



Fundamentos de Python (UFCD 10793)

Manual de Apoio

Formador: Sandra Liliana Meira de Oliveira



INTRODUÇÃO	13
ORIGEM DA LINGUAGEM PYTHON.....	16
INSTALAÇÃO DO AMBIENTE PYTHON.....	17
CARACTERÍSTICAS DA LINGUAGEM PYTHON.....	26
COMO ESTÁ ORGANIZADO O LIVRO?.....	28

CAPÍTULO II

EXECUÇÃO SEQUENCIAL	30
VARIÁVEIS E CONSTANTES.....	30
TIPOS DE DADOS.....	36
OPERADORES ARITMÉTICOS.....	39
OPERADORES DE ATRIBUIÇÃO.....	40
ENTRADA DE DADOS E CONVERSÃO DE TIPOS.....	41
FORMATAÇÃO DE STRINGS.....	44
EXERCÍCIOS RESOLVIDOS.....	48
EXERCÍCIOS PROPOSTOS.....	50

CAPÍTULO III

ESTRUTURAS CONDICIONAIS	54
ONDE ENCONTRAMOS ESTRUTURAS CONDICIONAIS?.....	54
OPERADORES.....	59
OPERADORES DE COMPARAÇÃO OU RELACIONAIS.....	60
OPERADORES LÓGICOS.....	60
TABELA VERDADE.....	62
ESTRUTURA IF...ELSE.....	65
ESTRUTURA ELIF.....	68
EXERCÍCIOS RESOLVIDOS.....	69
EXERCÍCIOS PROPOSTOS.....	79

CAPÍTULO IV

ESTRUTURAS DE REPETIÇÃO	84
ESTRUTURA WHILE.....	86
FUNÇÃO RANGE.....	91
ESTRUTURA FOR.....	93
COMANDO BREAK.....	95
EXERCÍCIOS RESOLVIDOS.....	97
EXERCÍCIOS PROPOSTOS.....	101

CAPÍTULO V

LISTAS	105
DECLARANDO UMA LISTA.....	106
INCLUINDO ELEMENTOS.....	109

EMBARALHANDO E SORTEANDO ELEMENTOS.....	111
ORDENANDO ELEMENTOS.....	112
REMOVENDO ELEMENTOS.....	113
CLONANDO E COMPARANDO LISTAS.....	116
INCLUINDO ELEMENTOS DE UMA LISTA EM OUTRA.....	117
OCORRÊNCIAS DE ELEMENTOS E COMPRIMENTO DA LISTA.....	118
MENOR, MAIOR E SOMA DE ELEMENTOS.....	119
RETORNANDO O ÍNDICE DE UM ELEMENTO.....	120
RETORNANDO ÍNDICE E ELEMENTO.....	123
LISTAS ANINHADAS.....	124
UMA MANEIRA DIFERENTE DE GERAR E MANIPULAR LISTAS.....	128
EXERCÍCIOS RESOLVIDOS.....	130
EXERCÍCIOS PROPOSTOS.....	137

CAPÍTULO VI

DICIONÁRIOS.....	141
ACEDENDO ELEMENTOS	142
INSERINDO E ATUALIZANDO ELEMENTOS.....	144
CONTANDO ELEMENTOS.....	145
VERIFICANDO A EXISTÊNCIA DE UMA CHAVE.....	145
OBTENDO VALORES E CHAVES.....	146
ORDENANDO ELEMENTOS.....	148
CLONANDO UM DICIONÁRIO.....	149
REMOVENDO ELEMENTOS.....	150
ESVAZIANDO UM DICIONÁRIO.....	150
EXERCÍCIOS RESOLVIDOS.....	151
EXERCÍCIOS PROPOSTOS.....	157

CAPÍTULO VII

FUNÇÕES.....	160
DECLARANDO FUNÇÕES.....	161
EXERCÍCIOS RESOLVIDOS.....	165
EXERCÍCIOS PROPOSTOS.....	171

CAPÍTULO VIII

INTRODUÇÃO À ORIENTAÇÃO A OBJETOS.....	175
CLASSES E OBJETOS.....	177
IMPLEMENTAÇÃO EM PYTHON.....	179
EXERCÍCIOS RESOLVIDOS.....	185
EXERCÍCIOS PROPOSTOS.....	193

CAPÍTULO IX

APLICAÇÕES DE SISTEMAS OPERATIVOS.....	196
CONCORRÊNCIA.....	196
ESCALONAMENTO.....	224

REFERÊNCIAS.....	235
-------------------------	------------

ORIGEM DA LINGUAGEM PYTHON

Python é uma linguagem de programação criada pelo matemático e programador Guido van Rossum em 1991. Embora muita gente faça associação direta com cobras, por causa das serpentes píton, a origem do nome deve-se ao grupo humorístico britânico *Monty Python*, que inclusive, em 2019, completou 50 anos de existência.



Figura 2 – Guido Van Rossum, criador da linguagem Python.
Fonte: https://pt.wikipedia.org/wiki/Guido_van_Rossum.



Figura 3 – *Monty Python*, série britânica de humor que deu nome à linguagem.
Fonte: <https://teoriageek.com.br/resenha-filme-monty-python-e-o-calice-sagrado/>.

A insistência em associar Python a serpentes ajudou na criação do símbolo oficial da linguagem. Veja a Figura 4.

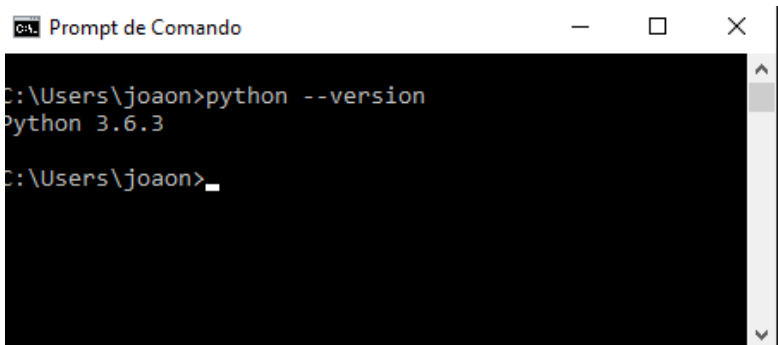


Figura 4 – Logo oficial da linguagem Python.
Fonte: <http://developernews.com.br/o-que-e-python/>.

INSTALAÇÃO DO AMBIENTE PYTHON

Como era de se esperar, o ambiente Python (linguagem, interpretador, módulos e ferramentas) é super simples de se instalar. No caso de sistema operativo (SO) Linux, a maioria das distribuições já tem o Python instalado, porque

vários *scripts* do SO já são executados em Python. Para verificar se já está instalado no Linux ou Windows, abra a consola de comandos do sistema operativo e digite o comando `python --version`. Veja a Figura 5, que apresenta o resultado do comando.



```
C:\Users\joaon>python --version
Python 3.6.3
C:\Users\joaon>
```

Figura 5 – Verifica instalação da linguagem Python.

No Linux, caso o Python não esteja instalado, utilize o gestor de pacotes conforme comandos na Tabela 1 abaixo:

Tabela 1 – Comandos para instalação no Python no Linux.

GERENCIADOR DE PACOTE DO LINUX	COMANDO	O QUE ESTÁ INSTALANDO?
apt-get	sudo apt-get install python3.8	Python
	sudo apt-get install python-pip	Gerenciador de módulos do Python. Serve para instalar extensões do Python.
Yum	sudo yum install python38	Python
	yum -y install python-pip	Gerenciador de módulos do Python.

No Windows, por padrão, não vem instalado, então deve efetuar download do instalador e realizar a instalação no computador. O site oficial é www.python.org. Veja a Figura 6.

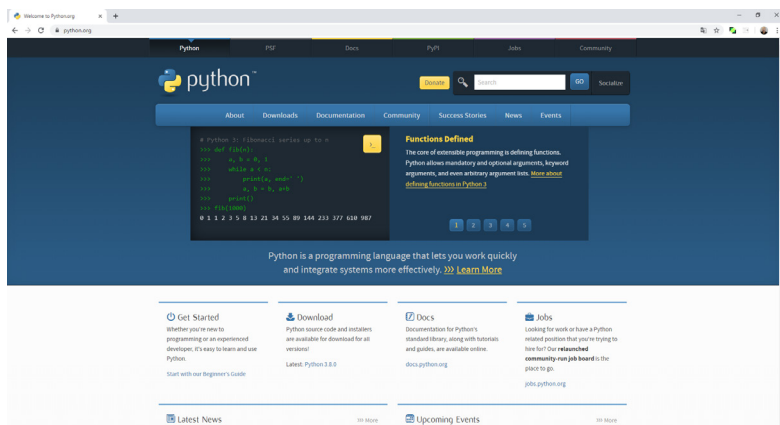


Figura 6 – Site oficial da linguagem Python.

No website, acesse a aba “downloads” e baixe o instalador Python. Na ocasião de escrita deste material, a versão do momento é a Python 3.8.0. Se na época em que estiver lendo este manual, existir uma versão mais recente, não há problema, todos os passos aqui indicados se aplicarão. Veja a Figura 7.

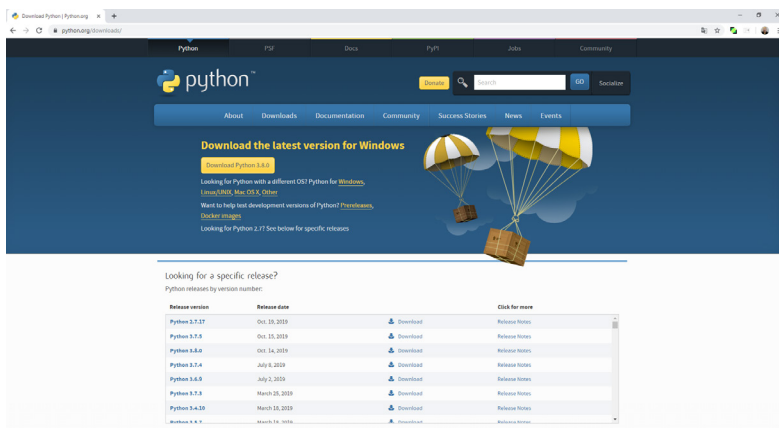


Figura 7 – Local para fazer o download o instalador Python para Windows.

Com o ficheiro de instalação na sua máquina, execute o mesmo com dois cliques ou pressione ENTER após a sua seleção. Em seguida, aplique o velho “Next, Next, Finish” das instalações do Windows. No primeiro ecrã, não se esqueça de marcar a opção “Add Python 3.8 to PATH” e clicar na opção “Install Now”. Veja a Figura 8.

De seguida, inicia-se o processo de instalação. Tenha cuidado, pois como a instalação vai modificar arquivos do sistema, o Windows pode perguntar se deseja continuar. Responda “Sim”. Veja a janela de progresso da instalação na Figura 9.

Segue-se uma janela informando que a instalação foi concluída com sucesso. Clique em *close* para fechar a janela. Veja a Figura 10.



Figura 8 – Primeira janela do instalador Python no Windows.

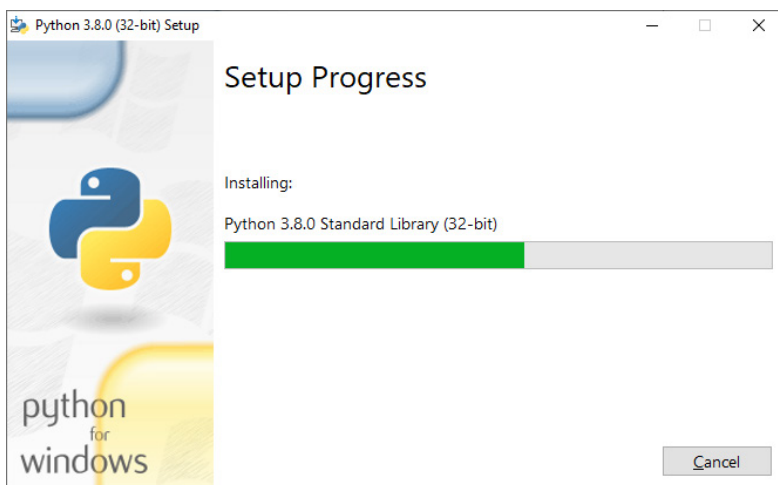


Figura 9 – Janela de progresso de instalação da linguagem Python.

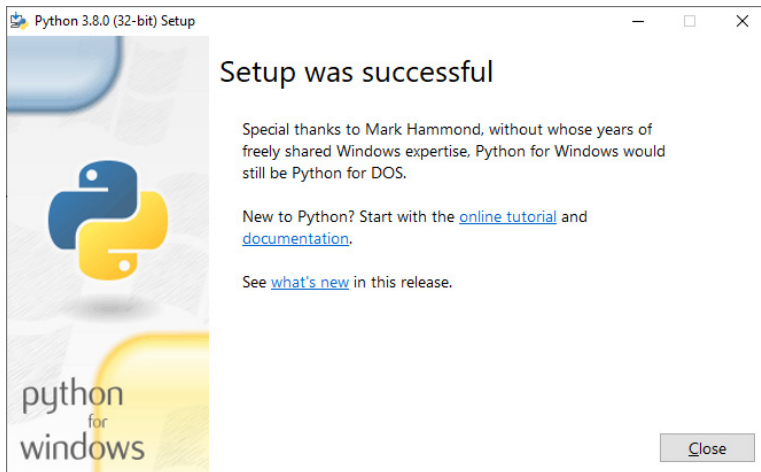


Figura 10 – Janela que informa o sucesso da instalação no Windows.

Agora, vamos testar. Nos programas instalados do Windows procure a pasta Python 3.8, execute a aplicação IDLE, conforme exemplo da Figura 11 abaixo.

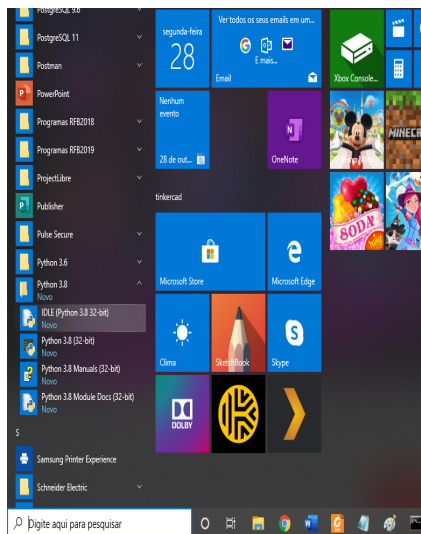


Figura 11 – Como iniciar a interface de programação IDLE.

Deve visualizar algo parecido com a imagem da Figura 12. O IDLE é um ambiente de desenvolvimento integrado para Python, que é lançado em cada atualização da ferramenta desde a versão 2.3. Não é a melhor ferramenta para se programar, mas para aprender programação é excelente. Veja o IDLE na Figura 12.

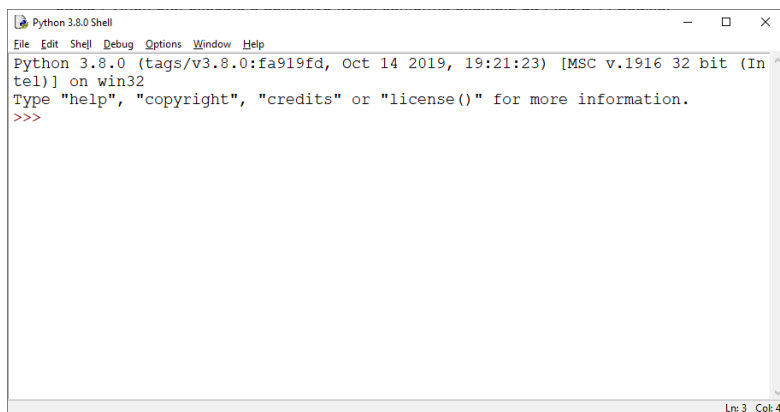


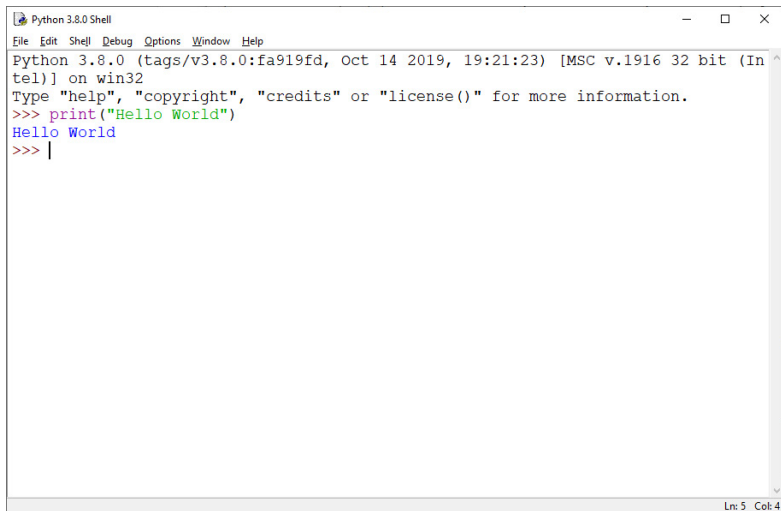
Figura 12 – O ambiente IDLE.

Recomendo contudo a utilização dos ambientes Pycharm da JetBrains ou Visual Studio Code. Pois permitem uma melhor produtividade e contêm excelentes funcionalidades. Veja a Figura 13.



Figura 13 – Tela inicial do Pycharm.

Após instalação da linguagem Python, para que
vejas veja o teu primeiro programa a funcionar, vamos escrever
um “Hello World”, conforme exemplo da Figura 14.

A screenshot of a Python 3.8.0 Shell window. The title bar reads "Python 3.8.0 Shell". The menu bar includes "File", "Edit", "Shell", "Debug", "Options", "Window", and "Help". The main text area shows the following: "Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:21:23) [MSC v.1916 32 bit (Intel)] on win32", "Type 'help', 'copyright', 'credits' or 'license()' for more information.", a prompt ">>>" followed by the command "print('Hello World')", the output "Hello World", and another prompt ">>>" with a cursor. The status bar at the bottom right indicates "Ln: 5 Col: 4".

```
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:21:23) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> print("Hello World")
Hello World
>>> |
```

Figura 14 – Exemplo de programa “Hello World” no Python.

A essa “altura do campeonato”, já deves ter executado o seu primeiro programa, que na verdade foi apenas um comando Python.

Esse programa foi executado na consola da linguagem Python, ou seja, um lugar onde escreve comandos, pressiona ENTER e eles são executados. Caso queira criar um módulo Python, que nada mais é do que uma sequência de comandos num arquivo, clique em “File”, em seguida “New File” e começa a programar, conforme Figura 15. Quando concluir o programa, pressiona a tecla F5, nomeia o arquivo e observa o resultado do seu programa.

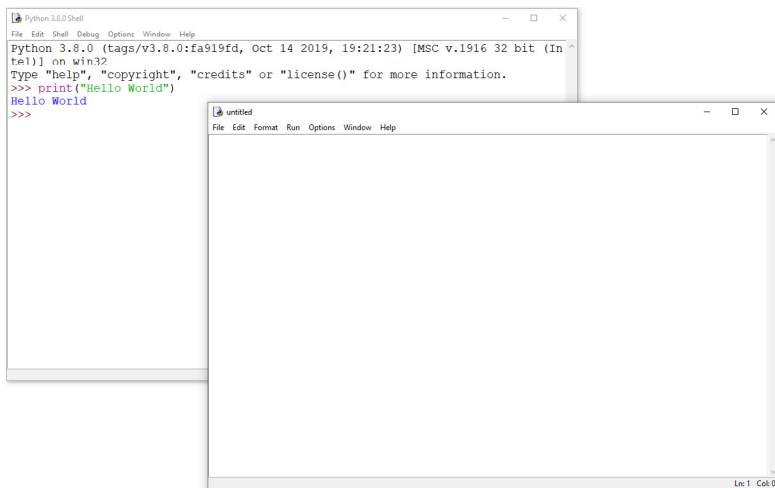


Figura 15 – Local onde se criam arquivos Python na ferramenta IDLE.

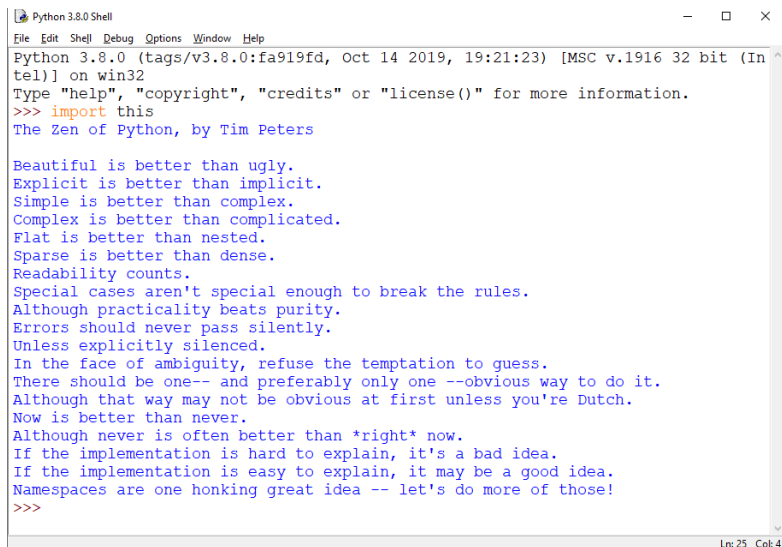
CARACTERÍSTICAS DA LINGUAGEM PYTHON

Python é conhecida como uma linguagem de aspectos bastante interessantes e de fácil aprendizagem. O objetivo inicial da linguagem era permitir código limpo e menos verboso, ou seja, com menos caracteres especiais, menos sintaxes complexas e mais estruturas de código simples. Por isso, se destaca:

- A facilidade para aprender, ler e compreender;
- Ser multiplataforma;
- Possuir modo interativo;
- Usa indentação para marcação de blocos;
- Quase nenhum uso de palavras-chave associadas com compilação;
- Possuir coletor de lixo para gerenciar automaticamente o uso de memória;
- Programação orientada a objetos;

- Programação funcional; e
- Uma imensidão de módulos de extensão, os quais permitem expandir o poder da linguagem Python.

Para muitos, Python possui uma filosofia de programação. Parte da cultura da linguagem gira ao redor de *The Zen of Python*, que é um poema escrito pelo programador Tim Peters. Podes conhecê-lo simplesmente digitando o `"easter egg"` `import this`. Veja a Figura 16.



```
Python 3.8.0 Shell
File Edit Shell Debug Options Window Help
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:21:23) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
>>>
```

Figura 16 – O poema Zen.

A linguagem Python tem sido a preferida para ensinar programação aos iniciantes. Universidades americanas como MIT, Stanford, Michigan e Rice mudaram para Python os seus cursos iniciais de programação. Universidade brasileiras renomadas, como IME, POLI-USP, Unicamp e UFRN também usam

Python nos seus cursos iniciais.

Aos que gostam de videoaulas, existem diversas iniciativas para o ensino de Python. Dentre as que se destacam, há o curso Python para Zumbis² do professor Fernando Masanori³,

o curso Python Birds⁴ do professor Renzo Nuccitelli⁵, e o Python Básico – Curso em Vídeo⁶ do professor Gustavo Guanabara⁷.

Além das importantes iniciativas dos professores Masanori, Nuccitelli e Guanabara, no portal Youtube, existe também um canal chamado Procópio na Rede⁸. L

COMO ESTÁ ORGANIZADO O LIVRO?

2 <https://www.pycursos.com/python-para-zumbis>

3 <https://twitter.com/fmasanori>

4 <https://www.python.pro.br/curso-de-python-gratis>

5 <https://twitter.com/renzoprobr>

6 <https://www.cursoemvideo.com/course/curso-python-3>

7 <https://twitter.com/guanabara>

8 <https://www.youtube.com/ProcopioNaRede>

O livro está organizado da seguinte forma: o Capítulo 2 trata das estruturas sequenciais de programação; o Capítulo 3 discorre sobre estruturas condicionais. O Capítulo 4 envolve as estruturas de repetição. No Capítulo 5, são trabalhadas as estruturas de dados conhecidas como listas. O Capítulo 6 explora uma outra estrutura de dados chamada dicionários. No Capítulo 7, mergulhamos no assunto intitulado funções. No Capítulo 8, fazemos a apresentação de um outro paradigma de programação conhecido como orientação a objetos. Por fim, no Capítulo 9, encerramos o livro com “chave de ouro” discutindo várias aplicações de sistemas operativos utilizando Py-thon.

EXECUÇÃO SEQUENCIAL

Este capítulo é iniciado com a apresentação dos conceitos de variáveis e de constantes. Na sequência, discutiremos os tipos de dados definidos na linguagem Python. Depois apresentaremos os operadores aritméticos da linguagem. Continuaremos com a apresentação da função `input()`, que é utilizada para receber dados de entrada, bem como algumas funções de conversão de tipos. Também apresentaremos *f-Strings*, uma forma melhorada para formatar **strings**. Ainda resolveremos algumas atividades e, por fim, serão propostos alguns exercícios.

VARIÁVEIS E CONSTANTES

No contexto das linguagens de programação, uma variável é um espaço alocado na memória de um computador. O papel de uma variável é armazenar dados ou expressões que estão sendo utilizados por um programa em execução. Variáveis existem durante um determinado espaço de tempo, isto é, do início até o fim da execução do programa que as declarou.

Para facilitar o entendimento de o que vem a ser uma variável, vamos fazer uma analogia com base em uma situação do nosso cotidiano. Imagine uma aula que acontece dentro de uma sala tradicional. Quando a aula é iniciada, várias cadeiras vão sendo ocupadas por alunos que assistirão a esta

aula. Eventualmente, por algum motivo, um determinado aluno poderá sair da sala de aula e, portanto, a sua cadeira ficará disponível para outro aluno. É possível que a cadeira desocupada anteriormente seja preenchida por um aluno que já está participando da aula ou por um aluno que acabou de entrar na sala de aula. A partir desse cenário hipotético, se fizermos uma comparação simplista com um programa, a sala de aula seria a memória do computador, as cadeiras seriam as variáveis e os alunos seriam os dados armazenados dentro das variáveis.

Para localizar a posição de uma variável dentro da memória, um computador utiliza um endereço na forma de um número na base hexadecimal⁹. No entanto, as linguagens de programação permitem que os programadores associem esses endereços de memória por meio da definição de nomes às variáveis. Em outras palavras, em vez de referenciar uma posição de memória por meio de um endereço hexadecimal como **0022EF71**, as linguagens de programação fazem referência a variáveis apenas declarando um nome para elas como, por exemplo, **aluno**.

Em Python, podemos declarar uma variável simplesmente definindo-lhe um nome e atribuindo-lhe um valor. Atribuir valores a variáveis nada mais é do que armazenar um valor (ou um conjunto deles) em um determinado endereço de memória referenciando-o por meio de um nome, conforme o Programa 1.

9 A base hexadecimal é um sistema de numeração que utiliza 16 símbolos para representação dos números. Além dos símbolos que fazem parte do sistema decimal (0, 1, 2, 3, 4, 5, 6, 7, 8 e 9), o sistema hexadecimal também inclui A, B, C, D, E e F.

```
1 aluno = "Ada Lovelace"
2 periodo_corrente = 10
3 ira = 9.99 #valor que define o IRA
4 laurea = True
```

Programa 1 – Declaração de variáveis e comentários.

Na linha 1, é declarada uma variável chamada **aluno**, cujo valor "**Ada Lovelace**", chamada de cadeia de caracteres ou simplesmente de **string**¹⁰, é atribuído. Na linha 2, a variável **periodo_corrente** é declarada e o valor que lhe é atribuído é o número inteiro 10. Já na 3, declara-se a variável **ira** (índice de rendimento acadêmico) cujo valor atribuído é o número real 9.99. Por fim, na linha 4, a variável **laurea** armazena o valor lógico **True**.

Observe que as variáveis declaradas no Programa 1 apresentam nomes significativos para o tipo de informação que desejamos armazenar sobre um aluno: nome, período corrente, índice de rendimento acadêmico (IRA) e se ele recebeu a láurea acadêmica. A definição de nomes semânticos para variáveis é sempre recomendada porque eles contribuem com a documentação do programa. Por inúmeras vezes, vemos alunos declarando nomes de variáveis que não apresentam o menor significado para a informação que está sendo armazenada. Por exemplo, se as variáveis **aluno**, **periodo_corrente**, **ira** e **laurea** fossem substituídos por **x**, **y**, **z** e **w**, respectivamente, e alguém lhe dissesse "Em **x**, é armazenada a **string** Alan Turing, em **y** o número 10, em **z** o número 9.98 e em **w** o valor **False**." conseguiria deduzir que se trata do nome de um

10 Para o interpretador Python, qualquer coisa que seja delimitada por um par de aspas simples ou dupla é considerada uma **string**. Por exemplo, "Olá", "3", '3.14' e 'Verdadeiro' são valores do tipo **string**.

aluno, do seu período corrente, do seu IRA e de que ele não foi o aluno laureado de sua turma? Certamente não. perce-be a importância da definição de nomes significativos para as variáveis?

Em Python, convencionou-se que o nome de uma variável deve começar com letras minúsculas como `aluno`, `periodo_corrente`, `ira` e `laurea`. Caso o nome de uma variável possua mais de uma palavra, deve-se separá-las com um *underscore* (`_`) como `periodo_corrente` ou `indice_rendimento_academico`. Essa última padronização é chamada de *snake case*. Há nomes de variáveis que não são permitidos em Python. Variáveis cujos nomes começam com números (por exemplo, `2aluno`) não são permitidas, embora `aluno2` seja um nome válido. Além disso, deve-se observar também que as palavras reservadas da linguagem Python não podem ser usadas como nomes de variáveis. Portanto, não é permitido declarar variáveis com os seguintes nomes:

<code>and</code>	<code>as</code>	<code>assert</code>	<code>break</code>	<code>class</code>	<code>continue</code>	<code>def</code>
<code>del</code>	<code>elif</code>	<code>else</code>	<code>except</code>	<code>False</code>	<code>finally</code>	<code>for</code>
<code>from</code>	<code>global</code>	<code>if</code>	<code>import</code>	<code>in</code>	<code>is</code>	<code>lambda</code>
<code>None</code>	<code>nonlocal</code>	<code>not</code>	<code>or</code>	<code>pass</code>	<code>raise</code>	<code>return</code>
<code>True</code>	<code>try</code>	<code>while</code>	<code>with</code>	<code>yield</code>		

Outra das características da linguagem Python é que ela é *case sensitive*. deve estar se perguntando “O que significa *case sensitive*?”. Então, vamos lá. Ao acessar a sua rede social, por exemplo, informa o seu utilizador e a sua senha, certo? Supondo que a sua senha é `python` e que, no momento de informar sua senha, digitou `Python`, o seu acesso será

libertado? Claro que não, não é verdade? Isso ocorre porque o processo de autenticação de utilizadores é do tipo *case sensitive*, isto é, há distinção de letras minúsculas para letras maiúsculas. Assim, em um programa, a variável `var1` ocupa uma posição de memória diferente da ocupada pela variável `Var1`. Portanto, `var1` e `Var1` são variáveis distintas.

Outro conceito importante que devemos ter em mente é o de constantes. Ao contrário de variáveis, cujos valores variam durante a execução de um programa, quando um valor é atribuído a um constante, este não poderá ser modificado até que o programa seja encerrado. Por exemplo, o valor aproximado de π 3.14159 e, portanto, ele nunca deverá ser modificado.

Python possui uma biblioteca chamada `math`, na qual são definidas algumas constantes matemáticas como `pi`, `tau` e o número de Euler. Para usar uma constante matemática, definida dentro da biblioteca `math`, basta escrever `from math import <nome_da_constant>`, como no exemplo a seguir.

```
1  from math import pi
2  from math import tau, e
3  print(pi)
4  print(tau, e)
```

Programa 2 – Exemplo de utilização da constante `pi`.

Na linha 1, é importada a constante `pi`. Na linha 2, são importadas simultaneamente o valor de `tau` e do número de Euler. Nas duas linhas seguintes, os valores das respectivas constantes são escritas na tela.

Mas o que fazer quando precisa a criar a sua própria constante, aquela que não é identificada pelo Python? Imagine a situação em que construiu um programa o qual, por padrão, no rodapé de todos os relatórios emitidos, deve aparecer “Programação Estruturada e Orientada a Objetos em Python”. Python não possui uma maneira explícita de declarar uma constante, diferentemente de outras linguagens de programação como C, C++ e Java, as quais usam as palavras reservadas **define**, **define** e **final**, respectivamente.

Voltando ao exemplo da emissão de relatórios, uma estratégia para “simular” o comportamento de uma constante seria, por exemplo, declarar uma função para retornar a **string** Curso de Programação Estruturada e Orientada a Objetos em Python ou simplesmente definir uma variável no início do programa e, em nenhum momento, alterar o seu valor.

```
1  #rodape será uma constante. Assim, seu valor não será alterado
2  rodape = "Programação Estruturada e Orientada a Objetos em Python"
3  print(rodape)
4  constante = obter_rodape()
5  print(constante)
6  def obter_rodape()
7  return "Programação Estruturada e Orientada a Objetos em Python"
```

Programa 3 – Exemplo da declaração de uma constante.

Na linha 2, uma **string** é atribuída à **rodape** e, como o seu valor não é modificado ao longo do programa, **rodape** possui o comportamento de uma constante. É declarada uma função chamada **obtem_rodape()**, nas linhas 06 e 07, que retorna Programação Estruturada e Orientada a Objetos em Python. Portanto, toda vez que for necessário obter essa **string** para escrever no rodapé dos relatórios, basta invocar a função

`obtem_rodape()`, como mostrado na linha 4 do Programa 3, ou utilizar a variável `rodape`, que possui o comportamento de uma constante.

TIPOS DE DADOS

As linguagens de programação, em geral, usam o termo “tipo primitivo” para representar a informação em sua forma mais elementar, tais como inteiro, real, lógico ou caractere. Na documentação oficial da linguagem Python¹¹, o termo “tipo primitivo” não é utilizado, mas sim “tipos *built-ins*” (ou tipos cons-truídos). O motivo é que, para Python, tudo é um objeto.

Também é importante entender que Python pos-sui vários tipos de dados e os principais deles são:

- **int** – armazena valores numéricos inteiros
- **float** – armazena valores numéricos com ponto flutuante
- **complex** – armazena valores numéricos complexos
- **bool** – armazena valores lógicos (True ou False). O valor True pode ser representado por 1 e o False por 0 e, por isso, alguns autores consideram valores do tipo **bool** como sendo do tipo inteiro

¹¹ Para consultar a documentação completa da linguagem Python, acesse o endereço <https://docs.python.org/3.7/>.

- **str** – armazena cadeias de caracteres
- **list** – armazena conjuntos de elementos que podem ser acessados por meio de um índice
- **dic** – armazena um conjunto de elementos que podem ser acessados por meio de uma chave.

Diferentemente de outras linguagens de programação como C, C++ e Java, Python possui tipagem dinâmica. Uma linguagem de programação que possui tipagem dinâmica como Python, PHP ou Perl não exige que o programador declare, explicitamente, o tipo de dado que será armazenado por cada variável. Essa característica permite que, ao longo da execução de um programa, uma mesma variável armazene valores de tipos distintos. Observe o exemplo no Programa 4.

```
1 exemplo = "Ada Lovelace"
2 exemplo = 10
3 exemplo = 9.99
4 exemplo = True
```

Programa 4 – Alteração de tipo de dados das variáveis.

Veja que, em cada uma das linhas do Programa 4, a variável **exemplo** sofre alteração no tipo de dado que é armazenado. Na primeira linha, é armazenado um valor do tipo **string**. Na segunda, um número inteiro. Na terceira, um número real e, na última linha, um valor lógico.

Para simplificar o que foi mencionado, vamos apresentar alguns exemplos utilizando a função **type()**, conforme Programa 5. **type()** é uma função que recebe como entrada um valor uma variável ou um objeto (por enquanto, vamos deixar o conceito de objeto para outro momento) e retorna a qual tipo o valor (ou a variável) pertence.

```

1  exemplo = "Ada Lovelace"
2  print(type(exemplo)) #Imprime <class 'str'>
3  '''Ada despertou uma admiração pelos trabalhos desenvolvidos por é
4  Charles Babbage que resultou no interesse de ser sua aluna. Babbage
5  considerado o projetista do primeiro computador de uso geral.'''
6  exemplo = 10
7  print(type(exemplo)) #Imprime <class 'int'>
8  exemplo = 9.99
9  print(type(exemplo)) #Imprime <class 'float'>
10 exemplo = True
11 print(type(exemplo)) #Imprime <class 'bool'>

```

Programa 5 – Uso da função `type()`.

Na linha 1, a **string** Ada Lovelace é atribuída à variável **exemplo**. Na linha seguinte, `type()` recebe como argumento o conteúdo armazenado em **exemplo** e retorna uma **string** informando que o valor armazenado é um objeto da classe **str**, isto é, `<class 'str'>`. De maneira semelhante, o mesmo acontece nas linhas 07, 09 e 11, cujos valores retornados pela função `type()` são `<class 'int'>`, `<class 'float'>` e `<class 'bool'>`, respectivamente.

Observe que nas linhas 02, 07, 09 e 11, aparecem `#Imprime <class 'str'>`, `#Imprime <class 'int'>`, `#Imprime <class 'float'>` e `#Imprime <class 'bool'>`, respectivamente. O que aparece depois de `#` é chamado de comentário. Em programação, os comentários, como o nome sugere, são notas adicionadas ao programa a fim de descrever, por exemplo, como funciona um determinado trecho dele. Outra utilidade dos comentários é garantir que algum trecho do programa não seja executado (porque ainda não foi concluído, porque apresenta erros ou porque ainda não foi exaustivamente testa-

do). É importante destacar que os comentários não são executados pelos programas.

Ainda no Programa 5, observe que nas linhas 03 – 05, aparece `'''Ada despertou uma admiração pelos trabalhos desenvolvidos por Charles Babbage que resultou no interesse de ser sua aluna. Babbage é considerado o projetista do primeiro computador de uso geral.'''`. Isso também é um comentário, porém ele é utilizado quando se deseja comentar mais de uma linha. Observe que esse tipo de comentário começa e termina com três aspas simples `'''`.

OPERADORES ARITMÉTICOS

Python oferece diversos conjuntos de operadores que podem ser utilizados em um programa. Para realizar operações matemáticas, podemos utilizar os operadores aritméticos, como mostrado no Quadro 1.

OPERADOR	DESCRIÇÃO	EXEMPLO DE APLICAÇÃO
+	Adição	<code>print(4 + 2)</code> <code>#resulta em 6</code>
-	Subtração	<code>print(4 - 2)</code> <code>#resulta em 2</code>
*	Multiplicação	<code>print(4 * 2)</code> <code>#resulta em 8</code>
/	Divisão	<code>print(4 / 3)</code> <code>#resulta em 1.3333</code>
//	Quociente inteiro da divisão	<code>print(4 // 3)</code> <code>#resulta em 1</code>

OPERADOR	DESCRIÇÃO	EXEMPLO DE APLICAÇÃO
%	Resto da divisão inteira	<code>print(4 % 2)</code> #resulta em 0
**	Potenciação	<code>print(4 ** 2)</code> #resulta em 16

Quadro 1 – Lista de operadores aritméticos.

Além de realizar as operações apresentadas no Quadro 1, os operadores de adição e de multiplicação apresentam um comportamento diferente quando são utilizadas **strings**. Por exemplo, o comando `print("Olá, " + "mundo!")` concatena a **string** `Olá` com a **string** `mundo!` e resulta em `Olá, mundo!`. Por outro lado, o operador de multiplicação, quando combinado a **strings**, repete a **string** N vezes, por exemplo, `print(2 * "ABC")` replica 2 vezes a **string** `ABC`, resultando em `ABCABC`.

Por fim, o operador de potenciação ****** pode ser substituído pela função `pow(base, exp)`, que recebe como entrada valores correspondentes à base e ao expoente. Assim para calcular 4^2 , pode-se escrever `4 ** 2` ou `pow(4, 2)`.

OPERADORES DE ATRIBUIÇÃO

Já que acabamos de tratar dos operadores aritméticos, agora apresentaremos os operadores de atribuição, desde aquele mais simples e direto – como o sinal de igual (=) que, nós, professores, gostamos de chamar de “recebe” – até as abreviações envolvendo operadores aritméticos e de atribuição.

OPERADOR	DESCRIÇÃO	EXEMPLO DE APLICAÇÃO
=	Atribuição simples	x = 2 #x recebe 2
+=	Atribuição de adição	x += 2 #equivale a x = x + 2
-=	Atribuição de subtração	x -= 2 #equivale a x = x - 2
*=	Atribuição de multiplicação	x *= 2 #equivale a x = x * 2
/=	Atribuição de divisão	x/= 2 #equivale a x = x / 2
%=	Atribuição de resto inteiro da divisão	x%= 2 #equivale a x = x % 2
=	Atribuição de potência	x= 2 #equivale a x = x ** 2
//=	Atribuição de quociente inteiro da divisão	x//= 2 #equivale a x = x // 2

Quadro 2 – Lista de operadores de atribuição.

ENTRADA DE DADOS E CONVERSÃO DE TIPOS

Até então, vínhamos declarando uma variável atribuindo valores a elas. No entanto, em um cenário real, geralmente o utilizador interage com o programa informando dados de entrada. Em Python, utiliza-se a função `input()`, que é exemplificada no Programa 6.

```

1  aluno = input("Digite seu nome: ")
2  periodo_corrente = input("Digite seu período corrente: ")
3  ira = input("Digite seu IRA(Índice de Rendimento Acadêmico): ")
4laurea = input("Informe se é o aluno laureado(a) da turma: ")

```

Programa 6 – Uso da função `input()`.

É importante destacar que a função `input()` sempre retornará o valor recebido no formato de uma `string`, mesmo que ele seja de um tipo de dado diferente como, por exemplo, um número real. Vamos testar?

```
1  aluno = input("Digite seu nome: ")
2  periodo_corrente = input("Digite seu período corrente: ")
3  ira = input("Digite seu IRA(Índice de Rendimento Acadêmico): ")
4  laurea = input("Informe se é o aluno laureado(a) da turma: ")
5  print(type(aluno))
6  print(type(periodo_corrente))
7  print(type(ira))
8  print(type(laurea))
```

Programa 7 – Verificação do tipo de dado armazenado usando a função `input()`.

Após executar o Programa 7 e considerando que o utilizador informou `"Ada Lovelace"`, `10`, `9.99` e `True`, observará que as variáveis `periodo_corrente`, `ira` e `laurea` armazenam valores do tipo `string`, embora a intenção do utilizador tenha sido digitar um número inteiro, um número real e um valor lógico, respectivamente. Como contornar esse problema? Fácil! Basta usar as funções de conversão `int()`, `float()` e `bool()`, respectivamente.

Os nomes das funções de conversão são bastante sugestivos. Além das apresentadas anteriormente, há outras como `str()` e `complex()`. Elas recebem como entrada um valor de um determinado tipo e tenta convertê-lo para uma `string` ou um número complexo. Então, para que as variáveis `periodo_corrente`, `ira` e `laurea` armazenem os tipos de dados desejados pelo utilizador, o Programa 8 pode

ser reescrito como segue:

```
1  aluno = input("Digite seu nome: ")
2  periodo_corrente = int(input("Digite seu período corrente: "))
3  ira = float(input("Digite seu IRA(Índice de Rend. Acadêmico): "))
4  laurea = bool(input("Informe se foi laureado(a) na turma: "))
5  print(type(aluno))
6  print(type(periodo_corrente))
7  print(type(ira))
8  print(type(laurea))
```

Programa 8 – Uso das funções de conversão.

Deve ter percebido que os valores digitados foram convertidos e, de fato, agora são do tipo inteiro, real e lógico, respectivamente. No entanto, deve estar a pensar: “Mas se eu não tivesse feito as conversões não haveria problema no meu programa.” É verdade. Agora imagine um programa que foi escrito para calcular a área de um retângulo.

Relembrando das aulas de geometria, sabemos que a área de um retângulo é o produto entre a sua base e a sua altura. Então, para entender a importância da conversão de tipos, vamos tentar calcular a área da figura sem fazer as conversões. Observe que o Programa 9 produzirá uma mensagem de erro.

```
1  base = input("Digite a base: ")
2  altura = input("Digite a altura: ")
3  area = base * altura
4  print(area)
```

Programa 9 – Cálculo da área do retângulo.

De uma forma simples, a mensagem apresentada diz

que não é possível realizar a multiplicação e o motivo disso é o fato de o multiplicador e o multiplicando serem **strings**. Então, para que o programa calcule corretamente a área do retângulo, os valores atribuídos às variáveis **base** e **altura** devem ser convertidos para um tipo numérico. Como exercício, o Programa 9 foi modificado para converter os valores da base e da altura em um número real usando a função **float()**. Para isso, a linha 1 deve ser reescrita para **base = float(input("Digite a base: "))** e a linha 2 para **altura = float(input("Digite a altura: "))**. Depois da modificação, execute o novo código e verifique que o programa calculou a área da figura corretamente.

FORMATAÇÃO DE STRINGS

Quando trabalhamos com **strings**, é bastante comum querermos formatá-las e, para isso, Python oferece diversas maneiras, tais como os marcadores de posição **%**, o método **format()** e a classe **Template**. No entanto, uma das maneiras mais simples de implementar a formatação de **strings** é utilizando a *Literal Strings Interpolation* ou, simplesmente, **f-Strings**. Foram incluídas na versão Python 3.6.

A sintaxe das **f-Strings** é bastante simples e o seu uso garante a incorporação de expressões dentro do texto

literal. É importante observar também que elas garantem a execução das expressões em tempo de execução.

Em um programa Python, as *f-Strings* são iniciadas com a letra **f** ou **F**, contendo expressões envolvidas por um par de chaves {...}, modificadas dentro da **string** a ser formatada. As *f-Strings* consideram tudo que está fora do par de chaves como sendo um texto literal e, portanto, na saída, o texto será replicado sem nenhuma alteração.

A maneira mais simples de formatar uma **string** é informando o(s) valor(es) que a comporá(ão), conforme exemplo a seguir:

```
1  from datetime import datetime
2  ano_atual = datetime.now().year
3  clube     = "SPFC"
4  campeonato_mundial = 3
5  ano_fundacao = 1930
6  print(f"{clube} possui {campeonato_mundial} títulos mundiais.")
7  print(F"São {ano_atual - ano_fundacao} anos de existência.")
```

Programa 10 – Um exemplo de utilização de *f-Strings*

Na linha 1, o módulo **datetime** é importado para que a classe com o mesmo nome seja utilizada. Na linha 2, o programa obtém o ano atual em curso. Nas linhas 3 – 5 são atribuídos valores às variáveis **clube**, **campeonato_mundial** e **ano_fundacao**, respectivamente. Observe que, na linha 6, as posições em que se encontram as expressões {clube} e {campeonato_mundial}, serão substituídas pelos seus respectivos valores, isto é, SPFC e 3. Já na linha 7, a expressão {ano_atual - ano_fundacao} será calculada

em tempo de execução. Considerando que este programa está sendo executado em 07 de novembro de 2020, o resultado da expressão é 90. Portanto, ao fim da execução do Programa 10, será exibido o seguinte:

```
SPFC possui 3 títulos mundiais.  
São 90 anos de existência.
```

Uma outra situação é o caso de a **string** a ser exibida possuir várias linhas. No exemplo abaixo, observe que reescrevemos o conteúdo exibido pelos comandos executados nas linhas 6 e 7, do Programa 10.

```
SPFC foi fundado em 30 de janeiro de  
1930. Em seus 90 anos de história, ob-  
teve 3 títulos mundiais.
```

Para modificarmos as **strings** exibidas pelo Programa 10, removeríamos as linhas 6 e 7 e incluiríamos o seguinte trecho de código:

```
print(f"{clube} foi fundado em 30 de  
janeiro de {ano_fundacao}."  
f" Em seus {ano_atual - ano_fundacao} anos  
de história,"  
f" obteve {campeonato_mundial}  
títulos mundiais.")
```

Perceba que, antes de cada uma das 3 linhas contendo as **strings**, adicionamos a letra **f**. Como exercício,

modifique o Programa 10 e verifique o resultado. No entanto, se eventualmente a letra `£` não for incluída, Python não conseguirá resolver a expressão que está dentro do par de chaves. O comando abaixo

```
print(f"{clube} foi fundado em 30 de  
janeiro de {ano_fundacao}."  
      " Em seus {ano_atual - ano_fundacao} anos de história,"  
      " obteve {campeonato_mundial} títulos mundiais.")
```

produziria, como saída, a seguinte `string`:

```
SPFC foi fundado em 30 de janeiro de  
1930. Em seus {ano_atual - ano_fun-  
dacao} anos de história, obteve {cam-  
peonato_mundial} títulos mundiais.
```

Agora, imagine uma situação em que, por algum motivo, precisamos formatar um número inteiro de modo que, se o valor for menor ou igual a 99.999, isto é, se o número possuir até 5 dígitos, devem ser preenchidos zeros à esquerda para que o número exibido sempre possua 6 dígitos. Observe os exemplos abaixo:

```
print(f"Matrícula do Charles Babbage é {25:06d}.")  
print(f"Matrícula do Alan Turing é {99999:06d}.")  
print(f"Matrícula da Ada Lovelace é {100387:06d}.")
```

Nos três exemplos anteriores, `%6d` diz “Exiba um número inteiro com 6 dígitos. Se ele possuir menos de 6 dígitos, preencha com zeros à esquerda até deixá-lo com 6 dígitos.”.

Também é possível formatar as **strings** incluindo valores que representam números com ponto flutuante. Veja o exemplo abaixo:

```
print(f"Valor de pi é {3.14159265352384626433}.")
```

Observe que o valor informado possui 20 casas decimais, no entanto, ao executar esse comando, Python fará o truncamento para 16 casas decimais. E se desejasse trabalhar, realmente, com a precisão de 20 casas decimais? Como faria? Bastaria informar que deseja utilizar um valor com ponto flutuante e a quantidade de casas decimais. No comando abaixo, fazemos essa especificação escrevendo `:.20f`. Execute o comando e veja o resultado.

```
print(F"Valor de pi é {3.14159265352384626433: .20f}.")
```

EXERCÍCIOS RESOLVIDOS

1. Construa um programa no qual um utilizador informe a sua estatura em metros e o programa converta-a para centímetros.

Comentários sobre a resolução

Para resolver esta questão, basta executar uma operação direta de conversão. Sabendo que o utilizador informará uma estatura em metros, o programa deverá convertê-la para centímetros apenas multiplicando-a por 100.

Vamos ao código:

```
1  estatura = float(input("Digite a sua estatura (em metros): "))
2  estatura = estatura * 100
3  print(f"Sua estatura é de {estatura} cm.")
```

Programa 11 – Exercício resolvido 2.1

2. Construa um programa que receba do utilizador a variação do deslocamento de um objeto (em metros) e a variação do tempo percorrido (em segundo). Ao fim, o programa deve calcular a velocidade média, em m/s, do objeto.

Comentários sobre a resolução

Recordando das aulas de Cinemática, sabemos que $V_m = \frac{\Delta s}{\Delta t}$.

O enunciado da questão diz que o utilizador informará a variação do deslocamento (isto é, Δs) e a variação do tempo (ou seja, Δt). Então, para calcular o que foi pedido, basta dividir o primeiro valor informado pelo utilizador pelo segundo valor. No entanto, é importante relembrar que a) `input()` é usada para receber os dados de entrada e b) ela sempre retorna uma **string**. Portanto, é preciso fazer as conversões dos valores de entrada para **float**, porque, por exemplo, o deslocamento pode ser 10,23 metros (valor real) e/ou o tempo percorrido 2,10 segundos (valor real).

Lembre-se que, em um valor real, por conta do separador decimal americano, quem separa a parte inteira da decimal é o ponto e não a vírgula. Portanto 3,14 deve ser substituído por 3.14.

Vamos ao código:

```
1  delta_s = float(input("Digite o deslocamento (em metros): "))
2  delta_t = float(input("Digite o tempo (em segundos): "))
3  velocidade = delta_s / delta_t
4  print(f"Vm = {velocidade:.2f} m/s")
```

Programa 12 – Exercício resolvido 2.2

3. Construa um programa para calcular a área de convivência de uma escola cujo formato é circular. Para isso, o utilizador deve informar o valor do raio.

Comentários sobre a resolução

Relembrando do conteúdo estudado em Geometria, a área limitada por uma circunferência corresponde ao produto entre a constante e o raio da circunferência, isto é, $\text{Área}_{\text{circ}} = \pi R^2$.

Vamos ao código:

```
1  from math import pi
2  raio = float(input("Digite o raio da área: "))
3  area = pi * raio ** 2 # mesmo que area = pi * pow(raio, 2)
4  print(f"Área = {area:.2f}.")
```

Programa 13 – Exercício resolvido 2.6.3

EXERCÍCIOS PROPOSTOS

1. Um aluno iniciou seus estudos em geometria plana e, para validar se suas respostas estão corretas, solicitou sua ajuda. Sabendo que $\text{Área}_{\text{triangulo}} = \frac{(\text{base} * \text{altura})}{2}$, construa um programa para auxiliar esse aluno.

2. Agora o mesmo aluno precisa que construa um programa para calcular o comprimento de uma circunferência. Para isso, o aluno informará ao programa o raio da circunferência.

Sabe-se que $Comp_{circ} = 2\pi r$.

3. Construa um programa que leia 2 números reais informados pelo utilizador. Ao fim, o programa deve calcular e imprimir:

- a. a soma dos dois valores
- b. o produto entre eles

4. Implemente um programa que converta o valor de uma velocidade média em km/h para m/s. Para isso, o utilizador deve informar o valor da velocidade média. Sabe-se que o fator utilizado para essa conversão é 3,6.

5. Construa um programa que tem como dados de entrada dois pontos quaisquer no plano cartesiano: P1 e P2. Considere que P1 é definido pelas coordenadas x_1 e y_1 , enquanto P2 por x_2 e y_2 . O programa deve calcular e escrever a distância entre os pontos P1 e P2. A fórmula que calcula a distância entre os dois pontos é dada por:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Para receber os dados de entrada, use as variáveis x_1 , y_1 , x_2 e y_2 .

Dica: A função que calcula a raiz quadrada de um número real é a `sqrt` (*square root*). Para usá-la, importe-a da biblioteca `math` da seguinte forma: `from math import sqrt`.

6. Considere uma equação do segundo grau, representada pela expressão

$$ax^2 + bx + c = 0.$$

Construa um programa no qual o utilizador informe os valores das constantes a, b e c. Ao fim, o programa deve calcular e imprimir o valor de $\Delta = b^2 - 4ac$.

7. Uma imobiliária paga aos seus corretores um salário base de R\$ 1.500,00. Além disso, uma comissão de R\$ 200,00 por cada imóvel vendido e 5% do valor de cada venda. Construa um programa que solicite o nome do corretor, a quantidade de imóveis vendidos e o valor total de suas vendas. Ao fim, o programa deve calcular e escrever o salário final do corretor de imóveis.

8. Construa um programa que receba do utilizador o valor do salário mínimo atual. Na sequência, o programa deve solicitar que o utilizador informe o valor do seu salário mensal. Ao fim, o programa deve calcular a quantidade de salários mínimos re-cebidos pelo utilizador.

9. Construa um programa que solicite que o utilizador informe 2 números inteiros positivos. O programa deve calcular:

- a) O cubo do segundo número
- b) A média geométrica entre o primeiro e o segundo número, isto é,

$$média_{geo} = \sqrt[2]{(num_1) * (num_2)}$$

10. Imagine a situação em que existem 2 copos com sumos de fruta. O primeiro copo está com sumo de laranja, enquanto o segundo está com sumo de maçã. Desejaa mudar os sumos de copo, isto é, colocar o suco de laranja no segundo copo e o suco de acerola no primeiro copo. No entanto, não é desejável que eles se misturem.

Agora, vamos transformar esta situação em um programa. Nas duas linhas abaixo, nós colocamos o suco de laranja no copo1 e o suco de acerola no copo2:

```
copo1="laranja"
```

```
copo2 = "maçã"
```

Continue o programa de modo a transferir o suco de maçã para o **copo1** e o suco de laranja para o **copo2**. Ao fim, imprima as variáveis **sumo1** e **sumo2**.

ESTRUTURAS CONDICIONAIS

Este capítulo trata das estruturas condicionais ou, simplesmente, desvios do fluxo de execução de um programa. Estamos falando das estruturas: `if`, `else` e `elif`. No começo deste capítulo, trataremos das explicações necessárias ao entendimento das estruturas condicionais. Na sequência, apresentaremos os operadores comparativos/relacionais e lógicos. Logo em seguida, explicaremos, de forma individualizada, cada estrutura e sua sintaxe. Concluiremos comentando exercícios resolvidos.

ONDE ENCONTRAMOS ESTRUTURAS CONDICIONAIS?

Conforme observado no capítulo anterior, por padrão, instruções de um programa Python são executadas na ordem em que foram inseridas no código, ou seja, uma após a outra, do início ao fim. Esse procedimento é conhecido como execução sequencial. Entretanto, em várias ocasiões, é necessário decidir a ordem de execução dos comandos a partir de condições pré-estabelecidas. Isso acontece, por exemplo, em situações cotidianas. Imagine o seguinte: tem uma consulta com horário marcado e, no deslocamento até o local da consulta, tem que decidir qual o caminho mais rápido para não se atrasar. Bom, conhece dois caminhos, um mais

distante e outro mais curto. Porém, ouve no rádio que o cami-nho mais curto se encontra congestionado. O que fazer? terá que decidir qual caminho deve escolher: o mais longo (sem congestionamento) ou o mais curto (com congestionamento). E aí? Algo similar acontece com os códigos de lingua-gem de programação.

Em outro exemplo, foi convidado a criar um programa para calcular a média aritmética das notas de um aluno qualquer e, em seguida, apresentar o resultado de aprovação do aluno com base nas seguintes regras: se a média aritmética for maior ou igual a 60 o aluno é considerado aprovado, caso contrário estará reprovado. Como resolveria isso? Vamos lá. Seu programa deve receber duas notas, as quais armazenaria em duas variáveis distintas, por exemplo, `nota1` e `nota2`. Em seguida, calcularia a média aritmética das notas e, só agora, se preocuparia com os testes da nota para aprovação ou reprovação.

Até o momento, com os recursos estudados no Capítulo 2, como resolveria isso? Vejamos o Programa 14.

```
1  nota1 = int(input("Informe a nota do bimestre 1 (0-100): "))
2  nota2 = int(input("Informe a nota do bimestre 2 (0-100): "))
3  media = (nota1 + nota2) / 2
4  estado_Aprovacao = media >= 60
5  # O que fazer?
```

Programa 14 – Cálculo da média aritmética de duas notas de um aluno qualquer.

Perceba que, até agora, conseguimos calcular a média e até saber o estado de aprovação, com base em um teste

realizado com a variável **media**. No entanto, não conseguimos avançar com respeito ao resultado da aprovação. Como faremos para apresentar o resultado "Aprovado" ou "Reprovado"? É aí que entram as estruturas condicionais, ou seja, a partir do resultado de um teste condicional, executaremos comandos e/ou deixaremos de executar outros. Certamente, agora é aquele momento que pensa: "Hamm?". Calma! Agora, vamos contar com o auxílio das palavras chaves **if** e **else**. Com elas, conseguiremos decidir se apresentaremos o resultado "**Aprovado**" ou "**Reprovado**", como se vê no Programa 15.

```
1  nota1 = int(input("Informe a nota do bimestre 1 (0-100): "))
2  nota2 = int(input("Informe a nota do bimestre 2 (0-100): "))
3  media = (nota1 + nota2) / 2
4  if media >= 60:
5      print("Aprovado")
6  else:
7      print("Reprovado")
```

Programa 15 – Cálculo da média aritmética de duas notas usando testes condicionais.

Observe que o código reflete nossa estratégia mental de solução do problema. Após a declaração e leitura das variáveis, calculamos a média e testamos esse valor para decidir se o aluno foi aprovado ou reprovado. O referido teste (linhas 4 – 7) pode ser interpretado da seguinte maneira: Se (**if**) a média for maior ou igual a 60, imprima a palavra Aprovado, senão (**else**), imprima a palavra Reprovado. Portanto, a palavra-chave **if** é um teste que vai decidir que comandos serão executados: **print("Aprovado")** ou **print("Reprovado")**.

Para consolidar melhor o seu entendimento, vamos

pensar em outro exemplo. Imagine que, mais uma vez, foi convidado para criar um programa Python, desta vez, para um banco que realiza empréstimos. Para isso, o banco lhe forneceu uma árvore de decisão¹³, conforme Figura 17, indicando quais clientes podem conseguir empréstimo.

Por onde começamos? Inicialmente, idealize como seria a solução do problema com os recursos que já possui na sua “caixa de ferramentas de programação”. Nós fazemos referência aos comandos que já aprendeu, tais como: criação de variáveis, leitura e impressão de valores, estrutura sequencial de comandos e agora, é claro, estruturas condicionais. Mãos à obra!

```
1  print("## Programa de empréstimos ##. Responda: (0 - Não e 1 - Sim) ")
2  nomeNegativado = int(input("Possui nome negativado? "))
3  if nomeNegativado == 1:
4      print("Não pode realizar empréstimo")
5  else:
6      carteiraAssinada = int(input("Possui carteira assinada? "))
7      if carteiraAssinada == 0:
8          print("Não pode realizar empréstimo ")
9      else:
10         possuiCasaPropria = int(input("Possui casa própria? "))
11         if possuiCasaPropria == 0:
12             print("Não pode realizar empréstimo")
13         else:
14             print("Conceder empréstimo")
```

Programa 16 – Exemplo de implementação de uma árvore de decisão.

13 Uma árvore de decisão é uma representação de uma tabela de decisão sob a forma de uma árvore, porém podem existir outras aplicações.

No Programa 16, por meio dos comandos **if** e **else**, observe que foi possível converter a árvore de decisão em um programa que, por sinal, parece muito com a Figura 17. Inicialmente, realizamos a leitura do estado de **nomeNegativado** do cliente. Logo em seguida, verificamos se o estado é negativado ou não.

Provavelmente, já notou que um comando **if** quer saber se algo é verdadeiro ou falso. A utilização de operadores lógicos e de comparação (discutidos logo mais) sempre geram como resultado um valor lógico, isto é, **True** ou **False**. Isso permite decidir que comandos serão executados e quais deixaram de ser. Portanto, no pseudocódigo do Programa 17, temos a seguinte análise.

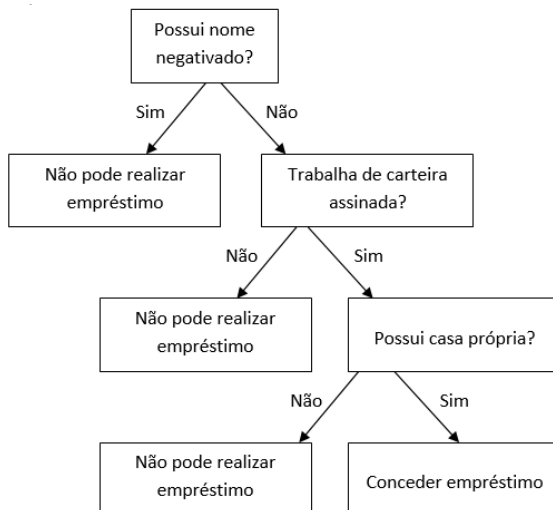


Figura 17 – Exemplo de árvore de decisão para criar programa de empréstimo.

```

1   if True:   #Se verdade
2       #Instrução 1 que será executada
3       #Instrução 2 que será executada
4       #...
5       #Instrução N que será executada
6   else:      #Se falso
7       #Instrução 1 que será executada
8       #Instrução 2 que será executada
9       #...
10      #Instrução N que será executada

```

Programa 17 – Pseudocódigo da estrutura `if/else`.

OPERADORES

Em programação, logo no início do processo de aprendizagem, quando começamos a receber dados do utilizador, percebemos a necessidade de realizar testes ou operações com eles. Por exemplo, comparar a igualdade ou diferença entre dois valores, comparações do tipo maior ou menor com ou sem igualdade e, por vezes, precisamos realizar testes complexos envolvendo mais de uma comparação. É nessa hora que surgem os operadores de comparação (também chamados de relacionais) e os operadores lógicos. Talvez esteja se perguntando: por que estamos tratando de operadores neste capítulo? Nós prontamente responderemos. Operadores comparativos e lógicos estão diretamente ligados com estruturas condicionais, pois na maioria das vezes que utilizamos comandos `if` ou `elif`, temos um teste associado com a utilização de operadores comparativos. Inclusive, a maioria dos exemplos, já tratados neste capítulo, relataram isso. Pois bem, vamos conhecer estes operadores.

OPERADORES DE COMPARAÇÃO OU RELACIONAIS

Os operadores de comparação, também chamados de operadores relacionais, como o próprio nome sugere, comparam ou relacionam valores. A linguagem Python possui operadores comuns encontrados em outras linguagens, o que facilita a migração da aprendizagem. Inclusive, não difere muito do aprendizado matemático visto, por exemplo, em assuntos como inequações ou intervalos. Conheça os operadores apresentados no Quadro 3.

OPERADOR	DESCRIÇÃO	EXEMPLO DE APLICAÇÃO
<code>==</code>	Igual	<code>3 == 2</code> #resulta em False
<code>!=</code>	Diferente	<code>3 != 2</code> #resulta em True
<code>></code>	Maior que	<code>3 > 2</code> #resulta em True
<code><</code>	Menor que	<code>3 < 2</code> #resulta em False
<code>>=</code>	Maior ou igual a	<code>3 >= 2</code> #resulta em True
<code><=</code>	Menor ou igual a	<code>3 <= 2</code> #resulta em False

Quadro 3 – Lista de operadores de comparação.

OPERADORES LÓGICOS

Python, assim como outras linguagens, possui três operadores lógicos para realização de testes compostos. Veja quem são esses operadores, conforme Quadro 4.

OPERADOR	EXPRESSÃO	EXEMPLO DE APLICAÇÃO
<code>and</code>	<code>X and Y</code>	<code>True and False</code> #resulta em False
<code>or</code>	<code>X or Y</code>	<code>True or False</code> #resulta em True
<code>not</code>	<code>not X</code>	<code>not False</code> #resulta em True <code>not True</code> #resulta em False

Quadro 4 – Lista de operadores lógicos.

Veja como funcionam os operadores lógicos no Programa 18.

```
1  print(not False)           #Imprime True
2  print(not True)            #Imprime False
3  print(False and False)     #Imprime False
4  print(False and True)      #Imprime False
5  print(True and False)      #Imprime False
6  print(True and True)       #Imprime True
7  print(False or False)      #Imprime False
8  print(False or True)       #Imprime True
9  print(True or False)       #Imprime True
10 print(True or True)        #Imprime True
```

Programa 18 – Exemplo do uso de operadores lógicos.

Para consolidar o seu entendimento, imagine que precisa ler três diferentes números inteiros e, ao fim, informar qual é o maior deles. Em Python, uma maneira de resolver este problema é apresentado no Programa 19.

```
1  numero1 = int(input("Digite o número 1: "))
2  numero2 = int(input("Digite o número 2: "))
3  numero3 = int(input("Digite o número 3: "))
4  if numero1 == numero2 or numero2 == numero3 or numero1 == numero3:
5      exit() #Encerra o programa
6  if numero1 > numero2 and numero1 > numero3:
7      print("O primeiro número é o maior")
8  if numero2 > numero1 and numero2 > numero3:
9      print("O segundo número é o maior")
10 if numero3 > numero1 and numero3 > numero2:
11     print("O terceiro número é o maior")
```

Programa 19 – Exemplo para encontrar maior valor.

No Programa 19, perceba que temos quatro estruturas condicionais. A primeira quer saber se temos pelos menos dois números iguais e, caso verdade, o programa será encerrado utilizando a função `exit()`. As demais estruturas condicionais querem saber se um dos números é maior que os demais. Foi implementado um teste para o `numero1`, um teste para o `numero2` e um teste para o `numero3`. Veja a diferença entre os operadores lógicos, na primeira estrutura condicional, onde utilizamos o operador `or`, se apenas um teste for verdadeiro, o resultado será verdadeiro. Nas demais estruturas condicionais que utilizam o operador `and`, é necessário que os dois testes sejam verdadeiros para que o resultado seja verdade.

TABELA VERDADE

A tabela verdade é um recurso utilizado no estudo da lógica matemática. Com o uso desta tabela, é possível definir o valor lógico de uma proposição¹⁴, isto é, saber quando dois ou mais testes comparativos resultam em verdadeiro ou falso, o que é bastante comum em programação. Veja o seguinte exemplo: para doar sangue, um requisito mínimo é a avaliação da idade: o doador tem que ter entre 16 e 69 anos. Um teste lógico que compreende esse requisito, em termos de programação Python, seria: `idade >= 16 and idade <= 69`. Veja que não é aceitável uma idade menor que 16 e nem maior que 69 anos, ou seja, os dois testes precisam ser verdadeiros. Portanto, para combinar testes simples e formar testes compostos, são utilizados conectivos lógicos. Estes conectivos

14 Proposição é uma sentença declarativa cujo conteúdo será considerado verdadeiro ou falso. Então, as proposições “A é uma vogal” resulta em um valor verdadeiro e “3 é par” resulta em um valor falso.

representam operações lógicas que formam as tabelas verdades. Na matemática, existem os conectivos lógicos, que são representados simbolicamente, como visto no Quadro 5.

CONECTIVO	SÍMBOLO	OPERAÇÃO LÓGICA	VALOR LÓGICO
Não	\sim	Negação	Valor falso quando o teste for verdadeiro e vice-versa
E	\wedge	Conjunção	Verdadeiro quando todos os testes da composição forem verdadeiros
Ou	\vee	Disjunção	Verdadeiro quando pelo menos um dos testes for verdadeiro

Quadro 5 – Lista de conectivos lógicos.

Assumindo V como verdadeiro, F como falso, T1 como Teste1 e T2 como Teste2, nos próximos 3 quadros, observe a tabela verdade para cada um dos conectivos lógicos.

NÃO	
T1	$\sim T1$
F	V
V	F

Quadro 6 - Tabela verdade do conectivo lógico Não.

E		
T1	T2	$T1 \wedge T2$
F	F	F
F	V	F
V	F	F
V	V	V

Quadro 7 – Tabela verdade do conectivo lógico E.

OU		
T1	T2	T1vT2
F	F	F
F	V	V
V	F	V
V	V	V

Quadro 8 – Tabela verdade do conectivo lógico Ou.

Em Python, para cada um dos conectivos apresentados anteriormente, a tabela verdade pode ser reescrita por meio da utilização de operadores lógicos, apresentados anteriormente.

NOT	
T1	not T1
False	True
True	False

Quadro 9 - Tabela verdade do operador lógico not em Python.

AND		
T1	T2	T1 and T2
False	False	False
False	True	False
True	False	False
True	True	True

Quadro 10 – Tabela verdade do operador lógico and, em Python.

OR		
T1	T2	T1 or T2
False	False	False
False	True	True
True	False	True
True	True	True

Quadro 11 – Tabela verdade do operador lógico or em Python.

ESTRUTURA IF...ELSE

Em Python, a estrutura básica de um comando `if` é bastante simples. Temos a palavra-chave `if`, seguida de um teste lógico, valor ou variável e o caractere dois pontos (:). No Programa 20, observe que nas linhas 3 e 4, temos um bloco de instruções, que em Python, é caracterizado por uma ou várias instruções indentadas (isto é, o recuo entre a margem e o início da instrução). Na prática, utiliza a tecla TAB para realizar o alinhamento de todas as instruções abaixo do `if`, criando assim um bloco. Veja o exemplo a seguir.

```

1  nome = input("Digite seu nome completo: ")
2  if len(nome) > 50: #len() retorna o número de caracteres da string
3      print("Seu nome é grande, ele possui mais de 50 letras. ")
4      print(f"Ele possui {len(nome)} caracteres.")

```

Programa 20 – Exemplo da sintaxe do comando `if`.

Em outras linguagens de programação, como Java ou C++, por exemplo, um bloco de instruções é definido pelo envolvimento de um par de chaves { } para o início e o fim do bloco. No caso de Python, por questões de melhoria da visualização do código, a indentação de instruções é utilizada. Então, de hoje

em diante, quando observar código indentado em Python, tenha certeza de que é um bloco de instruções. Inclusive, cada instrução **if** terá o seu bloco indentado.

Uma informação importante: em Python, assim como em outras linguagens de programação, uma expressão pode ser considerada verdadeira se ela possui valor diferente de zero. Caso contrário, é considerada falsa. Veja o exemplo abaixo:

```
1  valor = 10
2  if valor:
3      print("valor é diferente de zero => verdade.")
4  valor = 0
5  if not valor:
6      print("Valor é falso, mas not inverteu o resultado.")
```

Programa 21 – Considerando inteiros como expressões do comando **if**.

Já a palavra-chave **else** é aplicada nas condições em que é necessário executar instruções quando o teste do **if** não for satisfeito, ou seja, o resultado do teste é falso. Para nós, a palavra-chave **else** é mais bem entendida quando interpretada como a expressão “senão”. Sua sintaxe é simples, quando encerradas as instruções do **if**, no mesmo alinhamento do **if**, adiciona-se a palavra-chave **else** e o caractere dois pontos (:). Abaixo dessa linha, vêm as instruções indentadas formando um bloco. Veja o Programa 22.

```
1  valor = int(input("Número: "))
2  if valor % 2 == 0:
3      print("O valor é par")
4  else:
5      print("O valor é ímpar")
```

Programa 22 – Exemplo da sintaxe do comando **else**.

Deixemos de falar muito e vamos praticar. Para clarear o seu entendimento, começaremos do simples para o avançado. Inicialmente, construiremos o Programa 23 para avaliar se um número digitado pelo utilizador é par ou ímpar.

```
1 valor = int(input("Número: "))
2 if valor % 2 == 0:
3     print("O valor é par")
4 else:
5     print("O valor é ímpar")
```

Programa 23 – Programa par ou ímpar.

O código de resposta do problema é bem simples. Na linha 1, temos a leitura de um valor inteiro qualquer. Já na linha 2, para saber se um número é par, recorremos à seguinte lógica: para todo número par, quando dividido por 2, o resto da sua divisão é zero. Portanto, utilizamos o operador % que retorna o resto da divisão. Caso o resultado de `valor % 2` seja zero, imprimimos a mensagem “O **valor é par**”, caso contrário (exemplo de uso do **else**), imprimimos a mensagem “O **valor é ímpar**”.

Agora, foi convidado a resolver o seguinte problema: uma loja de roupas está vendendo camisas básicas com descontos, mas seus funcionários têm dificuldades no cálculo do valor final a ser cobrado. Por isso, seu Tanaka, dono da loja, convidou para criar um programa para calcular o preço final a partir do número de camisas. Seu Tanaka definiu as seguintes regras de desconto:

- Até 5 camisas, desconto de 3%
- Acima de 5 camisas e até 10 camisas, desconto de 5%; e
- Acima de 10 camisas, desconto de 7%.

Pronto, calcule e imprima o valor a ser pago pelo cliente, sabendo que o preço da camisa é R\$ 12,50. Veja a solução no Programa 24.

```
1  numeroCamisas = int(input("Quantidade de camisas: "))
2  valorCamisa = 12.50
3  valorFinal = numeroCamisas * valorCamisa
4  if numeroCamisas <= 5:
5      valorFinal = valorFinal * (1 - 3/100)
6  else:
7      if numeroCamisas <= 10:
8          valorFinal = valorFinal * (1 - 5/100)
9      else:
10         valorFinal = valorFinal * (1 - 7/100)
11  print(f"Valor final: R$ {valorFinal:.2f}")
```

Programa 24 – Programa da loja do Seu Tanaka.

Analisando o código, percebemos o seguinte: na primeira linha, temos o código para leitura da quantidade de camisas. Na segunda linha, declaramos uma variável para conter o valor individual de uma camisa. Na terceira linha, calculamos o valor final sem descontos. A partir da linha 4, iniciamos as análises para aplicação de desconto. O primeiro teste foi para verificar se o número de camisas é menor ou igual a 5, caso verdade, aplicamos 3% de desconto. Caso contrário, realizaremos um novo teste para saber se a quantidade de camisas é menor ou igual a 10 já que, certamente, a quantidade foi maior que 5. Nesse caso, se verdade, aplicaremos o desconto de 5%, caso contrário, ou seja, acima de 10, aplicaremos 7%.

ESTRUTURA ELIF

No Programa 24, utilizamos um comando `if` após o pri-

meiro comando **else**. Nesses casos, perceba que o código cresce lateralmente por causa do excesso de indentação, o que pode provocar dificuldades no entendimento do código. Para minimizar isso, o Python oferece uma solução elegante que diminui o uso de indentação, facilitando a visualização do código. Estamos falando da palavra-chave **elif**, que é uma espécie de contração do comando **else** e um comando **if**. Logo, ela é utilizada em substituição ao comando **else**, mas permitindo a realização de um novo teste. Veja a solução do mesmo problema agora usando a estrutura **elif**.

```
1  numeroCamisas = int(input("Número de camisas: "))
2  valorCamisa = 12.50
3  valorFinal = numeroCamisas * valorCamisa
4  if numeroCamisas <= 5:
5      valorFinal = valorFinal * (1 - 3/100)
6  elif numeroCamisas <= 10:
7      valorFinal = valorFinal * (1 - 5/100)
8  else:
9      valorFinal = valorFinal * (1 - 7/100)
10 print(f"Valor final: R$ {valorFinal:.2f}")
```

Programa 25 – Programa da loja do Seu Tanaka usando **elif**.

EXERCÍCIOS RESOLVIDOS

1. Uma empresa, que presta serviço à companhia de energia elétrica do estado, necessita de um programa que auxilie os seus eletricitas no cálculo das principais grandezas da Eletricidade que são Tensão, Resistência e Corrente. Sabe-se que

$$U= Ri,$$

onde, U é a Tensão (em V), R é a Resistência (em Ω) e i a Corrente (em A).

foi contratado(a) pela empresa para atender a essa solicitação. Construa um programa que apresente o seguinte menu:

```
*****
                CÁLCULO DE GRANDEZAS ELÉTRICAS
*****
1. Tensão (em Volt)
2. Resistência (em Ohm)
3. Corrente (em Ampére)
*****
Qual grandeza deseja calcular?
```

Em seguida, o programa deve solicitar que o eletricitista informe o valor das outras duas grandezas para realizar o cálculo. Quando o eletricitista escolher:

- Tensão, o programa deve solicitar que ele informe os valores da Resistência e da Corrente
- Resistência, o programa deve solicitar que ele informe os valores da Tensão e da Corrente
- Corrente, o programa deve solicitar que ele informe os valores da Tensão e da Resistência

Por fim, o programa deve calcular e apresentar o valor encontrado para a grandeza escolhida.

Comentários sobre a resolução

Para solução deste problema, a primeira coisa a ser feita é a construção do menu. Conseguimos isso por meio de um conjunto de comandos de impressão, ou seja, uma boa utilização do comando `print`.

Observe a linha 1. Nela temos uma **string** formatada que possui três campos de substituição "`{}` `{}` `{}`". O primeiro e o último campo são substituídos por uma **string** montada a partir do operador `*`, que quando utilizado com **string**, realiza a concatenação do seu conteúdo, o número de vezes do valor do operando que está ligado ao operador. Nesse caso, o valor 4 "`****`". As linhas 2 – 4 realizam a impressão das opções que compõem o menu. A linha 5 utiliza novamente o recurso de concatenação obtido com a utilização do operador `*`, que nesse caso realiza a impressão de 48 asteriscos na mesma linha.

Na linha 6, temos a utilização da função **input** para entrada de dados. A função **input** é envolvida pela função **int**, que converterá a **string** devolvida pela função **input** em um inteiro, que será atribuído a variável **op** (opção). Por que isso? Simplesmente, porque é melhor trabalhar com comparação de inteiros do que comparação de **strings**.

Na linha 7, temos um teste para validar o valor da variável **op**. Ela precisa estar com o valor entre 1 e 3, que são as opções do menu. No teste foi utilizado o operador lógico **or** para que quando o valor de **op** for menor que 1 ou maior que 3, o programa imprima a frase "**Opção inválida**". Caso contrário, o programa deve continuar com as análises do valor de **op**.

O primeiro teste de **op**, utilizando a palavra-chave **elif**, é com o valor 1 e o operador comparativo de igualdade `==`. Caso seja verdade, ou seja, **op** é igual a 1, segue-se a leitura das variáveis solicitadas e o cálculo da variável selecionada pela opção do menu. No caso da opção 1, temos a leitura do valor da resistência que

é atribuída a variável r , a leitura do valor de corrente que é atribuído a variável i e, por fim, o cálculo da variável v ($v = r * i$). Logo em seguida, ocorre a impressão da **string** formatada com o resultado da variável calculada.

Para as demais opções, ocorre algo semelhante. O utilizador escolhe valor 2 ou 3, informa as variáveis solicitadas, calcula o valor da variável escolhida e imprime uma **string** formatada com o resultado da operação.

Vamos ao código:

```
1  print(f"{'*' * 18} {'GRANDEZAS ELÉTRICAS '} {'*' * 18}")
2  print("1. Tensão (em Volt)")
3  print("2. Resistência (em Ohm)")
4  print("3. Corrente (em Ampère)")
5  print("** * 48)
6  op = int(input("Qual grandeza deseja calcular? "))
7  if op < 1 or op > 3:
8      print("Opção inválida.")
9  elif op == 1:
10     R = float(input("Digite o valor da corrente (em Ohm): "))
11     i = float(input("Digite o valor da corrente (em Ampère): "))
12     U = R * i
13     print(f"\nU = {U:.2f}")
14  elif op == 2:
15     U = float(input("Digite o valor da tensão (em Volt): "))
16     i = float(input("Digite o valor da corrente (em Ampère): "))
17     R = U / i
18     print(f"\nR = {R:.2f}")
19  else:
20     U = float(input("Digite o valor da tensão (em Volt): "))
21     R = float(input("Digite o valor da corrente (em Ohm): "))
22     i = U / R
23     print(f"\ni = {i:.2f}")
```

Programa 26 – Exercício resolvido 3.1

2. Considere um triângulo T, representado no sistema de coordena-

das cartesianas, definido por três pontos: $P_1(x_1, y_1)$, $P_2(x_2, y_2)$ e $P_3(x_3, y_3)$. Cada lado do triângulo é obtido por meio do cálculo da distância entre os pontos, conforme equação abaixo

$$distância = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}.$$

Para que T exista, é necessário que:

a) todos os lados sejam maiores que zero e;
b) um dos lados seja menor que a soma dos outros dois e maior que o valor absoluto da diferença entre os lados. Veja as restrições:

- i. $|B-C| < A < B+C$
- ii. $|A-C| < B < A+C$
- iii. $|A-B| < C < A+B$

Para cada conjunto de três pontos informados, o programa deve escrever o tipo do triângulo formado (Isósceles, Escaleno ou Equilátero), bem como os lados dos triângulos. Caso os pontos informados não formem um triângulo, o programa deve apresentar a mensagem **"Nenhum triângulo formado com os pontos informados."**

Comentários sobre a resolução

Vamos resolver este belo problema? Na primeira linha, ocorre a importação da função `sqrt` (cálculo de raiz quadrada) do módulo `math`, para evitar a escrita constante do comando `math`. Agora, só precisa utilizar `sqrt(valor)`.

Na linha 2, temos um comentário de explicação das duas próximas linhas. Dica: comentário é sempre bom, quando bem utilizado. Nas linhas 3, 4, 6, 7, 9 e 10 temos a leitura dos pontos do triângulo, no plano cartesiano. Na ocasião, utilizamos a função `float` para

converter a saída da função `input` para um valor real, porque será necessário trabalhar com números reais (engloba não só os inteiros e os fracionários, positivos e negativos, mas também todos os números irracionais).

A seguir, nas próximas três linhas, calculamos os lados do triângulo a partir dos pares formados pelos três pontos, ou seja, (P1 e P2), (P2 e P3) e (P1 e P3). O cálculo é o seguinte: `L1 = sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)`. Observe a boa utilização dos parênteses. Assim como na matemática, Python resolve as operações contidas nos parênteses mais internos, para no final calcular a operação do parêntese mais externo. Isso quer dizer que ele vai resolver primeiro as operações de subtração. Na sequência, por uma questão de precedência (vir primeiro), a operação de potência do operador `**` é realizada antes da operação de soma. Para concluir o cálculo da expressão, ocorre a solução da raiz quadrada com a função `sqrt`, cujo valor resultante será atribuído a variável `L1` (lado 1).

Nas linhas 16, 17 e 18 ocorre a criação de três variáveis auxiliares com valor inicial `True`. Em programação, é comum a utilização de variáveis auxiliares para solução de problemas. No caso das variáveis citadas, elas serão utilizadas para checar se algum problema ocorreu durante a avaliação conjunta dos lados. No caso, a primeira avaliação é útil para saber se pelo menos um dos lados é zero. Caso verdade, a variável `cond1` recebe o valor `False`, indicando que a formação de um triângulo não será possível. O segundo teste verifica se pelo menos um dos lados é maior que a soma dos outros dois. Caso isso ocorra, a variável `cond2` recebe o valor `False`, indicando que não será possível a formação de triângulo. O terceiro teste sequencial quer saber se algum dos lados é maior que a

soma absoluta dos outros dois. Caso isso ocorra, a variável `cond3` recebe valor `False`, indicando que não será possível a formação de triângulo.

Na linha 28, ocorre mais uma declaração de variável auxiliar. No caso em questão, a variável `triangulo` recebe inicialmente o valor `True`, indicando que até o momento, antes dos testes finais, é possível a formação de triângulo. Na linha 30, verificamos se pelo menos uma das variáveis `cond1`, `cond2` e `cond3` possui valor `False`. Caso verdade, ocorre a impressão da frase “**Nenhum triângulo formado**”. Logo à frente, nas linhas 33, 35 e 37 ocorrem os testes individualizados de cada variável para impressão dos motivos que impediram a formação dos triângulos.

Finalmente, se o resultado da linha 30 for `False`, analisaremos as condições que indicarão o tipo do triângulo. Primeiro se os três lados são iguais (linha 40). Caso seja verdade, ocorrerá a impressão da frase “**Triângulo equilátero**”, indicando o tipo do triângulo. Na linha 42, ocorre o teste que verifica se todos os lados são diferentes para informar, caso verdade, que o triângulo é escaleno. Caso não seja equilátero e nem escaleno, obviamente será isósceles. Na linha 47, a variável `triangulo` é testada e, se ela for verdadeira, os lados do triângulo serão impressos. Ufa! Este foi grande.

Para testar a existência de um triângulo:

- escaleno, use: $P1 = (1, 2)$, $P2 = (1, 7)$ e $P3 = (3, 3)$
- isósceles, use: $P1 = (2, 1)$, $P2 = (5, 1)$ e $P3 = (2, 4)$
- equilátero, use: $P1 = (1, 2)$, $P2 = (1, 7)$ e $P3 = (5.330127018922193, 4.5)$

Vamos ao código:

```

1  from math import sqrt
2  #Coordenadas x e y do Ponto 1
3  x1 = float(input("Digite a coordenada x do Ponto 1: "))
4  y1 = float(input("Digite a coordenada y do Ponto 1: "))
5  #Coordenadas x e y do Ponto 2
6  x2 = float(input("\nDigite a coordenada x do Ponto 2: "))
7  y2 = float(input("Digite a coordenada y do Ponto 2: "))
8  #Coordenadas x e y do Ponto 3
9  x3 = float(input("\nDigite a coordenada x do Ponto 3: "))
10 y3 = float(input("Digite a coordenada y do Ponto 3: "))
11 #Calcula os lados do triângulo
12 L1 = sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2) #Distância entre P1 e P2
13 L2 = sqrt((x3 - x1) ** 2 + (y3 - y1) ** 2) #Distância entre P1 e P3
14 L3 = sqrt((x3 - x2) ** 2 + (y3 - y2) ** 2) #Distância entre P2 e P3
15 #Considera que as três condições de existência são verdadeiras
16 cond1 = True
17 cond2 = True
18 cond3 = True
19 #Verifica se algum lado é igual a zero
20 if L1 == 0 or L2 == 0 or L3 == 0:
21     cond1 = False
22 #Verifica se algum lado é maior que a soma dos outros dois
23 if L1 > (L2 + L3) or L2 > (L1 + L3) or L3 > (L1 + L2):
24     cond2 = False
25 #Algum lado não é maior que o módulo da diferença entre os outros?
26 if L1 <= abs(L2 - L3) or L2 <= abs(L1 - L3) or L3 <= abs(L1 - L2):
27     cond3 = False
28 triangulo = True #Inicialmente, considera a existência do triângulo
29 #Alguma condição de inexistência foi identificada?
30 if cond1 == False or cond2 == False or cond3 == False:
31     triangulo = False #Triângulo inexistente
32     print("\nNenhum triângulo formado.\nMotivo(s):")
33     if cond1 == False:
34         print("Pelo menos um dos lados é igual a 0.")
35     if cond2 == False:
36         print("Pelo menos um lado é maior que a soma dos outros 2.")
37     if cond3 == False:
38         print("Um dos lados é menor ou igual ao módulo da diferença.")
39 #Triângulo existe
40 elif L1 == L2 == L3:
41     print("\nTriângulo equilátero.")
42 elif L1 != L2 and L1 != L3 and L2 != L3:

```



```

43     print("\nTriângulo escaleno.")
44 else:
45     print("\nTriângulo isósceles.")
46 #Todas as condições de existência do triângulo foram satisfeitas
47 if triangulo:
48     print(f"Medida do lado 1: {L1:.2f}")
49     print(f"Medida do lado 2: {L2:.2f}")
50     print(f"Medida do lado 3: {L3:.2f}")

```

Programa 27 – Exercício resolvido 3.2.

3. Escreva um programa que solicite ao utilizador a estatura de 3 pessoas. Ao fim, o programa deve imprimir as estaturas em or-dem decrescente.

Comentários sobre a resolução

Oba! Um problema comum em programação é a ordenação de elementos. Por isso, não poderia faltar aqui um exemplo de ordenação. Até sugerimos que pesquise sobre os métodos de ordenação e pesquisa. Existem vários. É uma excelente maneira de se aperfeiçoar em programação. Agora, tratando do código, adotamos a seguinte estratégia: nas três primeiras linhas, ocorre a leitura de três valores atribuídos às variáveis **alt1**, **alt2** e **alt3**. Na sequência, foram criadas três variáveis auxiliares: **mais_alto**, **est_mediana** e **mais_baixo**. Essas variáveis são utilizadas para considerar que inicialmente o valor mais alto, mais baixo e a mediana correspondem ao primeiro valor lido, no caso **alt1**.

Na linha 7, analisamos se a variável **alt1** possui o maior valor entre os três elementos. Caso sim, vamos considerar que **alt1** é o maior valor, armazenando o seu valor na variável **maior**. Aproveitamos também na linha 9 para perguntar se **alt2** é maior que **alt3**. Caso verdade, isso garante que **alt2** é o valor do meio e, por consequência, **alt3** o menor valor.

Os testes citados se repetem para a variável `alt2` e `alt3`, ou seja, primeiro identificamos o maior número e na sequência, encontramos o valor do meio e o menor valor. Fazendo assim, conseguimos reconhecer a ordem decrescente dos números.

Dica: Python permite a seguinte escrita de código: `if 10 > maior < 100`. Apesar de funcionar, recomendamos, por questões de compatibilidade com outras linguagens, que você escreva da seguinte maneira: `if maior < 10 and maior < 100`.

Vamos ao código:

```
1  alt1 = float(input("Digite a estatura da 1ª pessoa (em metros): "))
2  alt2 = float(input("Digite a estatura da 2ª pessoa (em metros): "))
3  alt3 = float(input("Digite a estatura da 3ª pessoa (em metros): "))
4  mais_alto   = alt1
5  est_mediana = alt1
6  mais_baixo  = alt1
7  if alt1 > alt2 and alt1 > alt3:
8      mais_alto   = alt1
9      if alt2 > alt3:
10         est_mediana = alt2
11         mais_baixo  = alt3
12     else:
13         est_mediana = alt3
14         mais_baixo  = alt2
15 elif alt2 > alt1 and alt2 > alt3:
16     mais_alto   = alt2
17     if alt1 > alt3:
18         est_mediana = alt1
19         mais_baixo  = alt3
20     else:
21         est_mediana = alt3
22         mais_baixo  = alt1
23 else:
24     mais_alto   = alt3
25     if alt1 > alt2:
26         est_mediana = alt1
```

```

27     mais_baixo = alt2
28     else:
29         est_mediana = alt2
30         mais_baixo = alt1
31     print(f"\n{mais_alto}m, {est_mediana}m e {mais_baixo}m")

```

Programa 28 – Exercício resolvido 3.3.

EXERCÍCIOS PROPOSTOS

1. Construa um programa que receba um número inteiro positivo informado pelo utilizador. Caso ele seja par, o programa deve calcular o seu quadrado. Mas, se ele for ímpar, deve ser calculado o seu cubo. Ao fim, o programa deve imprimir o valor calculado.

2. Construa um programa que solicite ao utilizador dois números positivos. Em seguida, o programa deve apresentar o seguinte menu:

1. Média ponderada, com pesos 2 e 3, respectivamente
2. Quadrado da soma dos 2 números
3. Cubo do menor número

Escolha uma opção:

De acordo com a opção informada, o programa deve calcular a operação apresentada no menu. Se a opção escolhida for inválida, o programa deve mostrar a mensagem “Opção inválida” e ser encerrado.

3. Construa um programa que apresente para o utilizador o menu abaixo:

**** TABELA VERDADE ****

1. Operador AND

2. Operador OR

3. Operador NOT

Escolha uma opção:

Se a opção escolhida for 1 ou 2, o programa deve solicitar que o utilizador informe 2 bits e, no caso da opção ser 3, o programa deve solicitar que o utilizador informe apenas 1 bit. Ao fim, o programa deve usar o(s) bit(s) informado(s) e apresentar o resultado da operação seleccionada, com base, na tabela verdade.

4. Suponha que o professor Fábio possui 2 *logins* na rede académica da instituição. Construa um programa que valide o acesso do professor à rede. Caso o par utilizador/senha informado esteja correto, o programa deve imprimir a mensagem “Seja bem vindo!”. Caso contrário, “utilizador e senha não conferem”.

login 1

utilizador: procopio s

enha: 12345

login 2

utilizador: paiva

senha: 54321

5. Uma empresa concederá um aumento de salário aos seus funcionários, variável de acordo com o cargo, conforme a tabela abaixo:

Cargo	Aumento (%)
Programador de Sistemas	30
Analista de Sistemas	20
Analista de Banco de Dados	15

Crie um programa que solicite ao utilizador o salário e o cargo de

um determinado funcionário. Na sequência, o programa deve calcular e imprimir o seu novo salário. Caso o cargo informado não esteja na tabela, o programa deve imprimir “Cargo inválido”.

6. Em sua programação semanal, uma rede de televisão apresenta os seguintes telejornais:

- Bom Dia Nação, apresentado por Zé PINHEIRO e por Ana Carla ARAÚJO
- Jornal Brasileiro, apresentado por Bill BONNER E CARLA VASCONCELOS

Crie um programa no qual o utilizador informe o sobrenome de um dos apresentadores. Se o sobrenome informado não estiver na lista acima, o programa deve mostrar a mensagem “Apresenta-dor(a) desconhecido(a)”. Em caso positivo, o programa deve mostrar o nome do telejornal apresentado pelo apresentador(a).

7. Crie um programa que solicite ao utilizador a informação de 3 estaturas. Caso existam estaturas iguais, o programa deve apresentar a mensagem “Há, pelo menos, 2 pessoas com a mesma estatura”. Caso contrário, o programa deve informar a maior estatura.

8. Considere uma equação $ax^2 + bx + c = 0$, representada pela expressão

Construa um programa no qual o utilizador informe os valores das constantes **a**, **b** e **c**. Ao fim, o programa deve calcular e imprimir as raízes reais da equação. Caso não existam raízes reais, o programa deve apresentar a mensagem “Não existem raízes reais.”.

9. Na última *Black Friday*, o gerente de uma loja de perfumes colocou todo o seu estoque em promoção, de acordo com a tabela a seguir:

Código	Condição de Pagamento	Desconto (%)
1	À vista (em espécie)	15
2	Cartão de débito	10
3	Cartão de crédito (vencimento)	5

Construa um programa que solicite ao operador do caixa o preço total da venda, bem como a forma de pagamento. Ao fim, o programa deve informar o valor final a ser pago.

10. O IMC (Índice de Massa Corporal) é utilizado para avaliar o peso de um indivíduo em relação à sua altura e, como resultado, indica se o indivíduo está com o peso ideal, acima ou abaixo do recomendado. A tabela a seguir indica a situação de um indivíduo adulto em relação ao seu IMC:

IMC	Situação
Abaixo de 18,5	Abaixo do peso
Acima de 18,5 e menor que 25	Peso normal
A partir de 25 e menor que 30	Sobrepeso
Acima de 30 e menor que 35	Obesidade grau 1
Acima de 35 e menor que 40	Obesidade grau 2
Acima de 40	Obesidade grau 3

Para calcular o IMC, é usada a fórmula $IMC = \frac{peso}{altura^2}$. Construa um programa no qual um utilizador informe seu peso e sua altura. A aplicação deve indicar o IMC calculado e a situação do indivíduo.

A resolução das questões propostas deste capítulo está disponível em:
https://github.com/peoolivro/codigos/tree/master/Cap3_Exercicios

Para complementar o conteúdo apresentado neste capítulo, pode acessar o canal Procópio na Rede e assistir às seguintes *playlists*:
<https://www.youtube.com/playlist?list=PLqsF5rntN2naZJmdDeMjQHAPvWvShcepO>
https://www.youtube.com/playlist?list=PLqsF5rntN2nb80Hy9SvvoaGEWPri_CS9I

ESTRUTURAS DE REPETIÇÃO

Há situações do nosso cotidiano que, às vezes, a repetição de uma atividade é necessária. Imagine, por exemplo, que está no final de semana e resolveu acessar uma plataforma de vídeos sob demanda para assistir a um filme. Não sabe ao certo a qual filme assistirá e, por isso, lê diversas sinopses a fim de encontrar aquele filme que espera ser interessante. Ao encontrá-lo, o seleciona e começa uma sessão de cinema na confortável poltrona da sua sala.

No exemplo anterior, a atividade que foi repetida diversas vezes foi a leitura da sinopse do filme. Após repetidas leituras, o fato de ter encontrado o filme ideal para o momento fez parar a leitura de várias sinopses, ou seja, isto foi a sua condição de parada.

Como em situações do mundo real, em um programa, pode ser necessário repetir um trecho diversas vezes até que uma determinada condição seja satisfeita. Em programação, para casos como esse, são usadas estruturas conhecidas como iteração (não é interação!), repetição, laço ou *loop*. Python implementa duas estruturas de repetição: **while** e **for**.

Antes de começarmos a falar sobre essas duas estruturas, vamos pensar na construção de um programa bas-

tante simples, daqueles que nunca ficaremos ricos com eles, porém úteis quando o objetivo é o entendimento.

Vamos construir um programa para imprimir quatro vezes a saudação “Olá!”. deve estar dizendo “Vocês estão de brincadeira!”. Não, nós não estamos de brincadeira. Vamos escrever esse programa.

```
1  print("Olá!")
2  print("Olá!")
3  print("Olá!")
4  print("Olá!")
```

Programa 29 – Impressão de quatro saudações.

Mas, se agora pedíssemos para construir um programa para escrever essa saudação quatro mil vezes, usaria a mesma estratégia da situação anterior, isto é, escreveria quatro mil vezes o comando `print("Olá!")`? Provavelmente, questionaria a existência de alguma estratégia alternativa que não fosse tão entediante como a utilizada no Programa 29. As quatro mil saudações podem ser escritas de uma maneira não enfadonha, utilizando uma estrutura de repetição, conforme o Programa 30. Não se preocupe: explicaremos o funcionamento deste programa logo mais.

```
1  cont = 1
2  while cont <= 4000:
3      print("Olá!")
4      cont += 1 #o mesmo que cont = cont + 1
5  print("FIM")
```

Programa 30 – Impressão de quatro mil saudações usando a estrutura **while**.

ESTRUTURA WHILE

O comportamento da estrutura de repetição **while** é muito simples e a sua ideia é a seguinte: enquanto uma condição (ou um conjunto delas) for verdadeira, uma instrução (ou um conjunto de instruções) que está dentro do laço deverá ser executada. Veja a sintaxe dessa estrutura no pseudocódigo do Programa 31.

```
1   while <condição (ou um conjunto delas) for verdadeira>:  
2       #Instruções a serem executadas  
3       #Instruções a serem executadas após o encerramento do loop
```

Programa 31 – Sintaxe da estrutura **while**.

Retornando ao Programa 30, percebemos que as instruções das linhas 3 e 4 estão deslocados para a direita, em relação à instrução da linha 2. Isso significa que as duas instruções estão dentro do fluxo de execução da estrutura **while** e, assim, elas só serão processadas enquanto a condição determinada pelo laço de repetição for verdadeira, isto é, **cont** <= 4000. Por outro lado, a linha 5 está com a mesma indentação da linha 2 e, portanto, não está dentro do laço. A seguir, detalharemos esse programa, linha por linha.

- Na linha 1, é definida uma variável chamada **cont**, que tem como papel contar a quantidade de vezes que o laço **while** está sendo executado. Em um programa, geralmente variáveis que têm esse papel são chamadas de variáveis contadoras.
- Na linha 2, é utilizada a estrutura de repetição **while**, cuja condição de execução é o valor da variável contadora **cont** ser menor ou igual a 4000.

Portanto, quando o valor de `cont` passar a ser maior que 4000, a condição de execução deixará de ser verdadeira e, consequentemente, as instruções definidas nas linhas 3 e 4 não serão executadas.

- Na linha 3, a `string` "Olá!" será exibida várias vezes porque a instrução `print("Olá!")` está dentro da estrutura `while`.
- Na linha 4, a variável `cont` é incrementada em uma unidade. Em outras palavras, todas as vezes que o programa entrar no laço, a variável contadora terá o seu valor aumentado em 1, isto é, `cont += 1` (vale lembrar que esta instrução poderia ser substituída por `cont = cont + 1`). Ao fim da execução da linha 4, o fluxo de execução retornará para a linha 2, verificando se o novo valor de `cont` é menor ou igual a 4000. Sendo verdadeiro, as linhas 3 e 4 serão executadas novamente (esse "vai e vem" entre as linhas 2, 3 e 4, é o motivo de a estrutura de repetição também ser chamada de laço ou de *loop*). Quando `cont` passar a ser maior que 4000, o laço não será mais executado e o fluxo do programa será deslocado para a linha 5.
- Na linha 5, finalmente o programa exibirá a `string` **FIM** e será encerrado.

Já vimos diversas vezes programadores iniciantes cometerem um erro sutil, porém capaz de causar um comportamento desagradável. Ele é chamado de "*loop* infinito". Esse termo é bastante sugestivo e representa a situação em que um determinado programa entra em um laço e não consegue "escapar".

Vamos continuar tomando como exemplo o Programa 30. Na linha 2, a instrução `cont <= 4000` monitora o valor da variável `cont` a fim de identificar se ainda é necessária a execução das instruções que estão dentro do laço. Agora, imagine que a instrução `cont += 1`, da linha 4, foi removida. Como já sabemos, essa instrução é responsável por fazer com que a variável contadora tenha seu valor acrescido em uma unidade cada vez que o laço for executado. Dessa forma, considerando que `cont` é inicializada com o valor 1, depois de 4000 vezes, seu valor será 4001, ou seja, a condição que sinaliza para o programa finalizar o laço.

Portanto se a linha 4 for removida do Programa 30, `cont` não terá o seu valor alterado, permanecendo sempre igual a 1. Percebe? Ela sempre será menor que 4000 e, assim, “até o fim dos tempos”, o laço será executado. Como diz o ditado, se é “como São Tomé, que precisa ver para crer”, faça o teste: modifique o Programa 30 removendo a linha 4. Em seguida, execute o programa e veja o que ocorrerá.

Em algumas situações é necessária a construção de trecho cujo comportamento seja semelhante a de um *loop* infinito. Uma maneira de escrevê-lo é utilizando a palavra reservada `True`, conforme Programa 32. Mais adiante, veremos outros exemplos usando essa estrutura.

```
1 while True:
2     print("Execução de um loop infinito")
```

Programa 32 – Exemplo de *loop* infinito.

Agora, vamos apresentar um novo exemplo, porém com um número finito de iterações. Imaginemos a situação na qual se deseja calcular e exibir a soma dos 10 primeiros números inteiros

positivos. Duas possíveis alternativas (porém, bastante mal elaboradas) são apresentadas no Programa 33.

```
1  #Primeira alternativa
2  print(1+2+3+4+5+6+7+8+9+10)
3  #Segunda alternativa
4  soma = 1+2+3+4+5+6+7+8+9+10
5  print(soma)
```

Programa 33 – Somatório dos 10 primeiros números inteiros positivos.

Neste somatório, podemos perceber a existência de uma sequência numérica finita que cresce, de forma padronizada, em uma unidade. Observa-se que o primeiro elemento é o 1 e o último é o 10 (1, 2, 3, ..., 10). Na Matemática, essa sequência numérica é chamada de progressão aritmética (PA)¹⁵.

Uma nova alternativa para somar os termos dessa PA, diferentemente do que foi apresentado no Programa 33, é utilizar uma estrutura de repetição, como visto no Programa 34.

```
1  soma = 0
2  termo = 1
3  while termo <= 10:
4      soma += termo
5      termo += 1 #geração do termo da PA, de razão 1
6  print(soma)
```

Programa 34 – Somatório dos 10 primeiros números inteiros positivos usando **while**.

Vamos aos comentários detalhados do Programa 34.

¹⁵ Progressão Aritmética (PA) é uma sequência numérica cuja diferença entre dois números (ou termos) consecutivos é chamada de razão r . Portanto, para gerar o termo a_n , basta somar o termo anterior a_{n-1} com a razão r , ou seja, $a_n = a_{n-1} + r$.

Na linha 1, é declarada a variável **soma**, cujo valor inicial atribuído é 0. Como vimos no Capítulo 2, é recomendável que as variáveis sejam declaradas com nomes significativos, isto é, que os nomes remetam a qual dado está sendo armazenado na variável. A cada iteração, **soma** é utilizada para acumular o somatório dos termos da PA. Por causa disso, variáveis com o comportamento semelhante ao de **soma**, ou seja, que acumulam valores a cada iteração de um laço, muitas vezes, são chamadas de variáveis acumuladoras. deve estar se perguntando: “Por que zero foi atribuído à variável soma?” A resposta é muito simples: porque 0 é o valor que não altera a operação de adição e, por isso, inicializamos a variável **soma** com esse valor.

- Na linha 2, a variável **termo** é declarada e inicializada com o valor 1. No programa, ela corresponde aos termos que serão gerados para a PA.
- Na linha 3, é definida a condição de execução do laço. Em uma descrição textual, poderíamos dizer que “enquanto a variável termo, utilizada para controlar a quantidade de termos gerados, for menor ou igual a 10, isto é, ainda não percorreu os 10 termos da PA, o bloco de instruções dentro do laço deverá se repetir.”
- Na linha 4, **soma** armazenará o resultado da adição entre o valor que ela já possui e o valor que está armazenado em **termo**. Achou difícil de entender? Calma! Na Tabela 2, simularemos o comportamento do que ocorrerá com a variável **soma** quando o laço estiver sendo executado.
- Na linha 5, a variável **termo** é incrementada em

uma unidade. Perceba que isso corresponde à geração do termo da PA, ou seja, o novo valor para termo é o valor que ele possuía na iteração anterior somado à razão 1: `termo += 1`.

- Na última linha, o valor da variável `soma` será exibido quando a condição de execução do laço se tornar falsa, ou seja, quando `termo` maior que 10.

Tabela 2 – Comportamento da variável soma, em função de termo.

ITERAÇÃO	QUANDO	ENTÃO
1	termo = 1	soma = 0+1
2	termo = 2	soma = 0+1+2
3	termo = 3	soma = 0+1+2+3
4	termo = 4	soma = 0+1+2+3+4
5	termo = 5	soma = 0+1+2+3+4+5
6	termo = 6	soma = 0+1+2+3+4+5+6
7	termo = 7	soma = 0+1+2+3+4+5+6+7
8	termo = 8	soma = 0+1+2+3+4+5+6+7+8
9	termo = 9	soma = 0+1+2+3+4+5+6+7+8+9
10	termo = 10	soma = 0+1+2+3+4+5+6+7+8+9+10

FUNÇÃO RANGE

Se fizermos uma tradução literal da língua inglesa para a língua portuguesa, a palavra *range* teria como significado “faixa”. Na prática, essa tradução faz sentido porque, em Python, a função `range()` gera uma sequência de números dentro de uma faixa especificada.

É bastante comum a utilização dessa função em conjunto com a estrutura de repetição `for`, a qual falaremos na próxima seção. A seguir, mostraremos duas maneiras de invocar a função `range()`:

1. `range(<quantidade_de_números_a_serem_gerados>)`
2. `range(<início_da_faixa>, <fim_da_faixa>[, <incremento>])`

Na primeira maneira, a função `range()` recebe um parâmetro o qual indica a quantidade **N** de números a serem gerados, onde **N** deve ser maior que 0. Para ficar claro como se dá a execução da primeira maneira, apresentaremos um exemplo utilizando uma lista, assunto que será abordado no próximo capítulo. Veja:

```
list(range(3)) #Saída: [0, 1, 2]
```

A saída dessa instrução é uma lista contendo 3 números: 0, 1 e 2. O detalhe importante é que o primeiro número da sequência gerada é o 0 e o último é o 2. Portanto, a instrução `range(N)` gera uma lista com **N** números, onde o primeiro número é 0 e o último é **N - 1**. Percebe? Então, agora, execute a instrução `list(range(10))` e observe que o primeiro número é 0 e o último 9.

Por outro lado, a segunda maneira de executar a função `range()` possui três parâmetros distintos: `<início_da_faixa>`, `<fim_da_faixa>` e `<incremento>` (opcional). Embora os nomes dos parâmetros sejam bastante significativos, `<início_da_faixa>` determina o primeiro número que deve ser gerado na faixa, `<incremento>` indica a razão (positiva ou negativa) da PA e, quando não especificado, seu valor padrão é 1. Por fim, o parâmetro `<fim_da_faixa>` está associado à razão da sequência. Quando `<incremento>` for positivo, o último número da sequência corresponde a `<fim_da_faixa> - <incremento>`. No caso de `<incremento>`

ser negativo, o último número da sequência é calculado da seguinte forma: $\langle \text{fim_da_faixa} \rangle + \langle \text{incremento} \rangle * (-1)$. deve estar dizendo: “Não entendi nada”. Calma! Observe os exemplos a seguir e, certamente, entenderá.

```
list(range(10, 16))    #Incremento = 1: [10, 11, 12, 13, 14, 15]
list(range(10, 16, 2)) #Incremento = 2: [10, 12, 14]
list(range(16, 10, -1))#Incremento = -1: [16, 15, 14, 13, 12, 11]
list(range(16, 10, -2))#Incremento = -2: [16, 14, 12]
```

Perceba que quando $\langle \text{incremento} \rangle$ é positivo $\langle \text{início_da_faixa} \rangle$ é menor que $\langle \text{fim_da_faixa} \rangle$. No entanto, quando $\langle \text{incremento} \rangle$ é negativo, $\langle \text{início_da_faixa} \rangle$ é maior que $\langle \text{fim_da_faixa} \rangle$. Agora, já podemos falar da outra estrutura de repetição: o **for**.

ESTRUTURA FOR

É comum programadores iniciantes se questionarem sobre qual das duas estruturas de repetição é mais adequada: **while** ou **for**. Costumamos dizer que depende da situação. Em geral, quando a quantidade de iterações é indeterminada, a estrutura **while** é uma boa alternativa. Por outro lado, quando o número de iterações é definido, a estrutura **for** é bastante adequada, cuja sintaxe é a seguinte:

```
1   for <variável> in range([maneira_1|maneira_2]):
2       #Instruções a serem executadas
3       #Instruções a serem executadas após o fim do loop
```

Programa 35 – Sintaxe da estrutura **for**.

Agora, vamos recodificar o Programa 30 e o Programa 34 de uma maneira diferente, modificando os programas para que utilizem a estrutura de repetição **for** ao invés da estrutura **while**.

```
1  for cont in range(4000):
2      print("Olá!")
3  print("FIM")
```

Programa 36 – Impressão de quatro mil saudações usando a estrutura **for**.

No Programa 36, a linha 1 é responsável pelo controle da execução do laço, a qual ocorrerá quatro mil vezes. Como dissemos anteriormente, **range()** gera uma sequência de números de acordo com o conjunto de parâmetros informados. Nesse programa, o parâmetro indica apenas a quantidade de números a serem gerados. Portanto o primeiro número da sequência será 0 e o último 3999, totalizando quatro mil números. A cada iteração, o valor gerado por **range()** será atribuído à variável **cont**, como resumido na Tabela 3. A linha 2, exibirá a mensagem **Olá!** em cada uma das quatro mil iterações. Por fim, a linha 3 será executada ao término do laço.

Tabela 3 – Valores gerados pela função **range()** ao longo de 4000 iterações.

ITERAÇÃO	ENTÃO
1	cont = 0
2	cont = 1
3	cont = 2
•	•
•	•
•	•
4000	cont = 3999

Observe que, na linha 2 do Programa 37, a função `range()` recebe 2 argumentos que correspondem à faixa de valores a ser gerada. Como não foi especificado o terceiro argumento, que corresponde à razão da PA, por padrão, ele assume o valor 1. Ainda é importante lembrar que o intervalo da faixa de valores é fechado no início e aberto no final, isto é, `[1, 11[`. Em outras palavras, estamos dizendo que o primeiro valor da faixa é 1 e o último é 10.

```
1 soma = 0
2 for termo in range(1, 11):
3     soma += termo
4 print(soma)
```

Programa 37 – Somatório dos 10 primeiros números inteiros positivos usando **for**.

Na linha 1, a variável acumuladora `soma` é inicializada com o valor 0 (lembre-se de que o motivo disso é o fato de 0 não alterar o resultado da operação de adição). Na linha 3, `soma` receberá o valor que ela já acumula somado ao valor que está armazenado na variável `termo`. Depois de 10 iterações da estrutura de repetição, a linha 4 será executada para mostrar o valor da variável `soma`.

COMANDO BREAK

Assim como fizemos ao falar da função `range()`, traduzindo *break* literalmente para a língua portuguesa, temos a palavra “quebra”. É justamente esse o papel do comando **break**: quebrar a execução de uma estrutura de repetição, isto é, forçar a saída do fluxo do programa de dentro do laço.

Para deixar essa explicação mais clara, vamos utilizar

como exemplo o Programa 38, cujo objetivo é solicitar inúmeras vezes que o utilizador informe dois números. O programa calculará a operação de divisão, na qual o primeiro número informado corresponde ao dividendo e o segundo ao divisor. Observe que, na linha 2, o laço tem uma estrutura de repetição infinita e, assim, a forma de sair dele é forçando a sua interrupção. Para isso, usamos o comando **break**, na linha 7.

```
1  print("*** Operação de divisão ***")
2  while True:
3      n1 = int(input("Informe o primeiro número: "))
4      n2 = int(input("Informe o segundo número: "))
5      if n2 == 0:
6          print("Divisor não pode ser 0.\nPrograma será encerrado...")
7          break
8      print(f"{n1} / {n2} = {(n1/n2):.2f}")
9  print("*** Fim da operação de divisão ***")
```

Programa 38 – Exemplo de utilização do comando **break**.

Portanto, enquanto o utilizador digitar um divisor (valor que será atribuído à variável **n2**) diferente de 0, o laço continuará sendo executado. Por outro lado, quando **n2** for igual a 0: a) será exibida a mensagem da linha 6, b) o fluxo do laço será interrompido pelo comando **break** na linha 7 e c) o fluxo do programa será desviado para a linha 9, a qual exibe a mensagem referente ao fim da execução do programa.

Ainda sobre a linha 7 do Programa 38, a substring **"\n"** indica que o cursor deve pular para a próxima linha e, portanto, a mensagem Programa será encerrado... será exibida na linha seguinte.

EXERCÍCIOS RESOLVIDOS

1. Crie um programa no qual o utilizador informe um número inteiro positivo N e armazene-o em uma variável. Em seguida, o utilizador deve digitar N números. Ao fim, o programa deve imprimir a média aritmética dos N números digitados.

Comentários sobre a resolução

Antes de começar a resolver esta questão, é importante destacar que isso se trata de um processo de repetição. Observe que a repetição ocorre porque o utilizador informará N vezes um determinado número, considerando N um número inteiro e maior que 0.

Nós poderíamos utilizar a estrutura **while** para resolver essa questão. No entanto, como sabemos a quantidade de vezes que o laço será executado, optamos pelo laço **for**.

Vamos ao código:

```
1  N = int(input("Digite a quantidade de números a informar: "))
2  soma = 0
3  for cont in range(N):
4      num = float(input("Digite um número: "))
5      soma += num
6  media = soma / N
7  print(f"Média = {media:.2f}")
```

Programa 39 – Exercício resolvido 4.1.

Na linha 1, a variável N armazena a quantidade de números informados pelo utilizador e, posteriormente, na linha 3, ela é utilizada para controlar a quantidade de vezes que as linhas 4 e

5 foram executadas dentro do laço. Na linha 5, a variável **soma** acumula os valores informados pelo utilizador.

Depois que todos os números foram informados, isto é, o laço foi executado N vezes, a linha 6 calcula a média aritmética entre os números digitados. Para calcularmos a média aritmética, foi preciso apenas dividir a soma dos números informados pela quantidade de números digitados. Na linha 7, o resultado da média aritmética é exibido.

2. Para construir o programa a seguir, considere que os usuários só informarão números inteiros positivos. Crie um programa que receba 5 números digitados e, ao fim, exibir quantos números são pares.

Comentários sobre a resolução

Mais uma vez temos uma situação em que se configura um processo de repetição: o utilizador informar 5 vezes um número inteiro positivo. Para resolver esta questão, optamos por utilizar novamente a estrutura **for**.

Vamos ao código:

```
1 pares = 0
2 for cont in range(5):
3     num = int(input("Digite um número: "))
4     if num % 2 == 0:
5         pares += 1
6     print(f"Quantidade de números pares digitados: {pares}")
```

Programa 40 – Exercício resolvido 4.2.

Na linha 2, é realizado o controle da quantidade de vezes que o laço será executado, isto é, 5 vezes. Na linha 4, o programa identifica se o número digitado é par. Sabe-se que quando um número é dividido por 2 e o resto inteiro dessa divisão é igual a 0, ocorre que o número é par. Assim, caso o número seja par, na linha 5, o programa contabiliza a quantidade de números pares digitados.

3. Construa um programa para fazer uma pequena entrevista com os alunos de uma turma. Na entrevista, são informados o sexo e a idade de cada aluno. Considere que o utilizador não sabe quantos alunos existem na turma. O programa deve exibir a quantidade de homens acima de 18 anos e a quantidade de mulheres de qualquer idade. Para encerrar o programa, o utilizador deve informar uma idade negativa.

Comentários sobre a resolução

Apesar de ser um cenário de um processo repetitivo, nós não sabemos quantas vezes essa repetição ocorrerá. Portanto, a estrutura `while` é mais adequada que a `for` para resolver o problema.

Vamos ao código:

```

1  mulheres = 0
2  homens_acima18 = 0
3  while True:
4      idade = int(input("Idade: "))
5      if idade < 0:
6          break
7      sexo = input("Sexo: ")
8      if sexo == "F" or sexo == "f":
9          mulheres += 1
10     elif sexo == "M" or sexo == "m":
11         if idade > 18:
12             homens_acima18 += 1
13     print(f"Total de mulheres: {mulheres}")
14     print(f"Total de homens acima de 18 anos: {homens_acima18}")

```

Programa 41 – Exercício resolvido 4.3.

Nas linhas 1 e 2, são declaradas as variáveis para contagem da quantidade de mulheres e de homens acima de 18 anos, respectivamente.

A instrução da linha 3 faz com que o laço se repita indefinidamente. A condição para que o programa seja encerrado é que uma idade negativa seja informada. Portanto, quando isso ocorre, o laço é interrompido com o comando **break**, na linha 6.

Na linha 8, o programa verifica se o sexo digitado é feminino. Para ele, os dados informados correspondem ao de uma mulher quando o sexo for "F" ou "f". Lembre-se que Python é *case sensitive* e, por isso, a letra F em maiúsculo é diferente da letra f em minúsculo. Portanto se o utilizador digitar "f" ou "F", o programa identificará que se trata de uma pessoa do sexo feminino.

Como resultado desse teste, na linha 9, o programa faz a contagem de uma pessoa do sexo feminino.

No entanto, se o valor digitado na linha 7 não corresponde ao sexo feminino, o programa verifica se corresponde ao sexo masculino ("M" ou "m"). Em caso positivo, a instrução da linha 11 é executada para verificar se a idade informada é superior a 18. Se sim, o programa contabiliza uma pessoa do sexo masculino com idade superior a 18 anos de idade.

Depois que o laço é encerrado, ou seja, uma idade negativa foi informada, o programa escreve o relatório informando a quantidade de mulheres (linha 13) e a quantidade de homens acima de 18 (linhas 14) que participaram da entrevista.

EXERCÍCIOS PROPOSTOS

1. Crie um programa no qual o utilizador informe 2 números inteiros: a e b. Para que o programa continue sua execução, verifique se $a < b$. Se sim, calcule a soma dos números inteiros no intervalo $[a, b]$. Caso contrário, informe uma mensagem de erro.
2. Um professor de Matemática deseja construir um programa para gerar uma Progressão Aritmética (PA). Para isso, devem ser informados 3 argumentos: a) primeiro termo, b) quantidade de termos e c) razão.
3. Construa um programa que receba o nome e o preço de 5 medicamentos de uma drogaria (considere que o utilizador in-

formou cinco medicamentos distintos). O programa deve informar o nome e o preço do medicamento mais barato, bem como a média aritmética dos preços informados.

4. Imagine um sistema de caixa eletrônico. Construa um programa que receba a senha de um correntista para validar o seu acesso ao sistema. Considere que a senha fictícia do correntista é 123456. Considere as seguintes restrições:

- quando a senha estiver correta, mostrar a mensagem: "Olá, <SEUNOME>. Seja bem-vindo ao nosso banco!"
- quando o utilizador errar a senha pela primeira vez, mostrar a mensagem: "Senha incorreta! ainda tem 2 tentativas."
- se o utilizador errar a senha pela segunda vez, mostrar a mensagem: "Senha incorreta! ainda tem 1 tentativa."
- se o utilizador errar a senha novamente, mostrar a mensagem "Sua senha foi bloqueada! Por favor, dirija-se a um de nossos caixas." e o programa deve ser encerrado.

5. Os professores de Educação Física estão organizando uma seletiva para montar a equipe de nataç o. Para isso, eles convocaram os 7 melhores tempos da  ltima competi  o e marcaram o tempo de cada um dos nadadores, na prova dos 25 metros, estilo livre.

Considerando que n o houve tempos iguais, construa um programa que leia o nome e o tempo (em segundos) de cada atleta e, em seguida, gere o seguinte relat rio:

- a. nome do nadador com o melhor tempo

- b. nome do nadador com o pior desempenho
- c. tempo médio dos nadadores e
- d. quantidade de atletas com o tempo entre 12s e 15s

6. Construa um programa que receba uma lista contendo a estatura dos alunos de uma escola. Crie um relatório que informe a

- a. menor estatura
- b. maior estatura
- c. média das estaturas informadas

7. Crie um programa no qual o utilizador informe a idade de um número indeterminado de alunos. Para encerrar a leitura dos dados, o utilizador deve informar uma idade negativa. No fim, o programa deve mostrar a média aritmética entre a maior e a menor idade.

8. Crie um programa no qual o utilizador informe o código do cargo de um funcionário (ver tabela abaixo) e o seu respectivo salário. Para encerrar a leitura dos dados, defina uma condição de parada (por exemplo, código do cargo igual a zero). Ao fim, o programa deve informar a média salarial dos nutricionistas.

CÓDIGO	CARGO
1	Enfermeiro
2	Nutricionista
3	Médico

9. Com base na tabela salarial da questão anterior, crie um programa que informe a quantidade de médicos com salários superiores a R\$ 4.500,00.

10. Uma turma da disciplina de Banco de Dados possui 5 alunos. Construa um programa que leia duas notas e calcule a média aritmética de cada aluno. Ao fim, de acordo com a tabela abaixo, indique o percentual de alunos em Prova Final.

FAIXA	RESULTADO
média < 2	Reprovado
$2 \leq \text{média} < 6$	Prova Final
média ≥ 6	Aprovado

A resolução das questões propostas deste capítulo está disponível em:
https://github.com/peoolivro/codigos/tree/master/Cap4_Exercicios

Para complementar o conteúdo apresentado neste capítulo, pode acessar o canal Procópio na Rede e assistir à seguinte *playlist*:
<https://bit.ly/Python-Módulo4>

LISTAS

Certamente, já estive em um edifício no qual todas as unidades existentes são comerciais. Para facilitar o entendimento deste novo conteúdo, imagine um edifício comercial no qual, em cada andar, existe uma empresa fictícia diferente:

0. No térreo, está localizada a recepção do edifício
1. No 1º andar, Tech Aplicativos
2. No 2º andar, Tech4Pet
3. No 3º andar, Clínica Dente Siso
4. No 4º andar, Postura – Clínica de Fisioterapia
5. No 5º andar, Soluções Inteligentes para Banco de Dados
6. No 6º andar, *Geek Game*
7. No 7º andar, Escola de Inglês BIT
8. No 8º andar, MyCar – Curso Técnico de Mecânica
9. No 9º andar, Centro de Estética Carla Morango

Nesse edifício, a equipe que trabalha na recepção associa cada andar a uma empresa. Por exemplo, se um cliente chegar ao balcão de informações e perguntar onde está localizada a empresa Soluções Inteligentes para Banco de Dados, o recepcionista responderá que está no 5º andar porque ele associou a empresa àquele andar. Da mesma forma que associou Tech4Pet ao 2º andar.

Em Python, poderíamos representar esse edifício utilizando uma estrutura de dados chamada **lista**, cujos elemen-

tos são organizados de forma linear. Cada elemento de uma lista pode ser acessado a partir do índice que representa a sua posição na coleção. É importante observar que o primeiro elemento de uma lista é armazenado no índice 0, o segundo no índice 1, e assim sucessivamente. Obviamente, o último elemento é posicionado no índice que corresponde à quantidade de elementos da lista menos um.

Os elementos de uma lista podem ser de diversos tipos, como os primitivos (**int**, **float**, **string** e lógico) ou os mais complexos como listas, dicionários, tuplas e objetos. As listas são formadas por uma sequência de elementos separados por vírgulas e envolvidos por um par de colchetes.

Em outras linguagens de programação, a estrutura de dados que se assemelha às listas é conhecida como vetor, *array* ou variável homogênea. Em C ou Java, por exemplo, um *array* só pode armazenar valores do mesmo tipo. No entanto, uma característica importante das listas em Python é que elas podem armazenar elementos de tipos distintos.

DECLARANDO UMA LISTA

Durante a declaração de uma variável do tipo lista, não é necessário especificar o tamanho máximo porque ela cresce dinamicamente. Uma lista pode ser declarada de duas formas: a) vazia ou b) preenchida. Veja as sintaxes básicas de cada uma dessas declarações:

```
<lista> = []
```

```
<lista> = [<elemento1>, <elemento2>, ..., <elementoN>]
```

onde `<lista>` é o nome da variável do tipo lista e `<elemento1>`, `<elemento2>` e `<elementoN>` são os elementos da lista `<lista>`. No Programa 42, veja como declarar uma lista vazia e uma lista com elementos pré-definidos.

```
1  # Lista vazia
2  idades = []
3
4  # Lista com elementos pré-definidos
5  idades = [27, 19, 33, 47]
```

Programa 42 – Declaração de listas.

Na linha 5 do Programa 42, os elementos da lista `idades` são todos do mesmo tipo. No entanto, se desejássemos armazenar dados sobre a atriz Paolla de Oliveira – tais como nome completo, quantidade de filhos, estatura e se utiliza Instagram¹⁶ – poderíamos representar esses dados, respectivamente, em uma lista como no exemplo abaixo:

```
dados = ["Caroline Paola Oliveira da Silva", 0, 1.70, True]
```

Observe que cada um dos elementos da lista `dados` é de um tipo distinto. Nome da atriz é uma **string**, a quantidade de filhos é um inteiro, a estatura é um valor real e a informação relacionada ao uso do Instagram é um valor lógico. Legal, não é?

E, se agora, desejássemos imprimir os dados dessa lista? Isso é muito simples! Basta executar o comando `print(dados)`. Se executar esse comando, perceberá

¹⁶ <https://www.instagram.com/>

que os dados da atriz serão exibidos na forma de uma lista em Python. No entanto, uma alternativa para formatar essa impressão seria, em cada linha, escrever os elementos da lista, algo como

```
Nome.....: Caroline Paola Oliveira da Silva
Filhos.....: 0
Estatura.....: 1.70m
Usa Instagram: Sim
```

Perceba que, para a impressão sair nesse formato, é necessário acessar o primeiro elemento da lista (nome da atriz), o segundo (quantidade de filhos), o terceiro (estatura) e o quarto (utilização do Instagram).

Para acessar um elemento, basta informar o nome da lista e em qual índice ele se encontra. A sintaxe básica para isso é a seguinte:

```
lista[<índice>]
```

onde <índice> é o índice em que se encontra o elemento desejado.

É importante esclarecer: o primeiro índice de uma lista é o 0, o segundo índice é o 1, e assim sucessivamente. Ora se o primeiro índice da lista é o 0, obviamente que o último será a quantidade de elementos da lista menos 1. deve estar dizendo: “Como assim? Não entendi.” Vamos lá! Observe a figura abaixo:

dados =	Caroline Paola Oliveira da Silva	0	1.70	True
Índice:	0	1	2	3

Figura 18 – Representação de uma lista.

A lista **dados** tem 4 elementos. O primeiro está armazenado no índice 0 e o último no índice 3. Assim, vemos que o último elemento está armazenado no índice que corresponde a **quantidade_de_elementos_da_lista - 1**. Percebe?

Agora, voltando à situação em que gostaríamos de imprimir os dados da atriz formatados, veja como poderíamos fazer isso no Programa 43.

```

1  dados = ["Caroline Paola Oliveira da Silva", 0, 1.70, True]
2  print(f"Nome.....: {dados[0]}")
3  print(f"Filhos.....: {dados[1]}")
4  print(f"Estatura.....: {dados[2]:.2f}m")
5  if dados[3] == True:
6      print("Usa Instagram: Sim")
7  else:
8      print("Usa Instagram: Não")

```

Programa 43 – Impressão de elementos de uma lista.

INCLUINDO ELEMENTOS

Uma alternativa para incluir elementos em uma determinada lista é utilizando o método **append()**. Ele faz com que o elemento seja adicionado no final da lista. A sua sintaxe básica é a seguinte:

<lista>.append(<elemento>)

onde `<lista>` é o nome da variável do tipo lista e `<elemento>` é o elemento que deve ser incluído no fim da lista. Veja o exemplo no Programa 44.

```
1   atrizes = ["Paolla de Oliveira"]
2   atrizes.append("Camila Queiroz")
3   while True:
4       nome = input("Digite o nome de uma atriz: ")
5       atrizes.append(nome)
6       resp = input("Deseja continuar [S|N]? ")
7       if resp.upper() == "N":
8           break
9   print(atrizes)
```

Programa 44 – Inclusão de elementos em uma lista usando o método `append()`.

Na linha 1, a lista `atrizes` é declarada com o elemento Paolla de Oliveira. Na linha seguinte, o elemento Camila Queiroz é incluído no fim da lista. Na linha 4, que está dentro de um laço, que começa na linha 3 e vai até a linha 8, o programa solicita que o utilizador informe o nome de uma atriz. O valor armazenado na variável `nome` é incluído no final da lista `atrizes`. Isso se repete até que o utilizador informe que não deseja mais continuar. Na linha 9, a lista `atrizes` será exibida. Observe como ficou a lista e veja que, de fato, os elementos foram incluídos no fim da lista.

Outra forma de incluir elementos em uma lista é utilizando o método `insert()`. A diferença desse método para o `append()` é que especifica em qual índice deseja incluir o novo elemento. Observe a sintaxe básica:

```
<lista>.insert(<índice>, <elemento>)
```

No Programa 45, apresentamos um exemplo de como utilizar esse método.

```
1 atrizes = ["Paolla de Oliveira"]
2 atrizes.append("Sheron Menezes")
3 atrizes.insert(1, "Raquel Nunes")
4 print(atrizes)
```

Programa 45 – Inclusão de elementos em uma lista usando o método `insert()`.

Ao fim da execução da linha 2, a lista `atrizes` estará da seguinte forma:

```
atrizes = ["Paolla de Oliveira", "Sheron Menezes"]
```

Na linha 3, será adicionada a atriz Raquel Nunes. No entanto, resolvemos adicionar esse novo elemento entre os dois já cadastrados. Para isso, utilizamos o método `insert()` indicando que o elemento deve ser incluído no índice 1, isto é, deslocando o elemento Sheron Menezes para o índice 2. Na execução do comando da linha 4, verá que a lista ficou assim:

```
["Paolla de Oliveira", "Raquel Nunes", "Sheron Menezes"]
```

EMBARALHANDO E SORTEANDO ELEMENTOS

Em algumas situações, é necessário escolher aleatoriamente um elemento dentro de uma lista. Um caso clássico dessa situação é um sorteio. Contudo, antes de realizar um sorteio, é realizado um embaralhamento das suas possíveis opções.

Para embaralhar os elementos de uma lista, utilizamos o método `shuffle()` que, antes de ser executado, necessita que a biblioteca `random` seja importada. Para sortear um dos elementos da lista, utiliza-se o método `choice()`. Abaixo são

exibidas, respectivamente, as sintaxes dos dois métodos.

random.shuffle(<lista>)

<lista>.choice()

No Programa 46, vemos um exemplo de como utilizar esses dois métodos combinados.

```
1  import random
2  atrizes = ["Adriana Prado", "Bárbara Borges", "Camila Queiroz",
3            "Danielle Winits", "Fernanda Paes Leme", "Helena Ranaldi",
4            "Paolla de Oliveira", "Raquel Nunes", "Viola Davis"]
5  random.shuffle(atrizes)          #Embaralha a lista
6  print(f"Lista embaralhada: {atrizes}")
7  sorteada = random.choice(atrizes) #Sorteia aleatoriamente
8  print(f"Atriz sorteada: {sorteada}")
```

Programa 46 – Embaralhamento e sorteio de elementos de uma lista.

Observe que, nas linhas 2 – 4, a lista **atrizes** é declarada com nove elementos ordenados alfabeticamente. Na linha 5, a lista é embaralhada. Executando a linha 6, pode verificar como a lista ficou após o embaralhamento. Por fim, um dos elementos embaralhados é sorteado, como apresentado na linha 7. O resultado do sorteio é apresentado na linha 8.

ORDENANDO ELEMENTOS

Para ordenar elementos de uma lista, Python oferece o método **sort()**, cuja sintaxe básica é a seguinte:

<lista>.sort(reverse = [False | True])

onde o argumento `reverse` indica se haverá inversão (`reverse = True`) ou se não haverá (`reverse = False`). Por padrão, o método `sort()` ordena do maior para o menor (isto é, `reverse = False`) e, portanto, o argumento `reverse` pode ser omitido. Veja um exemplo no Programa 47.

```
1  import random
2
3  atrizes = ["Adriana Prado", "Bárbara Borges", "Camila Queiroz",
4            "Danielle Winits", "Fernanda Paes Leme", "Helena Ranaldi",
5            "Paolla de Oliveira", "Raquel Nunes", "Viola Davis"]
6
7  # Embaralha elementos
8  random.shuffle(atrizes)
9  # Ordena elementos crescentemente
10 atrizes.sort() # mesmo que usar atrizes.sort(reverse = False)
11 print(atrizes)
12 # Ordena elementos decrescentemente
13 atrizes.sort(reverse = True)
14 print(atrizes)
```

Programa 47 – Ordenamento de elementos de uma lista.

REMOVENDO ELEMENTOS

Para remover elementos de uma lista, existem os métodos `remove()` e `pop()` e a função `del`. As sintaxes básicas de cada uma dessas maneiras de remover elementos de uma lista são mostradas a seguir:

```
<lista>.remove(<elemento>)
```

```
<lista>.pop(<índice>)
```

```
del <lista>[<índice>]
```

Para remover um elemento utilizando o método `remove()`, basta informar qual elemento se deseja excluir. Se, na lista, existir mais de um elemento com o valor informado, será removido apenas a primeira ocorrência encontrada.

O método `pop()` tem um funcionamento semelhante ao do `remove()`. Contudo, o seu argumento de entrada é o índice a ser removido e não o próprio elemento. Todavia, se o índice não for informado, o método removerá o último elemento da lista.

Por fim, para executar a função `del` informam-se o nome da lista e em qual índice o elemento se encontra. Se não for informado o índice da lista que se deseja remover o elemento, a função `del` destruirá a variável do tipo lista.

Acompanhe a execução do Programa 48 e observe o comportamento dos métodos `remove()` e `pop()` e da função `del`.

```

1  atrizes = ["Adriana", "Adriana", "Camila",
2            "Danielle", "Fernanda", "Helena",
3            "Paolla", "Raquel", "Viola"]
4
5  print("Removendo a primeira ocorrência do elemento Adriana...")
6  atrizes.remove("Adriana")
7  print(atrizes)
8
9  print("Removendo o elemento de índice 1, que é Camila...")
10 atrizes.pop(1)
11 print(atrizes)
12
13 print("Índice não foi informado. Remove o último elemento: Viola")
14 atrizes.pop()
15 print(atrizes)
16
17 print("Removendo o novo elemento de índice 1, que é Danielle...")
18 del atrizes[1]
19 print(atrizes)
20
21 del atrizes #Destroi a variável atrizes

```

Programa 48 – Remoção de elementos da lista.

Por fim, se desejarmos esvaziar uma lista, podemos utilizar o método `clear()`, cuja sintaxe básica é a seguinte:

```
<lista>.clear()
```

CLONANDO E COMPARANDO LISTAS

Para criar uma cópia de uma lista podemos utilizar a função `list()`, cuja sintaxe básica segue abaixo:

```
list(<lista>)
```

onde `<lista>` é a lista que está sendo clonada. Como resultado, o valor retornado pode ser atribuído a uma variável, como no exemplo abaixo:

```
1  atrizes = ["Adriana", "Adriana", "Camila",  
2            "Danielle", "Fernanda", "Helena",  
3            "Paolla", "Raquel", "Viola"]  
4  
5  copia = list(atrizes)  
6  print(atrizes)  
7  print(copia)
```

Programa 49 – Criando uma cópia de uma lista.

Vamos imaginar a situação em que desejamos remover todos os elementos de uma lista, mas, antes disso, precisamos fazer um *backup* (cópia) dessa lista. Caso a cópia seja igual à original, a lista original poderá ser esvaziada; caso contrário, um erro deverá ser exibido.

Em Python, duas listas são iguais se, e somente se, têm a mesma quantidade de elementos e todos eles são iguais nas mesmas posições. Abaixo, observe que `lista1` é igual à `lista2`, mas `lista1` é diferente de `lista3` (observe os elementos do índice 3 de cada uma delas) e, conseqüentemente, `lista2` também é diferente de `lista3`.


```
lista1 = [1, 2, 3, 4]
lista2 = [1, 2, 3, 4]
lista3 = [1, 2, 4, 3]
```

Voltando à situação inicial (copiar uma lista e, em seguida, esvaziá-la), observe o exemplo no Programa 50.

```
1  atrizes = ["Adriana Prado", "Bárbara Borges", "Camila Queiroz",
2           "Danielle Winits", "Fernanda Paes Leme", "Helena Ranaldi",
3           "Paolla de Oliveira", "Raquel Nunes", "Viola Davis"]
4
5  backup = list(atrizes)
6
7  # Verifica se a cópia é igual à original
8  if backup == atrizes:
9      atrizes.clear()
10     print("Lista de atrizes esvaziada")
11 else:
12     print("ERRO: Cópia gerada não é igual à original")
```

Programa 50 – Clonagem, esvaziamento e comparação de listas.

INCLUINDO ELEMENTOS DE UMA LISTA EM OUTRA

Caso deseje incluir os elementos de uma `listaB` no final de uma `listaA`, pode utilizar o método `extend()`. Veja a sua sintaxe:

```
listaA.extend(listaB)
```

Agora, observe a situação em que temos a lista `atrizes_internacionais` e desejamos incluir os seus elementos na lista `atrizes`.

```
1 atrizes = ["Adriana", "Bárbara", "Camila", "Danielle", "Fernanda"]
2 atrizes_internacionais = ["Angelina", "Viola"]
3
4 atrizes.extend(atrizes_internacionais)
5
6 print(atrizes)
```

Programa 51 – Exemplo de utilização do método `extend()`.

OCORRÊNCIAS DE ELEMENTOS E COMPRIMENTO DA LISTA

Quando falamos anteriormente sobre o método `remove()`, vimos a possibilidade de existir mais de uma ocorrência de um mesmo elemento. Assim, para contar quantas ocorrências de um determinado elemento existe em uma lista, utilizamos o método `count()`, conforme sintaxe básica abaixo:

`<lista>.count(<elemento>)`

No Programa 52, observe que na lista `atrizes` existem duas ocorrências do elemento `Adriana`.

```

1   atrizes = ["Adriana", "Adriana", "Camila",
2             "Danielle", "Fernanda", "Helena",
3             "Paolla", "Raquel", "Viola"]
4
5   ocorrencias = atrizes.count("Adriana")
6   print(f"O elemento Adriana se repete {ocorrencias} vezes.")

```

Programa 52 – Contagem de ocorrências de elementos em uma lista.

Mas e se quiséssemos contar quantos elementos existem em uma lista? A função `len()` nos retorna essa informação. Veja a sua sintaxe:

`len(<lista>)`

Vamos a mais um programa para exemplificar a função `len()`? verá que o programa informará nove elementos na lista.

```

1   atrizes = ["Adriana", "Adriana", "Camila",
2             "Danielle", "Fernanda", "Helena",
3             "Paolla", "Raquel", "Viola"]
4
5   elem = len(atrizes)
6   print(f"A lista possui {elem} elementos.")

```

Programa 53 – Verificação do comprimento da lista.

MENOR, MAIOR E SOMA DE ELEMENTOS

Quando necessário saber o menor e o maior elemento de uma lista ou, ainda, a soma dos elementos dessa lista, basta utilizarmos, respectivamente, as funções `min()`, `max()`

e `sum()`. As sintaxes são semelhantes à da função `len()`, cujo argumento é o nome da lista, isto é:

```
min(<lista>)
```

```
max(<lista>)
```

```
sum(<lista>)
```

E se desejássemos saber a média aritmética dos elementos de uma lista? Muito simples, basta somar os elementos dela e dividir pelo número de elementos, ou seja, `sum(lista) / len(lista)`. Veja o programa abaixo que implementa essas funções.

```
1  notas_turma = [4, 7.2, 9, 10, 1.75, 3.5, 8, 6.3]
2
3  menor = min(notas_turma)
4  maior = max(notas_turma)
5  media = sum(notas_turma) / len(notas_turma)
6
7  print(f"Menor nota....: {menor:.2f}")
8  print(f"Maior nota....: {maior:.2f}")
9  print(f"Média da turma: {media:.2f}")
```

Programa 54 – Exemplos das funções `min()`, `max()` e `sum()`.

RETORNANDO O ÍNDICE DE UM ELEMENTO

O método `index()` é utilizado para retornar o índice em que se encontra a primeira ocorrência de um elemento informado, cuja sintaxe é:

```
<lista>.index(<elemento>)
```

Antes de implementarmos esse método, observe a lista de atrizes abaixo:

```
atrizes=["Adriana Prado", "Bárbara Borges", "Camila Queiroz",  
        "Danielle Winits", "Fernanda Paes Leme",  
        "Helena Ranaldi", "Paolla de Oliveira",  
        "Raquel Nunes", "Viola Davis"]
```

Perceba que o elemento Adriana Padro está armazenado no índice 0, Bárbara Borges no índice 1 e Viola Davis está no índice 8. Mas, em qual índice está o elemento Marina Ruy Barbosa? Como se vê, esse elemento não está cadastrado.

Então se, na lista **atrizes**, invocássemos o método **index()** procurando por Marina Ruy Barbosa, seria exibido um erro indicando que o elemento não existe na lista: **ValueError: 'Marina Ruy Barbosa' is not in list**. Portanto, para evitar um erro como esse, antes de invocarmos o método **index()**, recomenda-se verificar a existência do elemento na lista e, em caso positivo, executá-lo. Caso contrário, evitar a execução do método.

Em Python, para verificar a existência de um elemento em uma lista, utilizamos o operador **in**. Veja o exemplo no programa a seguir.

```
1  atrizes = ["Adriana Prado", "Bárbara Borges", "Camila Queiroz",
2            "Danielle Winits", "Fernanda Paes Leme",
3            "Helena Ranaldi", "Paolla de Oliveira",
4            "Raquel Nunes", "Viola Davis"]
5
6  procurar_por = "Marina Ruy Barbosa"
7  if procurar_por in atrizes:
8      indice = atrizes.index(procurar_por)
9      print(f"{procurar_por} está no índice {indice}.")
10 else:
11     print(f"{procurar_por} não está na lista.")
```

Programa 55 – Procura pelo índice de um elemento na lista.

Na linha 6, substitua Marina Ruy Barbosa por Viola Davis e veja que o programa indicará que o elemento está posicionado no índice 8.

O operador **in**, além de verificar a existência de um elemento, também pode ser utilizado para percorrer uma lista elemento a elemento. Em geral, **in** é combinado com a estrutura de repetição **for**. Observe a lista a seguir:

atrizes=	Adriana	Bárbara	Camila	Danielle	Fernanda
índice:	0	1	2	3	4

Figura 19 – Representação de uma lista com nomes de atrizes.

Imaginemos a situação em que desejamos imprimir, em cada linha, os nomes cadastrados na lista. Na linha 3 do Programa 68, a cada iteração, um elemento é obtido de **atrizes** e, em seguida, armazenado na variável **nome**. Em outras palavras, na primeira iteração, o elemento posicionado no índice 0 é armazenado em **nome** (isto é, Adriana). Na segunda

iteração, o elemento do índice 1 (Bárbara) é armazenado na variável **nome**, e assim sucessivamente, até chegar no elemento que está armazenado no índice 4 (Fernanda). A linha 4 imprime cada um dos nomes cadastrados.

```
1   atrizes = ["Adriana", "Bárbara", "Camila", "Danielle", "Fernanda"]
2
3   for nome in atrizes:
4       print(nome)
```

Programa 56 – Percorrendo uma lista usando **in**.

RETORNANDO ÍNDICE E ELEMENTO

Muitas vezes precisamos percorrer uma lista e obter, simultaneamente, seus elementos e os respectivos índices. Para isso, podemos utilizar a função **enumerate()** que, em geral, também é combinada com a estrutura de repetição **for**. Veja como se pode utilizar **enumerate()** dentro de um **for**.

```
for <índice>, <elemento> enumerate(<lista>):
    <instruções>
```

Continuemos com a lista apresentada na Figura 19 para exemplificarmos, no Programa 57, como utilizar a função **enumerate()**.

```

1   atrizes = ["Adriana", "Bárbara", "Camila", "Danielle", "Fernanda"]
2
3   for indice, nome in enumerate(atrizes):
4       print(f"{nome} está armazenado no índice {indice}")

```

Programa 57 – Percorrendo uma lista usando `enumerate()`.

LISTAS ANINHADAS

Uma lista aninhada consiste na existência de uma lista como sendo elemento de uma outra lista. Como assim? Vamos, lá! Observe a lista `familia`, no exemplo abaixo:

```
familia = ["João Nascimento", ["João Vitor", "Maria Clara"]]
```

Veja que o elemento do índice 0 é uma `string` contendo o nome do professor João Nascimento, ao passo que, no índice 1, o elemento é uma outra lista, agora, contendo os nomes dos dois filhos do professor. A ideia é fácil, concorda? Veja que isto corrobora o que dissemos na introdução deste capítulo: os elementos de uma lista podem ser de diversos tipos como os primitivos (`int`, `float`, `string` e lógico) ou os mais complexos como listas, dicionários, tuplas e objetos.

As listas aninhadas podem ser utilizadas para a representação de matrizes. A matriz A, abaixo

$$A = \begin{bmatrix} 2 & 1 & -5 \\ 3 & 7 & 0 \\ 6 & 4 & 8 \end{bmatrix},$$

poderia ser representada por uma lista aninhada, como segue:


```

A = [
    [2, 1, -5],
    [3, 7, 0],
    [6, 4, 8]
]

```

Para acessar um elemento da lista aninhada **A** (ou simplesmente matriz), é preciso informar o índice correspondente e o índice da lista interna. A sua sintaxe é:

```

lista [<índice_externo>] [<índice_interno>]

```

onde <lista> é o nome da lista externa, <índice_externo> é o índice de <lista> em que se encontra a lista interna que se deseja acessar e <índice_interno> é o índice da lista interna. Tomando como exemplo a matriz **A**, definida anteriormente, <lista> corresponderia à própria lista **A** e [2, 1, -5], [3, 7, 0] e [6, 4, 8] correspondem às listas internas de **A**.

Observe que na matriz **A**, os elementos da diagonal principal são 2, 7 e 8. Consequentemente, o traço de **A** (isto é, a soma dos elementos da diagonal principal da matriz **A**) é dado por $tr(A) = 2 + 7 + 8 = 17$. Considerando a lista aninhada **A**, também definida anteriormente, podemos calcular $tr(A)$ como visto no Programa 58. Observe que, nas linhas 7 – 10, foi calculada a soma de todos os elementos da matriz.

```

1  A = [[2, 1, -5], [3, 7, 0], [6, 4, 8]]
2
3  tr_A = A[0][0] + A[1][1] + A[2][2]
4  print(f"tr(A) = {tr_A}")
5
6  # Soma dos elementos
7  soma_A = A[0][0] + A[0][1] + A[0][2]
8  soma_A += A[1][0] + A[1][1] + A[1][2]
9  soma_A += A[2][0] + A[2][1] + A[2][2]
10 print(f"soma(A) = {soma_A}")

```

Programa 58 – Manipulando elementos de uma matriz.

Agora, imagine o seguinte problema: calcular o traço da matriz quadrada **B** de ordem 500, bem como a soma dos seus elementos. Percebeu o esforço “braçal” para resolver esse problema? Possivelmente, não escreveria algo como

```

tr_B = B[0][0] + B[1][1] + B[2][2] + B[3][3] + ... + B[499][499]
soma_B = B[0][0] + B[0][1] + B[0][2] + ... + B[499][499]

```

porque já conhece as estruturas de repetição, não é verdade?! Então, veja no Programa 59 como poderíamos resolver esse problema para qualquer matriz quadrada de ordem **N**. Tomemos como base a matriz **A**.

```

1  A = [[2, 1, -5], [3, 7, 0], [6, 4, 8]]
2
3  tr_A = 0
4  soma_A = 0
5
6  lin = len(A)    # Qtde de elementos da lista
7  col = len(A[0]) # Qtde de elementos do 1º elemento da lista aninhada
8
9  for i in range(lin):
10     for j in range(col):
11         # Elemento da diagonal principal
12         if i == j:
13             tr_A += A[i][j]
14
15             soma_A += A[i][j]
16
17  print(f"tr(A) = {tr_A}")
18  print(f"soma(A) = {soma_A}")

```

Programa 59 – Percorrendo uma matriz.

Na linha 6, obtém-se a quantidade de elementos da lista aninhada, ou seja, a quantidade de linhas da matriz. Na linha 7, é verificada a quantidade de elementos que existem no elemento posicionado no índice 0 da lista aninhada, o que nos informa a quantidade de colunas da matriz. pode perguntar: “Por que `len(A[0])`?” Porque estamos contando a quantidade de elementos da lista interna, ou seja, a quantidade de colunas. Mas, obteríamos o mesmo resultado se utilizássemos `len(A[1])` ou `len(A[2])`.

As linhas 9 e 10 percorrem, respectivamente, as linhas e as colunas da matriz. Na linha 12, é verificado se o valor de `i` é igual ao de `j`. Lembremos que a definição mate-

mática da diagonal principal de uma matriz quadrada é dada pela coleção de elementos quando $i = j$. Quando a condição definida na linha 12 é verdadeira, a variável `tr_A` será acumulada com o respectivo valor do elemento da matriz. Por fim, na linha 15, cada valor do elemento que é obtido da matriz é acumulado na variável `soma_A`.

UMA MANEIRA DIFERENTE DE GERAR E MANIPULAR LISTAS

Existe uma maneira de se criar listas menos verbosa, ou seja, com menos código e mais compreensível. No mundo Python, isso se chama *List Comprehensions*, cuja sintaxe é:

```
novaLista = [expressao for item in lista [condicao]]
```

onde `novaLista` é a nova lista gerada pela operação, `expressao` corresponde a expressão que deve gerar o valor de um elemento na lista, `item` é o elemento iterado, `lista` o conjunto de elementos que devem ser iterados e `condicao` um teste opcional que pode ser `usado` para a geração de valores.

A motivação para que aprenda *List Comprehensions* pode vir da performance de execução dos códigos. Em geral, *List Comprehensions* são 35% mais rápidos que a estrutura de repetição `for`. Além disso, provavelmente se destacará entre os programadores iniciais.

Agora, vejamos alguns exemplos de utilização:

```

1  lista1 = [x ** 2 for x in range(10)]
2  print(lista1)
3  lista2 = []
4  lista2 = [x for x in range(1,20) if x % 2 == 0]
5  print(lista2)
6  lista3 = [i for i in "HACKATHON" if i in ["A","E","I","O","U"]]
7  print(lista3)
8  lista4 = [1,2,3]
9  lista4 = [i**3 for i in lista4]

```

Programa 60 – Exemplo de uso de *List Comprehensions* na **string** HACKATHON¹⁷.

Na linha 1, criamos uma variável do tipo lista chamada **lista1** contendo a potência dos valores [0,1,2,3,4,5,6,7,8,9], os quais são gerados pela função **range()**. Observe que a lista não existia anteriormente. Ela foi gerada pelo *List Comprehensions*.

A linha 2 exibe o resultado [0,1,4,9,16,25,36,49,64,81] e, na linha 3, uma lista vazia chamada **lista2** é inicializada, embora não houvesse necessidade. No entanto, fizemos isso apenas para deixar o código mais didático para .

A linha 4 gera uma nova lista de números pares no intervalo [1, 20] e faz a atribuição à variável **lista2**. Observe que a expressão contém uma estrutura condicional para a geração de valores na lista. A estrutura condicional testa os valores que são gerados pela função **range()** e caso sejam pares (isto é **i % 2 == 0**), esses valores serão inseridos na nova lista.

Já a linha 6 gera uma lista de vogais existentes na palavra

¹⁷ HACKATHON é uma maratona de programação na qual programadores, designers e outros profissionais de programação se reúnem a fim de construir soluções inovadoras para um problema apresentado.

“HACKATHON”, ou seja, a lista `[A, A, 0]`. Por fim, a linha 9 utiliza os elementos da lista4 para gerar o cubo de cada elemento, gerando a lista `[1, 8, 27]`.

EXERCÍCIOS RESOLVIDOS

1. Crie um programa para gerar duas matrizes quadradas aleatórias (A e B), no intervalo $[1, 10]$, de mesma ordem, a qual deve ser informada pelo utilizador. Ao fim, o programa deve calcular e imprimir a soma entre os elementos de A que estão abaixo da diagonal principal com os elementos de B que estão acima da diagonal principal.

Comentário sobre a resolução

Antes de começar a escrever o código desse programa, observe as

$$A = \begin{bmatrix} 2 & 1 & -5 \\ 3 & 7 & 0 \\ 6 & 1 & 8 \end{bmatrix}, B = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}.$$

No exemplo acima, a soma dos elementos de A abaixo da diagonal principal com os elementos de B acima da diagonal principal é $3+6+1+2+3+6=21$. Visualmente, conseguimos identificar os elementos acima e abaixo da diagonal principal. Mas como poderíamos “ensinar” ao nosso programa como ele identificar se um elemento está acima ou abaixo da sua diagonal principal? Veja a matriz genérica a seguir:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Os elementos abaixo da diagonal principal são a_{21} , a_{31} e a_{32} e os acima são a_{12} , a_{13} e a_{23} . Olhando para os índices i e j desses seis elementos, consegue enxergar algum padrão? Se identificou que quando o elemento está abaixo da

diagonal principal $i < j$ e, quando está acima, $i < j$, parabéns!
"matou a charada"!

Apesar de estes autores adorarem matemática, vamos ao que nos interessa neste momento. Vamos ao código:

```
1  from random import randrange
2  import random
3
4  linhas = int(input("Número de linhas: "))
5  colunas = int(input("Número de colunas: "))
6
7  A = [[randrange(1, 11) for i in range(colunas)]
8         for j in range(linhas)]
9  B = [[randrange(1, 11) for i in range(colunas)]
10         for j in range(linhas)]
11
12  #Para usar matrizes do exemplo, remova comentários das linhas 13 a 16
13  #A = [[2, 1, -5], [3, 7, 0], [6, 1, 8]]
14  #B = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
15  #linhas = len(A)
16  #colunas = len(A[0])
17
18  abaixoDP = 0
19  acimaDP = 0
20  for i in range(linhas):
21      for j in range(colunas):
22          if i > j: #Elemento abaixo da DP da matriz A
23              abaixoDP += A[i][j]
```

```

24         if i < j: #Elemento acima da DP da matriz B
25             acimaDP += B[i][j]
26
27     print("Matriz A:")
28     # Imprime A
29     for i in range(linhas):
30         for j in range(colunas):
31             print(A[i][j], end=' ')
32         print()
33
34     print("\nMatriz B:")
35     # Imprime B
36     for i in range(linhas):
37         for j in range(colunas):
38             print(B[i][j], end=' ')
39         print()
40
41     print(f"Soma = {abaixoDP + acimaDP}")

```

Programa 61 – Exercício resolvido 5.1.

As linhas 1 e 2 importam o módulo **range**, que é utilizado para gerar números aleatórios. Nas linhas 7 e 9, a instrução **randrange(1, 11)** gera um valor aleatório no intervalo que vai de 1 a 10. Para gerar um valor aleatório distribuído uniformemente, no intervalo [a, b], utiliza-se **randrange(a, b + 1)**. As linhas 7 e 8 montam a matriz aleatória **A**, enquanto 9 e 10 geram a matriz **B**.

As linhas 20 e 21 percorrem toda a matriz e, na linha 22, é realizado um teste para verificar se o elemento está abaixo da diagonal principal da matriz. Em caso positivo, a variável

abaixoDP (linha 23) é somada ao valor do elemento corrente da matriz **A**. Por outro lado, quando o elemento corrente da matriz **B** está acima da diagonal principal (linha 24), ele é somado à variável **acimaDP** (linha 25).

As linhas 27 – 39 são utilizadas apenas para formatação da impressão das matrizes. Por fim, na linha 41, é exibida a soma entre os elementos abaixo da diagonal principal da matriz **A** e os elementos acima da diagonal principal da matriz **B**.

2. Crie um programa no qual o utilizador digite o número de linhas e o número de colunas de uma matriz **M**. Ao fim, o programa deve informar se **M** é uma matriz triangular inferior.

Comentário sobre a resolução

Antes de começarmos, observe a matriz abaixo:

$$M = \begin{bmatrix} 2 & 0 & 0 \\ 3 & 0 & 0 \\ 6 & 1 & 8 \end{bmatrix}$$

Veja que todos os elementos acima da diagonal principal são iguais a zero. Então, quando os elementos acima da diagonal principal forem iguais a zero, ou seja, se for nulo todo elemento do tipo M_{ij} , para o qual $i < j$, a matriz **M** é chamada de triangular inferior.

Agora sim. Vamos ao código:

```

1  lin = int(input("Número de linhas: "))
2  col = int(input("Número de colunas: "))
3
4  # Se matriz não é quadrada não pode ser triangular
5  if lin != col:
6      print("Matriz não é triangular.")
7  else:
8      M = []
9      triangular = True
10     for i in range(lin):
11         ELEM = []
12         for j in range(col):
13             ELEM.append(int(input(f"M[{i+1}][{j+1}] = ")))
14         M.append(ELEM)
15
16     print("Matriz M:")
17     for i in range(lin):
18         for j in range(col):
19             print(M[i][j], end=` `)
20         print()
21
22     for i in range(lin):
23         for j in range(col):
24             # Elemento acima da DP?
25             if i < j:
26                 # Se algum elemento acima da DP é diferente de 0
27                 # M não pode ser triangular inferior
28                 if M[i][j] != 0:
29                     triangular = False
30                     break # Encerra a busca
31
32     if triangular == True:
33         print("M é uma matriz triangular inferior.")

```

```
34         else:
35             print("M não é uma matriz triangular inferior.")
```

Programa 62 – Exercício resolvido 5.2.

Na linha 8, declaramos a variável **M**, que é uma lista simples e, mais a frente, tornar-se-á uma lista aninhada.

Na linha 9, decidimos pela seguinte estratégia: afirmar que a matriz é triangular inferior, ou seja, **triangular = True**. deve estar se perguntando o motivo. Seguinte: para a lógica que definimos neste programa, é mais fácil provarmos que esta afirmação é falsa do que o contrário. Então, vamos prosseguir.

Na linha 11, declaramos a lista **ELEM** como vazia (daqui a pouco explicaremos essa “jogada”). Na linha 13, veja que o valor informado pelo utilizador será adicionado ao fim de **ELEM**. Perceba também que estamos incrementando as variáveis de controle *i* e *j* a 1. Isso é um artifício que utilizamos porque os valores dessas variáveis são iniciados em 0. No entanto, na matemática, a primeira linha de uma matriz é 1 e não 0 (o mesmo ocorre com as colunas da matriz). Percebe? Então, para que o programa fique mais amigável para o utilizador, seria exibido algo como

```
M[1] [1] =
M[1] [2] =
M[1] [3] =
```

Depois que o utilizador informa todos os elementos da linha atual da matriz, eles são incluídos em **M** (no código, veja a linha 14). Neste momento, **M** deixa de ser uma lista simples e passa a ser uma lista aninhada.

Sabemos que enquanto houver linhas a serem percorridas, o programa retornará para a linha 10 do código e, em seguida, executará novamente a linha 11. conseguiu identificar o motivo da nossa “jogada” (**ELEM** = [])? Fizemos isso porque a variável está armazenando os valores informados pelo utilizador na iteração anterior. Se **ELEM** não fosse esvaziada, os elementos informados na iteração anterior seriam incluídos na linha atual da matriz.

As linhas 16 – 20 imprimem a matriz informada. Agora, vamos ao “cérebro” do programa. Na linha 25, é verificado se o elemento está acima da diagonal principal. Em caso positivo, o programa identifica se o elemento atual é diferente de 0. Se sim, o programa altera o valor da variável **triangular** para **False** e encerra a busca na matriz, uma vez que ele já constatou que ela não é triangular. Viu como é mais fácil provar que a matriz não é triangular?

3. Crie um programa que leia um valor N, tal que $N > 0$. O programa deve gerar duas listas aleatórias (L1 e L2), com valores no intervalo [1 e 10]. A terceira lista, L3, deve ser gerada com base na soma entre os elementos de L1 e de L2. Ao fim, o programa deve imprimir as 3 listas.

Comentário sobre a resolução

Vamos imaginar que o valor N, informado pelo utilizador, é 4 e as listas geradas aleatoriamente são $L1 = [1, 3, 9, 2]$ e $L2 = [0, 5, 7, 4]$. Assim, $L3 = [1 + 0, 3 + 5, 9 + 7, 2 + 4]$. Portanto, obtemos $L3 = [1, 8, 16, 6]$. O somatório que fizemos corresponde a $L3[i] = L1[i] + L2[i]$.

Vamos ao código a seguir:

```
1  from random import randrange
2
3  N = int(input("Informe um valor para N: "))
4  if N > 0:
5      # Gera duas listas aleatórias no intervalo fechado [1, 10]
6      L1 = [randrange(1, 11) for i in range(N)]
7      L2 = [randrange(1, 11) for i in range(N)]
8
9      L3 = [] # Cria lista L3 vazia
10     for i in range(N):
11         # Inclui, em L3, a soma dos elementos de L1 e de L2
12         L3.append(L1[i] + L2[i])
13
14     print(f"L1 = {L1}")
15     print(f"L2 = {L2}")
16     print(f"L3 = {L3}")
17 else:
18     print("Erro: N deve ser maior que 0.")
```

Programa 63 – Exercício resolvido 5.3.

O código do Programa 63 é autoexplicativo porque o que é novidade para está nas linhas 1, 6 e 7. Como as linhas 6 e 7 estão comentadas no código, resta-nos dizer apenas que a linha 1 importa o pacote utilizado para geração de valores aleatórios em uma faixa de valores. Pronto! Simples assim! Teste o seu programa.

EXERCÍCIOS PROPOSTOS

1. Um professor de Matemática deseja construir um programa para gerar uma Progressão Aritmética (PA). Para isso, devem

ser informados 3 parâmetros de entrada: a) primeiro termo da PA, b) quantidade de termos da PA e c) razão dessa PA. Construa um programa para carregar e imprimir uma lista contendo os termos da PA, bem como a soma dos elementos da PA.

2. Os professores de Educação Física estão organizando uma seletiva para montar a equipe de natação. Para isso, eles convocaram os 7 melhores tempos da última competição e marcaram o tempo de cada um dos nadadores, na prova dos 25 metros, estilo nado livre.

Considerando que não houve tempos iguais, construa um programa que leia o nome e o tempo (em segundos) de cada atleta e, em seguida, gere o seguinte relatório:

- a. nadador com o melhor tempo;
- b. nadador com o pior desempenho;
- c. tempo médio dos nadadores e;

Dica: Crie duas listas: uma para cadastrar os nomes dos nadadores e outra para guardar os seus respectivos tempos.

3. Uma turma de formandos está vendendo rifas para angariar recursos financeiros para sua cerimônia de formatura. Construa um programa para cadastrar os nomes das pessoas que compraram a rifa. Ao fim, o programa deve sortear o ganhador do prêmio e imprimir o seu nome.

4. Crie um programa que solicite o utilizador um número N ím-par maior que 1. Em seguida, preencha uma lista com N números inteiros positivos (suponha que o utilizador sempre digitará números inteiros positivos). Selecione o elemento que está no

centro da lista. Ao final, calcule e escreva o fatorial do elemento selecionado.

5. Crie um programa que declare uma lista L, a qual pode preenchê-la manualmente. Em seguida, o programa deve calcular a média geométrica entre o menor e o maior elemento da lista L. Ao fim, o programa deve imprimir a média geométrica encontrada.

6. Crie um programa que leia um valor N, tal que $N > 1$. O programa deve gerar, aleatoriamente, uma lista L. Por fim, o programa deve calcular e imprimir a média geométrica dos N elementos da lista..

7. Crie um programa que gere, aleatoriamente, uma matriz M, com elementos no intervalo [0, 1]. A quantidade de linhas e de colunas de M devem ser informadas pelo utilizador. Ao fim, o programa deve informar se M é uma matriz nula.

8. Crie um programa no qual o utilizador informe o número de linhas e o número de colunas de uma matriz M e, em seguida, o utilizador deve digitar os elementos de M. Ao fim, o programa deve informar se M é uma matriz identidade.

9. Crie um programa que gere, aleatoriamente, uma matriz M. A quantidade de linhas e de colunas de M devem ser informadas pelo utilizador. Ao fim, o programa deve informar se M é uma matriz diagonal.

Exemplos de uma matriz diagonal:

$$\begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix} \quad \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 6 \end{pmatrix}$$

10. Uma matriz é considerada esparsa quando a quantidade de elementos iguais a zero supera a quantidade de elementos diferentes de zero. Crie um programa que gere, aleatoriamente, uma matriz M. A quantidade de linhas e de colunas de M devem ser informadas pelo utilizador. Ao fim, o programa deve informar se M é uma matriz esparsa.

A resolução das questões propostas deste capítulo está disponível em:
https://github.com/peoolivro/codigos/tree/master/Cap5_Exercicios

Para complementar o conteúdo apresentado neste capítulo, pode acessar o canal Procópio na Rede e assistir às seguintes *playlists*:

<https://www.youtube.com/playlist?list=PLqsF5rntN2naMOFTA56uUvMrAPLE2fdDH>

<https://www.youtube.com/playlist?list=PLqsF5rntN2nbVBXuRRRCihFGmst03ismX>

DICIONÁRIOS

No capítulo anterior, falamos sobre uma estrutura de dados conhecida como lista. Neste capítulo, falaremos de outra que é chamada dicionário. Diferentemente das listas, que utilizam índices para acessar os seus elementos, os dicionários utilizam chaves.

Vamos imaginar um sistema de informação que armazena o cadastro de alunos de uma instituição de ensino tais como matrícula, nome, telefone, sexo e curso. Na sua opinião, qual desses dados identificaria, unicamente, o cadastro de um determinado aluno no sistema? Se pensou “matrícula”, acertou! Em uma mesma instituição, não existem duas (ou mais) matrículas iguais, mas podem existir duas (ou mais) pessoas com o mesmo nome, ou matriculadas no mesmo curso, ou serem do mesmo sexo ou que usam o mesmo número telefônico. Portanto, matrícula é o único dado que viabiliza a identificação única de um aluno no sistema.

Então, se fôssemos implementar um dicionário, com base no exemplo imaginado, elegeríamos matrícula como a chave de acesso aos elementos e teríamos algo, hipoteticamente, parecido com a Figura 20.

CHAVE	VALOR			
MATRÍCULA	NOME	TELEFONE	SEXO	CURSO
1122	Ana Rocha	1111-1111	F	Eng. de Software
3344	Bruna Lima	2222-2222	F	Eng. Mecatrônica

CHAVE	VALOR			
MATRÍCULA	NOME	TELEFONE	SEXO	CURSO
5566	Carlos Silva	3333-3333	M	Eng. de Telecomunicações
7788	Heitor Brandão	4444-4444	M	Ciências da Computação
9910	Henrique Paiva	5555-5555	M	Eng. da Computação
1112	Letícia Campos	6666-6666	F	Eng. Mecânica
1314	Leila Campos	6666-6666	F	Eng. Elétrica

Figura 20 – Cadastro de alunos.

ACEDENDO ELEMENTOS

Imagine a situação em que queremos representar uma agenda de contatos do Instagram. Considerando que já estudou a estrutura de dados lista (e sabe como manipulá-la), uma solução seria definir duas variáveis desse tipo: uma para armazenar os nomes e a outra para armazenar as contas dos usuários. Se desejássemos imprimir o contato de Paolla de Oli-veira, poderíamos implementar algo como o Programa 64.

```

1  nomes = ["Camila Queiroz", "Paolla de Oliveira"]
2  instagram = ["@camilaqueiroz", "@paollaoliveirareal"]
3  ind = nomes.index("Paolla de Oliveira")
4  print(instagram[ind])

```

Programa 64 – Exemplo de listas para exemplificar as vantagens de utilizar dicionários.

No Programa 65, veja como ficaria a implementação do mesmo exemplo utilizando dicionários:

```
1 contato = {"@camilaqueiroz": "Camila Queiroz",
2           "@paollaoliveirareal": "Paolla de Oliveira"}
3 print(contato["@paollaoliveirareal"])
```

Programa 65 – Acessando elementos em um dicionário.

E aí, o que achou dessa nova estrutura de dados? Legal, não é?! Embora já tenhamos apresentado um exemplo que declara e acessa um dicionário, a sintaxe básica para tal é apresentada no pseudocódigo do Programa 66.

```
1 <dicionario> = {<chave1>: <valor1>,
2               <chave2>: <valor2>,
3               ...
4               <chaveN>: <valorN>}
```

Programa 66 – Sintaxe para declaração de dicionários.

onde **<dicionario>** é o nome da variável do tipo dicionário, **<chave1>**, **<chave2>** e **<chaveN>** são as chaves dos N elementos do dicionário e **<valor1>**, **<valor2>** e **<valorN>** são os correspondentes valores das chaves. Também podemos declarar um dicionário de outra forma. Observe o pseudocódigo do Programa 67.

```
1 <dicionario> = {}
2 <dicionario>[<chave1>] = <valor1>
3 <dicionario>[<chave2>] = <valor2>
4 ...
5 <dicionario>[<chaveN>] = <valorN>
```

Programa 67 – Outra forma de declarar dicionários.

Uma outra maneira de acessar os elementos de um dicionário é usando o método `get()`, cuja sintaxe é a seguinte:

`<dicionario>.get(<chave>)`

Observe o Programa 68 e perceba que ele é uma reescrita do Programa 65:

```
1 contato = {"@camilaqueiroz": "Camila Queiroz",  
2           "@paollaoliveirareal": "Paolla de Oliveira"}  
3 print(contato.get("@paollaoliveirareal"))
```

Programa 68 – Sintaxe do método `get()`.

INSERINDO E ATUALIZANDO ELEMENTOS

Para incluir ou alterar elemento(s) em um dicionário, utilizamos o método `update()`. Quando a chave já existe no dicionário, o valor associado a ela é alterado. Quando não, a chave e o valor são inseridos no dicionário. A sintaxe básica é a seguinte:

`<dicionario>.update({<chave>: <valor>})`

Veja o exemplo no Programa 69.

```
1 contato = {"@camilaqueiroz": "Camila Queiroz",  
2           "@paollaoliveirareal": "Paolla de Oliveira"}  
3 contato.update({"@paollaoliveirareal": "Paolla"})  
4 contato.update({"@sheronmenezes": "Sheron Menezes"})  
5 contato.update({"@bruna_iconica": "Bruna Marquezine"})  
6 print(contato)
```

Programa 69 – Incluindo e alterando elementos em um dicionário.

Na linha 3, o método `update()` é aplicado sobre a chave Paolla de Oliveira, que já existe no dicionário. Assim, ocorrerá que o valor Paolla Oliveira será alterado para Paolla Já nas linhas 4 e 5, os dados sobre as atrizes Sheron Menezes e Bruna Marquezine não estão no dicionário e, portanto, serão incluídos.

CONTANDO ELEMENTOS

A função `len()` recebe como argumento o nome de um dicionário e retorna a quantidade de elementos existentes. A sintaxe básica é a seguinte:

`len(<dicionario>)`

Observe o exemplo que imprime a quantidade de elementos existentes no dicionário `contato`.

```
1 contato = {"@camilaqueiroz": "Camila Queiroz",
2           "@paollaoliveirareal": "Paolla de Oliveira"}
3 print(len(contato))
```

Programa 70 – Exemplo da utilização da função `len()`.

VERIFICANDO A EXISTÊNCIA DE UMA CHAVE

Caso deseje verificar se uma determinada chave existe em um dicionário, pode utilizar o operador `in`. Caso a chave pesquisada esteja no dicionário, será retornado o valor **True** e, em caso negativo, o valor **False**. A sintaxe básica é a seguinte:

```
1 <chave> in <dicionario>:  
2     comandos
```

Programa 71 – Sintaxe do operador **in**.

Observe o exemplo que verifica a existência da chave **@paollaoliveirareal** no dicionário **contato**.

```
1 contato = {"@camilaqueiroz": "Camila Queiroz",  
2           "@paollaoliveirareal": "Paolla de Oliveira"}  
3 if "@paollaoliveirareal" in contato:  
4     print(f"{contato.get('@paollaoliveirareal')}")
```

Programa 72 – Verificando a existência de uma chave.

OBTENDO VALORES E CHAVES

Caso deseje obter as chaves de um dicionário, no formato de uma lista, utilize o método **keys()**, cuja sintaxe básica é a seguinte:

<dicionario>.keys()

Veja um exemplo no Programa 73.

```
1 contato = {"@camilaqueiroz": "Camila Queiroz",  
2           "@paollaoliveirareal": "Paolla de Oliveira",  
3           "@sheronmenezes": "Sheron Menezes",  
4           "@bruna_iconica": "Bruna Marqueline"}  
5 for insta in contato.keys():  
6     print(insta)
```

Programa 73 – Exemplo de utilização do método **keys()**.

Se desejar obter os valores de um dicionário, no formato de uma lista, utilize o método `values()`, cuja sintaxe básica é a seguinte:

`<dicionario>.values()`

Veja um exemplo no Programa 74.

```
1 contato = {"@camilaqueiroz": "Camila Queiroz",
2           "@paollaoliveirareal": "Paolla de Oliveira",
3           "@sheronmenezes": "Sheron Menezes",
4           "@bruna_iconica": "Bruna Marquezine"}
5 for nome in contato.values():
6     print(nome)
```

Programa 74 - Exemplo de utilização do método `values()`.

Caso deseje obter o par `<chave>: <valor>` de um dicionário, no formato de uma tupla, utilize o método `items()`, cuja sintaxe básica é a seguinte:

`<dicionario>.items()`

Veja um exemplo no Programa 75 e observe como ele parece com o método `enumerate()`, utilizado em listas.

```
1 contato = {"@camilaqueiroz": "Camila Queiroz",
2           "@paollaoliveirareal": "Paolla de Oliveira",
3           "@sheronmenezes": "Sheron Menezes",
4           "@bruna_iconica": "Bruna Marquezine"}
5 for insta, nome in contato.items():
6     print(f"{insta} --> {nome}")
```

Programa 75 - Exemplo de utilização do método `items()`.

ORDENANDO ELEMENTOS

Em um dicionário, para retornar os elementos de forma ordenada, tomando como base as chaves, podemos utilizar a função `sorted()`, cuja sintaxe básica é a seguinte:

```
sorted(<dicionario>, reverse = [True|False]),
```

onde `reverse` indica se o ordenamento deve ser reverso. Quando `reverse = True`, o ordenamento é decrescente e quando `reverse = False` (que é o valor padrão e, por isso, não é necessário explicitá-lo), o ordenamento é crescente. Veja o exemplo no Programa 76.

```
1 contato = {"@camilaqueiroz": "Camila Queiroz",
2           "@paollaoliveirareal": "Paolla de Oliveira",
3           "@sheronmenezes": "Sheron Menezes",
4           "@bruna_iconica": "Bruna Marquezine"}
5 for insta, nome in sorted(contato.items()):
6     print(f"{insta} --> {nome}")
```

Programa 76 – Exemplo de utilização da função `sorted()`.

No entanto, se precisássemos que o ordenamento do dicionário fosse baseado nos seus valores e não em suas chaves, a função `sorted()`, em sua sintaxe básica, não atenderia a nossa necessidade. Para tal, precisaríamos utilizar uma outra função chamada `itemgetter()`.

Possivelmente, utilizando o dicionário que definimos no Programa 76, não perceberá o funcionamento de `itemgetter()`. Portanto, faremos uma modificação no dicionário de forma que, em vez dos valores armazenarem nomes de atrizes, serão armazenadas as estaturas das atrizes em me-

tros. Assim, o ordenamento considerará o valor e não a chave, como se observa no Programa 77.

```
1  from operator import itemgetter
2  contato = {"@camilaqueiroz": 1.77,
3             "@paollaoliveirareal": 1.70,
4             "@sheronmenezes": 1.67,
5             "@bruna_iconica": 1.70}
6  for insta, estatura in sorted(contato.items(), key=itemgetter(1)):
7      print(f"{insta} --> {estatura:.2f}m")
```

Programa 77 – Exemplo de utilização da função `itemgetter()`.

Antes de utilizar a função `itemgetter()`, é preciso importá-la, como escrito na linha 1. Na linha 6, o valor 1, enviado à função, significa que o ordenamento será realizado com base no valor do dicionário e o valor 0 indica o ordenamento pela chave (porém já vimos que se precisa ordenar com base nas chaves, basta utilizar `sorted()` em sua sintaxe básica).

CLONANDO UM DICIONÁRIO

Para criar uma cópia de um dicionário, pode utilizar a função `dict()`, cuja sintaxe básica é a seguinte:

```
dict(<dicionario>)
```

onde `<dicionario>` é o dicionário que está sendo clonado. Como resultado, o valor retornado pode ser atribuído a uma variável, como no exemplo abaixo:

Veja um exemplo no Programa 78.

```
1 contato = {"@camilaqueiroz": "Camila Queiroz",
2           "@paollaoliveirareal": "Paolla de Oliveira",
3           "@sheronmenezes": "Sheron Menezes",
4           "@bruna_iconica": "Bruna Marquezine"}
5 copia = dict(contato)
6 print(copia)
```

Programa 78 – Exemplo de clonagem de um dicionário.

REMOVENDO ELEMENTOS

Para remover elementos de um dicionário, pode utilizar o método `pop()` informando qual chave deseja remover, cuja sintaxe básica é a seguinte:

`<dicionario>.pop(<chave>)`

Veja um exemplo no Programa 79.

```
1 contato = {"@camilaqueiroz": "Camila Queiroz",
2           "@paollaoliveirareal": "Paolla de Oliveira",
3           "@sheronmenezes": "Sheron Menezes",
4           "@bruna_iconica": "Bruna Marquezine"}
5 contato.pop("@bruna_iconica") #Remove os dados de Bruna Marquezine
6 print(contato)
```

Programa 79 – Exemplo de utilização do método `pop()`.

ESVAZIANDO UM DICIONÁRIO

Caso deseje esvaziar um dicionário, utilize o método `clear()`, mas cuidado! Se precisar obter esses dados

posteriormente, eles não estarão mais disponíveis porque todos os valores e chaves serão retirados da memória. Caso precise utilizá-los posteriormente, faça uma cópia do dicionário antes de limpá-lo. A sintaxe básica do método `clear()` é a seguinte:

`<dicionario>.clear()`

Veja um exemplo no Programa 80.

```
1 contato = {"@camilaqueiroz": "Camila Queiroz",
2           "@paollaoliveirareal": "Paolla de Oliveira",
3           "@sheronmenezes": "Sheron Menezes",
4           "@bruna_iconica": "Bruna Marquezine"}
5 copia = dict(contato)
6 contato.clear()
7 print(contato)
8 print(copia)
```

Programa 80 – Exemplo de utilização do método `clear()`.

EXERCÍCIOS RESOLVIDOS

1. Construa um programa que utilize um dicionário cujas chaves são os códigos do produto e os valores são o nome do produto, o preço unitário e a quantidade comprada, como no exemplo a seguir.

CHAVE	VALOR		
Código	Nome	Preço Unitário	Qtde Comprada
1	Monitor LED 24"	599,99	1
2	Teclado wireless	49,26	1
3	Mouse wireless	19,90	1
4	Cartucho colorido	54,00	2

A partir do dicionário, o programa deve imprimir os itens da compra e calcular o subtotal de cada um deles, ou seja, quantidade * preço unitário. Por fim, o programa deve apresentar o valor total da compra.

Comentários sobre a resolução

Se o problema não tivesse enunciado que o código do produto é a chave do dicionário, conseguiria perceber isso? Observe que ele não se repete e, portanto, ele é passível de ser a chave do dicionário.

Veja que os dados do produto podem ser modelados como sendo uma lista. Então, vamos pensar que o código do produto é a chave do dicionário e os valores são os elementos da lista.

Vamos ao código:

```
1  produtos = {1: ['Monitor LED 24"', 599.99, 1],
2              2: ['Teclado wireless', 49.26, 1],
3              3: ['Mouse wireless', 19.9, 1],
4              4: ['Cartucho colorido', 54, 2]}
5  total = 0
6  for cod, prod in produtos.items():
7      subtotal = produtos[cod][1] * produtos[cod][2]
8      print(f"{ prod[0]}: R$ {subtotal:.2f}")
9      total += subtotal
10 print(20 * "-")
11 print(f"Total: R$ {total:.2f}")
```

Programa 81 – Exercício resolvido 6.1.

Nas linhas 1 a 4, é realizada a declaração do dicionário. Na linha 6, a cada iteração, `produtos.items()` retornará a chave do dicionário (isto é, os códigos dos produtos) que será armazenada na variável `cod` e os valores que correspondem a cada chave (ou seja, uma lista contendo nome, valor unitário e quantidade

comprada) armazenada na variável **prod**.

Na linha 7, o subtotal de cada produto é calculado. Perceba que, na primeira iteração, **cod** armazenará o valor 1 e **prod** a lista `['Monitor LED 24"', 599.99, 1]`. Portanto, **produtos[cod][1]** significa “pegue o elemento do dicionário produtos cuja chave é 1 e, em seguida, acesse a lista, a qual está vinculada a essa chave, e obtenha o elemento com índice 1”. Na prática, o código estaria obtendo o preço unitário do produto. Por fim, **produtos[cod][2]** acessará o elemento do dicionário **produtos** com chave 1 e obterá o elemento do índice 2, que é a quantidade comprada do produto. Depois que esses dois valores são extraídos do dicionário, é realizada a multiplicação entre ambos e o resultado é armazenado na variável **subtotal**. Na linha 8, são escritos o nome do produto (que corresponde ao elemento armazenado no índice 0 da lista **prod**) e o valor do subtotal do produto. Na linha 9, os valores dos subtotais de cada produto são acumulados na variável **total**.

As instruções definidas nas linhas 7 – 9 ocorrem para todos os elementos do dicionário.

2. No exemplo anterior, criamos um dicionário contendo quatro elementos que foram definidos por nós mesmos. Porém, na prática, ocorre que os dados são informados pelo utilizador. Portanto, neste exercício, modificaremos o programa anterior de modo que os dados sejam informados pelo utilizador.

Comentários sobre a resolução

Vamos implementar esse programa imaginando que o progra-

ma ficará solicitando que o utilizador informe os dados até o momento que informar que não deseja mais continuar.

Vamos ao código:

```
1  produtos = {}
2  while True:
3      cod = int(input("Código: "))
4      nome = input("Nome: ")
5      preco = float(input("R$: "))
6      qtde = int(input("Qtde: "))
7      prod = []
8      prod.append(nome)
9      prod.append(preco)
10     prod.append(qtde)
11     produtos.update({cod: prod})
12     resp = input("Deseja continuar [S|N]? ")
13     if resp == "N" or resp == "n":
14         break
15     total = 0
16     for cod, prod in produtos.items():
17         subtotal = produtos[cod][1] * produtos[cod][2]
18         print(f"{prod[0]}: R$ {subtotal:.2f}")
19         total += subtotal
20     print(20 * "-")
21     print(f"Total: R$ {total:.2f}")
```

Programa 82 – Exercício resolvido 6.2.

Na linha 1, o dicionário **produtos** é declarado sem nenhum elemento. Na linha 3, deve ser informado o código do produto, o qual será utilizado como chave de **produtos**. Nas linhas 4 a 6, são informados nome, preço e quantidade comprada do produto, respectivamente.

Na linha 7, a lista `prod` é declarada vazia porque ela deve conter apenas os dados informados pelo utilizador e, nas linhas 8 a 10, esses dados são adicionados à lista `prod`. Na linha 13, caso o utilizador informe que não deseja continuar, o laço `while True` (linha 2) será encerrado. As linhas 15 a 21 comportam-se de forma semelhante às linhas 05 a 11 do Programa 81.

3. Crie um programa para cadastrar matrícula, nome e salário dos funcionários de uma empresa. Ao fim, o programa deve mostrar todos os funcionários cadastrados e, em seguida, aumentar para R\$ 1.800,00 o salário dos funcionários com salários inferiores a R\$ 1.500,00. Depois do aumento salarial, o programa deve exibir a nova listagem.

Comentários sobre a resolução

Podemos modelar esse problema considerando a matrícula como sendo a chave do dicionário e os outros dados como elementos de uma lista. Depois de cadastrar os funcionários, o programa deve utilizar uma estrutura condicional para verificar se o salário do funcionário é inferior a R\$ 1.500,00. Em caso positivo, aumentar o respectivo salário para R\$ 1.800,00 por meio do método `update()`.

Vamos ao código:

```

1  funcionarios = {}
2  while True:
3      mat = int(input("Matrícula: "))
4      nome = input("Nome: ")
5      sal = float(input("Salário: R$ "))
6      func = []
7      func.append(nome)
8      func.append(sal)
9      funcionarios.update({mat: func})
10     resp = input("Deseja continuar [S|N]? ")
11     if resp == "N" or resp == "n":
12         break
13     for matricula, dados in funcionarios.items():
14         if dados[1] < 1500:
15             dados[1] = 1800
16             funcionarios.update({matricula: dados})
17     for matricula, dados in funcionarios.items():
18         print(f"Matrícula: {matricula}")
19         print(f"Nome: {dados[0]}")
20         print(f"Salário: {dados[1]:.2f}\n")

```

Programa 83 – Exercício resolvido 6.3.

Na linha 13, observe que os dados do funcionário são armazenados na variável **dados** como sendo uma lista. Na linha 8, o salário do funcionário é incluído na lista **func**, no índice 1 (perceba que o nome do funcionário é o primeiro elemento adicionado à lista e, por isso, está no índice 0). Portanto, na linha 14, é verificado se o salário do funcionário é inferior a R\$ 1.500,00. Em caso positivo, o salário é alterado e, na linha 16, essa modificação é aplicada ao dicionário.

A impressão do dicionário é realizada nas linhas 17 a 20.

EXERCÍCIOS PROPOSTOS

Para todas as questões abaixo, utilize dicionários.

1. Um utilizador do departamento de vendas de uma empresa necessita de um relatório que apresente seus clientes potenciais. Para isso, é necessário que o relatório seja ordenado do cliente que mais comprou para o que menos comprou. Os dados de entrada são razão social e valor total de compras. Considere a razão social como sendo a chave identificadora do cliente.
2. Construa um programa no qual o utilizador informe o nome, a estatura e o peso de vários alunos de uma turma. Após o cada-tro, o programa deve imprimir o nome $IMC = \frac{\text{peso}}{\text{altura}^2}$. cada aluno ordenados pelo nome do aluno. Sabe-se que:
3. Construa um programa que cadastre diversos voos aéreos, bem como sua origem e seu destino. Considere o número do voo como sendo a chave. Com base no que foi armazenado no dicionário, o programa deve informar a quantidade de voos cuja origem é Natal.
4. Com base no dicionário da questão anterior, construa um programa para remover os voos cujo destino é Recife. Em seguida, imprima a nova listagem de voos.
5. Ainda com base no dicionário da questão 3, construa um programa em que, após os voos terem sido cadastrados, o utilizador possa modificar a origem e/ou o destino de um determinado voo. Ao fim, o programa deve imprimir a nova listagem de voos.

6. Crie um programa para uma nova plataforma de vídeo sob demanda o qual deve armazenar o título da série e o nome dos 2 principais atores. Ao final, o programa deve exibir uma listagem contendo, de forma ordenada, o nome da série e os nomes dos atores.

7. Modifique o programa anterior de modo que o utilizador informe o nome de uma série e o novo programa indique os nomes dos atores principais. Caso a série não esteja cadastrada, o programa deve informar isso ao utilizador.

8. Construa um programa que utilize um dicionário para representar a tabela abaixo.

Código	Nome	Valor (R\$)
1	Monitor LED 24"	599,99
2	Teclado wireless	49,26
3	Mouse wireless	19,90
4	Cartucho colorido	54,00

O programa deve aplicar um desconto de 10% sobre os produtos com o valor acima de R\$ 50,00 e acrescentar à descrição a **string** (em promoção). O novo dicionário ficará como a tabela abaixo.

Código	Nome	Valor (R\$)
1	Monitor LED 24" (em promoção)	539,99
2	Teclado wireless	49,26
3	Mouse wireless	19,90
4	Cartucho colorido (em promoção)	48,60

Utilize a tabela abaixo para implementar as questões 9 e 10.

CADASTRO DE FUNCIONÁRIOS

Matrícula	Nome	Sexo	Departamento	TempoServiço	Salário
1	Ana	F	TI	7	3200,00
2	Beatriz	F	TI	4	3720,00
3	Carla	F	TI	1	2100,00
4	Daniela	F	RH	2	3920,00
5	Emílio	M	RH	7	4235,12
6	Fernando	M	Marketing	7	1200,00
7	Gabriela	F	Marketing	8	7234,89
8	Hernandes	M	TI	6	4234,12
9	Ítalo	M	RH	13	13934,23
10	Janaína	F	RH	7	9341,89

9. Crie um dicionário para armazenar os dados da tabela anterior. Em seguida, o programa deve imprimir uma listagem de mulheres do setor de TI que recebem acima de R\$ 3.000,00.

10. Modifique a questão anterior para listar os homens que não são do setor de TI.

A resolução das questões propostas deste capítulo está disponível em:

https://github.com/peoolivro/codigos/tree/master/Cap6_Exercicios

Para complementar o conteúdo apresentado neste capítulo, pode acessar o canal Procópio na Rede e assistir à seguinte *playlist*:

<https://www.youtube.com/playlist?list=PLQsF5rntN2nY5VvFyUTqtD2M63MWVspS3>

FUNÇÕES

Todos nós temos tarefas e atividades rotineiras que são tediosas. Chato, correto? Agora, imagine a possibilidade de concentrar essas atividades em uma “palavra mágica”, que ao pronunciá-la, automaticamente realizaria todas as atividades. Não seria legal? Seria ótimo. Essa história parece um pouco com o que acontece com aquele herói famoso da DC, quando ele grita “Shazam!”. Brincadeiras à parte, é assim que acontece com o mundo da programação. Reunimos um conjunto de comandos e damos um nome para eles. Quando chamamos esse nome, todo o código é executado. E qual a vantagem disso? A resposta é o reuso de código, facilidade de manutenção e melhor entendimento deste.

O que acabamos de explicar no parágrafo anterior, em programação, chamamos de função. Portanto, função é um nome para um conjunto de comandos que pode ser chamado várias vezes em pontos diferentes do programa, sem a necessidade de repetição de código. Essa história é velha, começou em 1969 quando Dijkstra (vale a pena pesquisar sobre ele) provava que a modularização era uma chave para organização de códigos.

Continuando as explicações sobre funções, lembremos da matemática. Nessa ciência, um dos principais assuntos são as funções. São exemplos bem conhecidos: função do primeiro grau, função do segundo grau, funções exponenciais, funções logarítmicas, funções trigonométricas etc. E daí? Bom,

para facilitar, lembre-se das funções afins, também chamadas de funções do primeiro grau. Essas funções possuem a forma **$f(x) = ax + b$** , onde **a** e **b** são números reais e **a** não pode ser zero. Sabendo disso e de posse de uma função do primeiro grau, pode passar um valor para **x** e encontrar o valor de **f(x)**. Como exemplo prático, temos:

$$f(x) = ax + b \therefore f(x) = 3x + 4 \therefore f(2) = 3 * 2 + 4 = 10$$

Veja que no exemplo acima foi passado o valor 2 para x e o resultado foi $f(2) = 10$. Que maravilha! Podemos fazer algo semelhante com funções no mundo da programação. Pense em uma função como sendo uma caixa preta, onde en-via um valor e recebe um resultado. Até aqui, o mais interessante disso tudo é que já utilizamos várias funções em nossos programas Python. Quer uma prova? Lembre-se de `int()`, de `float()`, de `input()`, de `print()`, de `max()`, de `range()` etc. Então, vamos logo aprender o que são funções.

DECLARANDO FUNÇÕES

Para declarar uma função em Python, precisamos conhecer quatro coisas: a palavra-chave `def`, o nome da função, a lista de parâmetros e o corpo (bloco de comandos). A declaração de uma função deve ocorrer conforme a Figura 21.

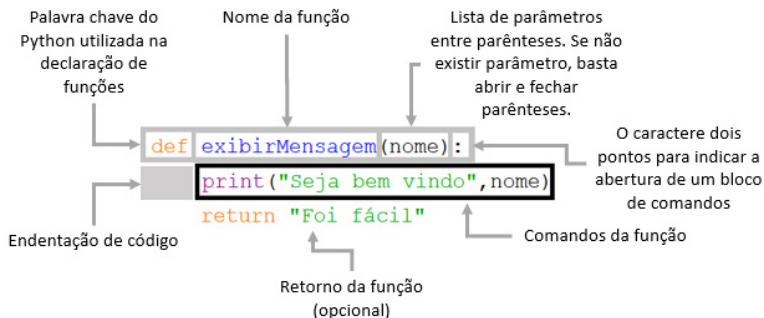


Figura 21 – Declaração de uma função em Python.

Conforme observado na Figura 21, o programador deve utilizar a palavra-chave **def** para iniciar a declaração de uma função. Em seguida, definir o nome da função, que não deve iniciar com números nem símbolos. Na sequência, abrem-se parênteses para inserir a lista de parâmetros que podem ser passados para função (falaremos mais sobre isso adiante). Agora, veja na prática um exemplo de declaração de função no Programa 84.

```
1 def exibirMensagem():  
2     print("Python é fácil")  
3 exibirMensagem()  
4 exibirMensagem()  
5 exibirMensagem()
```

Programa 84 – Exemplo de declaração de função.

No Programa 84, foi declarada uma função chamada **exibirMensagem**, cujo corpo, ou seja, bloco de comandos, tem apenas um comando de impressão com a frase **"Python é fácil"**. Perceba que uma vez declarada a função, podemos

chamá-la diversas vezes. É como se estivéssemos trocando o comando `print("Python é fácil")` por `exibirMensagem()`. Talvez, esse exemplo não faça muito sentido para , porque temos apenas um comando de impressão. Tudo bem! Mas e se tivéssemos mais comandos, como seria a situação? Vê o benefício agora?

Antes de prosseguir, uma informação importante: Python é uma linguagem interpretada, ou seja, a execução dos comandos ocorre sequencialmente, obedecendo, basicamente, a ordem e disposição das linhas de comando. Isso significa dizer que não se pode chamar uma função antes de sua declaração. Portanto, evite receber a seguinte mensagem: **NameError: name 'nomeFuncao' is not defined**. Veja um exemplo de simulação de erro no Programa 85.

```
1  exibirMensagemBoasVindas()  
2  
3  def exibirMensagemBoasVindas():  
4      print("Seja bem vindo")
```

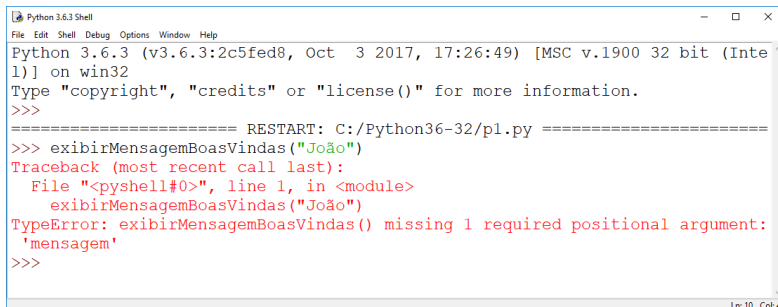
Programa 85 – Exemplo de chamada de função ainda não declarada.

Nos dois programas anteriores, declaramos funções sem parâmetros, mas como seria uma função com parâmetros? Vejamos no exemplo a seguir.

```
1  def exibirMensagemBoasVindas(pessoa, mensagem):  
2      print(f"{mensagem}, {pessoa}")  
3  
4  exibirMensagemBoasVindas("João","Oi") #Saída: Oi, João
```

Programa 86 – Exemplo de função com parâmetros.

Observe que entre os parênteses, utilizamos nomes para os parâmetros da função. Nesse caso, são parâmetros nomeados e obrigatórios, ou seja, não se pode tentar chamar a função sem a presença deles. Veja o que acontece se tentar fazer esse destempero.



```
Python 3.6.3 Shell
File Edit Shell Debug Options Window Help
Python 3.6.3 (v3.6.3:2c5fed8, Oct 3 2017, 17:26:49) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:/Python36-32/pl.py =====
>>> exibirMensagemBoasVindas("João")
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    exibirMensagemBoasVindas("João")
TypeError: exibirMensagemBoasVindas() missing 1 required positional argument:
'mensagem'
>>>
```

Figura 22 – Erro de chamada de função com parâmetros obrigatórios.

Caso seja necessário, pode passar os parâmetros nomeados. Nesse caso não importa a ordem dos parâmetros. Veja essa prática no Programa 87.

```
1 def exibirMensagemBoasVindas(pessoa, mensagem):
2     print(f"{mensagem}, {pessoa}")
3
4     exibirMensagemBoasVindas(mensagem = "Bom dia", pessoa = "Ana")
5 #Saída: Bom dia, Ana
```

Programa 87 – Passagem de parâmetros nomeada.

ainda pode declarar funções com parâmetros contendo valores padrão. Isso significa dizer que agora pode até ignorar a passagem de parâmetros. Veja no Programa 88.


```
1 def exibirMensagemBoasVindas(pessoa = "Fulano", mensagem = "Oi"):  
2     print(f"{mensagem}, {pessoa}")  
3  
4 exibirMensagemBoasVindas() #Saída: Oi, Fulano
```

Programa 88 – Função com parâmetros contendo valores padrão.

Depois de ter visto alguns exemplos de declaração de funções, veremos como resolver alguns exercícios clássicos de funções.

EXERCÍCIOS RESOLVIDOS

1. É muito comum precisar calcular a soma dos elementos de uma lista. Esse tipo de atividade pode ser necessário em vários pontos de um programa. Embora as listas possuam um método chamado `sum()`, que é utilizado para calcular a soma dos seus elementos, apenas para que compreenda a utilização de funções, crie uma função para calcular a soma dos elementos de uma lista de inteiros.

```
1 def somarElementosLista(inteiros):  
2     soma = 0  
3     for valor in inteiros:  
4         soma = soma + valor  
5     return soma  
6 print(somarElementosLista([3,4,6,9,10,23,13])) #Resultado 68
```

Programa 89 – Exercício resolvido 7.1.

Comentários sobre a resolução

Nós consideramos que esse tipo de solução é ferramenta obri-

gatória para todo aspirante a programador. Inclusive, já tratamos sobre esse assunto no Capítulo 5, mas como encapsular esse tipo de código em uma função? Esse é nosso trabalho.

A primeira coisa a fazer é definir um nome significativo para função. De preferência, um nome que qualifique bem o que ela faz. Não tenha medo de nomes grandes, só não use “espaços” e “caracteres especiais”. Falo de “ç”, “á”, “õ” etc. No exemplo acima, optamos por chamar a função de **somarElementosVetor**. Veja que a partir da leitura do nome, já temos elementos suficientes para entender o significado da função.

Em seguida, pensamos e definimos os parâmetros da função. Fica bem claro que se ela se destina a somar os elementos de uma lista, precisaremos de um parâmetro para recebê-la. Optamos então por chamar esse parâmetro de **inteiros**, já que se trata de uma lista de números inteiros. No corpo da função, criamos uma variável auxiliar, denominada **soma**, que é inicializada com zero. Em seguida, declaramos a estrutura de um laço **for** para listas. O objetivo da estrutura citada, já deve imaginar, é que cada elemento da lista tenha sequencialmente atribuído a variável valor. Dentro do **for**, ocorre o procedimento de soma utilizando a variável auxiliar **soma**. Após a finalização do **for**, a função retorna o valor da soma dos elementos da lista. Para isso, utiliza-se a palavra-chave **return**.

2. Crie uma função Python para calcular o fatorial de um número fornecido pelo utilizador.

```
1 def fatorial(n):
2     if n == 0 or n == 1:
3         return 1
4     else:
5         return n * fatorial(n - 1)
6 for valor in [0,1,2,3,4,5,6]:
7     print(fatorial(valor))
```

Programa 90 – Exercício resolvido 7.2.

Comentários sobre a resolução

Aproveitaremos o problema acima para explicar dois assuntos bastante interessantes: o conceito de pilha e a recursividade de funções.

Quando um programa está em execução, ele se chama processo. A área da memória RAM reservada para o processo é dividida em áreas com diferentes propósitos. Uma delas é a pilha, ou *stack*. Seu principal propósito é o armazenamento das chamadas de função que ocorrem durante a execução de um programa bem modularizado. Além disso, esse espaço de memória também é utilizado para alocação de variáveis locais de uma função, que nesse caso só existem enquanto a função está em execução. No entanto, vamos nos ater somente ao conceito de pilha.

A pilha, como o próprio nome se refere, é uma área de memória que funciona com a estratégia LIFO (*Last In First Out*). Tudo que é colocado lá fica no topo e aquilo que for colocado por último na pilha será o primeiro a ser retirado, exatamente como uma pilha de pratos ou de cartas de baralho. Bom, deve estar se perguntando, como isso se relaciona com a solução de nosso último problema? Vamos lá! Primeiro vamos lembrar

como se calcula um número fatorial:

$$0! = 1$$

$$1! = 1$$

$$2! = 2 * 1 = 2$$

$$3! = 3 * 2 * 1 = 6$$

$$4! = 4 * 3 * 2 * 1 = 24$$

Como se pode ver, o fatorial é a multiplicação dos termos antecessores de um número natural, até o número 1. Sabendo disso, na resposta do problema, foi criada uma função denominada fatorial, que recebe um parâmetro `n`. O valor `n` é testado na primeira linha de bloco de comandos, a partir do seguinte teste: `if n==0 and n==1`. Caso isso seja verdade, ou seja, `n` for igual a 0 ou 1, a função retornará o valor 1. Exatamente porque o fatorial de zero, por definição, é também igual a 1. Caminhando no código, temos a condição `else`, que abre caminho para execução do seguinte comando; logicamente, caso o resultado do primeiro teste seja falso: `return n*(fatorial(n-1))`.

Uma coisa interessante apareceu nesse código: uma chamada à própria função. Como diria um narrador esportivo, ao questionar o árbitro brasileiro que dirigiu a final da copa do mundo de 1982, na Espanha, entre Itália e Alemanha: “Pode isso, Arnaldo?” Podemos afirmar que sim. Esse recurso se chama recursividade, ou seja, quando uma função invoca a si mesma. Todo esquema de recursividade exige uma condição de parada. No nosso caso, a condição de parada do problema foi o teste do `if`, se `n` for 0 ou 1, a função não mais chama a si próprio. E o que acontece se não tivermos uma condição de parada? Ocorre um erro chamado de *stack overflow*, ou seja, estouro de pilha.

What? Para explicar com mais calma, veja a tabela abaixo:

Tabela 4 – Demonstração de uso da pilha com o armazenamento da chamada de função recursiva.

CÓDIGO	PILHA	COMENTÁRIO
fatorial(3)	fatorial(3)	A chamada da função fatorial com parâmetro 3 é colocada na pilha
return 3 * fatorial(2)	fatorial(2) fatorial(3)	O código de fatorial(3) está na pilha, porém, deve esperar a resposta de fatorial(2)
return 2 * fatorial(1)	fatorial(1) fatorial(2) fatorial(3)	Agora, estão na pilha: fatorial(3) e fatorial(2). Só que fatorial(2) exige a solução de fatorial(1) que também é colocada na pilha
return 1	fatorial(1) #Retorna 1 fatorial(2) fatorial(3)	No caso de fatorial 1, n é 1, então ele retornará 1. Assim, fatorial(1) deve sair da pilha.
return 2 * 1	fatorial(2) #Retorna 2 fatorial(3)	Como fatorial(1) foi resolvido, agora fatorial(2) pode ser resolvida também, utilizando o retorno de fatorial(1). Nesse caso, a resposta de fatorial(2) é 2.
return 3 * 2	fatorial(3) #Retorna 6	Por fim, fatorial(3) que aguardava na pilha, recebe a resposta de fatorial(2) e consegue calcular a resposta final, que no caso é 6.

O tal do estouro de pilha, citado no parágrafo anterior, significa que a área de memória destinada a pilha tem espaço limitado. Se colocar muitas chamadas de função dentro da pilha, pode estourar o limite de espaço disponível. Para finalizar, aplicamos um teste, passando elementos de um vetor para a função fatorial.

3. Agora, crie uma função Python que calcule o desvio padrão dos elementos de uma lista.

$$\text{Desvio padrão} = \sqrt{\frac{1}{n-1} \sum_{i=1}^n (v[i] - m)^2}$$

onde n é o número de elementos e m é a média dos valores.

```

1  def calcularMedia(lista):
2      soma = 0
3      for valor in lista:
4          soma += valor
5      return float(soma/len(lista))
6
7  def calcularDesvioPadrao(lista):
8      n = len(lista)
9      m = calcularMedia(lista)
10     soma = 0
11     import math
12     for valor in lista:
13         soma = soma + math.pow(valor - m, 2)
14     return math.sqrt((1/(n-1))*soma)
15
16  lista = [3,6,2,9,10,45,36,78,42,100]
17  print(calcularDesvioPadrao(lista))

```

Programa 91 – Exercício resolvido 7.3.

Comentários sobre a resolução

Uma primeira dica: quando se deparar com operações matemáticas que devem ser transformadas em código, gaste um tempo avaliando a expressão. O símbolo \sum significa um somatório que é controlado pelos limites abaixo e acima do símbolo. Isso geralmente se resolve com estruturas de repetição.

Iniciamos a resolução do problema com a definição de uma função para cálculo da média aritmética dos elementos de uma lista (`def calcularMedia(lista)`), porque o cálculo de desvio padrão exige o valor da média aritmética dos valores e porque modularizar o código é uma boa prática. Eventualmente pode precisar novamente do cálculo da média. Nesse caso, basta chamar a função que já está pronta. Na sequência, criamos a função para cálculo do desvio padrão (`def calcularDesvioPadrao(lista)`). A função média simplesmente soma os elementos de um vetor, conforme solução do Exercício 7.1 e, em seguida, divide a soma pelo número de elementos utilizando a função `len(soma)`. Na função `calcularDesvioPadrao`, temos a criação de três variáveis: `n`, o número de elementos do vetor, `m`, que recebe o cálculo da média dos elementos da lista e `soma`, que é inicializada com 0. Na sequência, utilizamos o laço `for` para realizar a operação de soma (`soma = soma + math.pow(valor - m, 2)`). Por fim, retornamos o cálculo da raiz quadrada `math.sqrt((1/(n-1))*soma)`.

EXERCÍCIOS PROPOSTOS

Crie um arquivo chamado `MinhasFuncoes.py`, o qual deverá conter todas as funções correspondentes às questões de 1 a 10. Para cada questão, crie um arquivo para testar as respectivas funções `TestaFuncao.py`.

Crie um programa que apresente o seguinte menu

Cálculo das áreas de figuras geométricas:

1. **Círculo**
2. **Triângulo**
3. **Retângulo**

Qual figura deseja calcular a área? ____

De acordo com a opção informada, o programa deve chamar a função `calcula_area_circulo()`, `calcula_area_triangulo()` ou `calcula_area_retangulo()`. Cada uma delas deve solicitar ao utilizador os parâmetros necessários para calculá-lo da área. As funções devem retornar o valor calculado.

2. Crie duas funções: a) `gera_matriz_aleatoria()` cujos argumentos de entrada são número de linhas, número de colunas, intervalo inicial dos valores gerados e intervalo final dos valores gerados a qual tem como retorno uma matriz e; b) `calcula_traco_matriz()`, cujo argumento é a matriz obtida por meio de `gera_matriz_aleatoria()`, deve retornar o traço da matriz. Ao fim, `TestaFuncao` deve escrever a matriz e o seu respectivo traço.

3. Crie uma função chamada `soma_matrizes()` cujos parâmetros são as matrizes A e B, de mesma ordem, obtidas por meio de `gera_matriz_aleatoria()`. A função deve retornar uma nova matriz C, resultado de $A + B$. O seu programa `TestaFuncao` deve escrever A, B e C.

4. Crie uma função chamada `multipla_matriz_por_constante()`, cujos parâmetros de entrada são a) uma matriz, obtida por meio de `gera_matriz_aleatoria()` e b) uma cons-

tante k. A função deve executar a operação $k * A$ e retornar a nova matriz C. **TestaFuncao** deve escrever A e $k * A$.

5. Crie uma função cujo argumento de entrada seja um dicionário com a estrutura

```
series = {título: [protagonista1, protagonista2]}.
```

A função deve retornar os nomes das séries e os nomes dos seus respectivos protagonistas ordenados alfabeticamente. **TestaFuncao** deve imprimir o dicionário original e o ordenado.

6. Crie uma função chamada **obtem_dados_funcionarios()**, que retorne um dicionário com os dados abaixo.

CADASTRO DE FUNCIONÁRIOS

Matrícula	Nome	Sexo	Departamento	TempoServiço	Salário
1	Ana	F	TI	7	3200,00
2	Beatriz	F	TI	4	3720,00
3	Carla	F	TI	1	2100,00
4	Daniela	F	RH	2	3920,00
5	Emílio	M	RH	7	4235,12
6	Fernando	M	Marketing	7	1200,00
7	Gabriela	F	Marketing	8	7234,89
8	Hernandes	M	TI	6	4234,12
9	Ítalo	M	RH	13	13934,23
10	Janaína	F	RH	7	9341,89

7. Com base no dicionário obtido por meio de **obtem_dados_funcionarios()**, construa uma função que retorne a quantidade de homens e de mulheres cadastrados.

8. Use o dicionário obtido por `obtem_dados_funcionarios()` e construa uma função que retorne um dicionário com os dados dos funcionários cujo tempo de serviço seja maior que 5 anos.

9. Ainda usando a função `obtem_dados_funcionarios()`, construa uma função que retorne um dicionário com os dados das funcionárias cadastradas. A função deve se chamar `lista_mulheres_por_setor()` e os parâmetros são o cadastro de todos os funcionários e um determinado setor da empresa.

10. Utilize o dicionário informado por `obtem_dados_funcionarios()` e construa uma função que recebe o sexo como argumento e retorna a média salarial dos funcionários daquele sexo.

A resolução das questões propostas deste capítulo está disponível em:

https://github.com/peoolivro/codigos/tree/master/Cap7_Exercicios

Para complementar o conteúdo apresentado neste capítulo, pode acessar o canal Procópio na Rede e assistir à seguinte *playlist*:

<https://www.youtube.com/playlist?list=PLqsF5rntN2nZpgE33AYNp6meEJ41pmdya>

INTRODUÇÃO À ORIENTAÇÃO A OBJETOS

Nos capítulos anteriores, foram apresentados os conceitos relacionados ao paradigma da programação estruturada. Nesse paradigma, os programas são compostos por três tipos de estruturas: sequência (comandos executados), decisões (condições) e iterações (repetições). A principal característica é a sequência contínua de comandos, podendo ser dividida em funções ou não, facilmente representada por um fluxograma. A programação estruturada é muito popular no aprendizado de programação, tanto que os currículos dos cursos da área de computação trazem o ensino da programação estruturada antes da programação orientada a objetos.

Falando em programação orientada a objetos, neste capítulo vamos tratar desse paradigma. Esse tema seria assunto para um livro inteiro. Por isso, faremos apenas uma introdução aqui. Este paradigma visa uma programação mais voltada ao mundo real, ao pensamento humano. Para viabilizar isso, fazemos uso do conceito de *abstração*.

Considere a compra de um carro zero. O cliente, quando vai em uma concessionária, está preocupado com algumas características do carro, dentre elas o valor, o modelo, a cor, a motorização, a desvalorização, os itens acessórios e o pós-venda. Difícilmente ele estará interessado onde o carro foi montado, qual o fabricante dos vidros e dos faróis ou qual a

marca das válvulas. Resumindo, o cliente deu mais importância a algumas características do que a outras no contexto da compra do carro, ou seja, ele focou no que achou necessário para tomar sua decisão. Em uma oficina mecânica, o contexto seria diferente e as características importantes, conseqüentemente, seriam outras. Considerando o paradigma da orientação a objetos, o cliente usou o conceito de abstração, pois ele ressaltou algumas características em detrimento de outras. Dizemos que as características não incluídas foram abstraídas.

A seguir, veremos alguns conceitos da programação orientada a objetos, bem como sua implementação na linguagem Python. Aliás, vale ressaltar que Python é uma linguagem totalmente orientada a objetos. Embora possa utilizar o paradigma da programação estruturada, tudo em Python é objeto, conforme podemos observar na Figura 23. Lembre-se que o comando `type()` retorna o tipo de uma variável ou de um valor. Note que as saídas para os valores testados sempre retornam `class` e logo após o tipo da classe.

```
Python 3.7.3 (default, Mar  6 2020, 22:34:30)
[Clang 11.0.3 (clang-1103.0.32.29)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> type(4)
<class 'int'>
>>> type(3.4)
<class 'float'>
>>> type('José')
<class 'str'>
>>> type(True)
<class 'bool'>
>>> type([2, 7, 3, 9])
<class 'list'>
>>> █
```

Figura 23 – Comando `type()` aplicado aos tipos de dados em Python.

CLASSES E OBJETOS

Embora o nome do paradigma seja Programação Orientada a Objetos, o conceito central é o de classe. É necessário que se defina uma classe para então se criar objetos com base nela. Assim, uma classe é a representação de um conjunto de objetos com características em comum (atributos e comportamentos), sendo uma abstração desses objetos no mundo real. A classe é o molde para criação de objetos, assim, um objeto é a concretização de uma classe. Dizemos que um objeto é a instância de uma classe.

Em uma agência bancária, por exemplo, temos os clientes e suas respectivas contas. No caso da modelagem inicial de um sistema bancário, teríamos uma classe representando os clientes e outra representando as contas, conforme a Figura 24, que seriam as abstrações dos conceitos reais já mencionados. O cliente pode possuir diversos atributos no mundo real, mas considere que para o sistema em desenvolvimento só importará o nome, o telefone e o CPF (os demais atributos foram abstraídos). De forma análoga, a conta contém os atributos número, saldo e cliente, e ainda foram definidos alguns comportamentos para essa classe: exibir o saldo, sacar e depositar.

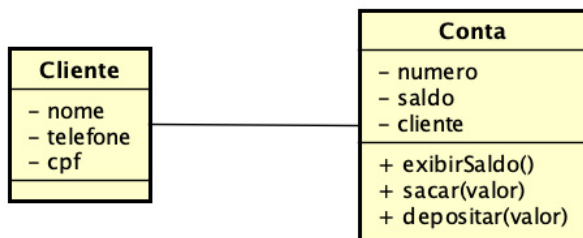


Figura 24 – Classes para um sistema bancário hipotético.

Falando em atributos e comportamentos, entenda os atributos como sendo a identidade do objeto e os comportamentos como sendo os métodos, algo que ele executa. Na Figura 24, os atributos definem os diferentes clientes e as diferentes contas, enquanto os métodos da classe **Conta** mudam o comportamento de um objeto à medida em que são executados. Na Figura 25, é mostrada a instanciação de objetos a partir da classe **Cliente**. Note que os valores dos atributos diferenciam cada objeto uns dos outros. Sobre o funcionamento dos métodos, falaremos deles com exemplos práticos na próxima seção.

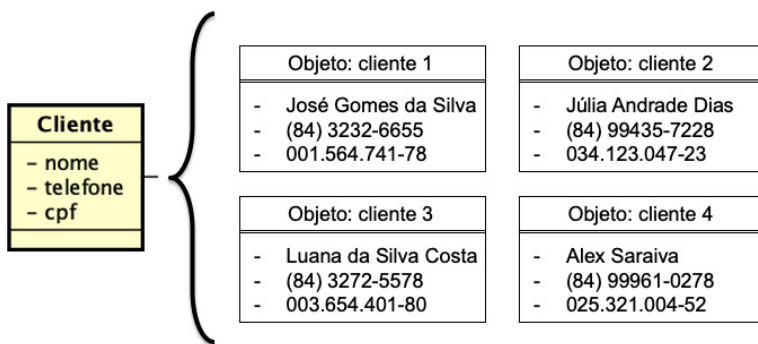


Figura 25 – Classe Cliente e seus objetos instanciados.

Na Figura 26, é mostrado um outro exemplo de classe e sua instanciação em objetos. A classe **Gato** possui os atributos **raça**, **nome**, **peso** e **idade** e, a partir dela, foram criados (isto é, instanciados) três objetos. Essa classe poderia ser utilizada, por exemplo, em um sistema de clínica veterinária.

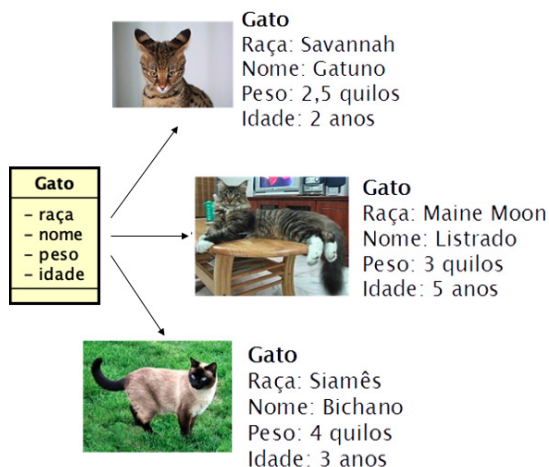


Figura 26 – Exemplo ilustrativo da classe **Gato** e seus objetos.

Existem diversos conceitos que envolvem a programação orientada a objetos, mas como este livro é introdutório, explicaremos alguns outros conceitos rapidamente na próxima seção, que mostra a implementação de classes e objetos em Python.

IMPLEMENTAÇÃO EM PYTHON

Para criar uma classe em Python, utilizamos a palavra reservada **class**, seguindo do nome da classe. De acordo com a PEP 8¹⁸, devemos colocar o nome da classe iniciando com letras maiúsculas. Se a classe tiver um nome composto, cada palavra deve iniciar também com letra maiúscula, seguindo a notação CamelCase. Por exemplo, no Programa 92 são criadas duas classes, **Conta** e **ContaBancaria**. O comando **pass**

18 disponível em <https://www.python.org/dev/peps/pep-0008/>

indica que as classes não fazem nada no momento.

```
1 class Conta:
2     pass
3
4 class ContaBancaria:
5     pass
```

Programa 92 – Criação de classes.

A partir de agora, descreveremos o exemplo do sistema bancário hipotético, baseado no descrito em Menezes (2014), com algumas modificações e explicando as linhas de código. O Programa 93 mostra a classe **Cliente**. Ela foi criada dentro do módulo `clientes.py`, que é um arquivo Python o qual conterà as classes relacionadas aos clientes.

```
1 class Cliente:
2     def __init__(self, nome, telefone, cpf):
3         self.nome = nome
4         self.telefone = telefone
5         self.cpf = cpf
```

Programa 93 – Módulo `clientes.py` com a classe `Cliente`.

Na linha 1, criamos a classe **Cliente** e, na linha 2, se já leu o Capítulo 7, pode notar que se trata de uma função. Quando utilizamos funções em classe, estas são chamadas de métodos. A diferença é que um método pertence a uma classe e sempre está atrelada a um objeto. Por conta disso, o primeiro parâmetro de um método é o objeto a ser recebido, chamado em

Python de **self**. Os demais parâmetros do método são os valores para instanciar o objeto a ser criado. O método `__init__()` é um método especial, chamado de construtor, que é utilizado na criação dos objetos. Os argumentos enviados para ele configuram os atributos do objeto que está sendo criado. Assim, as linhas 3 - 5 do Programa 93 representam a configuração dos parâmetros de um objeto ao ser criado. Existem outros métodos especiais para classes em Python. Caso tenha curiosidade, pesquise pelos métodos `__str__`, `__cmp__`, `__add__`, `__sub__` e veja como utilizá-los.

No Programa 94, é mostrado o código que cria os objetos com os clientes do banco¹⁹. A linha 1 importa a classe **Cliente** do módulo `clientes.py`. Para criar um objeto, basta utilizar a sintaxe `objeto = Classe(atributos)`. As linhas 2 - 5 representam a criação de objetos a partir da classe **Cliente**. O objeto **self** não é chamado explicitamente como parâmetro, mas de forma implícita quando executamos um método de um objeto.

```
1  from clientes import Cliente
2  cliente1 = Cliente('José Gomes', '(84) 3232-6655', '001.564.741-78')
3  cliente2 = Cliente('Júlia Dias', '(84) 99435-7228', '034.123.047-23')
4  cliente3 = Cliente('Luana Costa', '(84) 3272-5578', '033.654.401-80')
5  cliente4 = Cliente('Alex Saraiva', '(84) 3643-0278', '025.321.004-52')
```

Programa 94 – Módulo `banco.py` com a criação dos clientes.

O Programa 95 apresenta a criação da classe **Conta** no módulo `contas.py`, o qual apresenta novos conceitos que

¹⁹ Os CPFs são hipotéticos, criados aleatoriamente.

descreveremos a seguir. Na linha 1, tem-se o comando para criação da classe, enquanto nas linhas 2 – 5 tem-se o método construtor como os atributos de uma conta, conforme já explicado para a classe `Cliente`. Nas linhas 6 – 9, tem-se o método `exibirSaldo(self)`, que imprime o número, o nome do cliente e o saldo da conta do objeto. Note que o nome do cliente é oriundo do atributo `nome` de um objeto da classe `Cliente` (veja o Programa 93), exibido na linha 8. Nas linhas 10 – 12, tem-se o método `sacar(self, valor)`, que altera o atributo `saldo` subtraindo o valor enviado como parâmetro, caso não fique inferior a zero. Finalmente, nas linhas 13 – 14, tem-se o método `depositar(self, valor)`, que adiciona ao atributo `saldo` o valor passado como argumento.

```
1  class Conta:
2      def __init__(self, numero, saldo, cliente):
3          self.numero = numero
4          self.saldo = saldo
5          self.cliente = cliente
6      def exibirSaldo(self):
7          print(f'Conta: {self.numero} || ', end='')
8          print(f'Cliente: {self.cliente.nome}')
9          print(f'Saldo: R$ {self.saldo:.2f} \n'.)
10     def sacar(self, valor):
11         if self.saldo >= valor:
12             self.saldo -= valor
13     def depositar(self, valor):
14         self.saldo += valor
```

Programa 95 – Módulo `contas.py` com a classe `Conta`.

Note que em todos os métodos de uma classe, o primei-

ro parâmetro é o `self`, indicando a referência ao objeto utilizado. Note também que todos os atributos da classe são acessados por meio do `self`, pois eles são associados ao objeto instanciado.

O Programa 96 apresenta a criação de contas e o uso dos métodos, no mesmo módulo utilizado para criar os clientes (`banco.py`). Nas linhas 2 e 6, o objeto `conta1` foi associado ao objeto `cliente1` com um saldo de R\$ 100,00, enquanto o objeto `conta2` foi associado ao objeto `cliente3` com saldo de R\$ 1.500,50. Para invocar um método do objeto, basta chamar o objeto com o nome do método, separados por ponto (`.`), conforme mostrado nas linhas 3 – 5 e 7 – 9.

```
1  from contas import Conta
2  conta1 = Conta(1, 100.0, cliente1)
3  conta1.exibirSaldo()
4  conta1.depositar(200)
5  conta1.exibirSaldo()
6  conta2 = Conta(2, 1500.50, cliente3)
7  conta2.exibirSaldo()
8  conta2.sacar(500)
9  conta2.exibirSaldo()
```

Programa 96 – Módulo `banco.py` com a criação das contas.

O código completo e o resultado da execução do módulo `banco.py` completo é mostrado na Figura 27. O código desse módulo, juntamente com os módulos `clientes.py` e `bancos.py`, foram escritos no ambiente de desenvolvimento PyCharm²⁰ para facilitar a navegação entre eles. Contudo, eles

20 <https://www.jetbrains.com/pycharm/>

podem ser escritos no editor IDLE sem problemas.

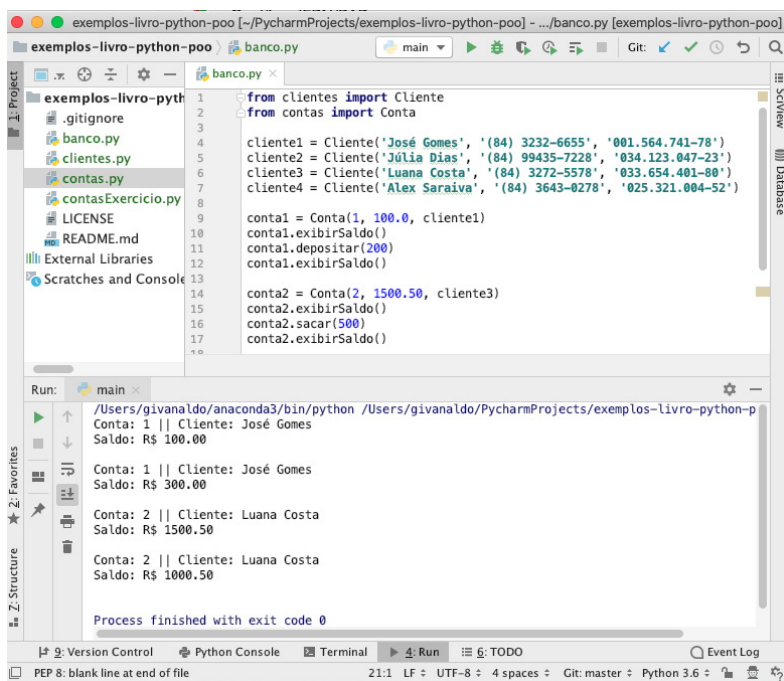


Figura 27 – Código completo e resultado da execução do módulo `banco.py`.

Esses foram os conceitos iniciais sobre a programação orientada a objetos, focando em classes, objetos, atributos e métodos. O paradigma possui muito mais conceitos envolvidos, que formariam um livro à parte. Por se tratar de um livro de introdução à programação, ficaremos por aqui no assunto. Nas seções a seguir, teremos exercícios resolvidos, que mostrarão mais alguns conceitos, além de exercícios propostos. Não deixe de estudar!

EXERCÍCIOS RESOLVIDOS

1. Implemente a classe **Disciplina** no módulo **escola.py** e crie um módulo de teste para ela. Um objeto é iniciado com o nome e as duas notas do aluno (entre 0,0 e 10,0) na disciplina em questão. O método **calcularMedia()** retorna a média aritmética do aluno na disciplina, enquanto o método **exibirSituacao()** retorna se o aluno está Aprovado, Em Recuperação ou Reprovado, considerando a média 6,0.

Disciplina
- nome : str - notas : list
+ calcularMedia() : float + exibirSituacao() : str

Comentários sobre a resolução

O primeiro passo é criar o módulo **escola.py**. Em seguida, criar a classe **Disciplina**, juntamente com o seu método construtor e os dois métodos (**calcularMedia()** e **exibirSituacao()**). O código é apresentado no Programa 97 e explicado logo a seguir.

```

1  class Disciplina:
2      def __init__(self, nome, notas):
3          self.nome = nome
4          self.notas = notas
5      def calcularMedia(self):
6          return (self.notas[0] + self.notas[1]) / 2
7      def exibirSituacao(self):
8          media = self.calcularMedia()
9          if media >= 6.0:
10             return 'Aprovado'
11             elif media >= 3.0:
12                 return 'Em Recuperação'
13             else:
14                 return 'Reprovado'

```

Programa 97 – Módulo `escola.py` com a criação da classe `Disciplina`.

Na linha 1, criamos a classe. Nas linhas 2 - 4 temos o método construtor, que recebe os parâmetros `nome` e uma lista com as duas notas do semestre. O método `calcularMedia()`, nas linhas 5 e 6, retorna o valor da média aritmética das notas do aluno. Note que o valor é acessado por meio do `self.notas[índice]`, ou seja, um atributo do objeto que é uma lista com dois elementos. Note também que o método retorna esse valor, da mesma forma que o retorno das funções vista no Capítulo 7. Por fim, o método `exibirSituacao()` é implementado nas linhas 7 - 14. Ele é composto por uma sequência de comandos condicionais comparando a média, retornando a situação na disciplina de acordo com a faixa da média.

O Programa 98 apresenta a segunda parte da questão: a criação do módulo `teste_escola.py` para testar a classe

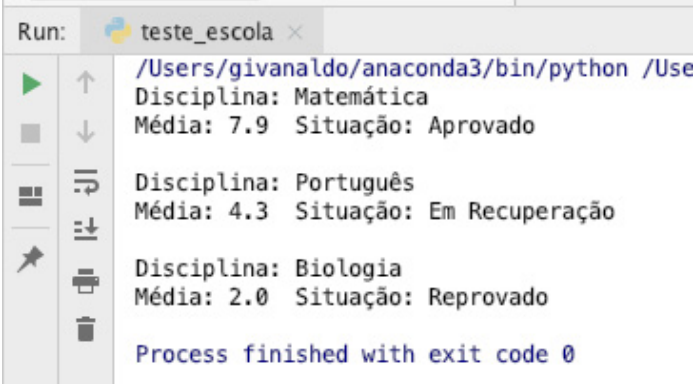
Disciplina. Na linha 1, importamos a classe do módulo `escola.py`, enquanto nas linhas 3 – 6, 8 – 11 e 13 – 16, são criados objetos para disciplinas (Matemática, Português e Biologia), atribuídos os seus respectivos nomes e notas. Para cada disciplina, após criado o objeto, é exibido o seu nome (`objeto.nome`). Note que podemos recuperar o valor de um atributo de um objeto chamando pelo nome criado no método construtor (linhas 4, 9 e 14). Existem algumas regras de acesso e modificação, que não fazem parte do escopo desse livro.

```
1  from escola import Disciplina
2
3  mat = Disciplina('Matemática', [6.3, 9.5])
4  print(f'Disciplina: {mat.nome}')
5  print(f'Média: {mat.calcularMedia():.1f}', end=' ')
6  print(f'Situação: {mat.exibirSituacao()}')
7
8  port = Disciplina('Português', [3.4, 5.2])
9  print(f'\nDisciplina: {port.nome}')
10 print(f'Média: {port.calcularMedia():.1f}', end=' ')
11 print(f'Situação: {port.exibirSituacao()}')
12
13 bio = Disciplina('Biologia', [1.0, 3.0])
14 print(f'\nDisciplina: {bio.nome}')
15 print(f'Média: {bio.calcularMedia():.1f}', end=' ')
16 print(f'Situação: {bio.exibirSituacao()}')
```

Programa 98 – Módulo teste_escola.py.

As duas instruções seguintes, para cada objeto, são para exibir a média e a situação de cada disciplina, utilizando os métodos já explicados, `calcularMedia()` e `exibirSituacao()`. A Figura 28

mostra o resultado da execução de `teste_escola.py`, testando com as três situações possíveis (Aprovado, Em Recuperação e Reprovado).



```
Run: teste_escola x
/Users/givonaldo/anaconda3/bin/python /Use
Disciplina: Matemática
Média: 7.9 Situação: Aprovado

Disciplina: Português
Média: 4.3 Situação: Em Recuperação

Disciplina: Biologia
Média: 2.0 Situação: Reprovado

Process finished with exit code 0
```

Figura 28 – Resultado da execução de `teste_escola.py`.

2. Tomando como base a classe **Conta**, mostrada na Figura 24 e implementada no Programa 95, altere a mesma para que um objeto conta possa ter um ou mais clientes associados, ou seja, possa ser uma conta individual ou uma conta conjunta.

Comentários sobre a resolução

Como agora a conta pode ter um ou mais clientes, o atributo cliente pode ser um objeto do tipo **Cliente** ou uma lista com vários objetos do tipo **Cliente**. Assim, nesse caso, alteraremos o método construtor, colocando o nome do atributo para clientes, e alteraremos o método `exibirSaldo()` para exibir os nomes dos clientes, caso seja mais de um. Os métodos `sacar()` e `depositar()` permanecem inalterados. O Programa 99 mostra a classe **Conta** com as novas modificações.


```

1  class Conta:
2      def __init__(self, numero, saldo, clientes):
3          self.numero = numero
4          self.saldo = saldo
5          self.clientes = clientes
6      def exibirSaldo(self):
7          print(f'Conta: {self.numero} ')
8          print('Cliente(s): ', end='')
9          if type(self.clientes) is list:
10             for cliente in self.clientes:
11                 print(f'{cliente.nome} | ', end='')
12             else:
13                 print(self.clientes.nome, end=' | ')
14             print(f'Saldo: R$ {self.saldo:.2f} \n')
15      def sacar(self, valor):
16          if self.saldo >= valor:
17              self.saldo -= valor
18      def depositar(self, valor):
19          self.saldo += valor

```

Programa 99 – Módulo `contas.py` com a classe `Conta` modificada.

A única modificação no método construtor foi a alteração do atributo `cliente` para `clientes`, apenas por questões de concordância, uma vez que a conta podem ter mais de um cliente. A alteração significativa foi no método `exibirSaldo()`, pois temos que testar se o atributo `clientes` é uma lista de contas ou apenas uma conta. Testamos essa condição na linha 9, comparando o atributo ao tipo `list`. Em caso afirmativo, utilizamos a estrutura de repetição `for` para enumerar todos os clientes. Caso contrário, apenas exibimos o atributo `nome` do objeto `clientes` (linha 13). A Figura 29 mostra o programa de

teste da classe `Conta` modificada e o resultado da sua execução.

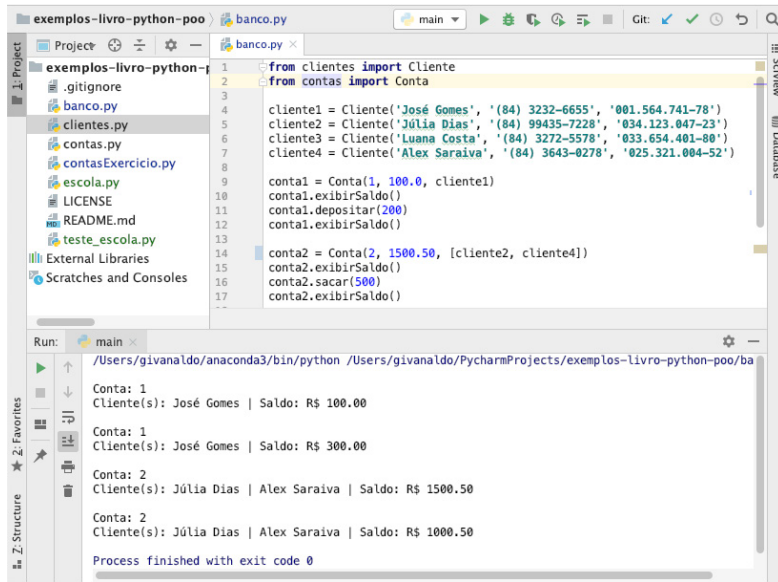


Figura 29 – Teste da classe `Conta` modificada

3. Implemente o método `exibirExtrato()` na classe `Conta` da questão anterior, que exibe todas as operações de saque e depósito de uma conta, com seus respectivos valores, além do saldo ao final.

Comentários sobre a resolução

Implementar o extrato da conta significa armazenar todas as operações de saque e de depósito realizadas em uma conta. Assim, precisaremos de uma estrutura para armazenar essas operações a cada vez que for realizado um saque ou um depósito.

O Programa 100 mostra a classe `Conta` com as modificações para

atender à questão. No método construtor, foi inserido o atributo `operacoes` (linha 6), que se inicia com uma lista vazia. Assim, quando se faz um saque (linha 19) ou um depósito (linha 22), o par `[Operacao, Valor]` é adicionado em operações, sendo `Operacao` o “Saque” ou o “Depósito”. Por fim, o método `exibirExtrato()` foi adicionado à classe nas linhas 23 – 29. O loop `for` é utilizado para percorrer a lista do atributo `operacoes` e exibi-lo na tela (linhas 26 e 27). O saldo da conta é exibido na linha 29.

```
1 class Conta:
2     def __init__(self, numero, saldo, clientes):
3         self.numero = numero
4         self.saldo = saldo
5         self.clientes = clientes
6         self.operacoes = []
7     def exibirSaldo(self):
8         print(f'Conta: {self.numero} ')
9         print('Cliente(s): ', end='')
10        if type(self.clientes) is list:
11            for cliente in self.clientes:
12                print(f'{cliente.nome} | ', end='')
13        else:
14            print(self.clientes.nome, end=' | ')
15        print(f'Saldo: R$ {self.saldo:.2f} \n')
16    def sacar(self, valor):
17        if self.saldo >= valor:
18            self.saldo -= valor
19            self.operacoes.append(['Saque', valor])
19            self.operacoes.append(['Saque\`', valor])
20    def depositar(self, valor):
```

```

21         self.saldo += valor
22         self.operacoes.append(['Depósito', valor])
23     def exibirExtrato(self):
24         print(f"\nExtrato da Conta {self.numero}")
25         print("-----")
26         for op in self.operacoes:
27             print(f"{op[0]} - R$ {op[1]:.2f}")
28         print("-----")
29         print(f"## Saldo: R$ {self.saldo:.2f}")

```

Programa 100 – Classe Conta com o método `exibirExtrato()`.

A Figura 30 mostra o programa de teste do método `exibirExtrato()` da classe `Conta` e o resultado da sua execução.

```

1 from clientes import Cliente
2 from contas import Conta
3
4 cliente1 = Cliente('José Gomes', '(84) 3232-6655', '001.564.741-78')
5 cliente2 = Cliente('Júlia Dias', '(84) 99435-7228', '034.123.047-23')
6 cliente3 = Cliente('Luana Costa', '(84) 3272-5578', '033.654.401-80')
7 cliente4 = Cliente('Alex Saraiva', '(84) 3643-0278', '025.321.004-52')
8
9 conta2 = Conta(2, 1500.50, [cliente2, cliente4])
10 conta2.exibirSaldo()
11 conta2.sacar(500)
12 conta2.sacar(235)
13 conta2.depositar(130)
14 conta2.exibirExtrato()

```

Run: main

```

/Users/givonaldo/anaconda3/bin/python /Users/givonaldo/PycharmProjects/exemplos-livro-python
Conta: 2
Cliente(s): Júlia Dias | Alex Saraiva | Saldo: R$ 1500.50

Extrato da Conta 2
-----
Saque - R$ 500.00
Saque - R$ 235.00
Depósito - R$ 130.00
-----
## Saldo: R$ 895.50

Process finished with exit code 0

```

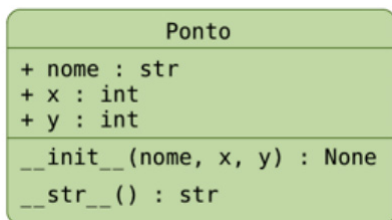
Figura 30 – Teste do método `exibirExtrato()` da classe `Conta`

EXERCÍCIOS PROPOSTOS

1. Crie uma classe chamada **Ingresso**, que possua o nome do evento e o valor do ingresso. Crie o método **exibirValor()**, que apenas retorne o valor do ingresso. Crie o método **__str__** que retorne o nome do evento seguido do valor do ingresso. Crie um programa para testar sua classe.

2. Crie uma classe chamada **Retangulo**, a qual possua os atributos **largura** e **altura**, e os métodos **calcularPerimetro()** e **calcularArea()**. No código de teste, crie um objeto e calcule, respectivamente, o perímetro e a área desse retângulo.

3. Crie uma classe **Ponto**, conforme a figura a seguir. O método **__str__** deve retornar os atributos do objeto no formato "nome: (x, y)". Crie em outro arquivo os testes para a classe **Ponto**, lendo diversos pontos e criando um objeto **ponto** para cada entrada lida. Coloque cada objeto da classe **Ponto** em uma lista e, ao final, imprima cada elemento dessa lista.



4. Crie uma classe **Lista** que receba um atributo do tipo **list** e que tenha um método o qual retorne os elementos da lista sem repetição. Crie o programa de teste.

5. Implemente uma calculadora que receba dois operadores utilizando os conceitos de orientação a objetos aprendidos. Os atributos `op1` e `op2` (operadores) são iniciados no construtor e os métodos `somar()`, `subtrair()`, `multiplicar()`, `dividir()` e `calcularPotencia()` realizam as respectivas ações nesses atributos. Crie o programa de teste para a classe `Calculadora`.

6. Crie a classe `Funcionario` com os atributos `nome` e `salario`, recebidos pelo método construtor, e o método `aumentarSalario(porcentagem)` cujo parâmetro é a porcentagem de aumento. Implemente um programa de teste para a classe, criando dois funcionários e simulando o aumento de salário de 20% para um e 50% para o outro.

7. Implemente uma classe `Carro`, que tenha as propriedades `consumo` e `combustível` refletindo o consumo do carro em km/l e o combustível no tanque, respectivamente. O consumo é atribuído no método construtor, enquanto o combustível inicia com 0. Crie o método `andar()` para simular o ato de dirigir o veículo por uma certa distância (isso vai reduzir a quantidade de combustível no tanque). Crie um método `exibirCombustivel()`, que mostrará o nível atual em que o carro se encontra e o método `abastecer(litros)` para aumentar o nível de combustível do carro. Agora, crie o programa de teste e simule o passeio de um veículo.

8. Implemente a classe `ContaInvestimento` semelhante a uma classe `ContaBancaria`, com a diferença de que se adicione um atributo `taxaJuros`. Forneça um construtor que configure tanto o saldo inicial como a taxa de juros. Forneça

um método `adicionarJuros()` para adicionar juros à conta (definido no atributo). Escreva um programa que construa uma poupança com um saldo inicial de R\$ 1.000,00 e uma taxa de juros de 10%. Depois aplique o método `adiconeJuros()` cinco vezes e imprima o saldo resultante.

9. Implemente a classe `Trigonometria` com o atributo `angulo` iniciado no método construtor (em graus) e o método `__str__`, para exibir o ângulo em radianos e os valores de seno, cosseno e tangente ao imprimir o objeto. Crie o programa de teste para três ângulos diferentes.

10. Aproveitando a classe `Ponto` da questão 3, adicione o método `calcularDistancia()`, que receba um outro ponto como argumento e calcule a distância euclidiana entre eles. Considerando dois pontos (P e Q), a distância euclidiana é calculada pela fórmula:

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

Crie o programa de teste, crie alguns pontos e calcule a distância entre eles.

A resolução das questões propostas deste capítulo está disponível em:

https://github.com/peoolivro/codigos/tree/master/Cap8_Exercicios

Para complementar o conteúdo apresentado neste capítulo, pode acessar o canal Procópio na Rede e assistir à seguinte *playlist*:

<https://www.youtube.com/playlist?list=PLqsF5rntN2nYn8urY0JKGif48hqSZwUKM>

APLICAÇÕES DE SISTEMAS OPERACIONAIS

Nos capítulos anteriores, nós vimos como a programação deve ser orientada a soluções do dia a dia. Nesse capítulo, vamos tratar de tópicos importantes da aplicação da linguagem Python em Sistemas Operacionais, ou seja, vamos ver como um sistema complexo utiliza soluções com programação. Afinal o que seria de nossos códigos sem o nosso sistema operativo para tornar tudo mais agradável ou acessível ao utilizador? Claro que nem sempre as coisas são tão “românticas” assim, mas vamos ver como ser um bom galante com nossos clientes.

CONCORRÊNCIA

Quando se fala de aplicações em sistemas operacionais, um assunto comumente abordado é *thread*. Segundo Anderson (2019), compreender que *threads* representam um trabalho executado em paralelo é um exercício de abstração no qual podemos pensar antes mesmo de implementar algo entrar na codificação.

O termo mais comum de um trabalho (*job*) são os processos que possuem uma forte relação com *threads* e que, muitas vezes, são confundidos. Se olhar da perspectiva do sistema operativo, a execução de um *software* é um trabalho, da mesma forma que pensar no envio de um pacote pela rede é

um trabalho. Por outro lado, se pensarmos que gravar alguma informação em um arquivo é um trabalho e se algum desses trabalhos precisar executar algo em paralelo, estaremos usando então uma *thread*.

Vamos aproximar usando um exemplo prático: imagine que está digitando um trabalho em um editor de textos. O que está digitando aparece simultaneamente na tela, então seu processador de textos precisa de uma *thread* para ler o te-clado e ao mesmo tempo mostrar a imagem em tela. Se usarmos um exemplo de uma tarefa mais cotidiana, podemos pensar em vários trabalhadores, cada um com uma marreta numa missão de baixar uma estaca. Podemos seguir de duas formas: cada trabalhador bate na estaca até cansar enquanto os outros ficam aguardando ou podemos sincronizar o trabalho de forma que cada um bata na estaca de forma ordenada, para que, ao final, todos estão cansados ao mesmo tempo, mas a estaca esteja colocada de forma mais rápida. pode ver no Youtube²¹ um vídeo que ficou famoso com base nesse exemplo.

Observe o Programa 101. Ele tem como objetivo executar simultaneamente várias *threads* e exibi-las em tela (HELLMAM, 2019).

```

1  import threading
2
3  def trabalhador(num):
4      while(1):
5          print(f'Eu sou o trabalhador: {num} \n')
6          # return
7
8  for i in range(5):
9      t = threading.Thread(target=trabalhador, args=(i,))
10     t.start()

```

Programa 101 – *Thread* trabalhadores.

Na linha 1, carregamos o módulo que permite a criação de *threads*. Nas linhas 3 - 6, criamos uma função que será executada por *threads* cujo papel é informar na tela qual seu identificador. Na linha 6, deixamos comentado o retorno da função, pois sem ele as *threads* ficam sendo executadas em loop infinito, já que não retornam ao código principal se remover o comentário observará que cada *thread* só irá executar uma vez. Na linha 8, criamos um *loop for* para criar mais de uma *thread* (neste caso colocamos 5). Na linha 10, chamamos a função *Thread*, em que no primeiro argumento nós temos a função alvo (**target**) que será executada, a qual, em nosso caso, chamamos a função *trabalhador* (**target = trabalhador**); e o segundo argumento tem o valor de *i* (**args=i**), que será nossa variável identificadora a *thread*. Na linha 10, iniciamos a *thread*. Com uma pequena modificação na linha 5, podemos observar o número de *threads* ativas, ou seja, que estão executando no processador. O Programa 102 mostra essa modificação.

```

1  import threading
2
3  def trabalhador(num):
4      while True:
5          ativas = threading.active_count()-1
6          print(f'Trabalhador {num} de {ativas} threads
7          ativas.')
7      #return
8
9  for i in range(5):
10     t = threading.Thread(target=trabalhador, args=(i,))
11     t.start()

```

Programa 102 – Número de *threads* ativas.

Agora imagine que seu professor lhe propôs a seguinte equação para ser calculada: $f(x) = x^2 + 3x + \sin(x)$. Perceba que cada parte da equação (x^2 , $3x$, $\sin(x)$) poderia ser calculada e depois somada no final para obter o resultado final. Imagine então que em vez de um cálculo simples, fosse algo mais complexo e que pudéssemos entregar cada parte deste cálculo a uma pessoa diferente. Vamos então imaginar que cada pessoa seja uma *thread* e que cada uma fará sua parte no cálculo da equação em separado para, no final, alguém só fazer o “fácil” trabalho de somar os resultados. O Programa 103 mostra a resolução.

```

1  import threading
2  from math import sin
3
4  def expo(x):
5      global resp_expo
6      resp_expo = x ** 2
7
8  def mult(x):
9      global resp_mult
10     resp_mult = 3 * x
11
12  def seno(x):
13     global resp_seno
14     resp_seno = sin(x)
15
16  if __name__ == '__main__':
17     x = 5
18
19     thread1 = threading.Thread(target=expo, args=(x,))
20     thread2 = threading.Thread(target=mult, args=(x,))
21     thread3 = threading.Thread(target=seno, args=(x,))
22     thread1.start()
23     thread2.start()
24     thread3.start()
25
26     fx = resp_expo + resp_mult + resp_seno
27     print('Equação f(x) = x^2 + 3*x + sen(x)')
28     print(f'Resultado para x = {x}: {fx}')

```

Programa 103 – Calcular função.

Nas linhas 1 e 2, importamos os módulos para executar *threads* e funções matemáticas, respectivamente; das linhas 4 - 6, a definição da função para calcular x^2 ; das linhas 8 - 10, a definição da função para calcular $3 \cdot x$; das linhas 12 - 14, a definição da função para calcular o **seno** (x); na linha 17, a definição da variável x ; nas linhas 19 - 21, a criação das *threads*, com cada uma apontando para uma função diferente; nas linhas 22 - 24, a inicialização das *threads* e, por fim, a função dos resultados na linha 26 e a impressão na tela nas linhas 27 e 28.

Uma coisa importante sobre *threads* é a sincronização. Dessa forma, mostraremos como podemos utilizar o método **join()** para nos ajudar. O cálculo da função funcionou corretamente porque o processador não gastou muito tempo em sua execução e porque quando o programa principal juntou as partes para atribuir na variável **fx** (linha 26), os valores para **resp_expo**, **resp_mult** e **resp_seno** já existiam. Agora vamos colocar um tempo de espera em cada *thread* de maneira que o programa principal solicitará a resposta das *threads*, as quais não tenham ainda feito o cálculo. Em uma situação análoga, se o programa principal é seu chefe e a é *thread*, ele pedirá um relatório, por exemplo, e quando informar que não tem tempo para entregar, provavelmente vai sofrer uma punição. O Programa 104 ilustra esse caso.

```

1  import threading
2  from math import sin
3  import time
4
5  def expo(x):
6      global resp_expo
7      time.sleep(10)
8      resp_expo = x * x
9
10 def mult(x):
11     global resp_mult
12     time.sleep(15)
13     resp_mult = 3 * x
14
15 def seno(x):
16     global resp_seno
17     time.sleep(20)
18     resp_seno = sin(x)
19
20 if __name__ == '__main__':
21     x = 5
22
23     thread1 = threading.Thread(target=expo, args=(x,))
24     thread2 = threading.Thread(target=mult, args=(x,))
25     thread3 = threading.Thread(target=seno, args=(x,))
26     thread1.start()
27     #thread1.join()
28     thread2.start()
29     #thread2.join()
30     thread3.start()
31     #thread3.join()
32
33     fx = resp_expo + resp_mult + resp_seno
34     print('Equação f(x) = x^2 + 3*x + sen(x)')
35     print(f'Resultado para x = {x}: {fx}')

```

Programa 104 – Cálculo da função com sincronização.

Veja que na linha 7, 12 e 17 usamos o método `sleep()` importado do módulo `time` na linha 3, de forma que nossas *threads* agora demoram um pouco mais a entregar sua resposta. O que acontece é que o programa principal vai tentar rodar a linha 33 e as variáveis `resp_expo`, `resp_mult` e `resp_seno` não vão existir. Voltando à analogia do chefe recebendo o relatório, desta vez nós temos um pouco mais de confiança e podemos dizer a ele para esperar o resultado. Então, nas linhas 27, 29 e 31, deixamos comentado o comando `join`, de maneira que, se tiramos o comentário, nosso código vai demorar um pouco mais para imprimir na tela, mas o resultado será o esperado pelo chefe.

1. Um problema bem conhecido da computação é o chamado produtor *versus* consumidor. O problema é simples e retrata condições como sincronização, concorrência, *deadlock* etc. A descrição do problema é o seguinte: existem as figuras do produtor (alguém que produz algo e guarda em um armazém) e a do consumidor (que vai ao armazém e utiliza o que foi produzido). Agora, vamos conhecer algumas situações:
2. O produtor e o consumidor terão acesso ao armazém, ou seja, eles precisam compartilhar a chave e nunca poderão brigar. Por exemplo, o produtor está tentando colocar alguma coisa e o consumidor já tentando usar. Eles são educados e o consumidor só pega alguma se o produtor já colocou lá em definitivo.
3. O produtor não deve desperdiçar seu produto, então se o armazém estiver cheio ele deve parar de

produzir. Da mesma maneira, o consumidor não deve perder a viagem e tentar consumir alguma coisa se o armazém estiver vazio.

4. Sempre que o produtor encher o armazém ele deve ir dormir. De forma semelhante, sempre que o consumidor consumir tudo ele deve ir dormir. Podemos colocar uma comunicação entre eles de maneira que, sempre que um for dormir, acorda o outro para trabalhar.
5. Sempre que houver a possibilidade de ambos trabalharem, devemos aproveitar, ou seja, se o armazém estiver no meio, (nem cheio nem vazio), tanto o produtor pode colocar coisas como o consumidor pode pegar coisas.

Existem várias soluções em diversas linguagens que usam semáforos, monitores ou outros tipos de estratégias que permitam que uma posição única seja acessada sem que haja corrupção dos dados. Vamos criar uma solução básica e ir aprimorando o código de maneira que possamos ir observando cada etapa. O primeiro teste que vamos fazer é tentar rodar o produtor e o consumidor usando *threads* e escrevendo em nosso armazém de forma concorrente (Akshar, 2013). O Programa 105 ilustra essa situação.


```

1  from threading import Thread, Lock
2  import time
3  import random
4
5  armazem = []
6  trava = Lock()
7
8
9  class ProdutorThread(Thread):
10     def run(self):
11         global armazem
12         while True:
13             trava.acquire()
14             armazem.append(random.randrange(0, 9))
15             print(f"Produzido {armazem[-1]}")
16             print(f"Lista do produtor {armazem}\n")
17             trava.release()
18             time.sleep(random.random())
19
20
21  class ConsumidorThread(Thread):
22     def run(self):
23         global armazem
24         while True:
25             trava.acquire()
26             if not armazem:
27                 print("Armazém vazio (Consumidor tentará consumir)")
28                 num = armazem.pop(0)
29                 print(f"Consumiu {num}")
30                 print(f"Lista do consumidor {armazem}\n")
31                 trava.release()
32                 time.sleep(random.random())
33
34
35  if __name__ == '__main__':
36     ProdutorThread().start()
37     ConsumidorThread().start()

```

Programa 105 – Produtor versus consumidor.

Na linha 5, criamos o local onde armazenaremos o que for produzido. Na linha 6, criamos uma variável que bloqueia o local que tenha condição de corrida, ou seja, onde haja o risco de corrupção de dados se duas *threads* tentarem acessar ao mesmo tempo. Nas linhas 9 - 18, temos o nosso produtor. Ele é responsável por, antes de colocar algo no armazém, bloqueá-lo (linha 13) e, depois que produzir, liberá-lo (linha 17). Na linha 14, ele produz número aleatórios de 0 - 9 e os guarda no armazém. Na linha 18, há uma simples parada aleatória de tempo. Nas linhas 15 e 16, imprime na tela o que produziu e todo o conteúdo do armazém, respectivamente. Nas linhas 21 a 32, temos nosso consumidor. Sempre que vai consumir ele também trava o armazém (linha 25) e destrava após o consumo (linha 31). Na linha 28, ele remove um elemento do armazém e nas linhas 29 e 30 imprime o que consumiu e como ficou o armazém, respectivamente. O detalhe na linha 26 é que se o armazém estiver vazio, ele avisará isso, mas provocará um erro na linha 28, informando que está tentando executar o método `pop()` `pop`, que serve para remover um elemento, mas o armazém está vazio. Na linha 32, ele descansará um tempo. Nas linhas 36 e 37, ocorrem as chamadas de execução da *threads*.

Vamos então fazer o seguinte: se o consumidor vê que o armazém está vazio, ele vai dormir e o produtor, sempre que produzir algo, acorda o consumidor. Para isso, vamos utilizar um objeto chamado `condition`.

Vamos ao código:

```

1  from threading import Thread, Condition
2  import time
3  import random
4
5  armazem = []
6  trava = Condition()
7
8
9  class ProdutorThread(Thread):
10     def run(self):
11         global armazem
12         while True:
13             trava.acquire()
14             armazem.append(random.randrange(0, 9))
15             print(f"Produzido {armazem[-1]}")
16             print(f"Lista do produtor {armazem}\n")
17             trava.notify()
18             trava.release()
19             time.sleep(random.random())
20
21
22  class ConsumidorThread(Thread):
23     def run(self):
24         global armazem
25         while True:
26             trava.acquire()
27             if not armazem:
28                 print("Nada no Armazém, Consumidor vai dormir")
29                 trava.wait()
30                 print("Produtor adicionou no armazém e "
31                       "avisou ao Consumidor")
32             num = armazem.pop(0)
33             print(f"Consumiu {num}")
34             print(f"Lista do consumidor {armazem}\n")
35             trava.release()
36             time.sleep(random.random())
37
38  if __name__ == '__main__':
39     ProdutorThread().start()
40     ConsumidorThread().start()

```

Programa 106 – Produtor versus consumidor que dorme.

Na linha 6, a **trava** mudou de **lock** para **condition**. Na linha 17, sempre que o produtor produz, avisa ao consumidor. Na linha 27, onde o consumidor verifica se o armazém está vazio, além de imprimir na tela que não tem nada e vai dormir, ele de fato dorme, pois na linha 29 estamos usando a função **wait**. Agora, quando o consumidor receber a notificação do produtor ele acordará e imprimirá que o produtor já adicionou alguma coisa e o está avisando na linha 30. Vale lembrar que o *notify* do consumidor não libera a trava, apenas acorda o consumidor, mas ele só terá acesso ao armazém depois da linha 18 ser executada, ou seja, o *release* da trava.

Até agora, nosso produtor pode produzir indefinidamente, mas como não existe armazém infinito, vamos limitar nosso armazém e colocar essa restrição ao nosso produtor. Ou seja, se o armazém estiver cheio, ele é quem pode ir dormir e aguardar que o consumidor o acorde. O Programa 107 ilustra esta situação.

```
1  from threading import Thread, Condition
2  import time
3  import random
4
5  armazem = []
6  trava = Condition()
7  TAM_ARM = 10
8
9
10 class ProdutorThread(Thread):
11     def run(self):
12         global armazem
13         while True:
14             trava.acquire()
15
```

```

16         if len(armazem) == TAM_ARM:
17             print("Armazém cheio, Produtor vai dormir")
18             trava.wait()
19             print("Existe espaço no armazém, "
20                   "Consumidor avisou o produtor")
21             armazen.append(random.randrange(0, 9))
22             print(f"Produzido {armazen[-1]}")
23             print(f"Lista do produtor {armazen}\n")
24             trava.notify()
25             trava.release()
26             time.sleep(random.random())
27
28     class ConsumidorThread(Thread):
29         def run(self):
30             global armazen
31             while True:
32                 trava.acquire()
33                 if not armazen:
34                     print("Nada no Armazém, o Consumidor irá dormir")
35                     trava.wait()
36                     print("Produtor adicionou algo ao armazém e avisou")
37                     num = armazen.pop(0)
38                     print(f"Consumiu {num}")
39                     print(f"Lista do consumidor {armazen}\n")
40                     trava.notify()
41                     trava.release()
42                     time.sleep(random.random())
43
44
45     if __name__ == '__main__':
46         ProdutorThread().start()
47         ConsumidorThread().start()

```

Programa 107 – Produtor versus consumidor com limitador do armazém.

Na linha 7, temos o tamanho do nosso armazém e o consumidor testa se ele já está cheio na linha 16. Se estiver, ele avisa em tela na linha 17 e vai dormir (linha 18). No consumidor, tivemos que colocar o aviso para o produtor na linha 40, e se o produtor acordar, ele imprimirá que existe espaço e que foi avisado pelo consumidor (linha 19). Aqui acontece da mesma forma que no código anterior, no qual o produtor só poderá acessar o armazém depois que o consumidor liberar a trava (linha 41).

Nosso produtor está funcionando muito bem, mas e se desejarmos ter mais de um produtor. Será que esse código vai funcionar? Vamos testar! O Programa 108 ilustra essa situação.

```
1  from threading import Thread, Condition
2  import time
3  import random
4
5  armazem = []
6  trava = Condition()
7  TAM_ARM = 10
8
9
10 class ProdutorThread(Thread):
11     def run(self):
12         global armazem
13         while True:
14             trava.acquire()
15
16             if len(armazem) == TAM_ARM:
17                 print("Armazém cheio, Produtor vai dormir")
18                 trava.wait()
19             print("Há espaço no armazém: produtor avisado.")
20             armazem.append(random.randrange(0, 9))
21             print(f"Produzido {armazem[-1]}")
22             print(f"Lista do produtor {armazem}\n")
23             trava.notify()
24             trava.release()
25             time.sleep(random.random())
```

```

25         time.sleep(random.random())
26
27
28     class ConsumidorThread(Thread):
29         def run(self):
30             global armazem
31             while True:
32                 trava.acquire()
33                 if not armazem:
34                     print("Nada no Armazém, o Consumidor irá dormir")
35                     trava.wait()
36                     print("Produtor adicionou algo no armazém e avisou.")
37                 num = armazem.pop(0)
38                 print(f"Consumiu {num}")
39                 print(f"Lista do consumidor {armazem}\n")
40                 trava.notify()
41                 trava.release()

```

Programa 108 – Produtor versus consumidor com várias instâncias.

A diferença está somente nas linhas 46-50, pois agora temos dois produtores, mas o resultado é totalmente diferente porque o produtor, que antes respeitava o tamanho do armazém, agora passa a não mais fazer isso pois recebe agora notificações não somente do consumidor, mas também do outro produtor. Isso poderia acontecer também se colocássemos outro consumidor e ele poderia receber notificação de outro consumidor e tentar consumir com um armazém ainda vazio.

Para corrigir esse problema de ser notificado por outros semelhantes, ou seja, produtores por outros produtores e consumidores por outros consumidores, vamos criar um objeto con-

dicional para cada um e garantir ainda que a chave do armazém seja bloqueada. Outra coisa que faremos é, em vez de somente verificar se o armazém está vazio (está com estoque igual a zero) ou se está cheio (se o estoque é igual a capacidade), iremos desta vez verificar se ele for maior que zero significa ou não, ou seja, qualquer consumidor pode ser executado ou se ele é menor que a capacidade do armazém ou não, isto é, qualquer produtor pode ser executado. O Programa 109 ilustra essa situação.

```
1  from threading import Thread, Lock, Condition
2  import time
3  import random
4
5  armazem = []
6  trava = Lock()
7  dorme_produto = Condition()
8  dorme_consumidor = Condition()
9  TAM_ARM = 10
10
11
12 class ProdutorThread(Thread):
13     def run(self):
14         global armazem
15         while True:
16             trava.acquire()
17             dorme_produto.acquire()
18             dorme_consumidor.acquire()
19             if len(armazem) < TAM_ARM:
20                 armazem.append(random.randrange(0, 9))
21                 print(f"Produzido {armazem[-1]}")
22                 print(f"Lista do produtor {armazem}\n")
23
24             dorme_consumidor.release()
25             dorme_produto.release()
```



```

26         else:
27             print("Armazém cheio, Produtor vai dormir")
28             if trava.locked():
29                 trava.release()
30                 dorme_consumidor.notify()
31                 dorme_consumidor.release()
32                 dorme_produtores.wait()
33
34             if trava.locked():
35                 trava.release()
36                 time.sleep(random.random())
37
38
39 class ConsumidorThread(Thread):
40     def run(self):
41         global armazem
42         while True:
43             trava.acquire()
44             dorme_produtores.acquire()
45             dorme_consumidor.acquire()
46             if len(armazem) > 0:
47                 num = armazem.pop(0)
48                 print(f"Consumiu {num}")
49                 print(f"Lista do consumidor {armazem}\n")
50
51                 dorme_consumidor.release()
52                 dorme_produtores.release()
53             else:
54                 print("Armazém vazio: Consumidor irá dormir")
55
56                 if trava.locked():
57                     trava.release()
58
59                 dorme_produtores.notify()
60                 dorme_produtores.release()
61                 dorme_consumidor.wait()

```

```

62
63         if trava.locked():
64             trava.release()
65             time.sleep(random.random())
66
67
68 if __name__ == '__main__':
69     Produtor1 = ProdutorThread()
70     Produtor2 = ProdutorThread()
71     Produtor1.start()
72     Produtor2.start()
73
74     Consumidor1 = ConsumidorThread()
75     Consumidor2 = ConsumidorThread()
76     Consumidor1.start()
77     Consumidor2.start()

```

Programa 109 – Produtor versus consumidor com várias instâncias corrigido.

Nas linhas 6 - 8, temos a trava do armazém, que irá garantir alteração única e as condicionais para colocar as *threads* para dormir. Nas linhas 16 - 18 e 43 - 45, estamos bloqueando o armazém e a resolução de ir dormir ou não. Veja que tanto o produtor quanto o consumidor precisam da variável condicional um do outro, pois eles notificarão quando forem dormir. Na linha 19, ele verifica se o armazém não está cheio, então ele produz nas linhas 20 - 22 e já libera a condicional do produtor e consumidor nas linhas 24 e 25. Se já estiver cheio (linha 26), avisar na tela (linha 27) e verificar se o armazém ainda está bloqueado (linha 28), pois outra *thread* já pode ter liberado, então fazer também a liberação na linha 29. Aqui é importante destacar que se esse teste de verificação está bloqueado, há

uma condição do objeto do tipo *lock*, ou seja, as **conditional** não possuem esse teste. Depois é necessário acordar os consumidores com uma notificação (linha 30), liberá-lo (linha 31) e colocar o produtor para dormir (linha 32). Se após seu retorno do sono o produtor encontrar a trava bloqueada (linha 34), ele desbloqueia (linha 35) e volta ao início. O consumidor segue lógica semelhante, mas fazendo a verificação se não o armazém não está vazio (linha 46) ou se estiver (linha 53). Neste último caso, ele dormirá conforme a linha 61. Veja na linha 69, 70, 74 e 75 que criamos mais de um produtor e consumidor.

Para encerrar nosso código do produtor e consumidor, gostaria de propor mais duas alterações meramente “estéticas”. Uma é que possamos criar vários produtores e consumidores sem ter que ficar copiando o código, mas sim informando o número; e outra diz respeito à identificação desses produtores e consumidores. O Programa 110 ilustra essa situação.

```
1  from threading import Thread, Lock, Condition
2  import time
3  import random
4
5  armazem = []
6  trava = Lock()
7  dorme_produtores = Condition()
8  dorme_consumidores = Condition()
9  TAM_ARM = 10
10
11
12 class ProdutorThread(Thread):
```

```

13     def __init__(self, threadID):
14         Thread.__init__(self)
15         self.threadID = threadID
16
17     def run(self):
18         global armazem
19         while True:
20             trava.acquire()
21             dorme_produtores.acquire()
22             dorme_consumidor.acquire()
23             if len(armazem) < TAM_ARM:
24                 armazem.append(random.randrange(0, 9))
25                 print(f"Produtor {self.threadID} produziu {armazem[-1]}")
26                 print(f"Lista do produtor {armazem}\n")
27
28                 dorme_consumidor.release()
29                 dorme_produtores.release()
30             else:
31                 print(f"Armazém cheio, Produtor {self.threadID} vai dormir")
32                 if trava.locked():
33                     trava.release()
34                     dorme_consumidor.notify()
35                     dorme_consumidor.release()
36                     dorme_produtores.wait()
37
38                 if trava.locked():
39                     trava.release()
40                 time.sleep(random.random())
41
42
43 class ConsumidorThread(Thread):
44     def __init__(self, threadID):
45         Thread.__init__(self)
46         self.threadID = threadID
47

```

```

48     def run(self):
49         global armazem
50         while True:
51             trava.acquire()
52             dorme_produtores.acquire()
53             dorme_consumidores.acquire()
54             if len(armazem) > 0:
55                 num = armazem.pop(0)
56                 print(f"Consumidor {self.threadID} consumiu {num}")
57                 print(f"Lista do consumidor {armazem}\n")
58
59                 dorme_consumidores.release()
60                 dorme_produtores.release()
61             else:
62                 print(f"Armazém vazio. Consumidor {self.threadID} irá dormir")
63
64                 if trava.locked():
65                     trava.release()
66
67                 dorme_produtores.notify()
68                 dorme_produtores.release()
69                 dorme_consumidores.wait()
70
71                 if trava.locked():
72                     trava.release()
73                 time.sleep(random.random())
74
75
76 if __name__ == '__main__':
77     numprod = 2 # quantidade de produtores
78     numcons = 2 # quantidade de consumidores
79
80     for i in range(numprod):
81         t = ProdutorThread(i)
82         t.start()
83
84     for i in range(numcons):
85         t = ConsumidorThread(i)
86         t.start()

```

Programa 110 – Produtor versus consumidor com número de instâncias numa variável.

Nas linhas 13 e 44, os métodos construtores associam o identificador de cada *thread*. Nas linhas 25, 31, 56 e 62, modificamos a impressão para mostrar a identificação das *threads*. Nas linhas 80 - 82 e 84 - 86, criamos as *threads* com um laço enviando com repetição baseada no número de *threads* definidas nas linhas 77 e 78 e enviando o índice como identificador da *thread* nas linhas 81 e 85.

Ainda gostaríamos de compartilhar com vocês um outro código. Bem, sempre propomos essas atividades em sala de aula e geralmente utilizamos a linguagem de programação C para mostrar a sua complexidade e, em seguida, utilizamos Python. Afinal, o que vem fácil também vai fácil. Talvez não tenha combinado bem o ditado popular, mas o fato é que um dos nossos alunos implementou, usando somente variáveis globais, um produtor e consumidor bem simples e funcional e deixamos para vocês fazerem modificações que desejarem, como por exemplo executar com vários produtores e consumidores. O código é mostrado no Programa 111.

```
1  import threading
2  import time
3  from random import randrange
4
5
6  class ProdutorConsumidor():
7
8      def produtor(self):
9          global gravando
10         global dados
11         dados = []
12
13         while True:
14             try:
15                 if len(dados) < 10:
16                     gravando = True
17                     dados.append(randrange(0, 9))
18                     print(f"\nProduziu {dados[-1]}")
19                     print(f"\nLista do produtor {dados}\n")
20                     gravando = False
21                     # time.sleep(0.6)
```

```

22         except KeyboardInterrupt:
23             print("Produtor parou")
24             break
25
26     def consumidor(self):
27         while True:
28             try:
29                 # time.sleep(1)
30                 if gravando == False and len(dados) > 0:
31                     print(f"\nConsumiu {dados[0]}")
32                     del (dados[0])
33                     print(f"\nLista do consumidor {dados}\n")
34             except KeyboardInterrupt:
35                 print("Consumidor parou")
36                 break
37
38
39 if __name__ == '__main__':
40     produtorConsumidor = ProdutorConsumidor()
41
42     p1 = threading.Thread(target=produtorConsumidor.produtor)
43     c1 = threading.Thread(target=produtorConsumidor.consumidor)
44
45     p1.start()
46     c1.start()

```

Programa 111 – Produtor versus consumidor com variáveis globais.

Na literatura clássica de sistemas operacionais, existe um outro problema bastante conhecido, capaz de representar bem a concorrência entre processos. Ele é chamado de “jantar dos filósofos”.

Imagine 5 filósofos sentados em uma mesa redonda, conforme a Figura 31. Todos eles têm seu próprio prato de macarrão e/ou estão comendo ou estão pensando. Os filósofos compartilham os garfos que estão um de cada lado de seu prato e só podem comer se conseguirem pegar os dois talheres, à direita e à esquerda. Imagine que o primeiro filósofo consegue pegar dois talheres e, na sequência, no sentido anti-horário, o próximo filósofo já não poderia comer pois estaria sem o talher da esquerda, mas o filósofo seguinte conseguiria pegar seus

talheres, mas inviabilizaria os próximos filósofos na sequência, o imediatamente seguinte pois ficaria sem o seu talher da esquerda e o outro que já está sem o seu talher da direita pois o primeiro filósofo já tinha pego. Assim quando esses dois filósofos encerrarem o jantar vão voltar a pensar e permitirão que outros possam comer. Eles se revezam neste processo de maneira que ninguém deve comer indefinidamente e deixe os outros morrerem de fome.

Um problema grave que pode acontecer é o seguinte: se cada filósofo pega o seu talher à direita, por exemplo, o que vai acontecer é que todos vão ficar esperando o talher da esquerda e esse nunca será liberado. Em sistemas operacionais, essa condição é chamada de *deadlock*. Existem várias soluções para o problema e Machado e Maia (2007) citam 3 possíveis soluções: a) permitir que apenas 4 filósofos sentem à mesa simultaneamente, b) permitir que um filósofo pegue o talher somente se o outro estiver disponível, e c) permitir que um filósofo par pegue o talher da esquerda primeiro e o filósofo impar pegue o talher da direita primeiro.

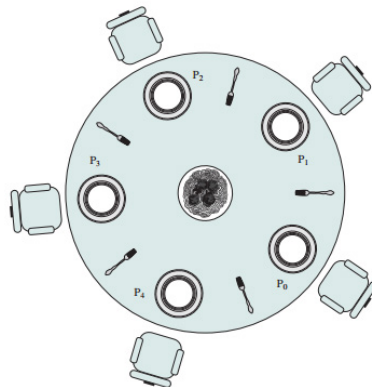


Figura 31 – O jantar dos filósofos.

Fonte: <https://www.thecrazyprogrammer.com/2017/06/dining-philosophers-problem-c-c.html>

O Programa 112 mostra a implementação desse problema. O código é bem simples e a ideia é: o filósofo tenta pegar o talher esquerdo primeiro. Se não conseguir, vai pensar. Se ele conseguir, ele tenta pegar o direito. Se não conseguir pegar, larga o esquerdo e volta a pensar, mas se conseguir, vai comer.

```
1  from threading import Thread
2  from random import uniform
3  from time import sleep
4
5
6  class Filosofo(Thread):
7      def __init__(self, nome, garfoEsq, garfoDir):
8          Thread.__init__(self)
9          self.nome = nome
10         self.garfoDir = garfoDir
11         self.garfoEsq = garfoEsq
12
13     def run(self):
14         while True:
15             sleep(uniform(2, 3))
16             print(f'> {self.nome}: estou PENSANDO')
17             self.janta()
18
19     def janta(self):
20         global garfos
21         while True:
22             sleep(uniform(5, 10))
23             esquerda = garfos[self.garfoEsq]
24             if esquerda == 0:
25                 garfos[self.garfoEsq] = self.nome
26                 print(f'> {self.nome}: Peguei o garfo da ESQUERDA')
27                 sleep(uniform(1, 3))
28                 direita = garfos[self.garfoDir]
29                 if direita == 0:
30                     garfos[self.garfoDir] = self.nome
31                     break
32             else:
```

```

33             print(f'> {self.nome}: O garfo da DIREITA está com '
34                   f'{direita}, soltei o da ESQUERDA')
35             garfos[self.garfoEsq] = 0
36         else:
37             print(f'> {self.nome}: Tentei pegar o garfo da '
38                   f'ESQUERDA, mas está com {esquerda}')
39             return
40
41         self.jantando()
42         garfos[self.garfoDir] = 0
43         garfos[self.garfoEsq] = 0
44
45     def jantando(self):
46         print(f'> {self.nome}: peguei o garfo da DIREITA, e '
47               f'COMECEI a comer')
48         sleep(uniform(5, 10))
49         print(f'> {self.nome} Terminei de comer, e soltei os garfos')
50
51
52 if __name__ == '__main__':
53     garfos = [0] * 5
54     nomes = ['Aristoteles', 'Kant', 'Platao', 'Socrates', 'Russel']
55     filosofos = []
56
57     for i, nome in enumerate(nomes):
58         fil = Filosofo(nome, i - 1, i)
59         filosofos.append(fil)
60
61     for f in filosofos:
62         f.start()

```

Programa 112 – Jantar dos filósofos.

Na linha 6, iniciamos a classe `Filosofo` instanciando uma *thread*. Na linha 53, criamos um vetor com os 5 talheres, preenchemos com zero, e é nele que vamos fazer o controle de quem está usando os talheres. A medida em que um filósofo pega

o talher, ele deverá alterar na posição do vetor de talheres de zero para seu nome. Da linha 7 - 11, no método `init`, definimos o nome de cada filósofo e qual o índice do talher esquerdo e direito. Isso é uma vantagem da linguagem Python, pois os índices dos vetores são circulares e podemos dizer que o primeiro filósofo ficará com o seu talher esquerdo com o índice -1, que é exatamente o índice do talher direito do último filósofo ou último talher. A linha 19 inicia o método `jantar`, que é onde está o controle dos talheres. Na linha 20, declaramos o vetor de talheres como global, de modo que todas as *threads* vejam o mesmo conteúdo. Associamos a variável `esquerda` ao valor indicado na posição do talher esquerdo do filósofo na linha 23. Na linha 24 consta o teste para verificar se alguém está usando e já colocou seu nome no vetor de talheres ou se está disponível para uso com o valor 0. Se estiver disponível, pegamos o talher colocando o nome do filósofo no vetor (linha 25) e avisando na tela que pegou o talher (linha 26). Agora é a vez de associar o talher da direita a variável `direita` (linha 28) e verificar se está disponível (linha 29). Se estiver disponível, podemos pegar o talher (linha 30) e sair da condicional (linha 31). A linha 32 apresenta o `else` da condição do talher da direita não estar disponível. Nesse caso, devemos soltar inclusive o talher esquerdo que havíamos pego. Avisamos que estamos liberando o talher esquerdo (linha 34) e "soltamos" (linha 35). A linha 36 apresenta o `else` da condição em que o talher esquerdo não está disponível e precisamos reiniciar o processo. A linha 40 apresenta o método `jantando`, definido nas linhas 45 a 48, em que é impresso a mensagem o garfo direito foi pego com sucesso (linha 46) e o filósofo vai jantar por um tempo e quando terminar avisa na linha 49. As linhas 42 e 43 indicam a devolução de fato dos talheres, zerando o vetor. A linha 54 apresenta a defi-

nição dos nomes para os filósofos e a linha 55 a criação do vetor para guardar as *threads* dos filósofos. Nas linhas 57 a 59, criamos uma lista de *threads* e nas linhas 61 e 62 as inicializações. Se seu interpretador permitir fazer *debug* e rodar linha a linha, pode-se executar a linha 62 aos poucos e colocar os filósofos à mesa vagarosamente e ver a disputa dos talheres. Repare que nas linhas 15, 22, 27 e 48 temos um tempo apenas para melhorar a visualização da saída, mas esses poderiam ser removidos em uma solução envolvendo um problema real (na verdade, deveriam ser removidos por questão de desempenho).

Assim como o problema dos filósofos, outros se seguiram, os quais não serão abordados aqui, mas que são interessantes para serem praticados, quais sejam: o problema do barbeiro, que pode ser visto no capítulo 7 do livro de Machado e Maia (2007), o problema dos leitores e escritores, que pode ser visto no Capítulo 2 de Tanenbaum (2010), entre outros.

ESCALONAMENTO

Vamos agora falar um pouco sobre nosso processador, ou melhor, como nosso sistema operativo envia processos para serem executados. A ideia que mais parece justa é que o processo que chegar primeiro seja o primeiro a ser executado, mas dependendo do tempo que estes demorem para serem executados, considerando não preempção, ou seja, um próximo processo só será executado quando o anterior completar sua execução, isso pode não funcionar adequadamente.

Processadores em lote não existem mais, porém esses algoritmos podem nos ajudar a melhorar o conceito de otimização e podemos também fazer adaptações para que o algoritmo

seja executado no ambiente preemptivo.

Para fazermos uma comparação, vamos considerar alguns tempos ou variáveis de tempo, baseado em (GEEKSFORGEEKS, s.d.): a) tempo de conclusão (*completion time*): tempo que o processo termina sua execução, b) tempo de chegada (*arrival time*): tempo que o processo chegou para ser executado, c) duração, tempo de execução ou tempo de estouro (*burst time*): tempo que o processo gasta para ser executado, d) tempo de retorno (*turnaround time*): tempo entre o instante que é submetido e o instante que é concluído, pode ser calculado pela diferença de tempo de conclusão e tempo de chegada, e) tempo de espera (*waiting time*): diferença de tempo entre o tempo de retorno e o de duração, f) tempo médio de espera: tempo total de espera dividido pelo número de processos, g) tempo médio de retorno: tempo total de retorno dividido pelo número de processos.

Vamos a um exemplo utilizando o algoritmo FIFO (*First-in, First-out*), conforme a Figura 32, ou seja, primeiro a entrar é o primeiro a sair. O Programa 113 ilustra essa situação.

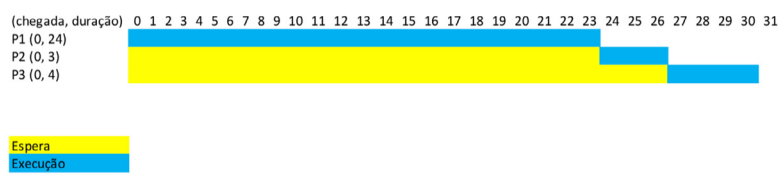


Figura 32 – Algoritmo FIFO.

Vamos primeiro verificar os tempos:

- a. tempo de conclusão dos processos é $P1 = 24$, $P2 = 27$ e $P3 = 31$
- b. o tempo de chegada de todos os processos foi 0.

- c. o tempo de duração de $P1 = 24$, $P2 = 3$ e $P3 = 4$
- d. tempo de retorno (a) - (b) então $P1 = 24 - 0 = 24$, $P2 = 27 - 0 = 27$, $P3 = 31 - 0 = 31$
- e. tempo espera (d) - (c) então $P1 = 24 - 24 = 0$, $P2 = 27 - 3 = 24$, $P3 = 31 - 4 = 27$
- f. tempo médio de espera $P1(e) + P2(e) + P3(e) / 3 = (0 + 24 + 27)/3 = 17$
- g. tempo médio de retorno $P1(d) + P2(d) + P3(d) / 3 = (24 + 27 + 31) / 3 = 27,33$

```
1 def calculandoTempoEspera(processos, n, td, te):
2     #considerando processo iniciando em 0
3     te[0] = 0
4
5     for i in range(1, n):
6         te[i] = td[i - 1] + te[i - 1]
7
8
9 def calcularTempoRetorno(processos, n, td, te, tr):
10     for i in range(n):
11         tr[i] = td[i] + te[i]
12
13
14 def calculandoTempoMedio(processos, n, td):
15     te = [0] * n
16     tr = [0] * n
17     total_te = 0
18     total_tr = 0
19
20     calculandoTempoEspera(processos, n, td, te)
21     calcularTempoRetorno(processos, n, td, te, tr)
```

```

22
23     print("Processos \t\t Tempo Duração \t\t"
24           " \t Tempo Espera \t\t Tempo Retorno")
25
26     for i in range(n):
27         total_te = total_te + te[i]
28         total_tr = total_tr + tr[i]
29         print(f"{i + 1} \t\t\t\t\t"
30               f"{td[i]} \t\t\t\t\t "
31               f"{te[i]} \t\t\t\t\t {tr[i]}")
32
33     print(f"Tempo médio espera = {total_te / n}")
39     print(f"Tempo médio retorno = {total_tr / n}")
35
36
37 if __name__ == "__main__":
38     processos = [1, 2, 3]
39     n = len(processos)
40     duracao = [24, 3, 4]
41     calculandoTempoMedio(processos, n, duracao)

```

Programa 113 – Algoritmo FIFO.

Nas linhas 1 a 6, temos a função para calcular o tempo de espera dos processos, lembrando que o primeiro processo inicia no tempo 0, conforme a linha 3, e os outros serão calculados de forma iterativa somando o tempo de duração e o tempo de espera do processo anterior. Nas linhas 9 a 11, temos a função para calcular o tempo de retorno de cada processo, que é o tempo de duração mais o tempo de espera, conforme podemos ver na linha 11. Nas linhas 14 a 34, temos o cálculo do tempo médio de espera e retorno. Iniciamos dois vetores para armazenar o tempo de espera e o tempo de retorno (linhas 15 e 16) e seus respectivos totais (linhas 17 e 18). Nas linhas 20 e 21, ocorrem as chamadas

das funções de calcular o tempo de espera e o tempo de retorno, respectivamente. Nas linhas 23 e 24, imprimimos o cabeçalho das colunas com os processos e seus tempos. Na linha 27, calculamos iterativamente o tempo total de espera como a soma de cada tempo de espera dos processos, e na linha 28 fazemos o mesmo para o tempo total de retorno. Nas linhas 29 a 31, imprimimos os valores de cada processo e nas linhas 33 e 34, calculamos e imprimimos os tempos médio de espera e de retorno, respectivamente. Na linha 38, definimos a quantidade de processos e uma numeração para eles. Na linha 40, definimos os tempos de duração de cada processo. Por fim, na linha 41, chamamos a função que calcula os tempos médio e fará todo o trabalho.

Vamos considerar processos chegando em tempos diferentes, conforme a Figura 33, e fazer novamente os cálculos. O Programa 114 ilustra essa situação.

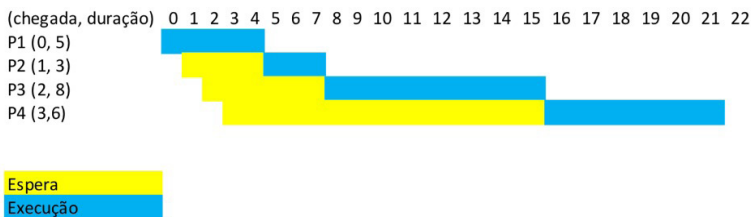


Figura 33 – FIFO com tempos de chegada diferentes.

Vamos primeiro verificar os tempos:

- tempo de conclusão é $P1 = 5$, $P2 = 8$, $P3 = 16$, $P4 = 22$
- o tempo de chegada de todos os processos foi $P1 = 0$, $P2 = 1$, $P3 = 2$, $P4 = 3$
- o tempo de duração de $P1 = 5$, $P2 = 3$ e $P3 = 8$, $P4 = 6$
- tempo de retorno (a) - (b) então $P1 = 5 - 0 = 5$, $P2 =$

- 8 - 1 = 7, P3 = 16 - 2 = 14, P4 = 22 - 3 = 19
- e. tempo de espera (d) - (c) então P1 = 5 - 5 = 0, P2 = 7 - 3 = 4, P3 = 14 - 8 = 6, P4 = 19 - 6 = 13
- f. tempo médio de espera $P1(e) + P2(e) + P3(e) + P4(e) / 4 = (0 + 4 + 6 + 13) / 4 = 5,75$
- g. tempo médio de retorno $P1(d) + P2(d) + P3(d) + P4(d) / 4 = (5 + 7 + 14 + 19) / 4 = 11,25$

```

1  def calculandoTempoEspera(processos, n, td, te, tc):
2      # considerando processo iniciando em 0
3      te[0] = 0
4
5      for i in range(1, n):
6          te[i] = (td[i - 1] + te[i - 1]) - (tc[i] - tc[i - 1])
7
8
9  def calcularTempoRetorno(processos, n, td, te, tr):
10     for i in range(n):
11         tr[i] = td[i] + te[i]
12
13
14  def calculandoTempoMedio(processos, n, td, tc):
15     te = [0] * n
16     tr = [0] * n
17     total_te = 0
18     total_tr = 0
19
20     calculandoTempoEspera(processos, n, td, te, tc)
21     calcularTempoRetorno(processos, n, td, te, tr)
22

```

```

23     print("Processos \t\t Tempo Duração \t\t"
24           " Tempo Espera \t\t Tempo Retorno")
25
26     for i in range(n):
27         total_te = total_te + te[i]
28         total_tr = total_tr + tr[i]
29         print(f"{i + 1} \t\t\t\t\t"
30               f"{td[i]} \t\t\t\t\t"
31               f"{te[i]} \t\t\t\t\t {tr[i]}")
32
33     print(f"Tempo médio espera = {total_te / n}")
34     print(f"Tempo médio retorno = {total_tr / n}")
35
36
37 if __name__ == "__main__":
38     processos = [1, 2, 3, 4]
39     chegada = [0, 1, 2, 3]
40     n = len(processos)
41     duracao = [5, 3, 8, 6]
42     calculandoTempoMedio(processos, n, duracao, chegada)

```

Programa 114 – FIFO com tempo de chegada diferentes

As diferenças de código entre os programas 113 e 114 estão na linha 1, onde na definição da função foi colocado mais um parâmetro: o tempo de chegada com a variável `tc`. Na linha 6, agora temos que considerar o tempo de chegada de cada processo, que é diferente de zero, e descontar do tempo de chegada do processo anterior. Na linha 39, definimos os tempos de chegada e por fim na linha 42 chamamos a função, agora, passando o vetor dos tempos de chegada. Podemos usar este último algoritmo como genérico e passar o tempo de chegada como 0 no vetor

chegada. A pergunta que fazemos é se isso funcionará se o tempo de chegada do processo for maior que o tempo de execução do processo anterior. Façam isso como atividade.

Vamos agora usar os dois exemplos utilizando o algoritmo do trabalho mais curto primeiro ou SJF (*Short Job First*) onde os processos de menor tamanho ou que ocupam menor tempo deverão ser executados primeiro. A ideia é pegar o menor processo, ou seja, com menor tempo de duração, e executá-lo primeiro. Temos que levar em consideração que no ambiente real o sistema operativo não tem como saber o tempo de duração de um processo no início, então ele iniciaria fazendo um sorteio para conhecer o processo e depois rodar algum algoritmo para fazer a estimativa da próxima vez que ele for rodar. Nosso SJF irá considerar que é possível já ter esse valor de forma antecipada como anteriormente. Então só precisamos ordenar nossos processos em ordem crescente e rodar nosso FIFO, conforme mostra a Figura 34.

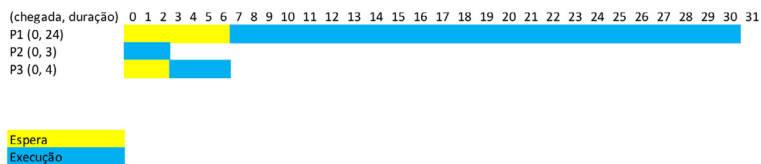


Figura 34 – Algoritmo SJF.

Vamos primeiro verificar os tempos:

- a. tempo de conclusão é $P1 = 31$, $P2 = 3$, $P3 = 7$
- b. o tempo de chegada de todos os processos foi 0
- c. o tempo de duração de $P1 = 24$, $P2 = 3$ e $P3 = 4$
- d. tempo de retorno (a) - (b) então $P1 = 31 - 0 = 31$, $P2 = 3 - 0 = 3$, $P3 = 7 - 0 = 7$

- e. tempo de espera (d) - (c) então $P1 = 31 - 24 = 7$, $P2 = 3 - 3 = 0$, $P3 = 7 - 4 = 3$
- f. tempo médio de espera $P1(e) + P2(e) + P3(e) / 4 = (7 + 0 + 3) / 3 = 3,33$
- g. tempo médio de retorno $P1(d) + P2(d) + P3(d) / 3 = (31 + 3 + 7) / 3 = 13,66$

Veja que diminuimos muito o tempo médio de espera e de retorno. Agora, vamos ao código no Programa 115.

```

1  def calculandoTempoEspera(processos, n, td, te, tc):
2      # considerando processo iniciando em 0
3      te[0] = 0
4
5      for i in range(1, n):
6          te[i] = (td[i - 1] + te[i - 1]) - (tc[i] - tc[i - 1])
7
8
9  def calcularTempoRetorno(processos, n, td, te, tr):
10     for i in range(n):
11         tr[i] = td[i] + te[i]
12
13
14  def calculandoTempoMedio(processos, n, td, tc):
15     te = [0] * n
16     tr = [0] * n
17     total_te = 0
18     total_tr = 0
19
20     calculandoTempoEspera(processos, n, td, te, tc)
21     calcularTempoRetorno(processos, n, td, te, tr)
22

```

```

23     print("Processos \t\t Tempo Duração \t\t"
24           " Tempo Espera \t\t Tempo Retorno")
25
26     for i in range(n):
27         total_te = total_te + te[i]
28         total_tr = total_tr + tr[i]
29         print(f"{i + 1} \t\t\t\t\t\t\t"
30               f"{td[i]} \t\t\t\t\t\t\t"
31               f"{te[i]} \t\t\t\t\t\t\t {tr[i]}")
32
33     print(f"Tempo médio espera = {total_te / n}")
34     print(f"Tempo médio retorno = {total_tr / n}")
35
36
37 if __name__ == "__main__":
38     processos = [1, 2, 3, 4]
39     chegada = [0, 0, 0, 0]
40     n = len(processos)
41     duracao = [5, 3, 8, 6]
42     duracao.sort()
43     calculandoTempoMedio(processos, n, duracao, chegada)

```

Programa 115 – Algoritmo SJF.

A única diferença está linha 42, onde modificamos a ordem dos processos.

Vamos ao segundo exemplo para comparar os resultados e, neste caso, voltamos a considerar que todos os processos chegaram ao mesmo tempo, senão a ordenação perde o sentido, ou melhor, não faria sentido. Por exemplo, parar o processo 1 depois de iniciado para rodar o processo 3, que chegou logo em seguida no tempo 1, e que teria prioridade na execução. Isso é feito de uma melhor maneira usando preempção, ilustrado na Figura 35.

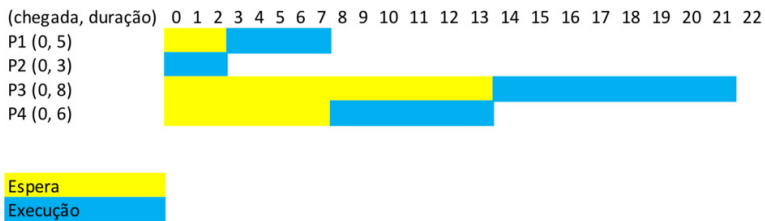


Figura 35 – Algoritmo SJF com preempção.

Vamos primeiro verificar os tempos:

- tempo de conclusão é $P1 = 8$, $P2 = 3$, $P3 = 22$, $P4 = 14$
- o tempo de chegada de todos os processos foi 0
- o tempo de duração de $P1 = 5$, $P2 = 3$ e $P3 = 8$, $P4 = 6$
- tempo de retorno (a) - (b) então $P1 = 8 - 0 = 8$, $P2 = 3 - 0 = 3$, $P3 = 22 - 0 = 22$, $P4 = 14 - 0 = 14$
- tempo de espera (d) - (c) então $P1 = 8 - 5 = 3$, $P2 = 3 - 3 = 0$, $P3 = 22 - 8 = 14$, $P4 = 14 - 6 = 8$
- tempo médio de espera $P1(e) + P2(e) + P3(e) + P4(e) / 4 = (3 + 0 + 14 + 8) / 4 = 6,25$
- tempo médio de retorno $P1(d) + P2(d) + P3(d) + P4(d) / 4 = (8 + 3 + 22 + 14) / 4 = 11,75$

Observamos que o tempo médio de espera e retorno nesse caso foram piores que no primeiro exemplo, que tinha tempo de espera menor pois iniciaram em tempos diferentes de 0.

REFERÊNCIAS

AKSHAR. ***Producer-consumer problem in Python***. Disponível em: <https://www.agiliq.com/blog/2013/10/producer-consumer-problem-in-python/>. Acessado em: 1 jul. 2019.

CARVALHO, Thiago Leite e. **Orientação a Objetos**: Aprenda seus conceitos e suas aplicabilidades de forma efetiva. Casa do Código: São Paulo, 2016.

CRUZ, Felipe. Python: **Escreva seus primeiros programas**. Casa do Código: São Paulo, 2015.

HUNT, John. ***A Beginners Guide to Python 3 Programming***. Springer International Publishing, 2019.

REAL PYTHON. ***An Introduction to Threading in Python***. Disponível em: <https://realpython.com/intro-to-python-threading/>. Acessado em: 26 jun. 2019.

PYMOTW. ***Threading***: manage concurrent threads. Disponível em: <https://pymotw.com/2/threading/>. Acessado em: 26 jun. 2019.

MACHADO, Francis Berenger; MAIA, Luiz Paulo. **Arquitetura de Sistemas Operacionais**. 4. ed. Rio de Janeiro: LTC, 2007.

MENEZES, Nilo Ney Coutinho. **Introdução à Programação com Python**: algoritmos e lógica de programação para iniciantes. 2. Ed. São Paulo: Novatec, 2014.

TANENBAUM, Andrew S.. **Sistemas Operacionais Modernos**. 3. ed. São Paulo: Pearson, 2010.

GEEKSFORGEEKS. ***Program for FCFS Scheduling | Set 1***. Disponí-

vel em: <https://www.geeksforgeeks.org/program-for-fcfs-cpu-scheduling-set-1/>. Acesso em: 17 jul. 2019.

GEEKSFORGEEEKS. ***Program for Shortest Job First (or SJF) CPU Scheduling | Set 1 (Non-preemptive)***. Disponível em: <https://www.geeksforgeeks.org/program-for-shortest-job-first-or-sjf-cpu-scheduling-set-1-non-preemptive/>. Acessado em: 17 jul. 2019.

GEEKSFORGEEEKS. ***Program for Best Fit algorithm in Memory Management***. Disponível em: <https://www.geeksforgeeks.org/program-best-fit-algorithm-memory-management/>. Acessado em: 24 jul. 2019.

SHAH, Ketan; MITRA, Anirban; MATANI, Dhruv. ***An $O(1)$ algorithm for implementing the LFU cache eviction scheme***. v. 1, p. 1-8, 2010.



Todos os direitos são reservados à Editora IFRN, não podendo ser comercializado em período de contrato de cessão de direitos autorais.

Em caso de reimpressão com recursos próprios do autor, está liberada a sua comercialização

A Editora do Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte (IFRN) já publicou livros em todas as áreas do conhecimento, ultrapassando a marca de 150 títulos. Atualmente, a edição de suas obras está direcionada a cinco linhas editoriais, quais sejam: acadêmica, técnico-científica, de apoio didático-pedagógico, artístico-literária ou cultural potiguar.

Ao articular-se à função social do IFRN, a Editora destaca seu compromisso com a formação humana integral, o exercício da cidadania, a produção e a socialização do conhecimento.

Nesse sentido, a EDITORA IFRN visa promover a publicação da produção de servidores e estudantes deste Instituto, bem como da comunidade externa, nas várias áreas do saber, abrangendo edição, difusão e distribuição dos seus produtos editoriais, buscando, sempre, consolidar a sua política editorial, que prioriza a qualidade.



editoraifrn





FÁBIO AUGUSTO PROCÓPIO DE PAIVA

Possui experiência em desenvolvimento de sistemas e em administração de banco de dados. Doutor (2016) em Engenharia Elétrica e de Computação pela Universidade Federal do Rio Grande do Norte (UFRN). Professor do Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte e desenvolve projetos com a Universidade de Coimbra, Portugal.



JOÃO MARIA ARAÚJO DO NASCIMENTO

Professor do Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte. Possui experiência em desenvolvimento de sistemas de informação e de automação industrial. Atua em projetos para ensino de robótica e de temáticas aeroespaciais. Mestre em Engenharia Elétrica e de Computação (2005) pela Universidade Federal do Rio Grande do Norte.



RODRIGO SIQUEIRA MARTINS

Professor do Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte. Experiência em administração e gerenciamento de redes de computadores, sistemas operacionais, IoT e áreas afins. Doutor (2014) em Sistemas e Computação pela Universidade Federal do Rio Grande do Norte e Pós-Doutorado na Universidade de Lancaster, Inglaterra.



GIVANALDO ROCHA DE SOUZA

Professor do Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte. Doutor em Ciência da Computação, na área de Algoritmos Experimentais, pela Universidade Federal do Rio Grande do Norte. Possui experiência em desenvolvimento de sistemas e já palestrou na Python Nordeste e Python Brasil.

Por que mais um livro abordando a linguagem Python? Essa foi uma das perguntas que fizemos ao longo do planejamento desta obra. época, reunimos vários livros de programação a fim de avaliá-los em termos de linguagem, de didática, de aplicações e de exercícios. Concluímos que há vários livros muito bons, porém não encontramos aquele que reunisse boa linguagem, diversos exercícios e aplicações práticas. Falando em aplicação prática, foi unânime a decisão, em abordarmos neste livro, algoritmos aplicados em Sistemas Operacionais. O motivo é o fato de esses algoritmos serem ferramentas excelentes para consolidação de conhecimentos de programação, além de serem essenciais na formação de um pro-gramador profissional. E mais, um sistema operativo é um dos melhores exemplos de programa-ção: são milhões de linhas de código que cobrem um grande espectro de problemas. Entender um pouco desse mundo pode fazer de o9i um pro-gramador melhor.

