

Unidade III

5 FUNÇÕES, STRINGS E ARQUIVOS

Neste tópico, trabalharemos com a manipulação de funções, strings e arquivos, além das funções embutidas nas bibliotecas do Python, como as que já vimos informalmente até aqui: `print()`, `input()`, `range()`, `sum()` etc. Aprenderemos como construir nossas próprias funções. Alguns conceitos importantes de escopo de variáveis também serão tema.

5.1 Definindo funções

Uma função é um bloco de código que somente vai ser executado se for explicitamente chamado. Podemos passar dados ou informações para as funções, que são chamados de parâmetros ou argumentos. Uma função pode ou não retornar um dado como resultado para quem a chamou; a decisão é do programador da função. Vamos agora colocar o conceito em prática para um melhor entendimento.

Exemplo 1:

```
def primeira_funcao():  
    print("Olá da minha primeira_funcao!")
```

Esse trecho de código sozinho não tem uma saída, pois a função foi definida, mas não foi explicitamente chamada. Toda função é definida com a palavra reservada `def` e, logo após, vem o seu nome, dado pelo programador. No caso do exemplo, `primeira_funcao`. A `primeira_funcao` não recebe nenhum argumento, por isso não existe nada entre os parênteses. O bloco de código interno da função deve ser indentado, como já vimos em outras estruturas. Depois de definida, a função pode ser chamada, como no próximo exemplo.

Exemplo 2:

```
def primeira_funcao():  
    print("Olá da minha primeira_funcao!")  
  
primeira_funcao()
```

Saída:
Olá da minha primeira_funcao!



Observação

Biblioteca, em linguagem de programação, é um conjunto de funções pré-escritas por outro programador que resolvam determinado domínio de problemas e que podem ser reutilizadas em nossos programas.

5.1.1 Parâmetros

Como comentado, podemos passar informações como parâmetros para as funções. Os parâmetros são especificados dentro dos parênteses, logo após o nome das funções. Podemos adicionar qualquer quantidade de parâmetros que for necessária; para isso, os separamos com vírgula.

No exemplo a seguir, a função recebe um parâmetro (nome). Quando a função é chamada, passamos o valor do parâmetro com ela. O parâmetro é usado dentro da função para imprimir a mensagem.

Exemplo:

```
def segunda_funcao(nome):  
    print("Olá, ", nome)
```

```
segunda_funcao("Maria")  
segunda_funcao("Jonas")  
segunda_funcao("Ana")
```

Saída:

```
Olá, Maria  
Olá, Jonas  
Olá, Ana
```

Uma função deve ser chamada com a quantidades de argumentos com os quais ela foi definida, a não ser que os argumentos tenham um valor padrão (default). O argumento default será visto em uma subseção futura.

```
def terceira_funcao(nome, idade):  
    print(nome, "tem", idade, "anos.")
```

```
segunda_funcao("Maria", 5)  
segunda_funcao("Jonas", 8)  
segunda_funcao("Ana", 10)
```

Saída:

Maria tem 5 anos.

Jonas tem 8 anos.

Ana tem 10 anos.

Uma função pode retornar um valor a quem a chama, usando a instrução `return`.

Exemplo:

```
def soma(x, y):  
    return x + y  
  
print(soma(3, 6))  
print(soma(5, 5))  
print(soma(134, 78))
```

Saída:

9

10

212



Observação

As palavras "parâmetros" e "argumentos" são usadas para designar o mesmo conceito. Na documentação do Python, eles são muitas vezes chamados pela abreviação `args`.

5.1.2 Variáveis locais e globais

Uma variável somente está disponível para acesso dentro da região onde ela foi criada. Esse conceito é chamado de escopo. De acordo com o conceito de escopo, uma variável criada dentro de uma função pertence ao escopo local dessa função, e somente pode ser usada dentro dessa mesma função.

Exemplo:

```
def escopo_funcao():  
    a = 25  
    print(a)  
  
escopo_funcao()
```

Saída:

25

Uma variável criada dentro do corpo principal do programa Python é chamada de variável global e pertence ao escopo global. Variáveis globais estão disponíveis para qualquer parte do programa Python, ou seja, dentro de qualquer escopo, ele sendo global ou local.

Exemplo:

```
a= 10
def imprime_a():
    print(a)
```

```
imprime_x()
print(a)
```

Saída:
10

Em Python, é possível criar uma função dentro de outra função. Nesse caso, variáveis criadas dentro da função "externa" podem ser acessadas dentro da função "interna". Para elucidar o conceito, segue o exemplo:

Exemplo:

```
def externa():
    a = 25
    def interna():
        print(a)
    interna()
```

```
externa()
```

É possível criar uma variável com escopo global dentro de uma função, utilizando a instrução global na criação da variável, como no exemplo a seguir:

Exemplo:

```
def funcao_var():
    global a
    a = 100
```

```
funcao_var()
print(a)
```

Saída:
100

5.1.3 Argumento default

Outra forma de enviar os argumentos para as funções é através da sintaxe chave = valor. Diferentemente da forma tradicional de chamar as funções, em que os parâmetros devem seguir a ordem correta, com a chave = valor, a ordem não importa. Vejamos o exemplo.

Exemplo:

```
def dados_usuario(nome, idade, cidade):  
    print("Nome:", nome)  
    print("Idade:", idade)  
    print("Cidade:", cidade)
```

```
dados_usuario(idade = 34, cidade = "São Paulo", nome = "João Silva")
```

Saída:
Nome: João Silva
Idade: 34
Cidade: São Paulo

É possível utilizar um valor de argumento padrão, caso a função seja chamada sem o argumento, e nesse caso a função usará o valor default (padrão).

Exemplo:

```
def fruta(fruta= "maça"):  
    print("Gosto de", fruta)
```

```
fruta("banana")  
fruta("uva")  
fruta()
```

Saída:
Gosto de banana
Gosto de uva
Gosto de maçã



Lembrete

Uma função é um bloco de código que somente vai ser executado se for explicitamente chamado. A função pode ser construída pelo programador ou pré-programada em bibliotecas Python.

5.2 Manipulação de strings

Até este ponto do estudo, já trabalhamos com as strings, em Python elas são caracteres, palavras, textos, contidos entre aspas simples ou duplas. Além dessa representação, podemos colocar textos multilinhas usando três aspas duplos ou simples.

Como outras linguagens de programação, as strings em Python são encaradas como uma matriz unidimensional de caracteres (vetor). Os colchetes podem ser usados para acessar os caracteres individuais da string e dentro dos colchetes a posição do caractere desejado. Note que a primeira posição da matriz começa em 0.

Exemplo 1:

```
a = """De médico e de  
louco todo mundo tem  
um pouco."""  
b= 'Bola'  
c= "Carro"  
print(a)  
print(b)  
print(c)
```

Saída:

```
De médico e de louco todo mundo tem um pouco.  
Bola  
Carro
```

Exemplo 2:

```
palavra = "humano"  
print(palavra[0])  
print(palavra[1])  
print(palavra[2])  
print(palavra[3])  
print(palavra[4])  
print(palavra[5])
```

Saída:

```
h  
u  
m  
a  
n  
o
```

Uma função bastante utilizada na manipulação de strings é `len()`. Ela recebe como parâmetro a string e retorna seu tamanho.

Exemplo:

```
clube = "Palmeiras"  
print(len(clube))
```

Saída:
9

É possível usar a instrução `in` para verificar se um caractere, palavra ou frase fazem parte da string. A instrução `not in` é o oposto de `in`.

Exemplo 1:

```
frase = "A rua está calma hoje!"  
print("calma" in frase)
```

Saída:
True

Exemplo 2:

```
frase = "A rua está calma hoje!"  
print("calma" in frase)
```

Saída:
True

Exemplo 3:

```
frase = "A rua está calma hoje!"  
if "calma" in frase:  
    print("A frase contém a palavra: calma")
```

Saída:
A frase contém a palavra: calma

Exemplo 4:

```
frase = "A rua está calma hoje!"  
print("calma" in frase)
```

Saída:
True

Exemplo 5:

```
frase = "A rua está movimentada hoje!"  
if "radar" not in frase:  
    print("A frase não contém a palavra: radar")
```

Saída:

A frase não contém a palavra: radar

Caso precisemos somente de parte da string, é possível fatiá-la (slice) usando o índice da matriz, especificando o começo e o final, separando com dois pontos. Vejamos os exemplos.

Exemplo 1:

```
a = "Parado"  
# imprime os índices 2 e 3 da string a  
print(a[2:4])
```

Saída

ra

Exemplo 2:

```
a = "Parado"  
# Sem especificar o início, a fatia começa no índice 0  
print(a[:4])
```

Saída:

Para

Exemplo 3:

```
a = "Parado"  
# Sem especificar o final, a fatia termina no último caractere  
print(a[2:])
```

Saída:

rado

Exemplo 4:

```
a = "Parado"
# Índices negativos começam a contar no último caractere
# -1 é o caractere 'o' de Parado
print(a[-5:-1])
```

Saída:
arad

A concatenação de strings pode ser feita com o operador + (soma). Concatenar é juntar duas strings.

Exemplo:

```
a = 'Maria'
b = 'Fernanda'
c = a + b
print(c)
```

Saída:
Maria Fernanda

As strings têm vários métodos prontos para serem usados. Esses métodos funcionam como funções: sempre retornam algo e não modificam diretamente o conteúdo variável.

A lista desses métodos é extensa. Nos próximos exemplos serão demonstrados alguns dos mais usados.

Exemplo 1:

```
a = "Maria Fernanda"
# retorna a string em letra minúscula
print(a.lower())
```

Saída:
maria fernanda

Exemplo 2:

```
a = "Maria Fernanda"
# retorna a string em letra maiúscula
print(a.upper())
```

Saída:
MARIA FERNANDA

Exemplo 3:

```
a = " Maria Fernanda "  
# retorna sem espaços em branco no início e fim: 'Maria Fernanda'  
print(a.strip())
```

Saída:
Maria Fernanda

Exemplo 4:

```
a = "Maria Fernanda"  
# Separa a string em pedaços de acordo com o delimitador escolhido  
# Nesse caso é o espaço  
print(a.split(" "))  
# O resultado é uma lista com dois itens
```

Saída:
['Maria', 'Fernanda']

Exemplo 5:

```
a = "Maria Fernanda"  
# Troca uma string por outra string  
print(a.replace("Maria", "Ana"))
```

Saída:
Ana Fernanda

Segue a lista de métodos que podem ser utilizados na manipulação de strings:

Quadro 9

Método	Descrição
capitalize()	Converte o primeiro caractere em maiúsculo
casefold()	Converte a string em minúsculo
center()	Retorna uma string centralizada
count()	Retorna o número de vezes que um valor é encontrado na string
encode()	Retorna a versão codificada da string
endswith()	Retorna True se a string finaliza com o valor especificado
expandtabs()	Configura o tamanho da tabulação da string
find()	Procura na string um valor e retorna a posição caso o valor for encontrado

Método	Descrição
<code>format()</code>	Formata valores especificados na string
<code>format_map()</code>	Formata valores especificados na string
<code>index()</code>	Procura na string um valor e retorna a posição caso o valor for encontrado
<code>isalnum()</code>	Retorna True se todos os caracteres da string forem alfanuméricos
<code>isalpha()</code>	Retorna True se todos os caracteres da string forem alfabéticos
<code>isdecimal()</code>	Retorna True se todos os caracteres da string forem decimais
<code>isdigit()</code>	Retorna True se todos os caracteres da string forem dígitos
<code>isidentifier()</code>	Retorna True se a string for um nome de variável
<code>islower()</code>	Retorna True se todos os caracteres forem minúsculos
<code>isnumeric()</code>	Retorna True se todos os caracteres forem numéricos
<code>isprintable()</code>	Retorna True se todos os caracteres forem passíveis de impressão
<code>isspace()</code>	Retorna True se todos os caracteres forem espaços em branco
<code>istitle()</code>	Retorna True se a string seguir as regras de um título
<code>isupper()</code>	Retorna True se todos os caracteres forem maiúsculos
<code>join()</code>	Junta os elementos de objetos com propriedades de iteração no final da string
<code>ljust()</code>	Retorna uma string justificada à esquerda
<code>lower()</code>	Converte a string em minúsculo
<code>lstrip()</code>	Retorna a string sem espaços à esquerda
<code>maketrans()</code>	Retorna uma tabela de tradução a ser usada em traduções
<code>partition()</code>	Retorna uma tupla onde a string é particionada em três
<code>replace()</code>	Retorna uma string onde um valor específico é substituído por outro
<code>rfind()</code>	Procura na string um valor especificado e retorna a última posição onde ele foi encontrado
<code>rindex()</code>	Procura na string um valor especificado e retorna a última posição onde ele foi encontrado
<code>rjust()</code>	Retorna a string justificada à esquerda
<code>rpartition()</code>	Retorna uma tupla onde a string é particionada em três
<code>rsplit()</code>	Separa a string com o delimitador especificado e retorna uma lista
<code>rstrip()</code>	Retorna a string sem espaços à direita
<code>split()</code>	Separa a string com o delimitador especificado e retorna uma lista
<code>splitlines()</code>	Separa a string na quebra de linha e retorna uma lista
<code>startswith()</code>	Retorna True se a string começar com o valor especificado
<code>strip()</code>	Retorna uma versão da string sem espaços à esquerda e à direita
<code>swapcase()</code>	Minúsculo se torna maiúsculo e vice-versa
<code>title()</code>	Converte o primeiro caractere de cada palavra em maiúsculo
<code>translate()</code>	Retorna uma string traduzida
<code>upper()</code>	Converte a string em maiúsculo
<code>zfill()</code>	Preenche o início de uma string com o número especificado de zeros



Observação

As strings possuem mais de 40 métodos diferentes. Para conhecê-los melhor, abra o console do Python e digite: `help(str)`

5.3 Abertura, leitura e gravação em arquivos

A manipulação de arquivos é importante em todo tipo de aplicação e a linguagem Python, assim como outras linguagens, possui inúmeras funções para criar, ler, atualizar e deletar arquivos.

A primeira função estudada para trabalharmos com arquivos é a `open()`, responsável por abrir o arquivo. Ela recebe dois parâmetros: o nome do arquivo e o modo de operação.

Existem quatro modos de operação para abrir o arquivo:

- "r" (Leitura): valor padrão. Abre o arquivo para leitura; se o arquivo não existir, mostrará um erro.
- "w" (Escrita): abre o arquivo para escrita; se o arquivo não existir, ele será criado.
- "a" (Adiciona): abre o arquivo para acrescentar dados; se o arquivo não existir, ele será criado;
- "x" (Cria): cria o arquivo; se o arquivo existir, mostrará um erro.

Também é possível especificar se o arquivo vai ser aberto em modo texto ou binário. Um exemplo de arquivo binário são as imagens e vídeos:

- "t" (texto): valor padrão. Modo texto.
- "b" (binário). Modo binário.

Exemplo 1:

```
arquivo = open("segredo.txt")
```

Exemplo 2:

```
# modo escrita e binário
```

```
arquivo = open("segredo_criptografado.txt", 'wb')
```

É interessante notar que os arquivos dos exemplos 1 e 2 devem estar na mesma pasta que o programa escrito em Python. É possível colocar, em vez de somente o nome dos arquivos, o caminho completo desses arquivos, como nos exemplos a seguir:

Exemplo 1:

no windows

```
arquivo = open("c:\\users\\administrator\\segredo.txt")
```

Exemplo 2:

modo escrita e binário

no linux

```
arquivo = open("/home/root/segredo_criptografado.txt", 'wb')
```



Observação

Tenha certeza de que o arquivo realmente exista caso seja aberto para leitura (opção 'r'). Do mesmo modo, tenha certeza de que o arquivo não exista caso seja aberto com a opção de criação 'x'; caso contrário, o Python mostrará um erro.

Agora que já sabemos abrir um arquivo, podemos usar a função `read()` e `readline()` para ler. Essa função tem várias opções de como ler o arquivo aberto.

Exemplo 1:

```
arquivo = open("segredo.txt", "r")  
# Lê todo o arquivo  
print(segredo.read())
```

Exemplo 2:

```
arquivo = open("segredo.txt", "r")  
# Lê os 5 primeiros caracteres do arquivo  
print(segredo.read(5))
```

Exemplo 3:

```
arquivo = open("segredo.txt", "r")  
# Lê a primeira linha do arquivo  
print(segredo.readline())
```

Exemplo 4:

```
arquivo = open("segredo.txt", "r")
# Lê a primeira e segunda linha do arquivo
print(segredo.readline())
print(segredo.readline())
```

Exemplo 5:

```
arquivo= open("segredo.txt", "r")
# lê linha a linha do arquivo
for linha in arquivo:
    print(linha)
```

Exemplo 6:

```
arquivo= open("segredo.txt", "r")
# lê linha a linha do arquivo
for linha in arquivo:
    print(linha)

# fecha o arquivo ao final do laço
arquivo.close()
```



Observação

É uma boa prática fechar o arquivo depois de terminar as operações nele. Por causa do buffer do sistema operacional, as modificações só aparecem depois do fechamento.

Para escrever em um arquivo, temos que abri-lo nos modos 'a' ou 'w'. A diferença entre eles é que o 'w' sobrescreve o arquivo existente e o 'a' adiciona o conteúdo no final do arquivo.

Exemplo 1:

```
arquivo = open("segredos.txt", "a")
arquivo.write("A senha é 123456")
arquivo.close()
```

Exemplo 2:

```
arquivo = open("segredos.txt", "w")
arquivo.write("Todo o conteúdo foi deletado!")
arquivo.close()
```



Observação

Para deletar um arquivo, é necessário importar o módulo 'os' do Python que contém a função `remove()`:

```
import os
os.remove("segredo.txt")
```

6 MATRIZES E ESTRUTURA DE DADOS

As estruturas de dados, incluindo as matrizes, facilitam bastante o processamento e diminuem a quantidade de código que precisa ser escrito. Elas implementam comportamentos e ideias que usamos na matemática (conjuntos, matrizes, sequências etc.) e padrões do mundo real (pilha, fila, lista etc.). A importância de estudo e do trabalho com essas estruturas é tão grande que existe uma disciplina inteira que trata somente delas.

Este tópico é uma introdução às estruturas de dados e mostrará como programar usando algumas delas.



Lembrete

Biblioteca, em linguagem de programação, é um conjunto de funções pré-escritas por outro programador que resolvam determinado domínio de problemas e que podem ser reusadas em nossos programas.

6.1 Manipulação básica de matrizes

As bibliotecas padrão do Python não incluem a estrutura de dados matrizes. É possível simular matrizes bidimensionais e multidimensionais usando listas de listas. O problema é que as listas não têm funções prontas para manipular matrizes e o programador deverá construir essas funções. Provavelmente, dependendo do trabalho matemático do programa com matrizes, ele terá problemas com desempenho, pelo fato de as listas não serem otimizadas para esse tipo de trabalho.

Existe uma alternativa para esse problema: a biblioteca NumPy, uma biblioteca gratuita e largamente utilizada na comunidade, que trabalha com matrizes e álgebra linear.

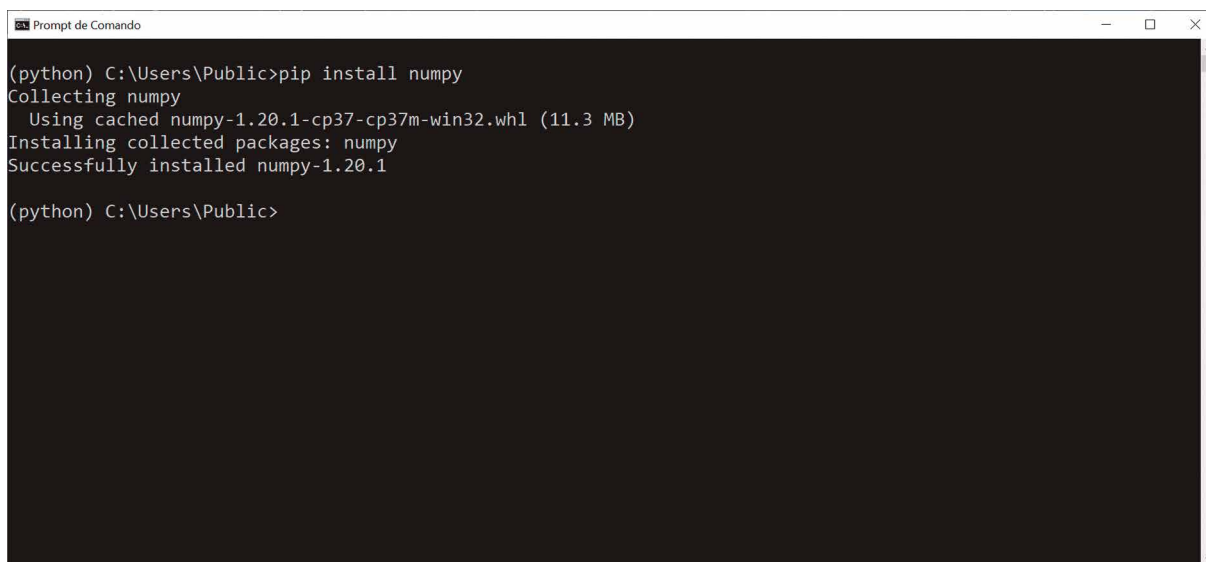
A NumPy foi escrita parcialmente em Python. A maior parte dela, que requer bastante processamento, foi escrita em linguagem C e C++.

Caso você tenha instalado o Python por meio da distribuição Anaconda, a biblioteca NumPy já está disponível para uso e nenhuma ação é requerida. Aqueles que fizeram a instalação do pacote básico do Python podem instalar a biblioteca a partir do instalador de pacotes Python (Pip).

Exemplo:

1 – Abra o terminal do Linux ou MacOS, ou o prompt de comando do Windows.

2 – Digite: `pip install numpy`



```
(python) C:\Users\Public>pip install numpy
Collecting numpy
  Using cached numpy-1.20.1-cp37-cp37m-win32.whl (11.3 MB)
Installing collected packages: numpy
Successfully installed numpy-1.20.1

(python) C:\Users\Public>
```

Figura 51 – Instalação do pacote NumPy no Windows

Após a instalação, podemos usar a biblioteca usando o comando `import` no Python.

Exemplo:

```
import numpy
matriz = numpy.array([1, 2, 3])
print(matriz)
```

Saída:
[1 2 3]

Matriz e array podem ser usados como sinônimos neste livro-texto. O exemplo que acabamos de mostrar importa a biblioteca `numpy` e utiliza um método (função) que recebe uma lista numérica, criando a variável `matriz` (nesse caso, um vetor ou matriz unidimensional). Podemos criar apelidos para as bibliotecas importadas, assim não precisamos escrever `numpy` toda vez que invocarmos uma função.

Exemplo 1:

```
import numpy as np
matriz = np.array([1, 2, 3])
print(matriz)
```

Saída:
[1 2 3]

Exemplo 2:

```
import numpy as np
# Matriz de dimensão zero (escalar)
matriz = np.array(50)
print(matriz)
```

Saída:
50

Exemplo 3:

```
import numpy as np
# Matriz 2D (lista de lista)
matriz = np.array([[1, 2, 3], [4, 5, 6]])
print(matriz)
```

Saída:
[[1 2 3]
 [4 5 6]]

Em matemática, a dimensão é definida como o número mínimo de coordenadas necessárias para descrever qualquer ponto dentro de um espaço, mas em NumPy é o mesmo que eixo ou eixos. 0D ou dimensão zero é um número escalar único e não é necessário eixo para defini-lo. Já em 1D ou uma dimensão, temos o eixo x; matrizes com uma linha e várias colunas têm uma dimensão. A matriz de duas dimensões contém várias linhas e várias colunas, ou seja, eixos x e y para defini-la.

Criamos matrizes de 0D, 1D e 2D, e é possível criar matrizes de n dimensão utilizando listas dentro de listas e assim por diante.

Após mostrarmos como descobrir a dimensão das matrizes, estudaremos como acessar os elementos da matriz e suas operações básicas.

Exemplo:

```
import numpy as np
m1= np.array(10)
m2= np.array([1, 2, 3, 4])
m3= np.array([[1, 2, 3], [4, 5, 6]])
m4= np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

print(m1.ndim)
print(m2.ndim)
print(m3.ndim)
print(m4.ndim)
```

Saída:

```
0
1
2
3
```

Trata-se do mesmo método que, como foi visto em como acessar um caractere de uma string, é usado para matrizes. A indexação das matrizes também começa em zero.

Exemplo 1:

```
import numpy as np
m1 = np.array([1, 2, 3, 4, 5, 6])
# imprime o primeiro elemento da matriz
print(m1[0])
```

Saída:

```
1
```

Exemplo 2:

```
import numpy as np
m1 = np.array([1, 2, 3, 4, 5, 6])
# imprime o último elemento da matriz
print(m1[-1])
```

Saída:

```
6
```

Exemplo 3:

```
import numpy as np
m1 = np.array([[1,2,3,4], [6,7,8,9]])
print('Terceiro elemento da primeira dimensão: ', m1[0, 2])
```

Saída:

Terceiro elemento da primeira dimensão: 3

Exemplo 4:

```
import numpy as np
m1 = np.array([[1,2,3,4], [6,7,8,9]])
print('Primeiro elemento da segunda dimensão: ', m1[1, 0])
```

Saída:

Terceiro elemento da primeira dimensão: 6

As operações com matrizes mantêm as regras matemáticas sobre matrizes. Isso quer dizer que uma operação de soma entre um número escalar e uma matriz mantém relação fiel com as regras da matemática, e o escalar é somado elemento a elemento da matriz e é retornada à nova matriz. No caso de soma entre duas ou mais matrizes, o primeiro elemento é somado ao primeiro elemento da segunda matriz, e assim com todos os elementos. As matrizes devem ter as mesmas dimensões. A boa notícia é que não precisamos nos preocupar com isso, pois, se algo estiver errado, o Python avisará!

Exemplo 1:

```
import numpy as np
m1 = np.array([[1,2,3,4], [6,7,8,9]])
n = 5
print('m1: ')
print(m1)
print('n: ', n)
print('m1 + n: ')
# soma de matriz com escalar
print(m1 + n)
```

Saída:

```
m1:
[[1 2 3 4]
 [6 7 8 9]]
n: 5
m1 + n:
[[ 6  7  8  9]
 [11 12 13 14]]
```

Exemplo 2:

```
import numpy as np
m1 = np.array([[1,2,3,4], [6,7,8,9]])
n = 5
print('m1: ')
print(m1)
print('n: ', n)
print('m1 - n: ')
# subtração matriz com escalar
print(m1 - n)
```

Saída:

```
m1:
[[1 2 3 4]
 [6 7 8 9]]
n: 5
m1 - n:
[[-4 -3 -2 -1]
 [ 1  2  3  4]]
```

Exemplo 3:

```
import numpy as np
m1 = np.array([[1,2,3,4], [6,7,8,9]])
n = 5
print('m1: ')
print(m1)
print('n: ', n)
# multiplicação matriz com escalar
print('m1 * n: ')
print(m1 * n)
```

Saída:

```
m1:
[[1 2 3 4]
 [6 7 8 9]]
n: 5
m1 * n:
[[ 5 10 15 20]
 [30 35 40 45]]
```

Exemplo 4:

```
import numpy as np
m1 = np.array([[1,2,3,4], [6,7,8,9]])
n = 5
print('m1: ')
print(m1)
print('n: ', n)
print('m1 / n: ')
# divisão matriz com escalar
print(m1 / n)
```

Saída:

```
m1:
[[1 2 3 4]
 [6 7 8 9]]
n: 5
m1 / n:
[[0.2 0.4 0.6 0.8]
 [1.2 1.4 1.6 1.8]]
```

Exemplo 5:

```
import numpy as np
m1 = np.array([[1,2,3,4], [6,7,8,9]])
m2 = np.array([[3,5,6,9], [1,2,4,8]])
print('m1: ')
print(m1)
print('m2: ')
print(m2)
print('m1 + m2: ')
# soma matriz com matriz
print(m1 + m2)
```

Saída:

```
m1:
[[1 2 3 4]
 [6 7 8 9]]
m2:
[[3 5 6 9]
 [1 2 4 8]]
m1 + m2:
[[ 4  7  9 13]
 [ 7  9 12 17]]
```

Exemplo 6:

```
import numpy as np
m1 = np.array([[1,2,3,4], [6,7,8,9]])
m2 = np.array([[3,5,6,9], [1,2,4,8]])
print('m1: ')
print(m1)
print('m2: ')
print(m2)
print('m1 - m2: ')
# subtração matriz com matriz
print(m1 - m2)
```

Saída:

```
m1:
[[1 2 3 4]
 [6 7 8 9]]
m2:
[[3 5 6 9]
 [1 2 4 8]]
m1 - m2:
[[-2 -3 -3 -5]
 [ 5  5  4  1]]
```

Exemplo 7:

```
import numpy as np
m1 = np.array([[1,2,3,4], [6,7,8,9]])
m2 = np.array([[3],[5]])
print('m1: ')
print(m1)
print('m2: ')
print(m2)
print('m1 * m2: ')
# multiplicação matriz com matriz
print(m1 * m2)
```

Saída:

```
m1:
[[1 2 3 4]
 [6 7 8 9]]
m2:
[[3]
 [5]]
m1 * m2:
[[ 3  6  9 12]
 [30 35 40 45]]
```

Exemplo 8:

```
import numpy as np
m1 = np.array([[1,2,3,4], [6,7,8,9]])
m2 = np.array([[3],[5]])
print('m1: ')
print(m1)
print('m2: ')
print(m2)
print('m1 / m2: ')
# divisão matriz com matriz
print(m1 / m2)
```

Saída:

```
m1:
[[1 2 3 4]
 [6 7 8 9]]
m2:
[[3]
 [5]]
m1 / m2:
[[0.33333333 0.66666667 1. 1.33333333]
 [1.2 1.4 1.6 1.8 ]]
```



Saiba mais

O código-fonte da biblioteca NumPy está disponível em:

GITHUB. *NumPy*. [s.d.]. Disponível em: <https://cutt.ly/GmdK3Cy>. Acesso em: 25 jun. 2021.

6.2 Estrutura de dados

As estruturas de dados mais utilizadas em Python são as listas e dicionários que serão detalhados na próxima unidade. A importância das listas em Python já foi notada. Mesmo sem conhecer seu conceito e detalhes, as usamos em estruturas for e matrizes. Os dicionários ganharam grande importância com a crescente utilização de dados hierárquicos na web e apps, com destaques a arquivos XML e mais recentemente JSON.

Neste tópico, estudaremos um pouco mais sobre listas, tuplas, sequências e conjuntos.



Observação

Em Python, existe o conceito de coleções de dados: listas e tuplas (ordenadas) e conjuntos e dicionários (não ordenados).

6.2.1 Conceito de lista

A lista em Python é uma coleção de elementos ordenada e mutável. Ordenada quer dizer que existe uma sequência e ordem; mutável significa que podemos modificar seus elementos. A lista também permite a inclusão de elementos duplicados.

Até o momento, vimos listas com um tipo de elemento, mas elas podem armazenar vários tipos em uma mesma lista.

Exemplo 1:

```
lista_1 = [ 65, True, 'humano', 3.333, ['outra', 'lista'], True, 65]
print(lista_1)
print('O tamanho da lista 1 é:', len(lista_1))
print('O tipo da lista 1 é:', type(lista_1))
```

Saída:

```
[65, True, 'humano', 3.333, ['outra', 'lista'], True, 65]
O tamanho da lista 1 é: 7
O tipo da lista 1 é: <class 'list'>
```

Exemplo 2:

```
lista_1 = [ 65, True, 'humano', 3.333, ['outra', 'lista'], True, 65]
print(lista_1)
# modificando o primeiro e o último elemento da lista
lista_1[0] = 1955
lista_1[6] = 'Maria'
print(lista_1)
# modificando a lista dentro da lista
lista_1[4][1] = 'lista???'
print(lista_1)
```

Saída:

```
[65, True, 'humano', 3.333, ['outra', 'lista'], True, 65]
[1955, True, 'humano', 3.333, ['outra', 'lista'], True, 'Maria']
[1955, True, 'humano', 3.333, ['outra', 'lista???'], True, 'Maria']
```


Existem várias funções que facilitam o programador a trabalhar com listas, tais como: adicionar elemento, remover elemento, inverter a ordem da lista, ordenar uma lista e outros. Essas facilidades serão apresentadas na próxima unidade.

6.2.2 Tupla

Uma tupla é uma coleção ordenada e imutável de elementos. As tuplas são geralmente utilizadas para guardar itens em uma variável simples. Diferente das listas que são representadas na linguagem Python com colchetes, as tuplas utilizam parênteses.

É importante notar que as tuplas podem conter itens de diferentes tipos e também repetidos, porém não é possível modificá-las (adicionar, remover ou modificar itens). Seus itens são acessados da mesma forma que nas listas.

Exemplo 1:

```
minha_tupla = ("fusca", "monza", "opala", "corsa", "palio")
print(minha_tupla)
```

Saída:

```
('fusca', 'monza', 'opala', 'corsa', 'palio')
```

Exemplo 2:

```
minha_tupla = ("fusca", "monza", "opala", "corsa", "palio")
# transformando tupla em lista
print(list(minha_tupla))
```

Saída:

```
['fusca', 'monza', 'opala', 'corsa', 'palio']
```

Exemplo 3:

```
minha_tupla = ("fusca", "monza", "opala", "corsa", "palio")
# tamanho da tupla
print(len(minha_tupla))
```

Saída:

```
5
```

Exemplo 4:

```
uni_tupla = ("Maria",)
# tupla de 1 elemento
print(type(uni_tupla))
```

Saída:

```
<class 'tuple'>
```

A característica de imutabilidade das tuplas oferece segurança aos dados armazenados. Por isso, uma das finalidades das tuplas é guardar dados ou informações que não serão alterados em outras partes do código.

Exemplo:

```
# Localização da Praça da Sé em São Paulo em latitude e longitude
praca_da_se = (-23.5499158, -46.6334289)

# Garantimos que essa informação nunca mude na execução do programa
print(praca_da_se)
```



Observação

Lembre-se que, para criar uma tupla com um único elemento, é necessário colocar uma vírgula após o item. Exemplo: (23,)

A tupla não é completamente imutável. Ao armazenar uma lista, por exemplo, os objetos da lista podem ser modificados. A estrutura da tupla porém, não é alterada.

6.2.3 Sequência

A sequência é um conceito importante em Python, pois as estruturas de dados sequências possuem funções e métodos específicos para sua manipulação.

Não estudaremos todas as estruturas de sequências. Algumas delas são: strings, tuplas, objetos range (faixa numérica), listas, sequências de bytes e matrizes de bytes. A principal característica das sequências é sua ordem determinística, que a difere das outras estruturas de dados.

As sequências de bytes são objetos imutáveis que guardam 1 byte por elemento, ou seja, cada posição pode receber de 0 até 255. A função bytes() pode receber uma lista de inteiros.

Exemplo 1:

```
a = bytes([0,1,2,3,11,15,255])
# os bytes são representados em hexadecimal
print(a)
```

Saída:
b'\x00\x01\x02\x03\x0b\x0f\xff'

Exemplo 2:

```
a = bytes('Maria', 'utf-8')
# recebendo uma string e mostrando em caracteres ascii
print(a)
```

Saída:
b'Maria'

As matrizes de bytes ou bytes arrays funcionam da mesma forma que as sequências de bytes, porém são mutáveis.

Exemplo:

```
a = bytearray(8)
print(a)
# modificando o primeiro byte
a[0] = 255
print(a)
```

Saída:
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00')
bytearray(b'\xff\x00\x00\x00\x00\x00\x00\x00')

Operações que podemos utilizar em sequências:

- **Concatenação:** adiciona o segundo elemento depois do primeiro, usando o operador +.
- **Multiplicação por um inteiro:** copia o elemento e o concatena x vezes, usando um inteiro x e o operador *.
- **Pertencimento:** verifica se um valor faz parte da sequência, usando o operador in.
- **Fatiamento:** podemos fatiar a sequência em partes, usando o operador de slice [:].
- **Funções len(), min() e max() podem ser usadas.**

6.2.4 Conjunto (set)

Conjuntos ou sets são bem conhecidos na matemática, e o Python implementa o mesmo conceito na linguagem de programação com as mesmas características. Conjuntos são utilizados para guardar itens em uma única variável. Eles não são ordenados, não são indexados e são mutáveis.

Os conjuntos são definidos com chaves, como na matemática.

Exemplo 1:

```
conjunto_a = {"banana", "maça", "melancia"}  
print(conjunto_a)
```

Saída:

```
{'maça', 'melancia', 'banana'}
```

Exemplo 2:

```
conjunto_a = {"banana", "maça", "melancia", "banana", "melancia"}  
# um conjunto não aceita itens repetidos  
print(conjunto_a)
```

Saída:

```
{'maça', 'melancia', 'banana'}
```

Exemplo 3:

```
conjunto_a = {"banana", "maça", "melancia"}  
# adicionando item ao conjunto  
conjunto_a.add('laranja')  
print(conjunto_a)
```

Saída:

```
{'laranja', 'maça', 'melancia', 'banana'}
```

Exemplo 4:

```
conjunto_a = {"banana", "maça", "melancia"}  
conjunto_b = {"uva", "maracuja"}  
# adicionando conjunto_b ao conjunto_a  
conjunto_a.update(conjunto_b)  
print(conjunto_a)
```

Saída:

```
{'uva', 'maracuja', 'melancia', 'maça', 'banana'}
```

Exemplo 5:

```
conjunto_a = {"banana", "maça", "melancia"}  
# removendo itens discard() e remove()  
conjunto_a.remove('maça')  
# remove() mostra erro quando o item removido existe, discard() não  
conjunto_a.discard('uva')  
print(conjunto_a)
```

Saída:

```
{'melancia', 'banana'}
```

Exemplo 6:

```
conjunto_a = {"banana", "maça", "melancia"}  
# deleta o conjunto  
del conjunto_a
```

Existem outros métodos para manipulação de conjuntos que replicam as características da Teoria dos Conjuntos. São eles: `union()`, `intersection()`, `symmetric_difference()` etc.



Observação

Os conjuntos possuem mais de 20 métodos diferentes. Para conhecê-los abra o console do Python e digite: `help(set)`



Saiba mais

A Maratona SBC de Programação é um evento de programação dirigido para alunos cursando Ensino Superior em Computação e áreas afins. Uma das linguagens aceitas na competição é o Python.

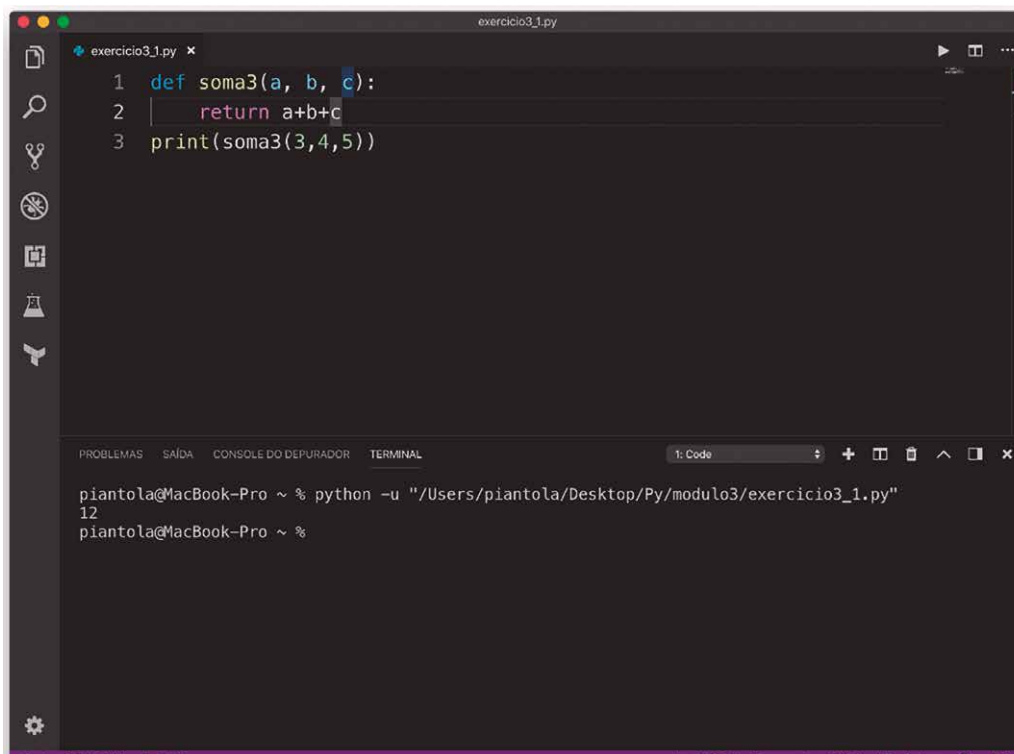
Para saber mais sobre a Maratona SBC de Programação, acesse:

Disponível em: <https://cutt.ly/vmSuHcA>. Acesso em: 13 jul. 2021.

6.3 Exercícios sugeridos

1. Construa um programa com a função 'soma3' que recebe como argumento três inteiros e retorna a soma desses três números. Imprima na tela o resultado.
2. Crie um programa com a função chamada 'positivo' que recebe um número e retorne verdadeiro se ele for positivo e falso se ele é negativo. Imprima o resultado.
3. Faça um programa que descubra se uma palavra é ou não um palíndromo. Um palíndromo é uma sequência de caracteres que, quando lidos do fim para o começo, são idênticos à palavra original. Exemplos: Ana, ovo, osso, ama e arara são palíndromos.
4. Crie um programa que receba uma string com uma frase digitada pelo usuário. Conte quantos espaços em branco existem na string e quantas vezes aparecem cada uma das vogais A, E, I, O e U.
5. Construa um programa que permita ao usuário entrar com uma string e mostre a string de trás para frente e em letras maiúsculas.
6. Faça um programa que receba um nome de arquivo e que leia esse arquivo linha a linha e imprima-o na tela.
7. Utilizando conjuntos (set) em Python, crie um programa que receba 10 números e imprima cada um deles, mas, caso tenha algum repetido, deve ser impresso somente uma vez.
8. Utilizando tuplas em Python, receba um par ordenado x1, y1 e x2, y2. Some os pares e imprima na tela o valor.
9. Faça um programa que receba duas matrizes 2x2 pelo usuário e some-as. Mostre o resultado na tela. Dica: Use o NumPy.
10. Faça um programa que receba uma matriz 3x3 e multiplique-a pelo escalar 5. Mostre o resultado na tela.

6.3.1 Respostas dos exercícios

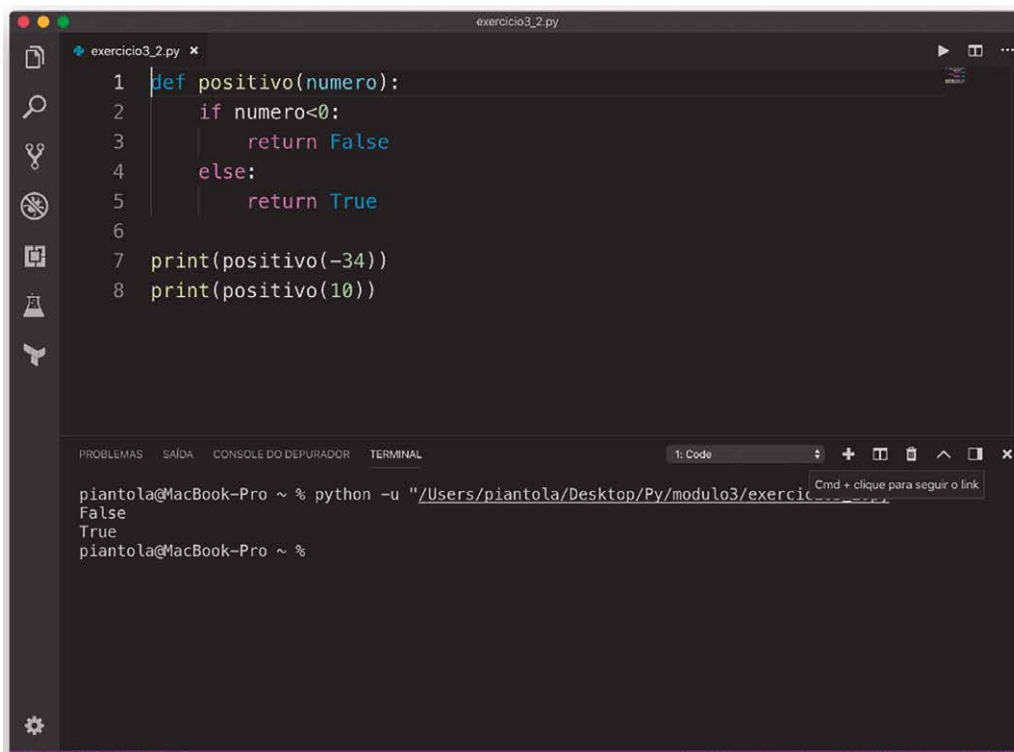


```
exercicio3_1.py
1 def soma3(a, b, c):
2     return a+b+c
3 print(soma3(3,4,5))

PROBLEMAS SAÍDA CONSOLE DO DEPURADOR TERMINAL
1: Code

piantola@MacBook-Pro ~ % python -u "/Users/piantola/Desktop/Py/modulo3/exercicio3_1.py"
12
piantola@MacBook-Pro ~ %
```

Figura 52 – Resolução do exercício 1

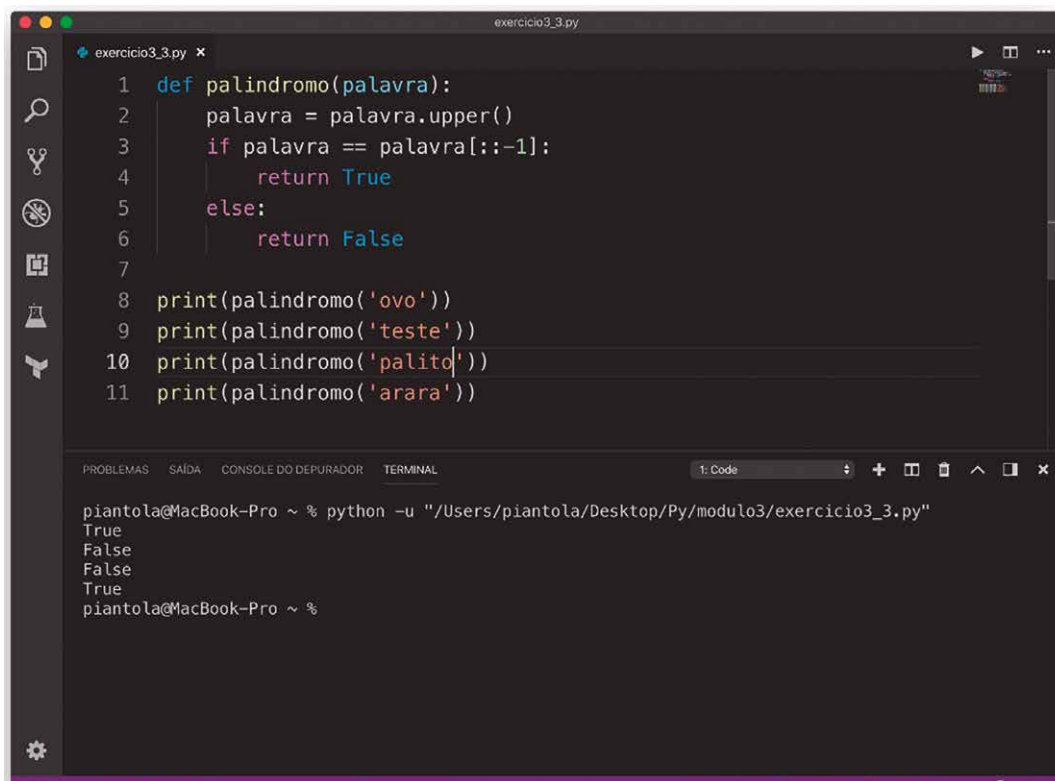


```
exercicio3_2.py
1 def positivo(numero):
2     if numero<0:
3         return False
4     else:
5         return True
6
7 print(positivo(-34))
8 print(positivo(10))

PROBLEMAS SAÍDA CONSOLE DO DEPURADOR TERMINAL
1: Code

piantola@MacBook-Pro ~ % python -u "/Users/piantola/Desktop/Py/modulo3/exercic
False
True
piantola@MacBook-Pro ~ %
```

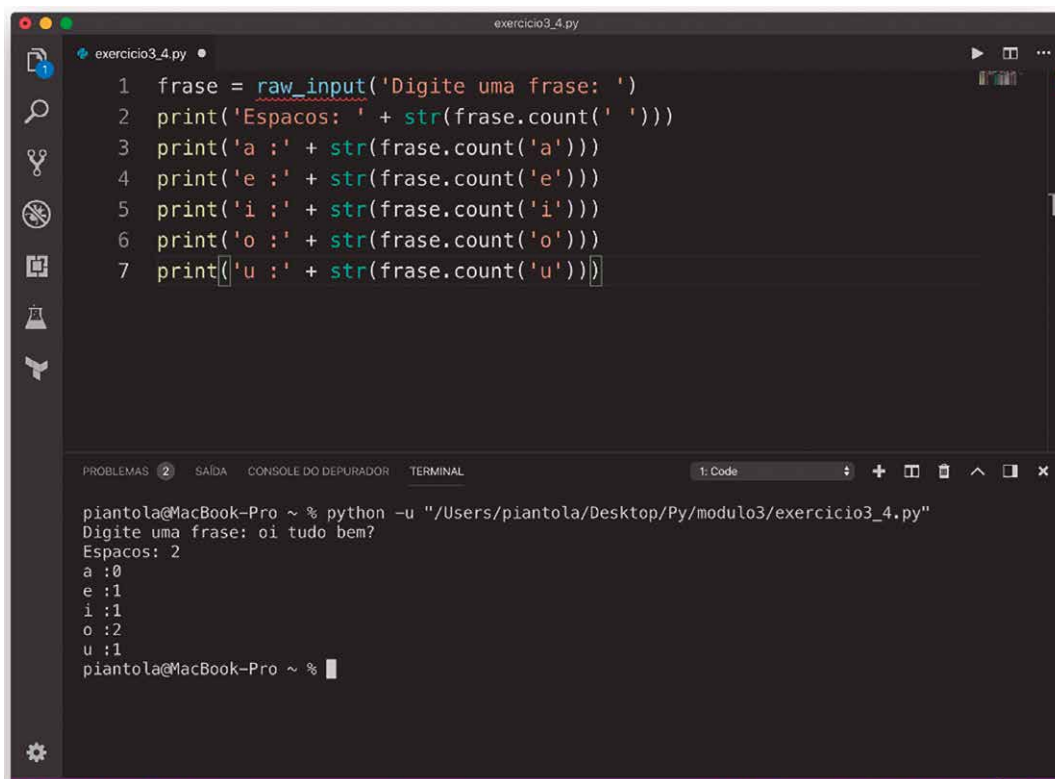
Figura 53 – Resolução do exercício 2



```
exercicio3_3.py
1 def palindromo(palavra):
2     palavra = palavra.upper()
3     if palavra == palavra[::-1]:
4         return True
5     else:
6         return False
7
8 print(palindromo('ovo'))
9 print(palindromo('teste'))
10 print(palindromo('palito'))
11 print(palindromo('arara'))

PROBLEMAS SAÍDA CONSOLE DO DEPURADOR TERMINAL
1: Code
piantola@MacBook-Pro ~ % python -u "/Users/piantola/Desktop/Py/modulo3/exercicio3_3.py"
True
False
False
True
piantola@MacBook-Pro ~ %
```

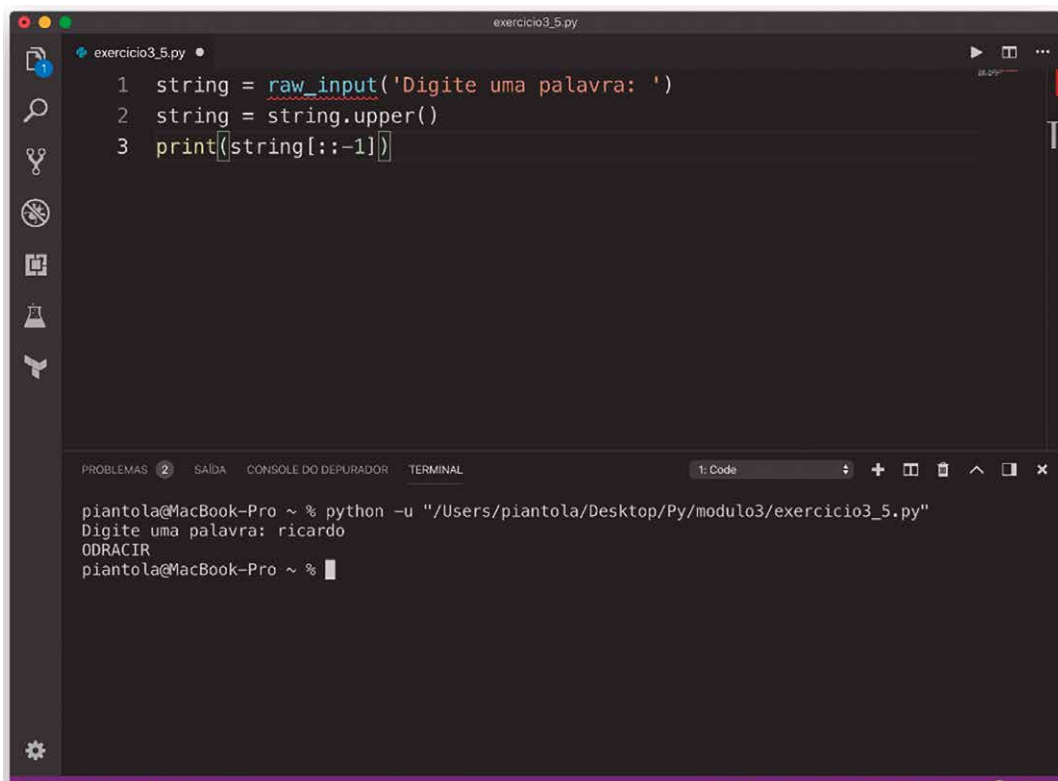
Figura 54 – Resolução do exercício 3



```
exercicio3_4.py
1 frase = raw_input('Digite uma frase: ')
2 print('Espacos: ' + str(frase.count(' ')))
3 print('a : ' + str(frase.count('a')))
4 print('e : ' + str(frase.count('e')))
5 print('i : ' + str(frase.count('i')))
6 print('o : ' + str(frase.count('o')))
7 print('u : ' + str(frase.count('u')))
```

```
PROBLEMAS 2 SAÍDA CONSOLE DO DEPURADOR TERMINAL
1: Code
piantola@MacBook-Pro ~ % python -u "/Users/piantola/Desktop/Py/modulo3/exercicio3_4.py"
Digite uma frase: oi tudo bem?
Espacos: 2
a :0
e :1
i :1
o :2
u :1
piantola@MacBook-Pro ~ %
```

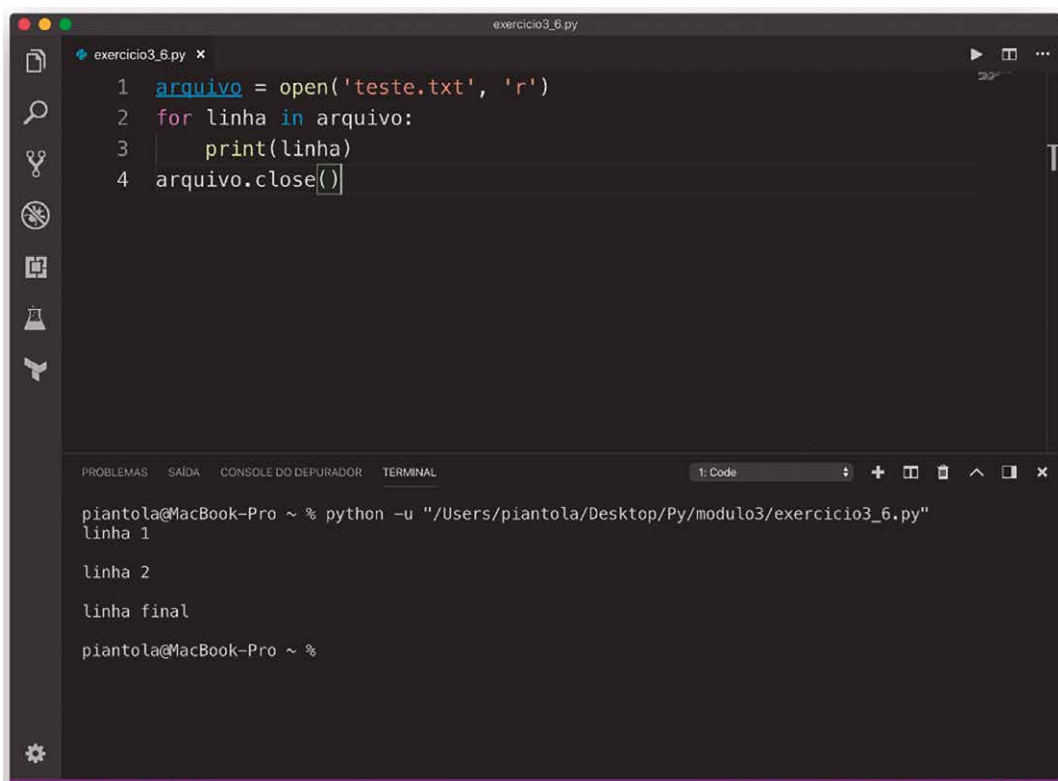
Figura 55 – Resolução do exercício 4



```
exercico3_5.py
1 string = raw_input('Digite uma palavra: ')
2 string = string.upper()
3 print(string[::-1])

PROBLEMAS 2 SAÍDA CONSOLE DO DEPURADOR TERMINAL 1: Code
piantola@MacBook-Pro ~ % python -u "/Users/piantola/Desktop/Py/modulo3/exercico3_5.py"
Digite uma palavra: ricardo
ODRACIR
piantola@MacBook-Pro ~ %
```

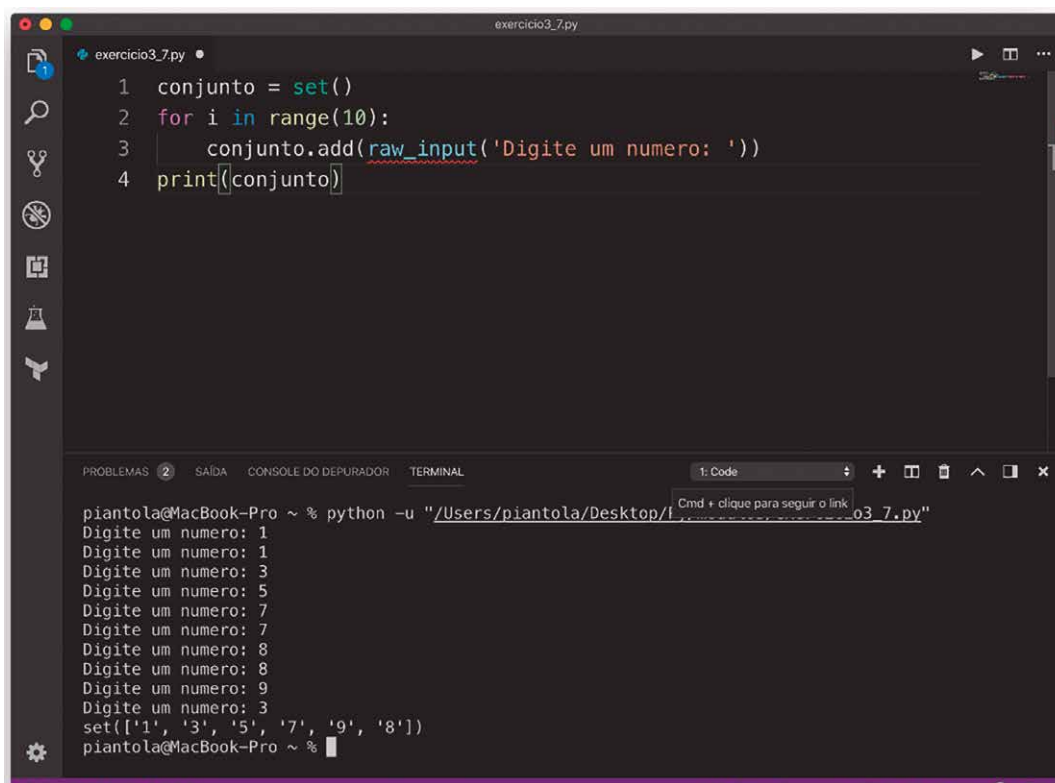
Figura 56 – Resolução do exercício 5



```
exercico3_6.py
1 arquivo = open('teste.txt', 'r')
2 for linha in arquivo:
3     print(linha)
4 arquivo.close()

PROBLEMAS SAÍDA CONSOLE DO DEPURADOR TERMINAL 1: Code
piantola@MacBook-Pro ~ % python -u "/Users/piantola/Desktop/Py/modulo3/exercico3_6.py"
linha 1
linha 2
linha final
piantola@MacBook-Pro ~ %
```

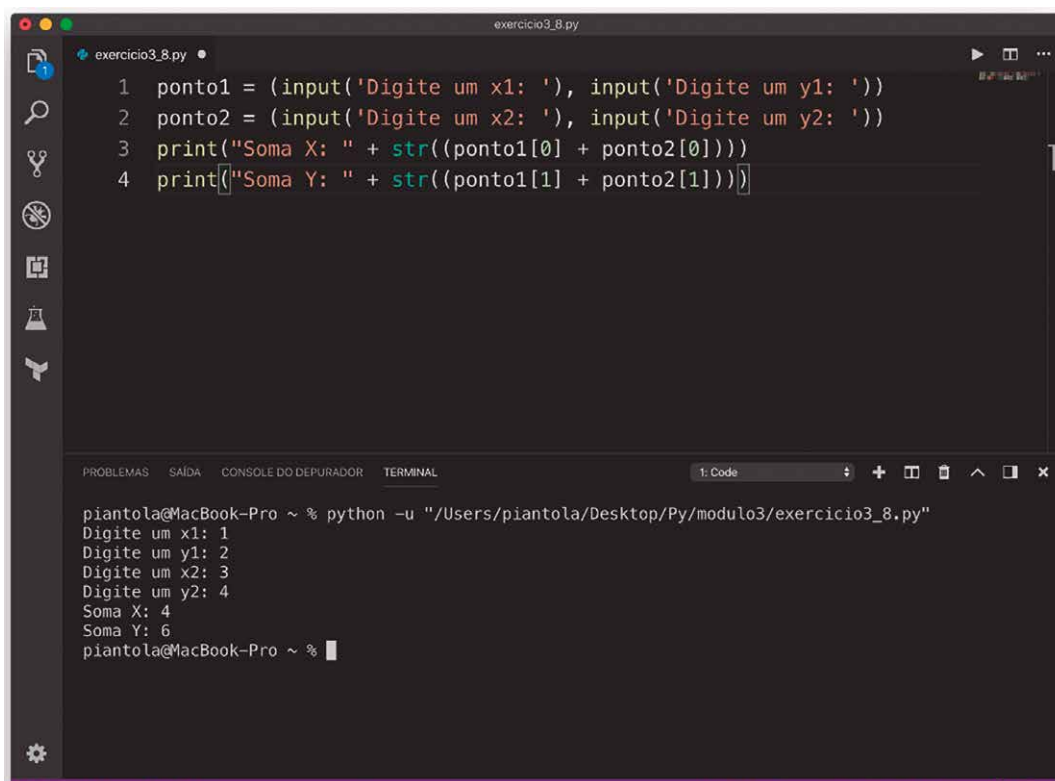
Figura 57 – Resolução do exercício 6



```
exercicio3_7.py
1 conjunto = set()
2 for i in range(10):
3     conjunto.add(raw_input('Digite um numero: '))
4 print(conjunto)

piantola@MacBook-Pro ~ % python -u "/Users/piantola/Desktop/Py/modulo3/exercicio3_7.py"
Digite um numero: 1
Digite um numero: 1
Digite um numero: 3
Digite um numero: 5
Digite um numero: 7
Digite um numero: 7
Digite um numero: 8
Digite um numero: 8
Digite um numero: 9
Digite um numero: 3
set(['1', '3', '5', '7', '9', '8'])
piantola@MacBook-Pro ~ %
```

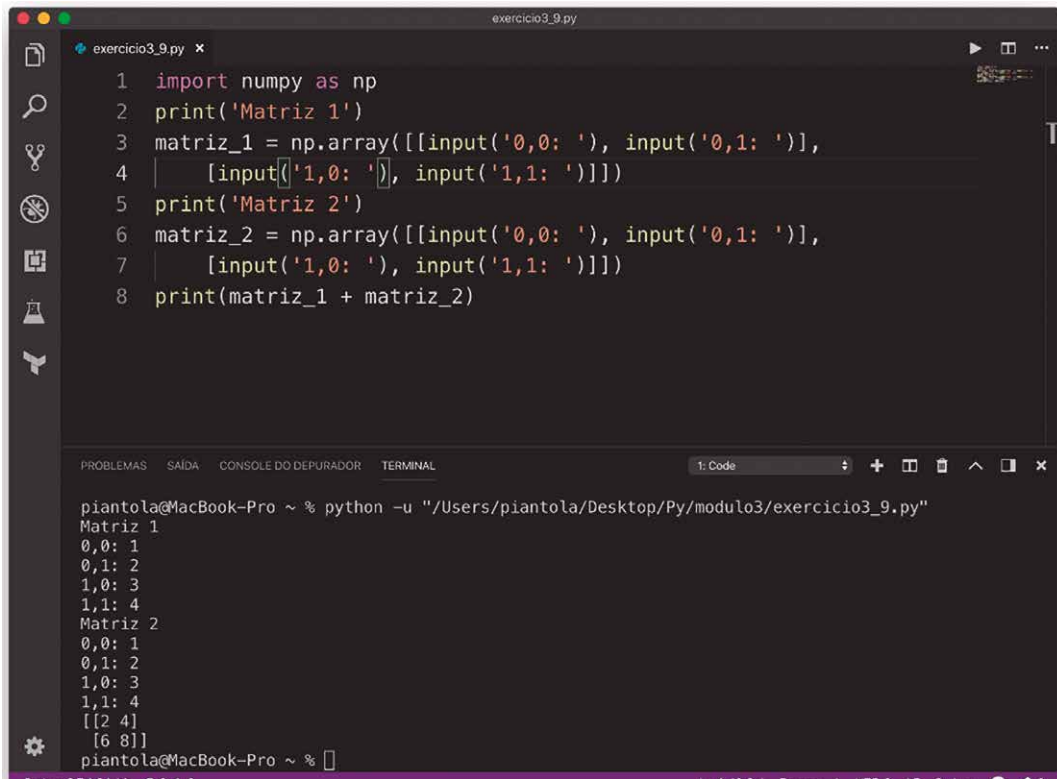
Figura 58 – Resolução do exercício 7



```
exercicio3_8.py
1 ponto1 = (input('Digite um x1: '), input('Digite um y1: '))
2 ponto2 = (input('Digite um x2: '), input('Digite um y2: '))
3 print("Soma X: " + str((ponto1[0] + ponto2[0])))
4 print("Soma Y: " + str((ponto1[1] + ponto2[1])))

piantola@MacBook-Pro ~ % python -u "/Users/piantola/Desktop/Py/modulo3/exercicio3_8.py"
Digite um x1: 1
Digite um y1: 2
Digite um x2: 3
Digite um y2: 4
Soma X: 4
Soma Y: 6
piantola@MacBook-Pro ~ %
```

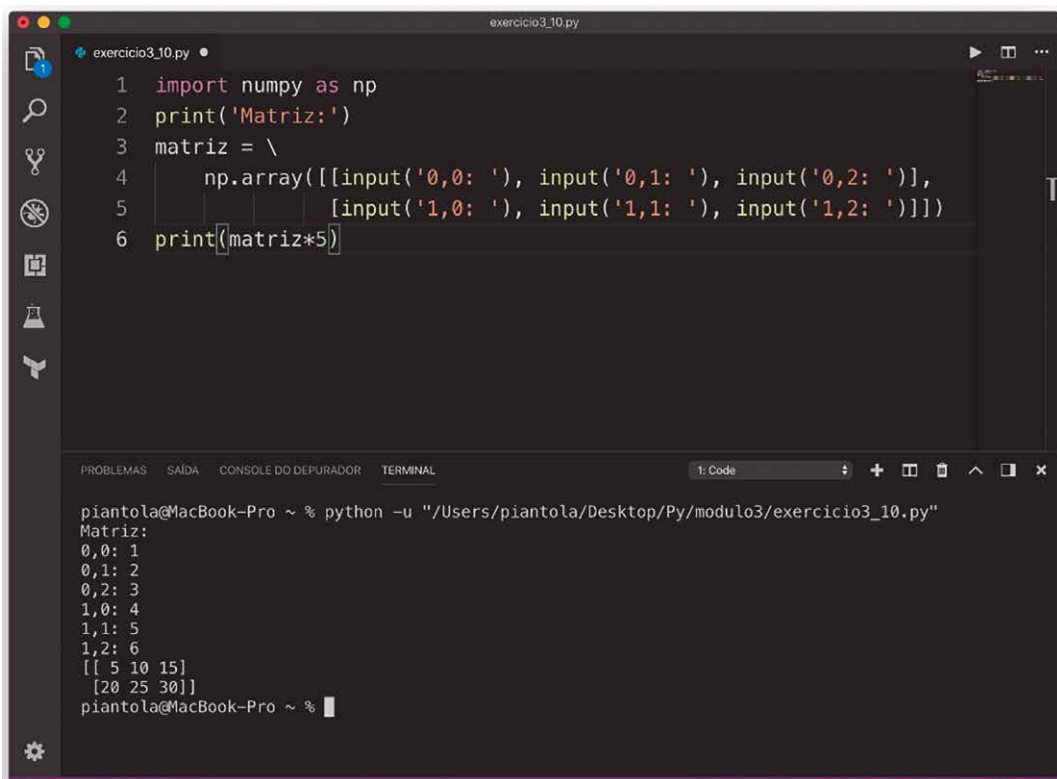
Figura 59 – Resolução do exercício 8



```
exercico3_9.py
1 import numpy as np
2 print('Matriz 1')
3 matriz_1 = np.array([[input('0,0: '), input('0,1: ')],
4                       [input('1,0: '), input('1,1: ')]])
5 print('Matriz 2')
6 matriz_2 = np.array([[input('0,0: '), input('0,1: ')],
7                       [input('1,0: '), input('1,1: ')]])
8 print(matriz_1 + matriz_2)

PROBLEMAS SAÍDA CONSOLE DO DEPURADOR TERMINAL 1: Code
piantola@MacBook-Pro ~ % python -u "/Users/piantola/Desktop/Py/modulo3/exercicio3_9.py"
Matriz 1
0,0: 1
0,1: 2
1,0: 3
1,1: 4
Matriz 2
0,0: 1
0,1: 2
1,0: 3
1,1: 4
[[2 4]
 [6 8]]
piantola@MacBook-Pro ~ %
```

Figura 60 – Resolução do exercício 9



```
exercico3_10.py
1 import numpy as np
2 print('Matriz:')
3 matriz = \
4     np.array([[input('0,0: '), input('0,1: '), input('0,2: ')],
5               [input('1,0: '), input('1,1: '), input('1,2: ')]])
6 print(matriz*5)

PROBLEMAS SAÍDA CONSOLE DO DEPURADOR TERMINAL 1: Code
piantola@MacBook-Pro ~ % python -u "/Users/piantola/Desktop/Py/modulo3/exercicio3_10.py"
Matriz:
0,0: 1
0,1: 2
0,2: 3
1,0: 4
1,1: 5
1,2: 6
[[ 5 10 15]
 [20 25 30]]
piantola@MacBook-Pro ~ %
```

Figura 61 – Resolução do exercício 10



Resumo

Trabalhamos, na unidade III, com funções, strings, arquivos e estruturas de dados. Apesar de não serem tópicos do núcleo da programação estruturada, eles não deixam de ser importantes, pois são o centro da própria programação. Uma função é um bloco de código que somente vai ser executado se for explicitamente chamado. Existem funções embutidas no Python para simplificar a vida do programador, como `print()`, `input()` e `range()`. É possível construir novas funções, que são responsáveis por organizar o código e podem ser reutilizadas em outros programas. Informações podem ser passadas para as funções através de parâmetros ou argumentos. É possível definir argumentos padrão e, nesse caso, se não for passado um valor para esse argumento, o valor padrão é utilizado.

O escopo de uma variável é a região em que ela é acessível. Variáveis criadas dentro de uma função não podem ser acessadas fora dela. Nesse caso, são chamadas de variáveis locais. Variáveis criadas no corpo principal do programa são acessadas globalmente, ou seja, em todo o programa, e são chamadas de variáveis globais.

Como visto, as strings são variáveis que guardam texto. Podem ser manipuladas como um vetor de caracteres e existem inúmeras funções em Python construídas especialmente para se trabalhar textos. Alguns exemplos são: transformar o texto em maiúsculo ou minúsculo, encontrar palavras dentro das strings, remover espaços ou outros caracteres, substituir texto e mais uma infinidade de facilidades.

Além das variáveis que guardam os dados voláteis na memória RAM, existem os arquivos que guardam dados de forma persistente nos discos. Então, é importante aprender a se trabalhar com arquivos. Os arquivos podem ser abertos somente para leitura, para gravação ou para leitura e gravação. A manipulação de arquivos pode ser feita em modo texto ou binário, e o Python tem funções como `open()` e `close()` para essa tarefa.

As estruturas de dados facilitam bastante o processamento e diminuem a quantidade de código que precisa ser escrito. Elas implementam comportamentos e ideias que usamos na matemática e padrões do mundo real. Exemplos: pilhas, filas, listas, conjuntos, matrizes etc.

As bibliotecas padrão do Python não incluem a estrutura de dados matrizes. É possível simular matrizes, utilizando listas de listas, mas não é o ideal, pois todo o comportamento das operações de matrizes teria que ser programado do zero também. Existe uma alternativa para isso, a

biblioteca NumPy, gratuita e largamente utilizada pela comunidade, que trabalha com matrizes e álgebra linear.

Listas e dicionários são as estruturas de dados mais utilizadas em Python. Além delas, temos as tuplas e conjuntos. A lista é uma coleção de elementos ordenada e mutável, já uma tupla é uma coleção ordenada e imutável. São utilizadas para guardar itens em uma variável simples. Já os conjuntos não são ordenados, nem indexados e são mutáveis. Todas as estruturas de dados têm características diferentes para manipulação de informação. É importante usar a estrutura adequada para a resolução do problema computacional.



Exercícios

Questão 1. O código Python a seguir implementa uma função para o cálculo da área de um triângulo e imprime o seu resultado:

```
def area_triangulo(base, alt):  
    return (base * alt)/2  
  
print('A área vale', area_triangulo(3, 4))
```

Nesse contexto, avalie as afirmativas:

- I – Ao ser executado, o programa imprime: A área vale 7.
- II – Foram definidos três parâmetros na função `area_triangulo`.

III – Os parâmetros da função `area_triangulo` não possuem valor default e, assim, é obrigatório passar esses valores como argumentos.

É correto o que se afirma em:

- A) I, apenas.
- B) II, apenas.
- C) III, apenas.
- D) II e III, apenas.
- E) I, II e III.

Resposta correta: alternativa C.

Análise das afirmativas

I – Afirmativa incorreta.

Justificativa: ao executar o código, será impresso: A área vale 6. A função `area_triangulo` recebe os valores 3 e 4 como argumentos e realiza a seguinte operação: $(3 \times 4)/2 = 6$. A função retorna o resultado 6 na posição em que foi chamada, dentro do comando `print()`.

II – Afirmativa incorreta.

Justificativa: foram definidos dois parâmetros na função `area_triangulo`. São eles: `base` e `alt`. Esses parâmetros são os nomes dados a atributos que a função deve receber ao ser chamada.

III – Afirmativa correta.

Justificativa: no código Python, vemos a linha: `def area_triangulo(base, alt):`. Nessa linha, foi definida a função denominada `area_triangulo`, cujos parâmetros são `base` e `alt`. Não observamos a inserção de nenhum valor padrão (default) para tais parâmetros. Logo, `base` e `alt` precisam necessariamente receber argumentos. Esses argumentos nada mais são do que os valores que os parâmetros da função devem assumir em determinado contexto. No caso, temos os argumentos 3 e 4 sendo passados aos parâmetros `base` e `alt`, respectivamente. Em alguns casos, é possível definir funções com valores padrão para seus parâmetros. Observe:

```
def area_triangulo(base=1, alt=1):  
    return (base * alt)/2  
  
print('A área vale', area_triangulo())
```

No caso, a função foi chamada sem a passagem de nenhum argumento, por meio do comando `area_triangulo()`. Contudo, note que a função foi definida de forma a adotar valores padrões para seus parâmetros. Na falta de argumentos, tanto `base` quanto `alt` assumem valor 1. Logo, a execução do código resulta em: A área vale 0.5.

Questão 2. Analise o código-fonte a seguir, no qual se utiliza uma conhecida estrutura de dados existente na linguagem Python:

```
fisicos = {'Newton', 'Hawking', 'Einstein'}  
matematicos = {'Fibonacci', 'Euclides', 'Newton'}  
  
c = fisicos.intersection(matematicos)  
d = fisicos.union(matematicos)  
  
print(c, e, d)
```

Considerando esse contexto, avalie as afirmativas:

- I – A estrutura de dados utilizada neste código é a tupla.
- II – O método intersection() retorna a interseção entre dois conjuntos.
- III – O conjunto fisicos é composto por três elementos, sendo que todos eles são do tipo string.
- IV – Ao ser executado, o código resulta na seguinte impressão: {'Newton'} e {'Euclides', 'Hawking', 'Einstein', 'Newton', 'Fibonacci'}

É correto o que se afirma em:

- A) I, apenas.
- B) II, apenas.
- C) III e IV, apenas.
- D) II, III e IV, apenas.
- E) I, II, III e IV.

Resposta correta: alternativa D.

Análise das afirmativas

- I – Afirmativa incorreta.

Justificativa: a estrutura de dados que se destaca nesse código-fonte é a set (que significa conjunto, em inglês). No Python, um conjunto é uma coleção desordenada de elementos, sendo que esses elementos não podem ser repetidos. Nas duas primeiras linhas do código, vemos a declaração do conjunto fisicos e do conjunto matematicos. Sabemos que se trata de conjuntos porque, logo após o sinal de igualdade, vemos elementos listados entre chaves, separados por vírgulas. Note que cada um dos dois conjuntos declarados possui três elementos. Para que tivéssemos a declaração de uma tupla (do termo em inglês tuple), deveríamos listar elementos entre parênteses. Vemos a sintaxe da declaração dessas estruturas a seguir:

```
nome_do_set = {elemento1, elemento2, elemento3}  
nome_da_tupla = (elemento1, elemento2, elemento3)
```

II – Afirmativa correta.

Justificativa: o método `intersection()` retorna um conjunto ao qual pertencem os elementos de interseção entre outros dois conjuntos. Dessa forma, na linha

```
c = fisicos.intersection(matematicos)
```

temos que o conjunto `c` é formado pelos elementos de interseção entre os conjuntos `fisicos` e `matematicos`. Logo, apenas o elemento `'Newton'` pertence a `c`, pois é o único elemento comum entre os conjuntos `fisicos` e `matematicos`.

III – Afirmativa correta.

Justificativa: o conjunto `fisicos` é composto por três elementos, que estão entre chaves separados por vírgula. Como há aspas simples abrangendo cada um dos elementos, entendemos que se trata de strings.

IV – Afirmativa correta.

Justificativa: a execução do código causará uma impressão em tela. Essa impressão exibirá um texto cujo conteúdo obedecerá à seguinte ordem:

- o valor (conteúdo) do conjunto `c`;
- o caractere `'e'`;
- o valor (conteúdo) do conjunto `d`.

Já foi discutido que apenas `'Newton'` pertence a `c`. Em seguida, será exibido o caractere `'e'`. Por último, será exibido o conteúdo do conjunto `d`, que é formado pela união entre os conjuntos `fisicos` e `matematicos`. Assim, todos os elementos pertencentes a cada um desses dois conjuntos será também um elemento de `d`. Lembre-se de que um conjunto é uma coleção desordenada de elementos e, portanto, não nos preocuparemos com a ordem de exibição dos elementos de `d`. Também sabemos que conjuntos não admitem repetições de elementos e, com isso, o elemento `'Newton'`, que aparece tanto em `fisicos` quanto em `matematicos`, aparecerá apenas uma vez em `d`. Portanto, ao executar o código, temos a seguinte impressão em tela:

```
{'Newton'} e {'Euclides', 'Hawking', 'Einstein', 'Newton', 'Fibonacci'}
```
