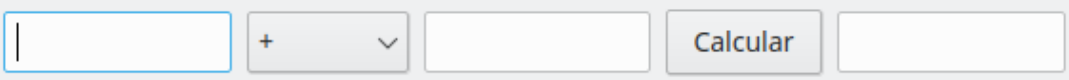

Figura 121 – Interface de calculadora simples
Fonte: Elaborado pelo Autor.

Como primeiro projeto no Qt Designer vamos criar a interface de uma calculadora simples como mostrado na Figura 121. Vamos salvar o arquivo como **calculadora.ui**. A construção da interface consistiu em inserir os componentes e alterar algumas propriedades. Mudamos os nomes dos componentes porque vamos referenciá-los no código. Portanto, é importante usar nomes significativos para facilitar a codificação.

Os componentes **LineEdits** foram renomeados como **edit_n1** (primeiro), **edit_n2** (segundo) e **edit_res** (terceiro). Os nomes dos componentes podem ser alterados com um clique duplo no *Object Inspector* ou na propriedade **objectName** do *Property Editor*. Os dois primeiros **LineEdits** receberão números digitados pelo usuário e o terceiro mostrará o resultado da operação quando o botão for pressionado. No componente **edit_res**, marcamos também a propriedade **readOnly** para que o componente seja somente leitura.

O **ComboBox** foi renomeado para **combo_op**. Através do clique com botão direto do mouse sobre o componente podemos acessar o menu *Edit Items...* para inserir os itens para o usuário selecionar. No caso do botão, clicamos duplo sobre ele para mudar seu texto e o renomeamos como **button_calc**. A janela (**QMainWindow**) foi renomeada para **window_calc** e colocamos o título "Calculadora" na sua propriedade **windowTitle**.

O conteúdo completo do arquivo é mostrado nas Figura 122 e Figura 123. Esse conteúdo foi mostrado apenas a título de curiosidade. Não é necessário editá-lo porque o Qt Designer gera esse conteúdo automaticamente.

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
 <class>window_calc</class>
 <widget class="QMainWindow" name="window_calc">
  <property name="geometry">
   <rect>
    <x>0</x>
    <y>0</y>
    <width>572</width>
    <height>49</height>
   </rect>
  </property>
  <property name="windowTitle">
   <string>Calculadora</string>
  </property>
  <widget class="QWidget" name="centralwidget">
   <widget class="QLineEdit" name="edit_n1">
    <property name="geometry">
     <rect>
      <x>6</x>
      <y>7</y>
      <width>122</width>
      <height>34</height>
     </rect>
    </property>
   </widget>
   <widget class="QLineEdit" name="edit_n2">
    <property name="geometry">
     <rect>
      <x>227</x>
      <y>7</y>
      <width>121</width>
      <height>34</height>
     </rect>
    </property>
   </widget>
   <widget class="QComboBox" name="combo_op">
    <property name="geometry">
     <rect>
      <x>134</x>
      <y>7</y>
      <width>87</width>
      <height>34</height>
     </rect>
    </property>
    <item>
     <property name="text">
      <string>+</string>
     </property>
    </item>
    <item>
```

Figura 122 – Conteúdo do arquivo **calculadora.ui** (parte 1)
Fonte: Elaborado pelo Autor.

```
 52        <property name="text">
 53         <string>-</string>
 54        </property>
 55       </item>
 56       <item>
 57        <property name="text">
 58         <string>*</string>
 59        </property>
 60       </item>
 61       <item>
 62        <property name="text">
 63         <string>/</string>
 64        </property>
 65       </item>
 66       <item>
 67        <property name="text">
 68         <string>**</string>
 69        </property>
 70       </item>
 71      </widget>
 72      <widget class="QPushButton" name="button_calc">
 73       <property name="geometry">
 74        <rect>
 75         <x>354</x>
 76         <y>6</y>
 77         <width>84</width>
 78         <height>36</height>
 79        </rect>
 80       </property>
 81       <property name="text">
 82        <string>Calcular</string>
 83       </property>
 84      </widget>
 85      <widget class="QLineEdit" name="lineEdit_3">
 86       <property name="geometry">
 87        <rect>
 88         <x>444</x>
 89         <y>7</y>
 90         <width>122</width>
 91         <height>34</height>
 92        </rect>
 93       </property>
 94       <property name="readOnly">
 95        <bool>true</bool>
 96       </property>
 97      </widget>
 98     </widget>
 99    </widget>
100    <resources/>
101    <connections/>
102   </ui>
```

Figura 123 – Conteúdo do arquivo **calculadora.ui** (parte 2)
Fonte: Elaborado pelo Autor.

Após a criação da interface, vamos criar o arquivo **calculadora.py** na mesma pasta do arquivo **calculadora.ui** e escrever o código como mostrado na Figura 124.

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

from PyQt5 import uic
from PyQt5.QtWidgets import QApplication, QMainWindow


# Classe herdando de QMainWindow
class CalculadoraUI(QMainWindow):

    def __init__(self):
        super().__init__()
        # Carrega o arquivo ui com a interface
        uic.loadUi('calculadora.ui', self)
        # Conecta clique do botão com o método calcula
        self.button_calc.clicked.connect(self.calcula)

    def calcula(self):
        # Obtém números digitados
        n1 = float(self.edit_n1.text())
        n2 = float(self.edit_n2.text())
        # Obtém operação selecionada
        op = self.combo_op.currentText()
        # Calcula de acordo com a operação
        if op == '+':
            res = n1 + n2
        elif op == '-':
            res = n1 - n2
        elif op == '*':
            res = n1 * n2
        elif op == '/':
            res = n1 / n2
        else:
            res = n1 ** n2
        # Mostra o resultado
        self.edit_res.setText(str(res))


if __name__ == '__main__':
    # Cria aplicação e a janela
    app = QApplication([])
    window = CalculadoraUI()
    # Exibe janela e executa a aplicação
    window.show()
    app.exec_()
```

Figura 124 – Código do arquivo **calculadora.py**
Fonte: Elaborado pelo Autor.

A criação da interfase no código é feita com a classe **CalculadoraUI** que herda da classe **QMainWindow**. No construtor usamos a função **loadUi** da biblioteca **uic** para carregar a interface do arquivo **calculadora.ui** para a interface atual (**self**). Ainda no

construtor, ligamos o evento de clique do botão (**clicked**) ao método **calcula()**. Assim, quando o usuário clicar no botão, o método **calcula()** será executado automaticamente.

No método **calcula()**, usamos o método **text()** para ler o conteúdo digitando nos **LineEdits edit_n1** e **edit_n2**. Utilizamos também o método **currentText()** do **combo_op** para obter a operação selecionada pelo usuário. Em seguida, de acordo com a operação escolhida, calculamos o resultado. Por último, chamamos o método **setText()** para exibir o resultado no **edit_res**. Na execução do código, criamos a aplicação (**app**) e a janela (**window**), exibimos a janela com o método **show()** e executamos a aplicação com o método **exec_()**.

4.5 Jogo da vela com interface gráfica

Nos capítulos anteriores já desenvolvemos um jogo da velha textual onde os jogadores precisavam informar a linha a coluna de suas jogadas. Agora, vamos criar esse jogo com interface gráfica para que os jogadores possam clicar nas posições desejadas do tabuleiro.

Nesse projeto, usamos um *layout* para posicionar melhor os componentes. Os layouts são configurações de disposição dos componentes para facilitar o desenho da interface e manter um proporção em caso de redimensionamento. A configuração do *layout* da janela e de *containers* pode ser feita com os botões de layout disponíveis na barra de ferramentas do Qt Designer. Os *containers* são componentes que podem conter outros componentes como é o caso do **GroupBox**.

Os principais *layouts* disponíveis nas interfaces Qt são horizontal, vertical, *grid* e *form*. O *layout* horizontal divide o *container* em colunas e os componentes internos são espalhados na horizontal. No layout vertical os componentes internos do *container* são dispostos na forma de linhas. O layout *grid* organiza o *container* como uma tabela e os componentes são ajustados preenchendo as linhas e colunas dessa tabela. Por fim o layout de *form* divide o *container* em duas com lunas com várias linhas sendo útil para formulários com rótulos e seus respectivos campos.

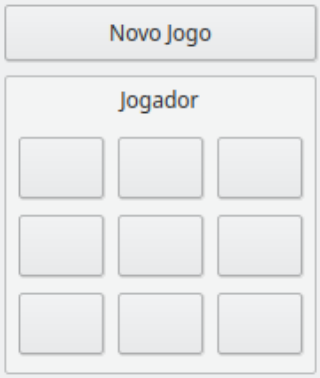

Figura 125 – Interface do jogo da velha
Fonte: Elaborado pelo Autor.

A Figura 125 mostra a interface do jogo criada no Qt Designer contendo um **PushButton** de novo jogo no topo (**button_novo**) e um **GroupBox** na parte inferior para

representar o tabuleiro (**group_jogo**). O *layout* da janela foi definido como vertical e o *layout* do **GroupBox** foi definido como *grid*.

Os botões do **GroupBox** receberam o nome **button_lc** onde com **l** e **c** representando a linha e a coluna do botão no tabuleiro, respectivamente. Os valores de **l** e **c** variam de 0 a 3. Isso facilitará mapear o clique do botão para a posição correspondente do tabuleiro. Outro foi a propriedade **maximumSize** desses botões que foi definida como **36x36** para que os mesmos tenham sempre um tamanho fixo.

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

from PyQt5 import uic
from PyQt5.QtWidgets import QApplication, QMainWindow, QPushButton,
QMessageBox
from random import random


class Velha:

    def __init__(self):
        # Inicializa tabuleiro
        self._tabuleiro = [
                [' ', ' ', ' '],
                [' ', ' ', ' '],
                [' ', ' ', ' ']]
        # Sorteia jogador
        if random() >= 0.5:
            self._jogador = 'X'
        else:
            self._jogador = 'O'

    def jogada(self, linha, coluna):
        # Verifica se a posição está vazia
        if self._tabuleiro[linha][coluna] == ' ':
            # Marca a posição com o simbolo do jogador
            self._tabuleiro[linha][coluna] = self._jogador
            return True
        return False

    def troca_jogador(self):
        # Troca o jogador
        if self._jogador == 'X':
            self._jogador = 'O'
        else:
            self._jogador = 'X'

    @property
    def eh_vencedor(self):
        # Testa se um jogador é vencedor
        linhas = self.todas_linhas()
        # O Jogador é vencedor ter tiver uma linha com 3 posições
        if tuple([self._jogador]*3) in linhas:
            return True
        return False
```

Figura 126 – Código inicial do arquivo **velha.py** com a classe **Velha**
Fonte: Elaborado pelo Autor.

A Figura 126 mostra o código inicial do nosso projeto com a classe **Velha**. O código é muito parecido com o projeto de jogo da velha desenvolvido nos capítulos anteriores. Porém, a classe atual apenas implementa o funcionamento da lógica do jogo, pois a exibição será feita com a interface gráfica. Na Figura 126 podemos ver construtor e os métodos **jogada()**, **troca_jogador()** e **eh_vencedor()**. O método **jogada()** retorna **True**

quando uma jogada é possível e **False** quando a jogada não é válida. Além disso, se a jogada for possível o método marca a posição da jogado com o símbolo do jogador atual. O método **troca_jogador()** troca o jogador atual e o método é vencedor verifica se o jogador atual já venceu o jogo.

```
# ...

class Velha:
    # ...

    @property
    def jogador(self):
        return self._jogador

    def tem_jogada(self):
        # Varre o tabuleiro procurando por posições vazias
        for linha in self._tabuleiro:
            if ' ' in linha:
                return True
        return False

    def todas_linhas(self):
        # Retorna todas as linhas possiveis em formato de tuplas
        # Linhas, colunas, diagonal e transversal
        todas = []
        # Linhas
        for linha in self._tabuleiro:
            todas.append(tuple(linha))
        # Colunas
        for cont in range(3):
            coluna = (self._tabuleiro[0][cont],
                      self._tabuleiro[1][cont],
                      self._tabuleiro[2][cont])
            todas.append(coluna)
        # Diagonal e transversal
        diagonal = []
        transversal = []
        for cont in range(3):
            diagonal.append(self._tabuleiro[cont][cont])
            transversal.append(self._tabuleiro[2 - cont][cont])
        todas.append(tuple(diagonal))
        todas.append(tuple(transversal))
        return todas
```

Figura 127 – Métodos **jogador()**, **tem_jogada()** e **todas_linhas()** da classe **Velha**
Fonte: Elaborado pelo Autor.

A Figura 127 mostra os demais métodos da classe Velha. O método **jogador()** serve para retornar o jogador atual. Já o método **tem_jogada()** retorna verdadeiro se ainda houverem posições vazias no tabuleiro. Por fim, o método **todas_linhas()** retorna todas as linhas possíveis do tabuleiro incluindo linhas horizontais, verticais, diagonal e transversal.

A Figura 128 mostra o código da classe **VelhaUI** com seu construtor e o método **marca()**. O construtor carrega a interface criada no Qt Designer (**velha.ui**), conecta os eventos dos botões e inicia o jogo. O clique do botão **button_novo** é associado ao método

novo(). Usamos o método **findChildren()** do **group_jogo** para percorrer todos os botões do tabuleiro e atribuir o método **marca()** ao evento de clique. No final, o método **novo()** é chamado para iniciar um novo jogo.

```
# ...

class Velha:
    # ...

# Classe herdando de QMainWindow
class VelhaUI(QMainWindow):

    def __init__(self):
        super().__init__()
        # Carrega o arquivo ui com a interface
        uic.loadUi('velha.ui', self)
        self.button_novo.clicked.connect(self.novo)
        for button in self.group_jogo.findChildren(QPushButton):
            button.clicked.connect(self.marca)
        self.novo()

    def marca(self):
        botao = self.sender()
        nome_botao = botao.objectName()
        linha = int(nome_botao[-2])
        coluna = int(nome_botao[-1])
        if self._jogo.jogada(linha, coluna):
            botao.setText(self._jogo.jogador)
            if self._jogo.eh_vencedor:
                msg = QMessageBox()
                msg.setText('Fim de jogo! \n O jogador ' +
                            self._jogo.jogador + ' venceu.')
                msg.exec()
                self.group_jogo.setDisabled(True)
            self._jogo.troca_jogador()
            self.group_jogo.setTitle('Jogador: ' + self._jogo.jogador)
            if not self._jogo.tem_jogada():
                msg = QMessageBox()
                msg.setText('Jogo empatado!')
                msg.exec()
                self.group_jogo.setDisabled(True)
```

Figura 128 – Classe **VelhaUI** com construtor e método **marca()**
Fonte: Elaborado pelo Autor.

Como o método **marca()** foi conectado ao clique de todos os botões do tabuleiro, usamos o método **sender()** da classe **QMainWindow** para identificar qual botão disparou o evento. Depois, pegamos a linha e a coluna do botão através seu próprio nome. Lembramos que o nome dos botões é **button_lc** onde com **l** e **c** representam a linha e a coluna do botão no tabuleiro.

De posse da linha e da coluna, fazemos a jogada no objeto **_jogo** da classe **Velha**. Se a jogada for válida, atribuímos o símbolo do jogador ao texto do botão pressionado. Em seguida, verificamos se o jogador venceu o jogo. Em caso afirmativo, mostramos uma mensagem de vencedor e desabilitamos o **GroupBox** do tabuleiro.

Se o jogador ainda não tiver vencido o jogo, trocamos de jogador e mostramos essa informação no título do **GroupBox**. No final, se não houverem mais jogadas, informamos que o jogo terminou empatado e desabilitamos o tabuleiro.

```
# ...

class Velha:
    # ...

class VelhaUI(QMainWindow):
    # ...

    def novo(self):
        self._jogo = Velha()
        self.group_jogo.setTitle('Jogador: ' + self._jogo.jogador)
        self.group_jogo.setEnabled(True)
        for button in self.group_jogo.findChildren(QPushButton):
            button.setText('')


if __name__ == '__main__':
    # Cria aplicação e a janela
    app = QApplication([])
    window = VelhaUI()
    # Exibe janela e executa a aplicação
    window.show()
    app.exec_()
```

Figura 129 – Métodos **novo()** da classe **Velha** e **if** de execução do arquivo
Fonte: Elaborado pelo Autor.

A Figura 129 apresenta o método **novo()** da classe **VelhaUI** e o **if** de execução do arquivo. O método **novo()** inicializa o objeto **_jogo** com a classe **Velha** e faz as devidas alterações nos componentes da interface. O **group_jogo** é habilitado e seu título mostra o jogador que começa o jogo. No final, o método limpa o texto dos botões que representam o tabuleiro para começar o jogo. O **if** de execução do arquivo apenas cria e exibe a interface.

4.6 Controle de gastos de veículos

Outro projeto que vamos desenvolver será um programa de controle de gastos de veículos. A interface do programa é apresentada pela Figura 130. Usamos o layout *vertical* na janela e o layout de *grid* para os componentes de **GroupBox**. Dentro do **GroupBox** superior incluímos um **ComboBox** para listar os veículos (**combo_veiculo**) e dois botões para incluir (**button_incluir_veiculo**) e excluir os veículos (**button_excluir_veiculo**). Para que o **ComboBox** possa permitir tanto a seleção como inclusão de novos veículos, marcamos sua propriedade **editable**.

Na parte inferior, incluímos um **TableWidget** para listar todos os gastos do veículo selecionado (**table_gastos**), um rótulo e um **LineEdit** para exibir o total de gastos (**edit_total**). Mas abaixo, incluímos os botões para incluir um gasto (**button_incluir_gasto**), excluir um gasto selecionado (**button_excluir_gasto**) e outro para salvar os gastos da tabela para o veículo selecionado (**button_salvar_gastos**). A personalização do **TableWidget** envolveu a criação das

colunas através do menu *Edit items...* disponível no clique do botão direito sobre o componente. Para o **edit_total**, modificamos apenas a propriedade **readOnly** para verdadeiro.

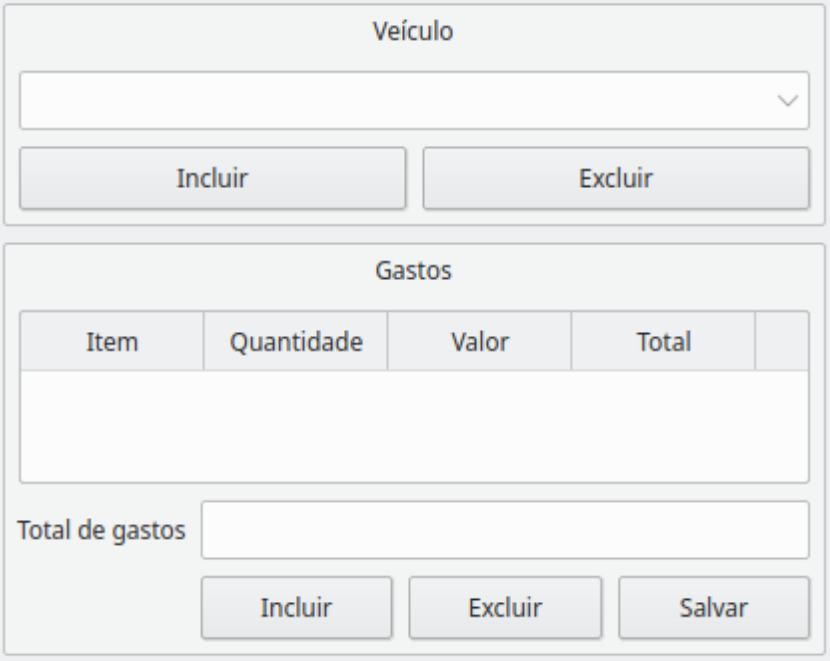

Figura 130 – Interface do programa de controle de gastos de veículos
Fonte: Elaborado pelo Autor.

Após a criação da interface, vamos escrever o código começando pelas importações e algumas variáveis que serão usadas em dicionários para salvar os gastos dos veículos. A Figura 131 mostra o código inicial com as importações e as variáveis que serão chaves de dicionário.

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

import json
from os.path import isfile
from PyQt5 import uic
from PyQt5.QtWidgets import QApplication, QMainWindow, \
    QMessageBox, QHeaderView, QTableWidget, QTableWidgetItem

GASTO = 'gasto'
QUANT = 'quantidade'
VALOR = 'valor'
```

Figura 131 – Métodos **novo()** da classe **Velha** e **if** de execução do arquivo
Fonte: Elaborado pelo Autor.

A primeira classe a ser criada é a classe **CadastroVeiculos** cujo código inicial é mostrado na Figura 132. Essa classe ficará responsável pelo controle dos dados dos veículos e seus respectivos gastos. O construtor da classe recebe o nome do arquivo de dados, inicializa os atributos e carrega dos dados do arquivo. O atributo **_dic_veiculos** é um dicionário para guardar os veículos e seus gastos.

O método **incluir_veiculo()** recebe o nome do veículo, verifica sua validade e faz a inclusão no dicionário. A chave do dicionário é o nome do veículo e o valor, inicialmente, é

uma lista vazia. Posteriormente, essa lista será preenchida com os gastos do veículo. O método **exclui_veiculo()** recebe o nome do veículo e, caso o nome exista, o remove do dicionário.

```
# ...

# Classe para manter o cadastro de veículos
class CadastroVeiculos:

    def __init__(self, nome_arquivo):
        # Inicializa atributos
        self._nome_arquivo = nome_arquivo
        self._dic_veiculos = {}
        # Lê arquivo JSON
        if isfile(self._nome_arquivo):
            with open(self._nome_arquivo) as arq:
                self._dic_veiculos = json.load(arq)

    def inclui_veiculo(self, veiculo):
        # Verifica se o novo veículo é válido (inexistente e não vazio)
        if veiculo not in self._dic_veiculos and veiculo != '':
            # Inclui veículo
            self._dic_veiculos[veiculo] = []

    def exclui_veiculo(self, veiculo):
        # Verifica se a veiculo existe
        if veiculo in self._dic_veiculos:
            # Exclui o veículo
            self._dic_veiculos.pop(veiculo)

    def lista_veiculos(self):
        # Retorna a lista de veículos
        return list(self._dic_veiculos.keys())
```

Figura 132 – Classe **CadastroVeiculos** com construtor e métodos iniciais
Fonte: Elaborado pelo Autor.

O método **lista_veiculos()** retorna uma lista com os nomes dos veículos cadastrados. A Figura 133 apresenta o código restante da classe **CadastroVeiculos**. Os métodos **inclui_gasto()**, **limpa_gastos()** e **lista_gastos()** sempre verificam se o veículo existe antes de executar suas ações. O método **inclui_gasto()** anexa um novo gasto ao veículo informado, o método **limpa_gastos()** remove todos os gastos de um veículo e o método **lista_gastos()** retorna uma lista com os gastos do veículo. Por fim, o método **salva()** armazena em arquivo o dicionário de veículo com seus gastos.

A Figura 134 mostra o código da classe **VeiculoUI** ligada a interface do programa. No construtor, carregador o arquivo da interface e configuramos seus componentes. No caso da tabela, modificamos seu cabeçalho (**header**) definindo a primeira coluna como autoajustável (**QHeaderView.Stretch**) com o método **setSectionResizeMode()**.

Na configuração dos eventos dos componentes, conectamos o evento de seleção da tabela (**itemSelectionChanged**) ao método **seleciona()** e o evento de alteração das células (**cellChanged**) ao método **altera_gasto()**. Para o componente **combo_veiculo**, ligamos o evento de alteração do veículo selecionado (**currentIndexChanged**) ao método

escolhe_veiculo(). No caso dos botões, conectamos os eventos de clique aos métodos apropriados.

```
# ...
class CadastroVeiculos:
    # ...

    def inclui_gasto(self, veiculo, gasto):
        # Verifica se o veículo existe
        if veiculo in self._dic_veiculos:
            # Obtém a lista de gastos do veículo
            lista_gastos = self._dic_veiculos[veiculo]
            # Adiciona o novo gasto
            lista_gastos.append(gasto)

    def limpa_gastos(self, veiculo):
        # Verifica se o veículo existe
        if veiculo in self._dic_veiculos:
            # Obtém e limpa a lista de gastos
            lista_gastos = self._dic_veiculos[veiculo]
            lista_gastos.clear()

    def lista_gastos(self, veiculo):
        # Verifica se o veículo existe
        if veiculo in self._dic_veiculos:
            # Retorna a lista de gastos do veículo
            return self._dic_veiculos[veiculo]
        return []

    def salva(self):
        with open(self._nome_arquivo, 'w') as arq:
            json.dump(self._dic_veiculos, arq, indent=2,
                      ensure_ascii=False)
```

Figura 133 – Demais métodos da classe **CadastroVeiculos**
Fonte: Elaborado pelo Autor.

Após a configuração dos eventos, criamos o atributo **_veiculos** com a classe **CadastroVeiculos** e chamamos o método **_atualiza_lista()**. O método **_atualiza_lista()** limpa os dados do **combo_veiculo** e insere a lista de veículos extraída do atributo **_veiculos**.

A Figura 134 apresenta também o método de classe **pergunta()** que, por sua vez, utiliza a classe **QMessageBox** para exibir um pergunta de sim ou não ao usuário com a mensagem recebida. O método retorna o botão pressionado pelo usuário (**QMessageBox.Yes** ou **QMessageBox.No**).

A Figura 135 exibe o código dos métodos **converte()** e **escolhe_veiculo()** da classe **VeiculoUI**. O método de classe **converte()** será usado para converter o texto das células da tabela em valores numéricos para a realização dos cálculos. Usamos o tratamento de exceção para retornar zero em caso de números inválidos.

```
# ...
# Classe herdando de QMainWindow
class VeiculoUI(QMainWindow):

    def __init__(self, nome_arquivo):
        super().__init__()
        # Carrega o arquivo ui com a interface
        uic.loadUi('veiculo.ui', self)
        # Personaliza cabeçalho da tabela
        header = self.table_gastos.horizontalHeader()
        # Primeira coluna autoajustável
        header.setSectionResizeMode(0, QHeaderView.Stretch)
        # Seleção na tabela
        self.table_gastos.itemSelectionChanged.connect(self.seleciona)
        # Alteração na tabela
        self.table_gastos.cellChanged.connect(self.altera_gasto)
        # Mudança no combobox
        self.combo_veiculo.currentIndexChanged.connect(
                self.escolhe_veiculo)
        # Clique dos botões
        self.button_incluir_veiculo.clicked.connect(self.inclui_veiculo)
        self.button_excluir_veiculo.clicked.connect(self.exclui_veiculo)
        self.button_incluir_gasto.clicked.connect(self.inclui_gasto)
        self.button_excluir_gasto.clicked.connect(self.exclui_gasto)
        self.button_salvar_gastos.clicked.connect(self.salva_gastos)
        # Cria o cadastro de veículos
        self._veiculos = CadastroVeiculos(nome_arquivo)
        # Atualiza a lista na interface
        self._atualiza_lista()

    def _atualiza_lista(self):
        # Limpa o combobox e adiciona os veículos existentes
        self.combo_veiculo.clear()
        self.combo_veiculo.addItems(self._veiculos.lista_veiculos())

    # Método para fazer perguntas sim ou não com caixa de diálogo
    @classmethod
    def pergunta(cls, mensagem):
        # Cria caixa de diálogo
        msg = QMessageBox()
        # Personaliza ícone, mensageme botões
        msg.setIcon(QMessageBox.Question)
        msg.setText(mensagem)
        msg.setStandardButtons(QMessageBox.Yes | QMessageBox.No)
        # Abre caixa de diálogo e retorna botão pressionado
        return msg.exec_()
```

Figura 134 – Código inicial da classe **VeiculoUI**
Fonte: Elaborado pelo Autor.

O método **escolhe_veiculo()** é chamado automaticamente quando o usuário seleciona um veículo no **combo_veiculo**. Observe que o parâmetro **index** é a posição do veículo selecionado no **ComboBox**. O método começa pegando o nome do veículo selecionado com o método **itemText()**. Com o nome do veículo, conseguimos obter sua

lista de gastos para exibir na tabela. Usamos o método **setRowCount()** da tabela para definir seu número de linhas igual a quantidade de gastos.

```
# ...
class VeiculoUI(QMainWindow):
    # ...

    # Método para converter texto em float
    @classmethod
    def converte(cls, texto, padrao=0):
        try:
            # Tenta converter
            return float(texto)
        except:
            # Retorna o padrão em caso de erro
            return padrao

    def escolhe_veiculo(self, index):
        # Veículo selecionado no combobox
        veiculo = self.combo_veiculo.itemText(index)
        # Obtém a lista de gastos do veículo
        lista_gastos = self._veiculos.lista_gastos(veiculo)
        # Modifica o número de linhas da tabela
        self.table_gastos.setRowCount(len(lista_gastos))
        # Percorre lista de gastos
        for linha, gasto in enumerate(lista_gastos):
            # Coluna com nome do gasto
            item = QTableWidgetItem()
            item.setText(gasto[GASTO])
            self.table_gastos.setItem(linha, 0, item)
            # Coluna da quantidade
            item = QTableWidgetItem()
            item.setText('{n:.2f}'.format(n=gasto[QUANT]))
            self.table_gastos.setItem(linha, 1, item)
            # Coluna do valor
            item = QTableWidgetItem()
            item.setText('{n:.2f}'.format(n=gasto[VALOR]))
            self.table_gastos.setItem(linha, 2, item)
            # Coluna do total
            item = QTableWidgetItem()
            total = gasto[QUANT] * gasto[VALOR]
            item.setText('{n:.2f}'.format(n=total))
            self.table_gastos.setItem(linha, 3, item)
        # Calcula total geral
        self.calcula_total()
```

Figura 135 – Métodos **converte()** e **escolhe_veiculo()** da classe **VeiculoUI**
Fonte: Elaborado pelo Autor.

Com a definição do número de linha, usamos um laço de repetição para percorrer cada gasto da lista e preencher as células da tabela. Para cada célula, criamos um objeto da classe **QTableWidgetItem** e usamos o método **setItem()** da tabela para atribuir o objeto à célula correspondente. O método **setItem()** recebe como parâmetros a linha e a coluna da tabela e o objeto que será atribuído a célula dessa posição. Por fim, o método **calcula_total()** é chamado para calcular o total geral de gastos.

```
# ...
class VeiculoUI(QMainWindow):
    # ...

    def inclui_veiculo(self):
        # Nome do veículo digitado no combobox
        veiculo = self.combo_veiculo.currentText().strip()
        # Inclui o veículo e atualiza o combbox
        self._veiculos.inclui_veiculo(veiculo)
        self._atualiza_lista()

    def exclui_veiculo(self):
        # Veículo atual do combobox
        atual = self.combo_veiculo.currentIndex()
        veiculo = self.combo_veiculo.itemText(atual)
        resp = self.pergunta('Deseja excluir o veículo ' +
                             veiculo + '?')
        if resp == QMessageBox.Yes:
            # Exclui o veículo e atualiza o combbox
            self._veiculos.exclui_veiculo(veiculo)
            self._atualiza_lista()

    def inclui_gasto(self):
        # Adiciona mais uma linha da tabela
        self.table_gastos.setRowCount(self.table_gastos.rowCount() + 1)

    def exclui_gasto(self):
        # Remove a linha selecionada na tabela
        atual = self.table_gastos.currentRow()
        self.table_gastos.removeRow(atual)
```

Figura 136 – Métodos para incluir e excluir veículos e gastos da classe **VeiculoUI**
Fonte: Elaborado pelo Autor.

Na Figura 136, podemos ver os métodos para incluir e excluir veículos e gastos. Esses métodos são chamados nos cliques dos botões correspondentes. Na inclusão de veículos, pegamos o nome do veículo digitado no **combo_box** usando o método **currentText()** e usamos o método **inclui_veiculo()** do atributo **_veiculos** para fazer o cadastro. No final, o método chama o método **_atualiza_lista()** para atualizar a lista de veículos mostrada no **combo_veiculos**. Na exclusão de veículos, o método **exclui_veiculo()** pega o veículo atual selecionado no **combo_veiculo**, confirma a exclusão com o usuário e precede com a exclusão chamando o método **exclui_veiculo()** do atributo **_veiculos**.

O método **inclui_gasto()** acrescenta mais uma linha na tabela para inclusão de mais um gasto. O número de linhas atual é obtido com o método **rowCount()**. Esse número é acrescido de um e usado como parâmetro para o método **setRowCount()**. O método **exclui_gasto()** pega a linha selecionada na tabela com o método **currentRow()** e remove essa linha com o método **removeRow()**.

A Figura 137 exibe o código do método **salva_gastos()** da classe **VeiculoUI** associado ao clique do botão de salvar. Inicialmente, verificamos se existe um veículo selecionado usando o método **currentIndex()** do **combo_veiculo**. Se o método retornar -1 (menos um), significa que não há nenhum veículo selecionado.

```
# ...
class VeiculoUI(QMainWindow):
    # ...

    def salva_gastos(self):
        # Pega a posição do veículo selecionado no combobox
        selecionado = self.combo_veiculo.currentIndex()
        # Testa se a seleção é válida
        if selecionado != -1:
            # Veículo selecionado
            veiculo = self.combo_veiculo.itemText(selecionado)
            # Limpa os gastos doe veículo
            self._veiculos.limpa_gastos(veiculo)
            # Para cada linha da tabela
            for linha in range(self.table_gastos.rowCount()):
                # Pega gasto, quantidade e valor
                gasto = ''
                item = self.table_gastos.item(linha, 0)
                if item is not None:
                    gasto = item.text()
                quantidade = ''
                item = self.table_gastos.item(linha, 1)
                if item is not None:
                    quantidade = self.converte(item.text())
                valor = ''
                item = self.table_gastos.item(linha, 2)
                if item is not None:
                    valor = self.converte(item.text())
                # Cria dicionário como gasto
                dic_gasto = {GASTO: gasto, QUANT: quantidade,
                             VALOR: valor}
                # Inclui nos gastos do veículo
                self._veiculos.inclui_gasto(veiculo, dic_gasto)
```

Figura 137 – Método **salva_gastos()** da classe **VeiculoUI**
Fonte: Elaborado pelo Autor.

Se houver um veículo selecionado, pegamos seu nome com o método **itemText()** do **combo_veiculo** e limpamos seus gastos existentes como método **limpa_gastos()** do atributo **_veiculos**. Em seguida, percorremos as linhas da tabela para coletar os gastos digitados pelo usuário. Para cada linha, pegamos as informações do gasto, quantidade e valor para compor um dicionario de gastos.

O conteúdo das células da tabela é obtido com o método **item()** que tem como parâmetro a linha e a coluna da célula. Nós usamos testamos se o conteúdo da célula é diferente de **None** porque, para as células vazias, o método **item()** retorna **None**.

Quando a célula já foi preenchida usamos o método **text()** de seu conteúdo para pegar o texto contido na célula. Os valores numéricos são convertidos com o método **converte()** explicado anteriormente. A última instrução do laço de repetição inclui cada gasto ao veículo selecionado.

A Figura 138 mostra os métodos **altera_gasto()** e **calcula_total()** da classe **VeiculoUI**. O método **altera_gasto()** está conectado ao evento de alteração das células da

tabela. Seu primeiro passo do é testar se a alteração ocorreu nas colunas de quantidade ou valor (colunas 1 e 2).

```
# ...
class VeiculoUI(QMainWindow):
    # ...

    def altera_gasto(self, linha, coluna):
        # Testa se a alteração foi na quantidade ou no valor
        if coluna in [1, 2]:
            # Pega as células de quantidade e de valor
            itemQuantidade = self.table_gastos.item(linha, 1)
            itemValor = self.table_gastos.item(linha, 2)
            # Testa se as células foram preenchidas
            if itemQuantidade is not None and itemValor is not None:
                # Converte os valores e calcula o total
                quantidade = self.converte(itemQuantidade.text())
                valor = self.converte(itemValor.text())
                total = quantidade * valor
            else:
                total = 0
            # Mostra o total na tabela
            itemTotal = QTableWidgetItem()
            itemTotal.setText('{t:.2f}'.format(t=total))
            self.table_gastos.setItem(linha, 3, itemTotal)
            # Calcula o total geral
            self.calcula_total()

    def calcula_total(self):
        total_geral = 0
        # Percore as linhas da tabela
        for linha in range(self.table_gastos.rowCount()):
            itemTotal = self.table_gastos.item(linha, 3)
            if itemTotal is not None:
                total_linha = self.converte(itemTotal.text())
                # Soma o total de cada linha
                total_geral += total_linha
        # Mostra o total geral
        self.edit_total.setText('{t:.2f}'.format(t=total_geral))
```

Figura 138 – Métodos **altera_gasto()** e **calcula_total()** da classe **VeiculoUI**
Fonte: Elaborado pelo Autor.

Quando a alteração ocorre na quantidade ou valor de um gasto, temos que recalcular o total do gasto. Assim, o método **altera_gasto()**, após converter o texto da quantidade e valor, calcula o total do gasto multiplicando a quantidade pelo valor. Se a célula da quantidade ou valor estiverem vazias, o total do gasto é definido como zero. Por fim, a célula referente ao total do gasto é preenchida e o método **calcula_total()** é chamado.

O método **calcula_total()** é responsável por calcular o total geral de gastos para um veículo. O cálculo é feito percorrendo a tabela para obter o valor total de cada gasto. Os valores são somados e apresentados no **edit_total**.

A Figura 139 apresenta os demais métodos da classe **VeiculoUI** e o **if** de execução do arquivo. O método seleciona está associado ao evento de seleção das células da tabela de gastos. Se a seleção for sobre a coluna de total do gasto, modificamos a tabela com o método **setEditTriggers()** para não permitir edições, pois o total do gasto é calculado automaticamente. Para as demais colunas, a alteração é permitida.

```
# ...
class VeiculoUI(QMainWindow):
    # ...

    def seleciona(self):
        # Pega a coluna selecionada na tabela
        coluna = self.table_gastos.currentColumn()
        # Se for a coluna 3 (total)
        if coluna == 3:
            # Não permite alterações
            self.table_gastos.setEditTriggers(
                    QTableWidget.NoEditTriggers)
        else:
            # Para as demais colunas, a alteração é permitida
            self.table_gastos.setEditTriggers(
                    QTableWidget.DoubleClicked |
                    QTableWidget.EditKeyPressed |
                    QTableWidget.AnyKeyPressed)

    # Evento de fechamento da janela
    def closeEvent(self, event):
        # Confirma se o usuário deseja sair
        if self.pergunta('Deseja sair?') == QMessageBox.Yes:
            # Salva o cadastro de veículo em arquivo
            self._veiculos.salva()
            # Confirma saída
            event.accept()
        else:
            # Cancela fechamento da janela
            event.ignore()


if __name__ == '__main__':
    # Cria aplicação e a janela
    app = QApplication([])
    window = VeiculoUI('veiculos.json')
    # Exibe janela e executa a aplicação
    window.show()
    app.exec_()
```

Figura 139 – Demais métodos da classe **VeiculoUI** e **if** de execução do arquivo
Fonte: Elaborado pelo Autor.

O método **closeEvent()** já existe na classe **QMainWindow**. Na classe **VeiculoUI**,o referido método é sobrescrito para confirmar a saída do programa. Usamos o método pergunta() para perguntar se o usuário deseja sair e o parâmetro **event** para aceitar ou cancelar o fechamento da janela.

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

4.7 Exercícios

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

A) Jogo da forca

Figura 140 – Interface para jogo da forca
Fonte: Elaborado pelo Autor.

- Implemente o jogo da forca com interface gráfica como mostrado na Figura 140. O jogo deve ler as palavras de um arquivo de texto simples previamente preenchido;

- Depois de desenhar a interface, crie a classe **Palavra** para representar a palavra do jogo. O construtor da classe deve receber um **str** com a palavra e guardá-la no atributo **_palavra**. O construtor deve inicializar também os atributos **_tela**, **_erros** e **_digitadas**. O atributo **_tela** é a representação da palavra a ser exibida no jogo e deve ser inicializado como uma lista contendo um hífen para cada letra da palavra. O atributo **_erros** é usado para contabilizar os erros e deve ser inicializado com zero. O atributo **_digitadas** é inicializado com **''** (texto vazio) e servirá para guardar todas as letras já digitadas;

- Crie os métodos **tela()**, **erros()** e **digitadas()** para a classe **Palavra**. O método **tela()** deve retornar um **str** com os elementos da lista **_tela**. O método **erros()** retorna o valor em **_erros** e o método **digitadas()** retorna o conteúdo do atributo **_digitadas**. Esses três métodos podem ser propriedades;

- Implemente o método **_letra_valida()** da classe **Palavra**. Esse método deve receber um parâmetro **str** e retornar **True** se o mesmo for uma letra válida. Um **str** é uma letra válida se contiver um único caractere, esse caractere for uma letra entre A e Z e não estiver nas letras já digitadas;

- Escreva o método **tem_letra()** para a classe **Palavra**. Esse método deve receber um a letra, verificar se a mesma é válida e, em caso positivo, percorrer a palavra para verificar se a letra existe. A letra deve ser acrescentada às letras digitadas e, se a letra existir na palavra, os hifens correspondentes à letra da representação de tela devem ser trocadas pela letra recebida. No final, o método deve retornar **True** se a letra foi encontrada na palavra;

- Para finalizar a classe **Palavra**, inclua o método **completada()** que deve retornar **True** se todos os hifens da representação de tela já foram substituídos por letras, ou seja, se não houverem mais hifens na representação de tela;

- Após a construção da classe **Palavra**, crie a classe **ForcaUI** herdando de **QMainWindow** para implementar a interface do jogo. O construtor da classe deve receber como parâmetro o nome do arquivo contendo as palavras. Além de carregar o arquivo de desenho da interface, o construtor deve inicializar os atributos da classe e conectar o evento de clique do botão de novo jogo. Os atributos a serem inicializados são **_nome_arquivo** (nome do arquivo recebido como parâmetro), **_lista_palavras** (lista vazia para guardar as palavras do jogo), **_palavra** (**None**, palavra atual do jogo) e **_jogando** (**False**, indica se o jogo está acontecendo). Após a inicialização dos atributos o construtor deve chamar um método para carregar as palavras do arquivo e o método de iniciar um novo jogo;

- Crie o método **carrega_palavras()** para ler o conteúdo do arquivo **_nome_arquivo** e preencher a **_lista_palavras**. Cada linha do arquivo deve ser uma palavra válida para o jogo. As palavras válidas devem possuir apenas letras de A a Z;

- Escreva o método **mostrar()** para atualizar a interface com as informações do jogo. As informações devem ser mostradas nos rótulos com a representação de tela da palavra, com as letras digitadas e com os erros;

- O método **novo()** da classe deve sortear uma palavra para o jogo, atribuir **True** ao atributo **_jogando** e chamar o método **mostrar()**. Já o método de classe **aviso()** para receber um texto como parâmetro e exibi-lo em uma mensagem com o **QMessageBox**;

- Implemente o método **jogar()** que deve receber um letra e inseri-lo no jogo. O método deve executar suas ações apenas se o atributo **_jogando** for **True**. Nesse caso, precisamos chamar o método **tem_letra()** do atributo **_palavra** para verificar se a letra existe. Se não existir, devemos verificar se o total de erros chegou a cinco e encerrar o jogo informando a derrota. Precisamos verificar também se a palavra foi completada e, em caso afirmativo, exibir a mensagem de fim de jogo;

- Por fim, sobrescreva o evento **keyPressEvent()** da classe **QMainWindow** para capturar as teclas digitadas pelo jogador. Esse método possui o parâmetro **event** e a tecla pressionada pode ser obtida com o o método **key()** desse parâmetro. De posse da tecla, o método deve verificar se a mesma é uma letra entre A e Z. Como a tecla é o código da letra, você pode usar a função **ord()** ou **chr()** para converter o código da tecla ou a letra. Se a tecla for válida, o método **jogar()** deve ser chamado passando como parâmetro a letra referente a tecla.

B) Agenda de contatos

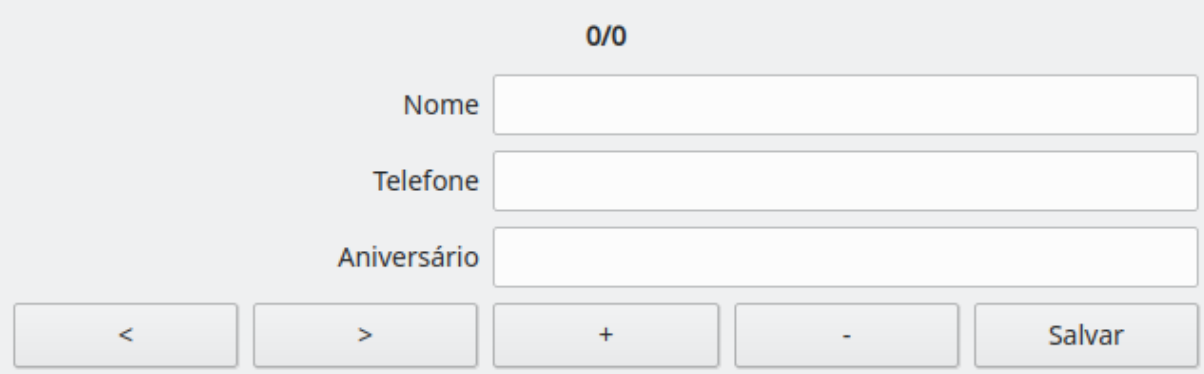

Figura 141 – Interface para agenda de contatos
Fonte: Elaborado pelo Autor.

- Implemente uma agenda de contatos com usando a interface exibida na Figura 141. A agenda deve salvar e ler a lista de contatos de um arquivo JSON;

- Os contatos serão armazenados em uma lista onde cada elemento é um dicionário com os dados do contato. Portanto, comece declarando as chaves do dicionário (**NOME**, **TELEFONE** e **ANIVERSARIO**);

- Crie a classe **AgendaUI** herdando de **QMaindWindow** para construção da interface. O construtor da classe deve receber como parâmetro o nome do arquivo de contatos. No construtor, carregue o arquivo de interface e inicialize os atributos **_nome_arquivo** (com o parâmetro recebido), **_contatos** (com uma lista vazia) e **_atual** (com menos um). O atributo **_nome_arquivo** guarda o nome do arquivo para ser usado na leitura e armazenamento dos dados, o atributo **_contatos** será usado para guardar a lista de contatos e o atributo **_atual** é a posição do contato atual a ser exibido na interface;

- Ainda no construtor da classe, leia o arquivo de contatos e, caso exista ao menos um contato, mude **_atual** para o primeiro contato. Conecte o evento de clique de cada botão ao método apropriado. A última instrução do construtor deve ser a chamada de um método para exibir o contato atual;

- Escreva o método **exibir()** para mostrar a posição do contato atual e o total de contatos no rótulo superior da interface e exiba as informações do contato atual (**_atual**) nas caixas de texto. Esse método deve ser chamado no final das ações de incluir, excluir e nos botões de navegação;

- Implemente o método de classe **pergunta()** que recebe uma mensagem textual, exibe essa mensagem com o **QMessageBox** e retorna o botão pressionado (sim ou não);

- Crie o método **incluir()** para criar um dicionário com os dados das caixas de texto e inseri-lo na lista de contatos (**_contatos**);

- Escreva o método **excluir()** para excluir o contato atual. Antes de excluir, verifique se posição atual é válida e confirme a exclusão com o usuário;

- Construa o método **salvar()** para atualizar os dados digitados pelo usuário na posição atual da lista de contatos;

- Implemente os métodos **anterior()** e **proximo()** para percorrer os contatos da lista. Lembre-se que o primeiro contato está na posição zero e a última posição é quantidade de contatos menos um;

- Sobrescreva o método **closeEvent()** para confirmar se o usuário deseja realmente sair e salvar os contatos em arquivo antes de fechar o programa.

4.8 Respostas dos exercícios

A) Jogo da forca

Interface

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
 <class>MainWindow</class>
 <widget class="QMainWindow" name="MainWindow">
  <property name="geometry">
   <rect>
    <x>0</x>
    <y>0</y>
    <width>571</width>
    <height>256</height>
   </rect>
  </property>
  <property name="windowTitle">
   <string>Agenda de contatos</string>
  </property>
  <widget class="QWidget" name="centralwidget">
   <layout class="QVBoxLayout" name="verticalLayout">
    <item>
     <widget class="QPushButton" name="button_novo">
      <property name="text">
       <string>Novo Jogo</string>
      </property>
     </widget>
    </item>
    <item>
     <widget class="QLabel" name="label_palavra">
      <property name="font">
       <font>
        <pointsize>16</pointsize>
        <weight>75</weight>
        <bold>true</bold>
       </font>
      </property>
      <property name="text">
       <string>-----</string>
      </property>
      <property name="alignment">
       <set>Qt::AlignCenter</set>
      </property>
     </widget>
    </item>
```

```
42      <item>
43       <widget class="QLabel" name="label_digitadas">
44        <property name="font">
45         <font>
46          <pointsize>14</pointsize>
47         </font>
48        </property>
49        <property name="text">
50         <string>Letras digitadas:</string>
51        </property>
52        <property name="alignment">
53         <set>Qt::AlignLeading|Qt::AlignLeft|Qt::AlignVCenter</set>
54        </property>
55       </widget>
56      </item>
57      <item>
58       <widget class="QLabel" name="label_erros">
59        <property name="font">
60         <font>
61          <pointsize>14</pointsize>
62         </font>
63        </property>
64        <property name="text">
65         <string>Erros:</string>
66        </property>
67        <property name="alignment">
68         <set>Qt::AlignLeading|Qt::AlignLeft|Qt::AlignVCenter</set>
69        </property>
70       </widget>
71      </item>
72     </layout>
73    </widget>
74   </widget>
75   <resources/>
76   <connections/>
77  </ui>
```

Código

```
 1  #!/usr/bin/env python3
 2  # -*- coding: utf-8 -*-
 3
 4  from PyQt5 import uic
 5  from PyQt5.QtWidgets import QApplication, QMainWindow, QMessageBox
 6  from random import shuffle
 7  from os.path import isfile
 8
 9
10  class Palavra:
11
12      def __init__(self, palavra):
13          # Palavra
14          self._palavra = palavra
15          # Um traço para cada letra para mostra na tela
16          self._tela = ['-'] * len(palavra)
17          # Erros e letras digitadas
```

```
18          self._erros = 0
19          self._digitadas = ''
20
21      @property
22      def tela(self):
23          # Retorna a representação da palavra na tela
24          return ''.join(self._tela)
25
26      @property
27      def erros(self):
28          return self._erros
29
30      @property
31      def digitadas(self):
32          return self._digitadas
33
34      def _letra_valida(self, letra):
35          # A letra deve ser um único caracter
36          if len(letra) != 1:
37              return False
38          # Dever ser entre A e Z e não ter sido digitada antes
39          if letra < 'A' or letra > 'Z' or letra in self._digitadas:
40              return False
41          return True
42
43      def tem_letra(self, letra):
44          # Verifica se a letra é valida
45          if self._letra_valida(letra):
46              # Acrescenta às letras já digitadas
47              self._digitadas += letra
48              # Supões que não tem a letra
49              encontrada = False
50              # Percorre os caracteres da palavra
51              for cont, carac in enumerate(self._palavra):
52                  # Verifica se a letra é igual ao caractere atual
53                  if carac == letra:
54                      # Letra encontrada
55                      encontrada = True
56                      # Mostra a letra na representação da tela
57                      self._tela[cont] = letra
58              # Contabilita o erro se a letra não for encontrada
59              if not encontrada:
60                  self._erros += 1
61              return encontrada
62
63      @property
64      def completada(self):
65          # A palavra está é completada quando não tem mais traços
66          return not ('-' in self._tela)
67
68
```

```
69  class ForcaUI(QMainWindow):
70
71      def __init__(self, nome_arquivo):
72          super().__init__()
73          # Carrega o arquivo ui com a interface
74          uic.loadUi('forca.ui', self)
75          # Conecta clique do botão
76          self.button_novo.clicked.connect(self.novo_jogo)
77          # Inicializa nome do arquivo e carrega o conteúdo
78          self._nome_arquivo = nome_arquivo
79          self._lista_palavras = []
80          self.carrega_palavras()
81          # Palavra
82          self._palavra = None
83          # Indicado de jogo ativo
84          self._jogando = False
85          self.novo_jogo()
86
87      def carrega_palavras(self):
88          # Verifica se o arquivo existe
89          if isfile(self._nome_arquivo):
90              # Abre o arquivo e lê as palavras
91              with open(self._nome_arquivo) as arq:
92                  for linha in arq:
93                      palavra = linha.strip()
94                      self._lista_palavras.append(palavra)
95
96      def mostrar(self):
97          # Mostra as informaçõe sdo jogo nos labels
98          self.label_palavra.setText(self._palavra.tela)
99          texto = 'Letras digitadas: ' + \
100             ', '.join(self._palavra.digitadas)
101         self.label_digitadas.setText(texto)
102         texto = 'Erros: ' + str(self._palavra.erros)
103         self.label_erros.setText(texto)
104
105     def novo_jogo(self):
106         if len(self._lista_palavras) < 5:
107             self.aviso('O arquivo ' + self._nome_arquivo +
108                        ' deve ter ao menos 5 palavras')
109             return
110         # Embaralha lista de palavras
111         shuffle(self._lista_palavras)
112         # Sorteia (primeira palavra da lista embaralhada)
113         self._palavra = Palavra(self._lista_palavras[0])
114         # Inicia o jogo
115         self._jogando = True
116         # Mostra o estado do jogo
117         self.mostrar()
118
```

```
119      @classmethod
120      def aviso(cls, mensagem):
121          # Cria caixa de diálogo
122          msg = QMessageBox()
123          # Personaliza ícone, mensageme botões
124          msg.setIcon(QMessageBox.Information)
125          msg.setText(mensagem)
126          msg.setStandardButtons(QMessageBox.Ok)
127          # Abre caixa de diálogo e retorna botão pressionado
128          msg.exec_()
129
130      def jogar(self, letra):
131          # Verifica se o jogo está ativo
132          if self._jogando:
133              # Testa se a letra existe na palavra
134              if not self._palavra.tem_letra(letra):
135                  # Testa se já tem 5 erros
136                  if self._palavra.erros == 5:
137                      # O jogador perde e o jogo termina termina
138                      self.mostrar()
139                      self._jogando = False
140                      self.aviso('Você perdeu!')
141              # Verifica se a palavra foi compeltada
142              if self._palavra.completada:
143                  # O jogador vence e o jogo termina
144                  self.mostrar()
145                  self._jogando = False
146                  self.aviso('Você acertou!')
147              # Mostra o estado do jogo
148              self.mostrar()
149
150      # Evento de pressionamento de tecla
151      def keyPressEvent(self, event):
152          # Pega o código da tecla pressionada
153          tecla = event.key()
154          # Verifica se a tecla é válida
155          if tecla >= ord('A') and tecla <= ord('Z'):
156              # Joga com a letra referente a tecla
157              self.jogar(chr(tecla))
158
159
160  if __name__ == '__main__':
161      # Cria aplicação e a janela
162      app = QApplication([])
163      window = ForcaUI('palavras.txt')
164      # Exibe janela e executa a aplicação
165      window.show()
166      app.exec_()
```

B) Agenda de contatos

Interface

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
 <class>MainWindow</class>
 <widget class="QMainWindow" name="MainWindow">
  <property name="geometry">
   <rect>
    <x>0</x>
    <y>0</y>
    <width>637</width>
    <height>200</height>
   </rect>
  </property>
  <property name="windowTitle">
   <string>Agenda de contatos</string>
  </property>
  <widget class="QWidget" name="centralwidget">
   <layout class="QGridLayout" name="gridLayout">
    <item row="0" column="0" colspan="5">
     <widget class="QLabel" name="label_atual">
      <property name="font">
       <font>
        <weight>75</weight>
        <bold>true</bold>
       </font>
      </property>
      <property name="text">
       <string>0/0</string>
      </property>
      <property name="alignment">
       <set>Qt::AlignCenter</set>
      </property>
     </widget>
    </item>
    <item row="1" column="2" colspan="3">
     <widget class="QLineEdit" name="edit_nome"/>
    </item>
    <item row="2" column="0" colspan="2">
     <widget class="QLabel" name="label_3">
      <property name="text">
       <string>Telefone</string>
      </property>
      <property name="alignment">
       <set>Qt::AlignRight|Qt::AlignTrailing|Qt::AlignVCenter</set>
      </property>
     </widget>
    </item>
    <item row="2" column="2" colspan="3">
     <widget class="QLineEdit" name="edit_telefone"/>
    </item>
    <item row="3" column="0" colspan="2">
     <widget class="QLabel" name="label_4">
      <property name="text">
```

```
 53          <string>Aniversário</string>
 54         </property>
 55         <property name="alignment">
 56          <set>Qt::AlignRight|Qt::AlignTrailing|Qt::AlignVCenter</set>
 57         </property>
 58        </widget>
 59       </item>
 60       <item row="3" column="2" colspan="3">
 61        <widget class="QLineEdit" name="edit_aniversario"/>
 62       </item>
 63       <item row="4" column="0">
 64        <widget class="QPushButton" name="button_anterior">
 65         <property name="text">
 66          <string>&lt;</string>
 67         </property>
 68        </widget>
 69       </item>
 70       <item row="4" column="1">
 71        <widget class="QPushButton" name="button_proximo">
 72         <property name="text">
 73          <string>&gt;</string>
 74         </property>
 75        </widget>
 76       </item>
 77       <item row="4" column="2">
 78        <widget class="QPushButton" name="button_incluir">
 79         <property name="text">
 80          <string>+</string>
 81         </property>
 82        </widget>
 83       </item>
 84       <item row="4" column="3">
 85        <widget class="QPushButton" name="button_excluir">
 86         <property name="text">
 87          <string>-</string>
 88         </property>
 89        </widget>
 90       </item>
 91       <item row="4" column="4">
 92        <widget class="QPushButton" name="button_salvar">
 93         <property name="text">
 94          <string>Salvar</string>
 95         </property>
 96        </widget>
 97       </item>
 98       <item row="1" column="0" colspan="2">
 99        <widget class="QLabel" name="label_2">
100         <property name="text">
101          <string>Nome</string>
102         </property>
103         <property name="alignment">
104          <set>Qt::AlignRight|Qt::AlignTrailing|Qt::AlignVCenter</set>
105         </property>
106        </widget>
107       </item>
```

```
    </layout>
   </widget>
  </widget>
  <resources/>
  <connections/>
 </ui>
```

Código

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

import json
from os.path import isfile
from PyQt5 import uic
from PyQt5.QtWidgets import QApplication, QMainWindow, QMessageBox

NOME = 'nome'
TELEFONE = 'telefone'
ANIVERSARIO = 'aniversário'


# Classe herdando de QMainWindow
class AgendaUI(QMainWindow):

    def __init__(self, nome_arquivo):
        # Criação da interface
        super().__init__()
        uic.loadUi('agenda.ui', self)
        # Inicialização de atributos
        self._nome_arquivo = nome_arquivo
        self._contatos = []
        # Posição na lista de contatos
        self._atual = 0
        # Leitura do arquivo
        if isfile(self._nome_arquivo):
            with open(self._nome_arquivo) as arq:
                self._contatos = json.load(arq)
        if len(self._contatos) == 0:
            self._atual = -1
        # Clique dos botões
        self.button_anterior.clicked.connect(self.anterior)
        self.button_proximo.clicked.connect(self.proximo)
        self.button_incluir.clicked.connect(self.incluir)
        self.button_excluir.clicked.connect(self.excluir)
        self.button_salvar.clicked.connect(self.salvar)
        self.exibir()

    # Exibe o contato da posição atual
    def exibir(self):
        # Total de contatos
        total = len(self._contatos)
        # Exibe a posicao atual e total de contatos
        self.label_atual.setText(str(self._atual+1) +
```

```
46                                      '/' + str(total))
47          # Testa se a posição é válida
48          if self._atual >= 0 and self._atual < len(self._contatos):
49              # Pega o contato da posição
50              contato = self._contatos[self._atual]
51              # Mostra os dados nos Edits
52              self.edit_nome.setText(contato[NOME])
53              self.edit_telefone.setText(contato[TELEFONE])
54              self.edit_aniversario.setText(contato[ANIVERSARIO])
55
56      # Perguntas sim ou não com caixa de diálogo
57      @classmethod
58      def pergunta(cls, mensagem):
59          # Cria caixa de diálogo
60          msg = QMessageBox()
61          # Personaliza ícone, mensageme botões
62          msg.setIcon(QMessageBox.Question)
63          msg.setText(mensagem)
64          msg.setStandardButtons(QMessageBox.Yes | QMessageBox.No)
65          # Abre caixa de diálogo e retorna botão pressionado
66          return msg.exec_()
67
68      def incluir(self):
69          # Pega dados do contato
70          nome = self.edit_nome.text()
71          telefone = self.edit_telefone.text()
72          aniversario = self.edit_aniversario.text()
73          contato = {NOME: nome,
74                     TELEFONE: telefone,
75                     ANIVERSARIO: aniversario}
76          # Adiciona a lista
77          self._contatos.append(contato)
78          # Vai para a última posição da lista (contato adicionado)
79          self._atual = len(self._contatos) - 1
80          self.exibir()
81
82      def excluir(self):
83          # Testa se a posição é válida
84          if self._atual >= 0 and self._atual < len(self._contatos):
85              # Confirma exclusão
86              resp = self.pergunta('Deseja excluir o contato atual?')
87              if resp == QMessageBox.Yes:
88                  # Remove contato
89                  self._contatos.pop(self._atual)
90                  self.exibir()
91
92      def salvar(self):
93          # Testa se a posição é válida
94          if self._atual >= 0 and self._atual < len(self._contatos):
95              # Pega os dados dos edits
96              nome = self.edit_nome.text()
97              telefone = self.edit_telefone.text()
98              aniversario = self.edit_aniversario.text()
99              contato = {NOME: nome,
100                         TELEFONE: telefone,
```

```
101                             ANIVERSARIO: aniversario}
102                 # Atualiza contato
103                 self._contatos[self._atual] = contato
104
105         def anterior(self):
106             # Vai para posição anterior
107             self._atual -= 1
108             # A menor posição é 0 (zero)
109             if self._atual < 0:
110                 self._atual = 0
111             self.exibir()
112
113         def proximo(self):
114             # Vai para a próxima posição
115             self._atual += 1
116             # A maior posição é a última da lista
117             if self._atual >= len(self._contatos):
118                 self._atual = len(self._contatos) - 1
119             self.exibir()
120
121         # Evento de fechamento da janela
122         def closeEvent(self, event):
123             # Confirma se o usuário deseja sair
124             if self.pergunta('Deseja sair?') == QMessageBox.Yes:
125                 with open(self._nome_arquivo, 'w') as arq:
126                     json.dump(self._contatos, arq, indent=2,
127                               ensure_ascii=False)
128                 # Confirma saída
129                 event.accept()
130             else:
131                 # Cancela fechamento da janela
132                 event.ignore()
133
134
135     if __name__ == '__main__':
136         # Cria aplicação e a janela
137         app = QApplication([])
138         window = AgendaUI('agenda.json')
139         # Exibe janela e executa a aplicação
140         window.show()
141         app.exec_()
```

4.9 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 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/

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

Feito por (professor-autor)	Data	Revisão de *layout*	Data	Versão
Marcos Roberto Ribeiro	20/03/2022	Desginado pela proex		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