

# **Introdução à Python para Economistas**

**Charles Massami Kumahara**

## Sumário

<b>Introdução .....</b>	<b>1</b>
<b>Capítulo 1 Variáveis.....</b>	<b>8</b>
<b>Capítulo 2 Algumas noções sobre funções e métodos em Python .....</b>	<b>10</b>
<b>Capítulo 3 Tipos básicos de dados .....</b>	<b>12</b>
<b>Capítulo 4 Tipos Básicos de estrutura de dados (Listas, Tuplas, Dicionários e Conjuntos).....</b>	<b>18</b>
<b>Capítulo 5 Operações matemáticas básicas .....</b>	<b>34</b>
<b>Capítulo 6 Operadores relacionais e lógicos.....</b>	<b>41</b>
<b>Capítulo 7 Estruturas condicionais de fluxo (if).....</b>	<b>47</b>
<b>Capítulo 8 Loops (estruturas de repetição do tipo “for ___ in ___” e “while”) .....</b>	<b>53</b>
<b>Capítulo 9 Importação de bibliotecas .....</b>	<b>62</b>
<b>Capítulo 10 Introdução à elaboração de gráficos em Python .....</b>	<b>67</b>
<b>Posfácio .....</b>	<b>78</b>
<b>Bibliografia e cursos consultados .....</b>	<b>79</b>

## Introdução

### I.1 Breves considerações sobre o livro

Este é um livro de introdução à Python para economistas. Queremos dizer com isto que buscamos apresentar o básico desta linguagem ao mesmo tempo que motivamos o seu uso com assuntos ligados à economia.

Escrevemos esse texto na versão 3.7 de Python. Como essa linguagem é continuamente desenvolvida, é possível que alguns comandos não funcionem corretamente no momento em que o leitor estiver executando-os. Por se tratar de um texto básico, esperamos que isto raramente ocorra (ou mesmo que isto não ocorra).

Escolhemos a organização do texto de maneira que possa servir de fonte de consultas. Após o domínio dos assuntos aqui tratados, a sequência natural deve ser o estudo das bibliotecas de interesse do leitor (veremos isso em “**I.5 Bibliotecas e o estudo de Python**”) em paralelo à outros cursos e textos introdutórios<sup>1</sup>. Uma advertência: não se deve esperar conhecer toda a linguagem Python para depois buscar aplicações. É importante que se tente usos tão logo quanto possível.

### I.2 O que é um programa?

Um programa é simplesmente um “arquivo-texto, escrito em um formato especial (linguagem)” (MENEZES, 2014, p.36). De forma complementar à definição anterior, podemos dizer que um programa é um conjunto de códigos escritos de acordo com regras específicas que o computador interpreta e executa.

Neste livro, quando nos referirmos a um programa, o código a ser digitado será apresentado após o símbolo “[in]:”. Já o resultado que esperamos será apresentado após “[out]:”. Por exemplo:

#### Exemplo I.1

```
[in]: print(5 + 5)
```

```
[out]: 10
```

---

<sup>1</sup> Após o estudo deste livro, recomendamos o início de estudos mais aprofundado de Python da seguinte forma (nesta ordem): 1) Curso “Python – Aprenda os fundamentos de Rafael Santos no site [www.udemy.com](http://www.udemy.com); 2) Livro: Curso Intensivo de Python: Uma introdução prática e baseada em projetos à programação, de MATTHES, Eric (2016), 3) Livro: Automatize Tarefas Maçantes com Python, de SWEIGART, Al (2015).

No programa acima, devemos digitar “`print(5 + 5)`” e esperamos que o programa gere como resultado o número 10.

Um outro exemplo de um programa escrito em Python poderia ser:

### Exemplo I.2

```
[in]: print('Olá, bom dia')
```

```
[out]: Olá, bom dia
```

Neste exemplo, devemos digitar o código “`print('Olá, bom dia')`”. Após a execução do programa, o computador deverá exibir como saída a mensagem “Olá, bom dia”.

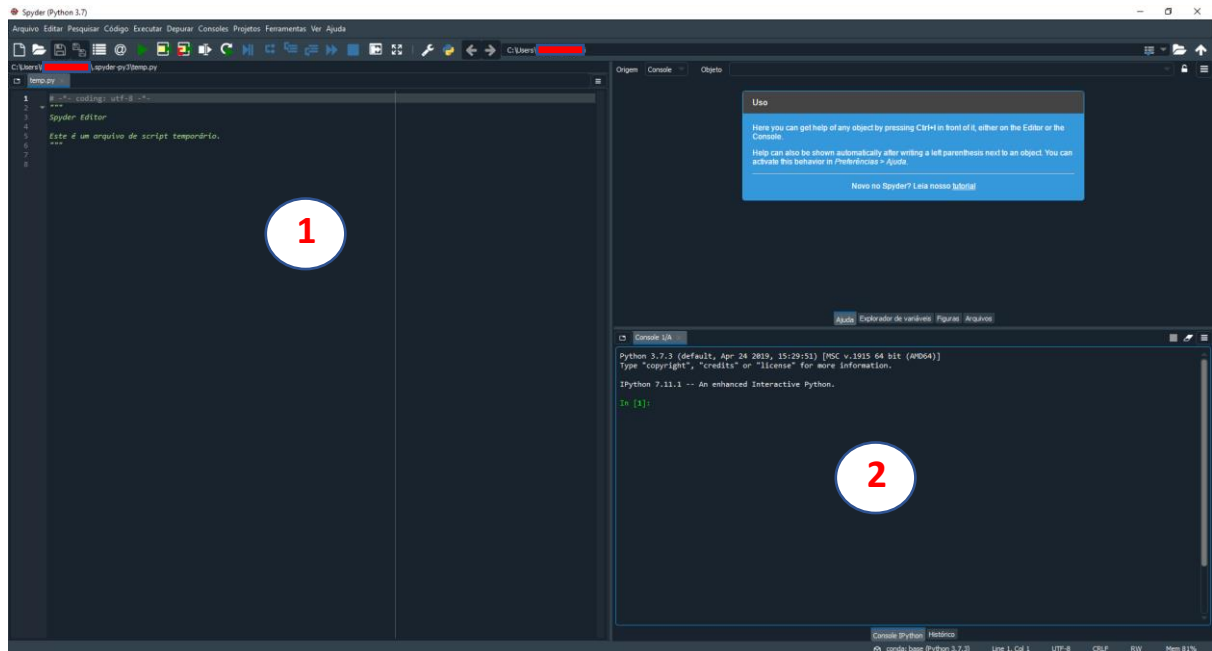
Estes são exemplos de programas extremamente simples escritos na linguagem Python. Ao estudar programação, poderemos ver que é possível desenvolver programas com diferentes objetivos e complexidades.

### I.3 Instalação de Python

Há diferentes maneiras de se instalar Python, de acordo com o sistema operacional. Uma forma simples de instalação é via pacote Anaconda. Basta acessar <https://www.anaconda.com/> → Download → Escolher a versão.

Após o download, será possível acessar o “Anaconda Navigator”, que apresenta diferentes aplicativos. Utilizaremos Spyder para desenvolver nossos exemplos neste livro. Após abrir o Spyder, veremos a seguinte tela:

Figura I.1: Tela do Spyder



Em “1”, iremos escrever nossos programas. Em “2”, obteremos a “saída” (resultado) do programa.

Vamos escrever nosso primeiro programa utilizando o exemplo abaixo:

### Exemplo I.3

[in]: `print(5 + 6)`

[out]: 11

Para reproduzir o exemplo acima, devemos seguir os seguintes passos:

- 1) Abrir o Spyder
- 2) Digitar o seguinte código na tela 1 da Figura I.1: `print(5 + 6)`
- 3) Clicar no botão F5 do teclado ou, alternativamente, localizar o menu “Executar” e clicar em “Executar”.

Caso tenha seguido os passos 1 a 3 corretamente, Python deverá retornar na tela 2 o resultado da soma  $5 + 6$ , ou seja, 11. Para salvar o programa, devemos escolher uma pasta e, ao nomear o arquivo, não esquecer de incluir a extensão “.py” (por exemplo, teste.py).

Spyder é um exemplo de um aplicativo em que podemos escrever programas na linguagem Python. Os ambientes em que podemos escrever os nossos programas são chamados

de IDE's (Integrated Development Environment). Outros exemplos de IDE's para escrever em Python são o Jupyter Notebook (também instalado com o Anaconda) e o Google Colaboratory, disponível no google drive (necessário instalação). Quem optar por utilizar um destes dois, deve nomear seu arquivos incluindo a extensão “.ipynb”.

#### **I.4 Algumas características básicas da linguagem Python**

Ao escrever um programa em Python, devemos observar algumas regras básicas:

- 1) Ao rodar um programa, o computador lê e executa o programa linha a linha. Ou seja, o computador executa a primeira linha, passa para a segunda, executa-a, e assim por diante, até chegar à última linha.
- 2) A linguagem Python diferencia letras maiúsculas e minúsculas. Assim, um código que deve ser escrito com letra minúscula mas escrita com maiúscula não funcionará (e vice-versa).
- 3) O alinhamento no início de cada linha do código é importante. Espaços em branco no início da linha é chamado “indentação” e indicam para Python a existência de um bloco de códigos dentro do programa. Geralmente, para padronização, recomenda-se quatro “espaços” ou um “tab” para marcar uma indentação.
- 4) Em Python, tudo é um objeto e cada objeto possui métodos associados, dependendo do tipo de dado do objeto (isso ficará mais claro à medida que nos familiarizamos com a linguagem).
- 5) É possível escrever comentários ao longo de um programa. Para isso, temos duas opções: usar cerquilha (#) ou aspas triplas (’’’). Numa determinada linha, tudo o que estiver escrito após # não será levado em conta por Python. Quando queremos escrever um comentário de diversas linhas, devemos escrever entre aspas triplas. Veja os exemplos:

##### **Exemplo I.4**

```
[in]: print(5 + 5)  # operação de adição de cinco com cinco
[out]: 10
```

##### **Exemplo I.5**

```
[in]: '''
    Este programa realiza uma operação de soma
    '''
[in]: print( 5 + 5)
```

[out]: 10

No exemplo I.4, o que está escrito após # não é considerado. No exemplo I.5, as linhas escritas entre as aspas triplas também são ignoradas quando Python estiver executando o programa.

6) Em algumas IDE's, como em Spyder, sempre será necessário utilizar a função print() para que o computador informe algo como saída de um programa. Por exemplo:

Exemplo I.5

[in]: 5 + 5

[out]:

No programa acima, quando executado na IDE Spyder, o computador não “retorna” nenhum valor como saída. É necessário a função print( ), como no exemplo I.6 abaixo:

Exemplo I.6

[in]: print( 5 + 5)

[out]: 10

No entanto, caso estivermos usando a IDE Jupyter Notebook, o programa acima não exige que a função print() seja utilizada para que o computador exiba o resultado desta operação (em outros casos, porém, a utilização da função print( ) é necessária nesta IDE). Isto não é motivo de preocupação pois é algo que nos habituaremos facilmente. Apenas comentamos essa peculiaridade pois é a primeira coisa que podemos notar quando estamos acostumados a utilizar o Spyder e, eventualmente, passamos para o Jupyter Notebook.

## **I.5 Bibliotecas e o estudo de Python**

De acordo com os interesses de cada um, é necessário importar em nossos programas bibliotecas que contenham funções e códigos desenvolvidos para aplicações específicas. Após o domínio do básico em Python, o próximo passo natural seria estudar as bibliotecas criadas

para os fins que desejamos. Abaixo, podemos ver alguns exemplos<sup>2</sup> de bibliotecas e suas utilidades.

→ Bibliotecas básicas para Análise de Dados:

- Pandas
- Numpy

→ Bibliotecas para geração de gráficos:

- Matplotlib
- Seaborn
- Plotly
- Bokeh
- Altair
- ggplot
- Vpython
- networkx

→ Bibliotecas para matemática simbólica (álgebra, funções, cálculo, etc.):

- SymPy
- Theano
- CVXPY

→ Biblioteca para Estatística:

- Statsmodels
- Scikit-Learn
- PyMC (análise bayesiana)
- Pystan (análise bayesiana)
- Keras
- ScyPy

---

<sup>2</sup> Baseado principalmente no curso Python – Aprenda os fundamentos, de Rafael Santos, aula “Principais bibliotecas”, disponibilizado em [www.udemy.com](http://www.udemy.com)



→ Biblioteca com algumas ferramentas para economia:

- Quantecon

→ Outras bibliotecas importantes:

- Pytables
- Tensor Flow

Para utilizar uma biblioteca em um determinado programa, são necessários alguns códigos específicos para sua importação. Veremos como isso é feito no capítulo 9 “Importação de bibliotecas”.

Apenas como guia para estudos futuros, a lista abaixo apresenta as bibliotecas Python essenciais, segundo o livro “Python para Análise de Dados”, de MCKINNEY, Wes (2018):

- NumPy
- pandas
- matplotlib
- ScyPy
- scikit-learn
- statsmodels

## Capítulo 1 Variáveis

As variáveis exercem papel fundamental em Python e são utilizadas em praticamente todos os programas. Por este motivo, abordaremos este importante assunto para iniciarmos.

Primeiramente, esclarecemos que não nos preocuparemos em dar uma definição precisa de variável. Apenas nos limitaremos a dizer que as variáveis em Python exercem papel análogo às variáveis utilizadas em matemática, com pelo menos uma diferença importante: em Python é possível atribuir textos à elas.

Assim como na matemática, utilizamos o sinal “=” de forma a atribuir um valor à uma variável. Vejamos:

### Exemplo 1.1

```
[in]: a = 5      # L1: atribuímos à variável “a” o número 5
      print(a)  # L2
[out]: 5
```

Em L1, atribuímos à variável “a” o número 5. Em L2, passamos a variável “a” como argumento da função print() para que Python apresente como saída o valor atribuído à variável “a”.

As variáveis em Python são “*case sensitive*”, ou seja, Python diferencia variáveis em que o nome apresenta letras maiúsculas e minúsculas (por exemplo, uma variável, digamos, “x”, seria diferente de uma outra “X”). Além disso, há regras específicas para darmos nomes às variáveis. De forma a sermos o mais direto possível não iremos listar essas regras, apenas notando que se violarmos uma delas Python simplesmente apresentará um erro quando executarmos o código. De qualquer forma, essas regras podem ser encontradas em qualquer texto introdutório específico da linguagem Python ou em uma simples pesquisa na internet.

Python nos permite, com apenas uma linha, gerar diversas variáveis e atribuir valores à elas. Vejamos abaixo:

### Exemplo 1.2

```
[in]: x, y, z = 10, 60, 2      # com isso, teremos: x = 10, y = 60, z = 2
      print(x)
      print(y)
```

```
print(z)
[out]: 10
      60
      2
```

Na primeira linha do programa acima, Python atribui a cada variável `x`, `y`, `z` os respectivos valores: 10, 60 e 2.

Agora voltemos ao exemplo 1.1 para um breve comentário. Nele, como explicamos, atribuímos à variável “`a`” o número 5. Veremos posteriormente que essa variável passa a ser do tipo inteiro (ou *integer*). Também podemos atribuir às variáveis outros tipos e estruturas de dados, como *strings*, *floats*, listas, tuplas, Data Frame, dentre outros possíveis em Python. Os tipos básicos de dados e de estruturas de dados serão discutidos adiante após abordarmos de forma introdutória no próximo capítulo um outro elemento básico da linguagem Python: as funções.

## Capítulo 2 Algumas noções sobre funções e métodos em Python

### 2.1 Uma noção sobre funções

A linguagem Python possui várias funções que podemos utilizar em nossos programas. Dentre elas, existem algumas que são chamadas de funções nativas (também chamadas de funções *built-in*). As funções nativas (*built-in*) são aquelas que podem ser utilizadas diretamente, sem a necessidade de importação de qualquer biblioteca ou módulo. Por exemplo, a função “`print( )`” utilizada em exemplos anteriores é nativa de Python e está disponível sem a necessidade de importação de nenhuma biblioteca. Algumas outras funções necessitam da importação de bibliotecas para nosso programa. Iremos ver como importar bibliotecas para uso de suas funções posteriormente.

### 2.2 Noções sobre métodos

Os métodos são comandos que exercem algum tipo de ação sobre objetos. Para ilustrar como utilizamos um método, vamos lançar mão de dois exemplos:

#### Exemplo 2.1

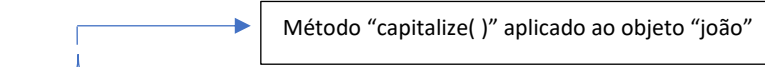
```
[in]: print('joão')  
[out]: joão
```

Neste primeiro exemplo, passamos o argumento “joão” para a função `print()`. Como resultado, Python retorna como saída “joão”.

No segundo exemplo, vamos utilizar um método para que Python retorne como resultado “João”, com a primeira letra maiúscula. Vejamos:

#### Exemplo 2.2:

```
[in]: print('joão'.capitalize( ))  
[out]: João
```



Método “`capitalize( )`” aplicado ao objeto “joão”

Neste exemplo, ‘joão’ é um objeto e “`capitalize( )`” é um método. O método `capitalize( )` faz com que Python transforme a primeira letra do objeto “joão” em letra maiúscula. Veja que a saída apresenta como resultado João (com a primeira letra maiúscula).

Para aplicar um determinado método, utilizamos a estrutura “objeto.método()”, sem as aspas. Mas não vamos nos aprofundar neste assunto (sobre métodos) pois o objetivo de tocarmos neste tópico foi o de apenas dizer que cada tipo de dados e estruturas de dados possui métodos próprios. Veremos adiante que ‘joão’ é do tipo *string*. Um dado do tipo *string* possui métodos que podemos aplicar a ele, assim como outros tipos e estruturas de dados possuem métodos “próprios”. Desta forma, é sempre importante saber com que tipo ou estrutura de dados estamos trabalhando, pois um método que pode ser aplicado a determinado tipo ou estrutura pode não ser aplicável a um outro. Por isso, a seguir, iremos estudar os tipos e estruturas de dados mais básicos que encontraremos em Python.

## Capítulo 3 Tipos básicos de dados

É importante saber distinguir os diferentes tipos de dados. Eles determinam quais operações e “métodos” podem ser aplicados, bem como os resultados dos operadores que utilizamos. Por exemplo, veremos que o operador `+`, quando aplicado aos inteiros e aos *floats*, funciona como o operador matemático da adição. No entanto, quando aplicado à *strings*, o operador `+` funciona como um “concatenador” de texto.

É possível saber o tipo de dado que estamos utilizando pela aplicação da função *builtin*<sup>3</sup> `type()`. Vejamos o exemplo abaixo:

### Exemplo 3.1

```
[in]: b = 5.0
      type( b )
[out]: float
```

No código acima, passamos a variável “b” como argumento da função “`type()`”. Com isso, Python nos informa que “b” é do tipo *float*. Este é um dos tipos básicos de dados. Como era de se esperar, existem outros. Neste capítulo, veremos os seguintes:

- Strings
- Inteiros
- Floats
- Booleanos

## 3.1 Strings

### 3.1.1 Ideias Gerais sobre as Strings

Dito de uma forma simples, uma *string* é um tipo de dado interpretado por Python como sendo um texto<sup>4</sup>. Utilizamos aspas simples ou duplas para declarar uma *string*. Vamos ver um exemplo:

---

<sup>3</sup> Ou seja, função nativa de Python.

<sup>4</sup> Ou, um pouco mais formalmente, uma string é uma sequência ordenada de caracteres.

### Exemplo 3.2

```
[in]: t = "Bom dia!" # L1
      type( t )      # L2
[out]: str
```

No código acima, em L1 geramos a variável *t* e atribuímos à ela uma *string* (podemos ver isso pelas aspas duplas). Agora, para Python, *t* é do tipo *string* e, sendo assim, é interpretado como um texto. Em L2 utilizamos a função nativa `type( )` para saber o tipo de dado de *t* e, assim, Python imprime como resultado `str` (abreviação para *string*).

#### 3.1.2 Inserindo o valor de uma variável dentro de uma *string*

Podemos acrescentar variáveis dentro de *strings* e fazer com que o valor atribuído à esta variável apareça como parte do texto. Para isto, utilizamos a seguinte estrutura<sup>5</sup>:

```
f"texto_qualquer {nome_variável}"
```

em que (i) devemos acrescentar *f* antes da abertura das aspas, (ii) *texto\_qualquer* é um texto de escolha livre, (iii) em *nome\_variável* devemos colocar o nome da variável que queremos que faça parte da *string* e (iv) *{nome\_variável}* pode ser incluído em qualquer lugar do texto. Vejamos um exemplo simples:

### Exemplo 3.3

```
[in]: name = Aline
      print(f"Bom dia, {name}!")
[out]: Bom dia, Aline!
```

Note que colocamos a letra *f* antes de abrir aspas e o nome da variável entre chaves. Assim, dentro da *string*, Python substitui *{name}* pelo valor atribuído à variável *name*.

Também poderíamos colocar *{name}* no início do texto (ou em qualquer parte que escolhermos):

---

<sup>5</sup> Esta estrutura chamada de *f-string* está presente apenas a partir da versão 3.6 do Python.

#### Exemplo 3.4

```
[in]: name = Aline
      print(f'{name}, bom dia!')
```

[out]: Aline, bom dia!

### 3.1.3 Alguns métodos relacionados às *strings*

Vamos ver dois exemplos para ilustrar alguns métodos associados às *strings*:

#### Exemplo 3.5: O método `title()`

```
[in]: frase = "amanhã vai ser outro dia"
      print(frase.title())
```

[out]: Amanhã Vai Ser Outro Dia

Ao aplicar o método `title()` à uma *string*, cada palavra que a forma passa a ter a primeira letra maiúscula.

#### Exemplo 3.6: O método `upper()`

```
[in]: frase = "amanhã vai ser outro dia"
      print(frase.upper())
```

[out]: AMANHÃ VAI SER OUTRO DIA

Ao aplicar o método `upper()` à uma *string*, todas as suas letras passam a ser maiúsculas.

## 3.2 Inteiros

Os Inteiros e o *Float* (*Float* será o próximo tipo básico de dados a ser visto) são tipos básicos de dados que se referem a valores numéricos. A diferença entre eles é que os Inteiros são números sem ponto decimal, enquanto os *Floats* são números com ponto decimal. Tratando-se de números, podemos realizar diversas operações matemáticas utilizando ambos. Essas operações serão abordadas apenas posteriormente.

Vejamos um exemplo de uma variável em que é atribuído um Inteiro:



### Exemplo 3.7

```
[in]: n = 4
      type(n) #L1
[out]: int
```

Neste exemplo, foi atribuído à variável `n` o número 4. Note que não colocamos ponto decimal, dessa forma Python entende “n” como Inteiro. Por isso, quando pedimos para Python verificar o tipo de `n` (linha L1), a resposta é “int”, ou seja, inteiro (em inglês, *integer*).

### 3.3 Float (ou Ponto Flutuante)

Os dados do tipo *float* (também chamados de Ponto Flutuante) são valores numéricos em que colocamos ponto decimal. Veja abaixo:

### Exemplo 3.8

```
[in]: n = 4.0
      type(n)
[out]: float
```

Note que agora, ao atribuir o valor 4 à variável `n`, colocamos o ponto decimal (diferentemente do padrão no Brasil, a parte decimal é separada da parte inteira por um ponto e não por uma vírgula). Assim, quando solicitamos para Python o tipo da variável `n`, a resposta é *float*.

### 3.4 Dados do tipo booleano (True ou False)

Os dados do tipo booleano assumem dois valores possíveis: True ou False (note que a primeira letra deve ser maiúscula). Essas são duas palavras reservadas por Python e **não** podem ser utilizadas para outros fins (por exemplo, não podemos criar uma variável de nome True).

Os dados do tipo booleano são muito importantes e utilizados com muita frequência. Veremos diversos exemplos posteriormente.

### 3.5 Algumas observações adicionais

Como vimos, os tipos de dados determinam quais operações podem ser realizadas, bem como os resultados dos operadores que utilizamos. O exemplo do operador “+” nos ilustra isto: quando aplicado aos inteiros e aos *floats*, funciona como o operador matemático da adição. Quando aplicado à *strings*, + funciona como um “concatenador” de texto. Vamos ver dois exemplos para esclarecer este ponto:

#### Exemplo 3.9

```
[in]: a = 5
      b = 2
      print(a + b)
[out]: 7
```

#### Exemplo 3.10

```
[in]: a = “Olá, ”
      b = “bom dia!”
      print(a + b)
[out]: Olá, bom dia!
```

No exemplo 3.9, o operador + realiza a soma dos inteiros 5 e 2. Já quando utilizamos “+” com duas *strings* (exemplo 3.10), temos a concatenação dos dois textos.

O que queremos enfatizar é que o operador + exerce determinada funcionalidade dependendo do tipo ou estrutura de dados que estamos trabalhando. E da mesma forma ocorre com muitas outras operações em Python.

Podemos saber quais métodos estão disponíveis para cada tipo de dado utilizando a função *built-in* “dir()”. Por exemplo, podemos saber quais métodos disponíveis para *strings* pelo seguinte comando:

#### Exemplo 3.11

```
[in]: dir(str)
```

O resultado do código acima, que suprimimos por ser muito extenso, será uma lista que contém todos os métodos disponíveis para os dados do tipo *string*. Para saber o que cada método específico faz, o mais simples é realizar uma busca na internet. Uma das vantagens de Python é que ela é uma linguagem que possui uma extensa comunidade ativa que nos permite encontrar praticamente todo tipo de informação sobre suas funcionalidades.

## Capítulo 4 Tipos Básicos de estrutura de dados

### (Listas, Tuplas, Dicionários e Conjuntos)

Neste capítulo, veremos alguns dos tipos mais básicos de estruturas de dados que podemos encontrar em Python: (i) listas, (ii) tuplas, (iii) dicionários e (iv) conjuntos. A necessidade principal de se estudar este tópico é o mesmo do capítulo 3: saber qual o tipo e estrutura de dados que estamos trabalhando nos permite entender as operações que podemos realizar e quais serão os seus resultados.

#### 4.1 Listas

##### 4.1.1 Listas: Ideias Gerais

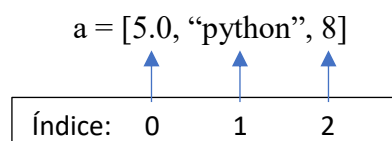
Uma lista é um tipo de estrutura de dados que nos permite atribuir diversos dados à uma única variável. Para gerar uma lista, utilizamos colchetes [ ]. Seus itens são armazenados de acordo com a ordem em que os colocamos. Abaixo, um exemplo de uma lista atribuída à uma variável:

##### Exemplo 4.1

```
[in]: a = [5.0, "python", 8]
      print(a)
[out]: [5.0, "python", 8]
```

Os elementos de uma lista podem ser de diferentes tipos: inteiros, *floats*, *strings*, etc. Também podemos ter estruturas de dados como um elemento de uma lista, até mesmo uma outra lista (ou seja, uma lista pode conter uma ou mais listas, além de outras estruturas de dados).

Cada um de seus elementos possui um índice associado, começando por 0. Assim, o primeiro elemento de uma lista possui índice 0, o segundo índice 1, e assim por diante. Esquemáticamente, no exemplo 4.1 temos:



#### 4.1.2 Acessando um elemento de uma lista

Como os elementos das listas possuem uma ordem, é possível acessá-los utilizando seus respectivos índices. Vamos mostrar como isto é feito através de alguns exemplos:

##### Exemplo 4.2

```
[in]: países = ['Brasil', 'México', 'China', 'Japão']
      moeda = ['Real', 'Peso mexicano', 'Yuan', 'Iene']
      print(países[0]) # L1
      print(moeda[0]) # L2
[out]: Brasil
      Real
```

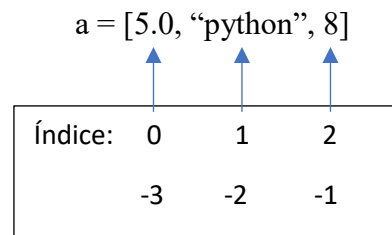
No programa acima, em L1 utilizamos a expressão `países[0]` para acessar o primeiro elemento da lista “países”. O número zero entre colchetes refere-se ao índice do elemento da lista que queremos acessar. A função `print( )` foi utilizada para Python imprimir o elemento como saída. Da mesma forma, em L2 escrevemos `moeda[0]` para obter o primeiro elemento da lista “moeda”. Também poderíamos, por exemplo, acessar o elemento de índice 2 (China) da lista `países` utilizando o código `países[2]`. Vejamos:

##### Exemplo 4.3

```
[in]: países = ['Brasil', 'México', 'China', 'Japão']
      moeda = ['Real', 'Peso mexicano', 'Yuan', 'Iene']
      print(países[2])
[out]: China
```

Note que o primeiro elemento da lista tem índice 0, o segundo elemento tem índice 1, e assim por diante. Assim, se quisermos o *i*-ésimo elemento de uma lista, devemos lembrar que ele possui índice *i*-1.

Vamos voltar ao exemplo 4.1 (com a lista `a = [5.0, “python”, 8]`). Também é possível localizar o elemento da lista por um índice que vincula o último elemento com -1, o penúltimo com -2 e assim por diante. Vejamos abaixo:



Vamos utilizar esse índice “inverso” para obter alguns elementos da lista “países” e também da lista “moeda” no exemplo abaixo:

#### Exemplo 4.4

```
[in]: países = ['Brasil', 'México', 'China', 'Japão']
      moeda = ['Real', 'Peso mexicano', 'Yuan', 'Iene']
      print(países[-1])
      print(moeda[-1])
[out]: Japão
       Iene
```

Por fim, podemos acessar diversos elementos de uma lista utilizando apenas um código:

#### Exemplo 4.5

```
[in]: países = ['Brasil', 'México', 'China', 'Japão']
      print(países[0:3])
[out]: ['Brasil', 'México', 'China']
```

No exemplo acima, é utilizado o código `países[0:3]` para acessar os elementos de índice 0 a 2 da lista `países`. Nele, o elemento de índice 0 é inclusivo e o 3 é exclusivo.

### 4.1.3 Alterando um elemento de uma lista

Podemos alterar o elemento de índice 1 da lista “países” da seguinte forma:

## Exemplo 4.6

```
[in]: países = ['Brasil', 'México', 'China', 'Japão']
      países[1] = 'Portugal'    # L1: altera o elemento de índice 1 da lista países
      print(países)
[out]: ['Brasil', 'Portugal', 'China', 'Japão']
```

Naturalmente, no exemplo acima, poderíamos alterar qualquer elemento da lista “países” substituindo na linha L1 o número 1 entre colchetes pelo índice do elemento que estamos interessados em modificar.

**4.1.4 Excluindo um elemento de uma lista**

Também é possível excluir os elementos de uma lista. O código abaixo exclui o elemento de índice 2 da lista países.

## Exemplo 4.7

```
[in]: países = ['Brasil', 'México', 'China', 'Japão']
      del países[2]    # exclui o elemento de índice 2 da lista países (3º elemento)
      print(países)
[out]: ['Brasil', 'México', 'Japão']
```

**4.1.5 Utilizando a função range() para gerar uma lista**

Em Python, podemos gerar uma sequência numérica com a função nativa range(). Para utilizá-la, incluímos três argumentos, seguindo a estrutura “range(i, f, r)”, sendo que i, f e r devem ser números inteiros, i será o primeiro número da sequência, f onde será finalizada (f não é incluído) e r a distância entre os seus elementos. Vejamos abaixo a criação de uma sequência com o uso da função range( ).

## Exemplo 4.8

```
[in]: s = range(8, 20, 2)
```

No exemplo acima, geramos uma sequência numérica que inicia em 8, termina em 20 (20 não incluído na sequência) com distância entre os seus elementos de 2 unidades (ou seja, a sequência gerada é 8, 10, 12, 14, 16 e 18).

Comumente utilizamos a função `range()` com apenas um argumento. Por exemplo:

#### Exemplo 4.9

```
[in]: s1 = range(20)
```

Neste caso, a função `range()` gera, por padrão, uma sequência que se inicia em 0 e termina em 20 (20 não incluído), com distância de 1 unidade entre os seus elementos (ou seja, a sequência gerada é 0, 1, 2, 3, ..., 19).

Note que, nos exemplos 4.8 e 4.9, geramos uma sequência numérica e não uma lista. Para transformar a sequência numérica em uma lista, devemos combinar a função `range()` com uma outra função nativa, a função `list()`. Uma das vantagens de se fazer isto é que podemos criar uma lista longa de uma forma bem simples. Vejamos:

#### Exemplo 4.10

```
[in]: lista = list(range(20))    #L1
      print(lista)
[out]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

No código acima, em L1 utilizamos as duas funções nativas, `range()` e `list()`. “`range(20)`” gera uma sequência de 0 a 19 (20 não é incluído). Já a função `list()` transforma essa sequência em uma lista, como podemos ver na saída do programa. Note que a função `range()` foi utilizada como argumento para `list()`.

Se quisermos que a lista comece em, digamos, cinco, podemos utilizar o seguinte comando:

#### Exemplo 4.11

```
[in]: lista2 = list(range(5, 10))
      print(lista2)
```



```
[out]: [5, 6, 7, 8, 9]
```

Para gerar uma lista com uma distância específica entre os seus elementos, devemos colocar os três argumentos na função `range( )`. O código abaixo gera uma lista cuja distância entre seus elementos é de duas unidades.

#### Exemplo 4.12

```
[in]: lista3 = list(range(0, 10, 2))  
      print(lista3)  
[out]: [0, 2, 4, 6, 8]
```

Repare que, neste caso, precisamos passar 3 argumentos para a função `range()`, mesmo se quisermos que a sequência comece em zero.

#### 4.1.6 A função `len()`

Com a função `len()` obtemos o número de elementos de uma lista. Vejamos abaixo:

#### Exemplo 4.13

```
[in]: lista = ["a", "b", "c"]  
      print(len(lista))  
[out]: 3
```

No programa acima passamos a lista `["a", "b", "c"]` como argumento da função `len()` e Python nos retorna o seu número de elementos. À primeira vista essa não parece uma função tão importante mas, de fato, ela é muito utilizada em diferentes contextos.

#### 4.1.7 O método `append( )`

Podemos adicionar um elemento ao final de uma lista pelo método `append( )`. Veja o código abaixo.

#### Exemplo 4.14

```
[in]: x = [1, 2, 3, 4]
      x.append(5)
      print(x)
[out]: [1, 2, 3, 4, 5]
```

Neste exemplo, em L1 utilizamos o método `append( )` passando como argumento o número cinco. Isto adiciona este valor ao final da lista `x`.

#### 4.1.8 O operador + utilizado com duas listas

O operador `+`, quando utilizado entre duas listas, faz a junção delas.

#### Exemplo 4.15

```
[in]: países = ['Brasil', 'México', 'China', 'Japão']
      moeda = ['Real', 'Peso mexicano', 'Yuan', 'Iene']
      print(países + moeda)
[out]: ['Brasil', 'México', 'China', 'Japão', 'Real', 'Peso mexicano', 'Yuan', 'Iene']
```

Note que, se tivermos duas listas cujos elementos são números (inteiros ou *floats*) e utilizarmos o operador `+` entre eles, Python **não** somará os elementos de mesmo índice, mas realizará a junção das duas listas. Vejamos:

#### Exemplo 4.16

```
[in]: a = [2, 4, 7]
      b = [1, 2, 3]
      print(a + b)
[out]: [2, 4, 7, 1, 2, 3]
```

## 4.2 Exercícios resolvidos

(Utilize a lista a ser criada no exercício 1 para todos os outros exercícios)

1. Crie duas listas, uma chamada de “países” e outra de “moedas”, as mesmas utilizadas ao longo deste capítulo. Gere a lista “moeda” de forma que exista uma correspondência entre seus elementos e os países. Utilize a função `print( )` para que a saída do programa apresente as duas listas.

Solução:

```
[in]: países = ['Brasil', 'México', 'China', 'Japão']
      moeda = ['Real', 'Peso mexicano', 'Yuan', 'Iene']
      print(países)
      print(moeda)
```

```
[out]: ['Brasil', 'México', 'China', 'Japão']
       ['Real', 'Peso mexicano', 'Yuan', 'Iene']
```

2. Acesse o terceiro elemento da lista “países” (China)

Solução

```
[in]: países = ['Brasil', 'México', 'China', 'Japão']
      print(países[2])    # China é o 3º elemento da lista, mas seu índice é 2
[out]: China
```

3. Acesse o último elemento da lista “países” utilizando índices negativos.

Solução

```
[in]: países = ['Brasil', 'México', 'China', 'Japão']
      print(países[-1])   # Último elemento da lista países: índice -1
[out]: Japão
```

4. Acesse o 2º e o 3º elemento da lista países (México e China).

Solução

```
[in]: países = ['Brasil', 'México', 'China', 'Japão']
      print(países[1:3])   # índice 1, inclusivo; índice 3, exclusivo
[out]: ['Mexico', 'China']
```

5. Altere o 1º elemento da lista “países” de Brasil para Argentina (elemento de índice 0):

Solução

```
[in]: países = ['Brasil', 'México', 'China', 'Japão']
      países[0] = 'Argentina'
      print(países)
[out]: ['Argentina', 'México', 'China', 'Japão']
```

6. Exclua o segundo elemento da lista países (México).

Solução

```
[in]: países = ['Brasil', 'México', 'China', 'Japão']
      del países[1]      # exclui o 2º elemento da lista países, cujo índice é 1
      print(países)
[out]: ['Brasil', 'China', 'Japão']
```

7. Utilize o método append() para incluir “Chile” como último país da lista “países”

Solução

```
[in]: países = ['Brasil', 'México', 'China', 'Japão']
      países.append("Chile")
      print(países)
[out]: ['Brasil', 'México', 'China', 'Japão', 'Chile']
```

8. Crie um programa que tenha como saída o país México e sua moeda.

Solução sugerida 1

```
[in]: países = ['Brasil', 'México', 'China', 'Japão']
      moeda = ['Real', 'Peso mexicano', 'Yuan', 'Iene']
      i = 1
      print(países[i])
      print(moeda[i])
[out]: México
      Peso mexicano
```

Solução sugerida 2<sup>6</sup>

```
[in]: países = ['Brasil', 'México', 'China', 'Japão']
```

---

<sup>6</sup> Utilizamos aqui a estrutura f-string vista no capítulo 3, item 3.1.2 Inserindo uma variável dentro de uma string.

```

moeda = ['Real', 'Peso mexicano', 'Yuan', 'Iene']
i = 1
print(f'País: {países[i]}, moeda: {moeda[i]}')

```

[out]: País: México, moeda: Peso mexicano

## 4.3 Tuplas

### 4.3.1 Tuplas: Ideias Gerais

As tuplas são estruturas de dados parecidas com as listas, mas com a importante diferença de que não podemos alterar seus elementos depois de sua criação. Para gerar uma tupla, usamos parênteses. Vamos ver um exemplo:

Exemplo 4.17

```
[in]: t = (30, 9, 5)
```

No programa acima, criamos uma tupla e atribuímos à ela uma variável que chamamos de `t`.

Podemos acessar seus elementos utilizando uma estrutura como a que usamos com listas:

Exemplo 4.18

```

[in]: t = (30, 9, 5)
      print(t[0])      # L1
[out]: 30

```

Note que o índice do primeiro elemento é zero, assim como nas listas. Com isso, o código `t[0]` na linha L1 acessa o primeiro elemento da tupla `t` criada, como podemos ver na saída do programa.

Como não podemos alterar os elementos das tuplas depois desta ser criada, se tentarmos utilizar uma estrutura do tipo `t[0] = 5` para alterar o primeiro elemento de `t` para o número cinco, obteremos um erro.

Para identificar uma tupla, podemos verificar se os elementos que a compõem estão envoltos de parênteses. Em caso afirmativo, trata-se de uma tupla. No entanto, cabe observar que uma tupla, em realidade, é definida com o uso de uma vírgula e não com uso de parênteses. De fato, para criarmos uma tupla não precisamos utilizar os parênteses. Vejamos um exemplo:

#### Exemplo 4.19

```
[in]: t = 5, 8, 7, 20  # L1
      type(t)
[out]: tuple
```

Como podemos ver, a variável `t` é do tipo “tupla” mesmo sem a utilização dos parênteses para sua criação em L1.

#### 4.3.2 “Desempacotando” uma tupla

Podemos atribuir cada elemento de uma tupla à uma variável distinta através de uma operação chamada “desempacotamento”. Os dois exemplos abaixo ilustram essa operação.

#### Exemplo 4.20

```
[in]: t = (30, 9, 5)  # L1
      a, b, c = t      # L2
      print(a)         # L3
      print(b)         # L4
      print(c)         # L5
[out]: 30
       9
       5
```

Na linha L1, geramos uma tupla. Em L2 fizemos a operação de “desempacotamento” na qual os valores do primeiro, segundo e terceiro elementos da tupla `t` são atribuídos às

variáveis a, b e c, respectivamente.<sup>7</sup> De L3 a L5 apenas solicitamos à Python que nos retorne os valores dessas variáveis.

No segundo exemplo, vamos ver algo que nos será importante posteriormente<sup>8</sup>:

#### Exemplo 4.21

```
[in]: nomes = ("aline", "maria luiza") # L1
      nome1, nome2 = nomes             # L2
      print(nome1.title())              # L3
      print(nome2.upper())              # L4
[out]: Aline
      MARIA LUIZA
```

Novamente, em L1 geramos uma tupla de dois elementos e em L2 fazemos o seu “desempacotamento”. Com isso, o primeiro elemento da tupla “nomes” é atribuído à variável “nome1” e o segundo elemento é atribuído à variável “nome2”. Como estes elementos são *strings*, tanto “nome1” quanto “nome2” também o são. Assim, podemos aplicar os métodos `title()` e `upper()` vistos anteriormente à essas variáveis. É o que fazemos nas linhas L3 e L4, ao mesmo tempo em que pedimos para Python imprimir o resultado como saída.

## 4.4 Dicionários

Um dicionário é um tipo de estrutura de dados caracterizado pela abertura e fechamento de chaves (`{}`) em que cada um dos seus elementos é composto por uma chave e um valor à ela relacionado (estrutura “chave-valor”). Para separar seus elementos, utilizamos vírgula. Vamos ver um exemplo:

#### Exemplo 4.22

```
[in]: d = {'Aline': 33, 'Maria Luiza': 3}
```

<sup>7</sup> É importante (e interessante) esclarecer que em L2 o código “a, b, c = t” NÃO pode ser escrito como “t = a, b, c”. Em “a, b, c = t” atribuímos os elementos da tupla t às variáveis a, b e c. Se escrevêssemos “t = a, b, c” estaríamos tentando criar uma tupla t cujos elementos são os valores das variáveis a, b e c (que, no código do exemplo, não existem previamente à linha L2).

<sup>8</sup> Nos exercícios 2 e 3 do capítulo 10 (“Introdução à elaboração de gráficos em Python”), utilizaremos a operação de “desempacotamento” de tupla ao chamarmos a função `plt.subplots()` para geração de gráficos.

```
print(type(d))
[out]: dict
```

Neste exemplo, criamos uma variável “d” e atribuímos à ela um dicionário com dois elementos, separados por vírgula. Cada um deles possui uma estrutura “chave-valor”. No primeiro elemento a chave é “Aline”, sendo 33 o valor à ela relacionado. No segundo, “Maria Luiza” é a chave e 3 é o valor vinculado à ela.

Os dicionários são mutáveis, ou seja, depois de criado, podemos alterá-lo. Vamos ver alguns exemplos de alterações que podemos realizar:

Exemplo 4.23 (Alterando o valor associado à uma chave)

```
[in]: frutas = { 'Abacate': 6, 'Laranja': 20, 'Limão': 4}    # cria o dicionário frutas
      frutas['Limão'] = 8    # altera o valor associado à Limão de 4 para 8
      print(frutas)
[out]: frutas = { 'Abacate': 6, 'Laranja': 20, 'Limão': 8}
```

Exemplo 4.24 (Adicionando um elemento)

```
[in]: frutas = { 'Abacate': 6, 'Laranja': 20, 'Limão': 4}    # cria o dicionário frutas
      frutas['Morango'] = 10    # acrescenta um elemento cuja chave é 'Morango' e o valor 10
      print(frutas)
[out]: { 'Abacate': 6, 'Laranja': 20, 'Limão': 4, 'Morango': 10}
```

Exemplo 4.25 (Excluindo um elemento)

```
[in]: frutas = { 'Abacate': 6, 'Laranja': 20, 'Limão': 4}    # cria o dicionário frutas
      del frutas['Laranja']    # exclui o elemento cuja chave é “Laranja”
      print(frutas)
[out]: { 'Abacate': 6, 'Limão': 4}
```

Apesar dos dicionários serem mutáveis, dentro da estrutura chave-valor de cada um dos seus elementos não podemos alterar uma chave já criada. Além disto, as chaves não são necessariamente do tipo *string*. De forma geral, para as chaves, podemos utilizar os tipos e



estruturas de dados imutáveis (como inteiros, *floats*, tuplas). Já os valores (na estrutura chave-valor) podem ser de diferentes tipos, inclusive mutáveis, como *strings*, listas, inteiros, *floats*, etc., até mesmo outro dicionário.

Para finalizar este tópico sobre dicionários, vamos acessar o valor de uma chave específica. Vejamos isto através do seguinte exemplo.

#### Exemplo 4.26

```
[in]: frutas = {'Abacate': 6, 'Laranja': 20, 'Limão': 4} # L1:cria um dicionário de nome frutas
      print(frutas['Laranja']) #L2
[out]: 20
```

No programa acima, na linha L1 criamos uma variável “frutas” e atribuímos à ela um dicionário. Em L2, para acessar o valor associado à chave “Laranja”, usamos a estrutura `frutas['Laranja']`, em que “frutas” é o nome da variável à qual atribuímos o dicionário e entre colchetes temos a chave do elemento que buscamos.

## 4.5 Conjuntos

### 4.5.1 Algumas ideias gerais sobre conjuntos

Em Python, os conjuntos são uma coleção não ordenada de objetos. Dentre suas características mais importantes podemos listar:

- 1) Por ser não ordenado, não há índices pelos quais seus elementos possam ser localizados;
- 2) Seus elementos são únicos, ou seja, um determinado objeto aparece uma única vez num conjunto (`{1,2,2}` e `{1,2}` são considerados iguais)
- 3) seus elementos não precisam ser necessariamente números.

Para gerar um conjunto, utilizamos `{ }` ou a função `set()`. Vamos ver dois exemplos:

#### Exemplo 4.27

```
[in]: c = {1, 2, 3, 4, 5}      #gera um conjunto cujos elementos são inteiros
      d = set([1, 2, 3, 4, 5]) #gera um conjunto idêntico ao conjunto c
```

```
e = {'a', 'b', 'c'}      #gera um conjunto cujos elementos são strings
```

Para gerar um conjunto vazio, devemos usar o seguinte comando:

#### Exemplo 4.28

```
[in]: s = set()          # gera um conjunto vazio e atribui à variável s
```

**Não podemos** criar um conjunto vazio escrevendo `s = {}`. De fato, este comando gera um dicionário.

### 4.5.2 Algumas operações com conjuntos

Podemos fazer operações entre conjuntos como União, Intersecção e Diferença. Veja abaixo:\.

#### Exemplo 4.29

```
[in]: a = {4, 9, 15}
      b = { 9, 20}
      u = a | b # União entre o conjunto a e b
      i = a & b # Intersecção entre o conjunto a e b
      d = a - b # Diferença entre conjunto a e b
      print(f'União: {u}')
      print(f'Intersecção: {i}')
      print(f'Diferença: {d}')
[out]: União: {4, 9, 15, 20}
      Intersecção: {9}
      Diferença: {4, 15}
```

### 4.6 Algumas observações sobre outros tipos de estrutura de dados

Além dos tipos mais básicos de estruturas de dados (como as listas, as tuplas, os dicionários e os conjuntos), ao utilizarmos Python nos depararemos com outras (Data Frame,

Series, Times Series, Arrays e etc). Estas aparecem à medida que precisamos recorrer à bibliotecas para aplicações específicas. Por exemplo, quando utilizamos a biblioteca Pandas (uma das mais importantes para análise de dados), é comum o uso de uma estrutura de dados chamada *Data Frame*. Ao utilizarmos a biblioteca numpy, é comum nos depararmos com os Arrays e as matrizes. Como poderíamos esperar, essas estruturas de dados possuem seus próprios métodos e características.

Não é viável apresentar todas as possíveis estruturas de dados que existem em Python. Elas devem ser estudadas à medida que aparecem no uso das bibliotecas. Assim, ao longo da análise das bibliotecas específicas, poderão ser vistos os métodos e atributos vinculados às diferentes estruturas que aparecerão ao longo das aplicações.

## Capítulo 5 Operações matemáticas básicas

Neste capítulo vamos ver como efetuar operações matemáticas em Python. Naturalmente, serão utilizados nestas operações os dados numéricos (tipos inteiro e *float*).

### 5.1 Adição

As operações de adição são realizadas pelo operador “+”. Vejamos:

#### Exemplo 5.1

```
[in]: print(2 + 7)
[out]: 9
```

Poderíamos atribuir esta soma à uma variável e depois usar a função `print( )` para que Python apresente o resultado. O exemplo abaixo ilustra o que queremos dizer.

#### Exemplo 5.2

```
[in]: a = 2 + 7
      print(a)
[out]: 9
```

Este tipo de atribuição pode ser feito em todas as operações matemáticas apresentadas a seguir. Mas, para sermos sucintos, não faremos.

### 5.2 Subtração

As operações de subtração são realizadas pelo operador “-”:

#### Exemplo 5.3

```
[in]: print(20 - 14)
[out]: 6
```

### 5.3 Multiplicação

Utilizamos asterisco (\*) para multiplicação:

#### Exemplos 5.4

```
[in]: print(5 * 2)
```

```
[out]: 10
```

### 5.4 Potenciação

As operações de potenciação são realizadas com dois asteriscos (\*\*):

#### Exemplo 5.5

```
[in]: print(4 ** 2)
```

```
[out]: 16
```

### 5.5 Divisão

Podemos efetuar operações de divisão com “/”:

#### Exemplo 5.6

```
[in]: print(10 / 2)
```

```
[out]: 5.0
```

### 5.6 Resto de uma divisão (x % y)

Podemos obter o resto de uma divisão de dois números utilizando o símbolo “%”.

Vejamos abaixo:

#### Exemplo 5.7

```
[in]: print(18.5 % 3)
```

```
[out]: 0.5
```

Acima, o programa retorna o resto da divisão de 18.5 por 3.

### 5.7 A parte inteira de uma divisão (x // y)

Utilizando duas barras, obtemos a parte inteira da divisão entre dois números:

## Exemplo 5.8

```
[in]: print(18.5 // 3)
```

```
[out]: 6.0
```

**5.8 Operações “dinâmicas”**

Uma característica interessante das variáveis em Python é que podemos escrever expressões do seguinte tipo:

## Exemplo 5.9

```
[in]: z = 2          # L1: atribuímos 2 à variável z
      ↓
      z = 1 + z      # L2: z = 1 + 2, portanto, z passa a valer 3
      ↓
      z = 1 + z      # L3: z = 1 + 3, portanto, z passa a ser 4
      print(z)
```

```
[out]: 4
```

Nos exercícios resolvidos abaixo, veremos como isto pode ser útil para desenvolvermos modelos econômicos que nos apresentam o comportamento das variáveis ao longo do tempo (ou seja, modelos dinâmicos).

**5.9 Exercícios Resolvidos**

1. Dados:

$L = 100$ , o número de trabalhadores de um determinado país no tempo  $t = 0$

$g = 3\%$ , o crescimento do número de trabalhadores a cada período

Compute o número de trabalhadores deste país a cada período, de  $t = 0$  a  $t = 5$ .

Solução sugerida 1:

```
[in]: L = 100 # número inicial de trabalhadores em t = 0
```

```
      g = 0.03 # taxa de crescimento de número de trabalhadores por período
```

```

print(L)
L = L*(1 + g)  # número de trabalhadores em t = 1
print(L)
L = L*(1 + g)  # número de trabalhadores em t = 2
print(L)
L = L*(1 + g)  # número de trabalhadores em t = 3
print(L)
L = L*(1 + g)  # número de trabalhadores em t = 4
print(L)
L = L*(1 + g)  # número de trabalhadores em t = 5
print(L)
[out]: 100
      103.0
      106.09
      109.2727
      112.550881
      115.92740743

```

Solução sugerida 2:

```

[in]: trabalho = [ ]  # gera uma lista vazia
      g = 0.03
      L = 100  # número inicial de trabalhadores em t = 0
      trabalho.append(L)
      L = L*(1 + g)  # número inicial de trabalhadores em t = 1
      trabalho.append(L)
      L = L*(1 + g)  # número inicial de trabalhadores em t = 2
      trabalho.append(L)
      L = L*(1 + g)  # número inicial de trabalhadores em t = 3
      trabalho.append(L)
      L = L*(1 + g)  # número inicial de trabalhadores em t = 4
      trabalho.append(L)
      L = L*(1 + g)  # número inicial de trabalhadores em t = 5
      trabalho.append(L)

```

```
print(trabalho)
```

```
[out]: [100, 103.0, 106.09, 109.2727, 112.550881, 115.92740743]
```

2. Considere uma economia com as seguintes características:

→  $PIB = Y = F(K, L) = (K * L)^{\alpha}$

→  $\alpha = 0.5$

→  $K = 100$  (Estoque de capital da economia em unidades reais)

→  $L$  segue a trajetória vista no exercício anterior

A partir dos dados acima, compute o PIB do país de  $t = 0$  a  $t = 5$

Solução sugerida:

```
[in]: pib = [ ] # gera uma lista vazia
```

```
alpha = 0.5
```

```
K = 100 # Estoque de capital da economia
```

```
g = 0.03 # crescimento do número de trabalhadores na economia a cada período
```

```
L = 100 # número inicial de trabalhadores em t = 0
```

```
Y = (K*L)**alpha # PIB em t = 0
```

```
pib.append(Y)
```

```
L = L*(1 + g) # número de trabalhadores em t = 1
```

```
Y = (K*L)**alpha # PIB em t = 1
```

```
pib.append(Y)
```

```
L = L*(1 + g) # número de trabalhadores em t = 2
```

```
Y = (K*L)**alpha # PIB em t = 2
```

```
pib.append(Y)
```

```
L = L*(1 + g) # número de trabalhadores em t = 3
```

```
Y = (K*L)**alpha # PIB em t = 3
```

```
pib.append(Y)
```



```
L = L*(1 + g)    # número de trabalhadores em t = 4  
Y = (K*L)**alpha    # PIB em t = 4  
pib.append(Y)
```

```
L = L*(1 + g)    # número de trabalhadores em t = 5  
Y = (K*L)**alpha    # PIB em t = 5  
pib.append(Y)
```

```
print(f'PIB = {pib}')
```

```
[out]: PIB = [100.0, 101.49, 103.0, 104.53, 106.09, 107.67]
```

## Apêndice

### A.5.1 Uma estrutura especial para operações matemáticas

Ao utilizarmos Python, é comum nos depararmos com uma estrutura especial nas operações matemáticas. Os exemplos seguintes mostram essa estrutura.

#### Exemplo 5.9

```
[in]: x = 1
      x += 2 # equivale ao código: x = x + 2 (no exemplo: x = 1 + 2)
      print(x)
[out]: 3
```

#### Exemplo 5.10

```
[in]: x = 3
      x -= 2 # equivale ao código: x = x - 2 (no exemplo, x = x - 2)
      print(x)
[out]: 1
```

#### Exemplo 5.11

```
[in]: x = 6
      x /= 3 # equivale ao código: x = x / 3 (no exemplo, x = 6 / 3)
      print(x)
[out]: 2
```

#### Exemplo 5.12

```
[in]: x = 10
      x **= 4 # equivale ao código: x = x ** 4 (no exemplo, x = 10 ** 4)
      print(x)
[out]: 10000
```

## Capítulo 6 Operadores relacionais e lógicos

Os operadores relacionais e lógicos são importantes ferramentas utilizadas como base para diversos outros tópicos em programação, dentre eles assuntos relacionados com controle do fluxo de execução dos programas e algumas estruturas de repetição. Ambos, controle de fluxo de execução e estruturas de repetição, serão temas dos próximos dois capítulos e aumentam substancialmente o potencial para criação de programas úteis.

### 6.1 Operadores relacionais (==, !=, >, >=, <, <=)

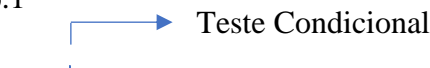
Os operadores relacionais comparam dois elementos, podendo ser utilizados para construir os chamados “Testes Condicionais” em que Python retorna um resultado do tipo booleano (*True* ou *False*, nunca ambos). Veremos os seguintes operadores relacionais:

- Operador relacional de igualdade: ==
- Operador relacional de “diferença”: !=
- Operadores relacionais de desigualdades: >, >=, <, <=

#### 6.1.1 Operador relacional de igualdade ( == )<sup>9</sup>

Podemos utilizar o operador “==” para avaliar a equivalência entre dois elementos. Se os dois elementos forem considerados iguais por Python, teremos como retorno *True*. Caso forem considerados diferentes, o resultado será *False*. Vejamos dois exemplos.

##### Exemplo 6.1



[in]: print(10 == 10.0)

[out]: True

Teste Condicional

Como Python considera 10 igual a 10.0, o resultado da aplicação do operador “==” é *True*.

---

<sup>9</sup> Note que o operador relacional “==” compara dois elementos e resulta num valor booleano (*True* ou *False*). O sinal “=” é um operador de atribuição, em que se atribui algo à uma variável.

### Exemplo 6.2

```
[in]: print(10 == 11)
```

```
[out]: False
```

Como 10 é diferente de 11, o resultado da aplicação do operador “==” é *False*.

#### 6.1.2 Operador relacional de “diferença” ( != )

Utilizamos o operador “!=” para verificar se dois elementos são diferentes. Se os elementos que estão sendo comparados forem considerados diferentes um do outro, o resultado da comparação será *True*. Caso os elementos sejam iguais, será *False*.

### Exemplo 6.3

```
[in]: print(5 != 10)
```

```
[out]: True
```

No exemplo acima, como 5 é diferente de 10, o resultado é *True*.

### Exemplo 6.4

```
[in]: print(4 != 4)
```

```
[out]: False
```

Neste exemplo, como 4 é igual à ele mesmo, o resultado é *False*.

#### 6.1.3 Operadores relacionais de desigualdades ( >, >=, <, <= )

Os operadores >, >=, <, <= correspondem ao usual na matemática. Vamos vê-los através de alguns exemplos.

### Exemplo 6.5

```
[in]: print(10 > 8)
```

```
[out]: True
```

#### Exemplo 6.6

```
[in]: print(8 >= 10)
```

```
[out]: False
```

#### Exemplo 6.7

```
[in]: print(4 < 4)
```

```
[out]: False
```

#### Exemplo 6.8

```
[in]: print(3 <= 3)
```

```
[out]: True
```

### 6.1.4 Nota sobre formatação

É recomendável, ainda que não obrigatório, deixar um espaço em branco antes e depois dos operadores relacionais (por exemplo, é melhor escrever “`x <= y`” do que “`x<=y`”). Estes espaços não alterarão o funcionamento do código escrito, porém torna a sua leitura mais agradável.<sup>10</sup>

## 6.2 Operadores lógicos (and, or, not)

Para o que segue, vamos considerar o seguinte vocabulário:

**Proposição simples:** Uma proposição simples é uma sentença que pode assumir apenas dois valores: ou é falso ou é verdadeiro, não podendo ser ambos ao mesmo tempo. Utilizaremos a abreviação P, P1 e P2 para nos referir às proposições simples.

**Proposição Composta:** Uma proposição composta é uma proposição que possui duas ou mais proposições simples, ligadas entre si pelos operadores lógicos “and” ou “or”.

**Conjunção:** Uma conjunção é uma proposição composta do tipo “P1 and P2”.

---

<sup>10</sup> A linguagem Python possui uma série de recomendações para padronização da sua escrita, algo como sugestões de “bons hábitos”. Um dos objetivos disto é homogeneizar os códigos escritos por diferentes pessoas, tornando-os mais fáceis de entender e mais agradáveis de ler. Sobre este assunto, recomendamos procurar pela documentação PEP 8 na internet.

**Disjunção:** Uma disjunção é uma proposição composta do tipo “P1 or P2”.

**Negação:** A negação de uma proposição P é escrita em Python como “not P”.

Seguiremos as seguintes regras:

R1. Uma conjunção (P1 and P2) é verdadeira somente se as duas proposições simples que a compõe são verdadeiras.

R2. Uma disjunção (P1 or P2) é falsa somente se as duas proposições simples que a compõe são falsas.

R3. A negação de uma proposição simples P (not P) é falsa somente se P for verdadeira.

Agora vamos aplicar em Python o que dissemos acima.

#### Exemplo 6.9

```
[in]: 5 >= 1 and 4 == 4      # L1: (P1 and P2) é uma conjunção
      └──┬──┘ └──┬──┘
        P1      P2
```

[out]: True

Vamos entender porque Python retorna o valor booleano *True* neste exemplo 6.9. Primeiramente vemos que em L1 temos uma conjunção (P1 and P2). Utilizaremos, portanto, R1 para verificar se o resultado lógico de L1 é *False* ou *True*. Como P1 e P2 são verdadeiros, por R1 temos que o resultado desta conjunção é *True*.

#### Exemplo 6.10

```
[in]: 5 >= 8 and 4 == 4      # L1: (P1 and P2) é uma conjunção
      └──┬──┘ └──┬──┘
        P1      P2
```

[out]: False

Neste exemplo, também temos uma conjunção (P1 and P2). No entanto, P1 é *False*. Por R1 temos que o resultado de L1 é *False*<sup>11</sup>.

#### Exemplo 6.11

[in]:  $7 >= 9$  or  $4 > 4$       # L1: (P1 or P2) é uma disjunção  
           P1            P2

[out]: False

Acima, temos uma disjunção (P1 or P2) em que P1 e P2 são ambas *False*. Por R2, temos que o resultado de L1 é *False*.

#### Exemplo 6.12

[in]:  $4 >= 2$  or  $1 > 8$       # L1: (P1 or P2) é uma disjunção  
           P1            P2

[out]: True

Aqui, também temos uma disjunção (P1 or P2) em que P1 é *True* e P2 é *False*. Por R2, temos que o resultado de L1 é *True*.

#### Exemplo 6.13

[in]: not  $8 > 1$       # L1: (not P) é uma negação de P  
           P

[out]: False

---

<sup>11</sup> Por R1, “Uma conjunção (P1 and P2) é verdadeira somente se as duas proposições que a compõe são verdadeiras”. Como neste exemplo P1 é *False*, temos que (P1 and P2) não pode ser *True*. Como uma proposição ou é *False* ou é *True*, não podendo ser as duas ao mesmo tempo e não havendo um terceiro valor que ela possa assumir, concluímos que a conjunção é *False* neste exemplo.

Neste exemplo, temos a negação de P. Como P é verdadeiro então, por R3, temos como resultado *False*.

### Exemplo 6.14

[in]: not  $9 < 2$       # L1: (not P) é uma negação de P

P

```
[out]: True
```

Neste exemplo, temos a negação de P em que P é falso. Por R3, temos como resultado  $True^{12}$ .

<sup>12</sup> Como “not P” é falso somente se P for verdadeiro, se P for falso então (not P) é verdadeiro.



## Capítulo 7 Estruturas condicionais de fluxo (if)

Neste capítulo, vamos ver três estruturas que nos permitirão controlar o fluxo de execução de nossos programas, de acordo com alguma condição. Ou seja, com essas sintaxes, imporemos condições que vão fazer com que nosso programa execute ou não determinado bloco de códigos.

### 7.1 Estrutura “if condição:”

Considere o código com a seguinte estrutura:

if *condição*:

*bloco 1*

Na sintaxe acima, devemos colocar após “if” uma condição de nosso interesse. Se essa condição for verdadeira (*True*, um tipo booleano), Python irá executar “bloco 1”, que pode conter uma ou mais linhas. Caso essa condição seja falsa (o booleano *False*), Python irá ignorar “bloco 1” e continuará a execução do restante do programa. Note que “bloco 1” não está alinhado à esquerda, ou seja, possui indentação<sup>13</sup>. Essa indentação é importante para que Python possa identificar o bloco a ser executado caso a condição seja *True* (e ignorado caso seja *False*).

Vamos ver um exemplo simples:

#### Exemplo 7.1

```
[in]: x = 3
      if x > 1:
          print(x)
[out]: 3
```

No programa acima, a condição é dada por “ $x > 1$ ” e o bloco a ser executado caso essa condição seja avaliada como verdadeira é dado por “`print(x)`”. Como atribuímos o número inteiro 3 à  $x$  no início do programa,  $x > 1$  é avaliada como *True*. Com a condição sendo

---

<sup>13</sup> Como vimos, os espaços em branco no início da linha são chamados “indentação” e indicam para Python a existência de um bloco de códigos dentro do programa. Por recomendação (PEP 8), convém utilizar quatro espaços ou a tecla Tab para marcar a indentação.

verdadeira, Python executa o bloco indentado “print(x)”. Como resultado, obtemos como retorno do programa o valor de x, ou seja, o número inteiro 3.

Vamos ver um segundo exemplo:

### Exemplo 7.2

```
[in]: x = 0
      if x > 1:
          print(x)
[out]:
```

Neste exemplo 7.2, atribuímos o número zero à x. Neste caso, a condição  $x > 1$  é *False* e, como consequência, o bloco indentado dado por “print(x)” não é executado.

Nos exemplos acima os blocos possuem apenas uma linha, mas eles podem conter quantas forem necessárias. As linhas que fazem parte de um determinado bloco precisam estar alinhadas na mesma indentação pois Python as identifica por este espaçamento inicial. Vejamos um outro exemplo:

### Exemplo 7.3

```
[in]: nota = 5                                # L1
      if nota == 5:                            # L2
          nota = nota + 1                      # L3
          print(nota)                         # L4
          print("Você ganhou um ponto extra") # L5
      print("O programa será finalizado")     # L6
[out]: 6
      Você ganhou um ponto extra
      O programa será finalizado
```

Neste exemplo, em L1 atribuímos o número inteiro cinco à variável “nota”. Em L2 temos uma condicional que indica para Python executar o bloco indentado caso a condição “nota == 5” seja *True* (ou seja, se a nota for igual a cinco). Desta vez o bloco indentado possui três linhas (L3, L4 e L5). Note que a linha L6, por não estar indentada, será executada

independentemente da estrutura condicional. Quando rodamos o programa, Python identifica a condição “nota == 5” como verdadeira (*True*) e executa o bloco indentado. Isto adiciona 1 à nota inicial (linha L3) e imprime como saída do programa tanto a nota atualizada (L4) como a string “Você ganhou um ponto extra” da linha L5. Após o término do bloco indentado, Python continua a execução do programa e a *string* contida em L6 também aparece como saída.

Vamos ver o mesmo exemplo anterior, atribuindo um valor inicial diferente de 5 à variável “nota”:

#### Exemplo 7.4

```
[in]: nota = 3
      if nota == 5:
          nota = nota + 1
          print(nota)
          print("Você ganhou um ponto extra")
      print("O programa será finalizado")
[out]: O programa será finalizado
```

Note que, neste caso, a condição “nota == 5” é *False* (pois  $x = 3$ ). Assim, todo o bloco indentado na estrutura condicional não é executada. No entanto, como a última linha do programa não faz parte da estrutura condicional (pois não está indentada), ela é executada independentemente da condição ser *True* ou *False*.

### 7.2 Estrutura “if / else”

Nos programas anteriores, não estabelecemos nenhum bloco de execução no caso em que a condição em “if *condição*” seja falsa. Vamos fazer isto neste item. Vejamos a seguinte estrutura:

```
if condição:
    bloco 1
else:
    bloco 2
```

Nesta estrutura condicional, também escolhemos uma condição de nosso interesse. Caso essa “condição” seja verdadeira (ou seja, o tipo booleano *True*), o bloco 1 é executado e o bloco 2 é ignorado. Caso seja falsa (*False*), o bloco 1 é ignorado e o bloco 2 é executado. Note que “*else*” está alinhando à esquerda com “*if*” e tanto o bloco 1 quanto o bloco 2 possuem indentação.

Vamos ver um exemplo simples em que utilizamos essa estrutura:

#### Exemplo 7.5

```
[in]: x = 10
      if x > 8:
          print("x é maior do que 8")
      else:
          print("x é menor do que ou igual à 8")
[out]: x é maior do que 8
```

No programa acima, iniciamos atribuindo o número inteiro 10 à variável *x*. Nossa condição escolhida é “*x > 8*”. Como essa condição é verdadeira (*True*), Python executa o primeiro bloco “*print(“x é maior do que 8”)*” e ignora o bloco que pertence à “*else*”.

Vejamos outro exemplo:

#### Exemplo 7.6

```
[in]: x = 7
      if x > 8:
          print("x é maior do que 8")
      else:
          print("x é menor do que ou igual à 8")
[out]: x é menor do que ou igual à 8
```

Neste exemplo, atribuímos o inteiro 7 à *x*. Como a condição escolhida *x > 8* é falsa (*False*), Python ignora o primeiro bloco “*print(“x é maior do que 8”)*” e executa o bloco relacionado à “*else*”.

### 7.3 Estrutura “if / elif/ elif/ ... /else”

A estrutura vista neste item permite o estabelecimento de diversas condições. Vejamos abaixo:

```
if condição1:
    bloco 1
elif condição2:
    bloco 2
elif condição3:
    bloco 3
    ...
elif condição n-1:
    bloco n-1
else:
    bloco n
```

Python irá executar o bloco pertencente à primeira condição que seja *True*, deixando de executar todos os blocos posteriores, mesmo que haja outras condições cujo valor deva ser avaliada como *True*. Ou seja, se, digamos, a condição 3 for a primeira a ser *True*, o bloco 3 será executado e Python não executará nenhum outro bloco posterior, mesmo se houver outra condição tida como *True*. Caso nenhuma das n-1 condições forem *True*, Python irá executar o bloco n, que diz respeito à “else.”<sup>14</sup>.

Vejamos um exemplo:

#### Exemplo 7.7

```
[in]: nota = 5.8
      if nota >= 9:    # Condição 1
          print("Passou com nota alta")
      elif nota >= 6:  # Condição 2
          print("Passou com nota acima da média")
```

---

<sup>14</sup> A estrutura “if-elif-else” pode ser utilizada sem o “else”. Ou seja, também podemos escrever um programa que contenha apenas uma estrutura do tipo “if-elif”, em que “else” é omitido.

```
elif nota >= 5:  # Condição 3
    print("Passou com nota mínima")
elif nota >= 3:  # Condição 4
    print("Deverá fazer prova substitutiva")
else:
    print("Reprovado")
[out]: Passou com nota mínima
```

No programa acima, atribuímos 5.8 (um *float*) à variável “nota”. Note que tanto a Condição 1 (nota  $\geq 9$ ) quanto a Condição 2 (nota  $\geq 6$ ) são falsas (*False*). Desta forma os blocos referentes à essas condições não são executados. Já a Condição 3 (nota  $\geq 5$ ) é verdadeira (*True*) e, assim, Python executa o bloco que pertence à ela, como podemos ver na saída do programa. Note que a Condição 4 (nota  $\geq 3$ ) também deve ser considerada verdadeira (*True*), mas Python não executa o seu bloco pois não há a execução dos blocos relacionados às condições que são posteriores à primeira avaliada como *True*.

## Capítulo 8 Loops: estruturas de repetição “for x in X” e “while”

Os laços de repetição (também chamados *loops*) permitem que um determinado bloco de códigos de um programa seja executado repetidas vezes. Neste capítulo iremos apresentar dois tipos de estruturas de *loops*:

→ item 8.1: “for \_\_\_\_ in \_\_\_\_:”

→ item 8.3: “while *condição*:”.

### 8.1 Laços do tipo “for \_\_\_\_ in \_\_\_\_:”

Veremos duas formas de utilizar os laços de repetição do tipo “for \_\_\_\_ in \_\_\_\_:”:

(i) utilizando listas;

(ii) utilizando a função `range( )`

Esses laços apresentarão as seguintes estruturas gerais:

Estrutura Geral 1

for \_\_\_\_ in *lista*:

bloco de repetição

Estrutura Geral 2

for \_\_\_\_ in `range( )`:

bloco de repetição

Também podemos criar laços de repetição com dicionários, mas não abordaremos essa forma por ser de utilidade limitada para nossos propósitos (introduzir a linguagem Python como instrumental para estudos de tópicos em economia)<sup>15</sup>.

#### 8.1.1 Laços do tipo “for \_\_\_\_ in *lista*:”

Primeiramente, veremos como construir os laços de repetição do tipo “for \_\_\_\_ in *lista*:”. Vamos utilizar um exemplo para explicar como podemos fazer isto.

---

<sup>15</sup> A construção de *loops* com dicionários pode ser vista, por exemplo, no livro de Matthes (2016), “Curso Intensivo de Python: Uma Introdução Prática e Baseada em Projetos”, capítulo 6 “Dicionários”.

## Exemplo 8.1

```
[in]: X = [2, 4, 6, 8] # L1
      for x in X:      # L2
          print(x+1)   # L3
      print("Fim")     # L4
[out]: 3
       5
       7
       9
       Fim
```

Na primeira linha do exemplo, atribuímos uma lista à variável X. Em L2 utilizamos a estrutura geral “for \_\_\_\_ in *lista*:”, em que x é uma letra qualquer de nossa escolha<sup>16</sup> e X é uma lista. Note que o bloco de repetição (L3) está indentado. Esta indentação sinaliza para Python o bloco de códigos a serem executados na repetição do *loop*.

A instrução “for x in X:” faz com que Python realize a seguinte sequência:

- 1) Python gera uma variável “x” e atribui à ela o primeiro valor da lista X (neste exemplo, o inteiro 2);
- 2) Python executa o bloco caracterizado pela indentação, a linha L3. Como nesta primeira execução temos x = 2, Python retorna como resultado 3, a primeira linha da saída do programa;
- 3) Ao terminar de executar o bloco indentado (linha L3), Python retorna à linha L2 e, desta vez, atribui à variável “x” o segundo elemento de X (neste exemplo, o inteiro 4).
- 4) Novamente, Python executa as linhas que constituem o bloco de repetição (L3) e, como agora x = 4, Python retorna o valor 5, a segunda linha da saída do programa. Python irá executar esse “*loop*” até passar por todos elementos da lista X.
- 5) Após ter percorrido todos os elementos da lista X, Python continua a execução do restante do programa. Neste exemplo, é executado a linha L4 que contém o código “print(“Fim”). Note que Python entende que essa linha não faz parte do bloco de repetição porque ela não está indentada.

---

<sup>16</sup> Podemos escolher uma letra ou um nome, o que preferirmos.



### 8.1.2 Laços do tipo “for \_\_\_\_ in range( ):”

Os laços de repetição construídos com a função `range( )` seguem a mesma execução dos laços utilizando listas<sup>17</sup>. A única diferença é que na estrutura geral “for \_\_\_\_ in *lista*:” substituímos a lista pela função `range( )`, tornando a estrutura “for \_\_\_\_ in `range( )`:”.

Primeiramente devemos recordar que a função `range( )` gera uma sequência numérica. Quando fazemos um laço de repetição com `range( )`, Python percorre cada um dos elementos da sequência numérica gerada, executando o bloco de repetição com cada um desses elementos, da mesma forma que ocorre quando utilizamos listas em nossos laços de repetição. Vamos ver o exemplo abaixo:

Exemplo 8.2:

[in]: for x in range(4):

```
    print(x)      # bloco de repetição, identificada pela indentação
    print("Fim")
```

[out]: 0

1

2

3

Fim

Neste exemplo, `range(4)` gera a sequência 0, 1, 2, 3. Já a instrução “for x in `range(4)`:” faz com que Python realize os passos de 1 a 5 vistos no item anterior, mas desta vez percorrendo os valores da sequência numérica gerada com a função `range(4)`. Note que, neste caso, o bloco de repetição é o comando “`print(x)`”, como podemos ver pela indentação.

## 8.2 Exercícios Resolvidos

1. Vamos utilizar Python para calcular o estoque de capital de um país do período 0 ao 5, dadas as seguintes condições:

- O estoque inicial é  $K = 10$
- Não há depreciação
- Investimento é igual a 2 unidades a cada período.

---

<sup>17</sup> A função `range( )` foi abordada no capítulo 4, item 4.1.5 (Utilizando a função `range()` para gerar uma lista)

Solução sugerida 1: Loop com lista

```
[in]: tempo = [0, 1, 2, 3, 4, 5]
      investimento = 2
      K = 10 # Estoque de capital em t = 0
      for t in tempo: # L1
          print(K) # L2
          K = K + investimento # L3
```

```
[out]: 10
        12
        14
        16
        18
        20
```

Podemos notar que o “t” na expressão “for t in tempo:” não consta em nenhuma expressão do bloco de repetição. Neste caso, “t in tempo:” é utilizado apenas para que o *loop* seja executado 6 vezes (o período de 0 a 5), mas a variável t não faz parte de nenhum cálculo.

As linhas de L1 a L3 fazem o seguinte: Python gera uma variável t e atribui à ela o inteiro 0 (o primeiro elemento da lista tempo). Após rodar o bloco assinalado pela indentação (L2 e L3), Python retorna ao início (L1) e atribui à variável t o número inteiro 1 (o segundo elemento da lista). Novamente, Python executa as linhas do bloco indentado (agora com t = 1), retorna à linha L1, atribui à t o terceiro elemento da lista (t = 2) e executa o bloco de repetição. Esse procedimento é repetido até que t assuma o valor 5, último elemento da sequência da lista tempo.

Solução sugerida 2: Loop com a função range( )

Também é possível executar este tipo de *loop* utilizando a função nativa range(). Vejamos abaixo:

```
[in]: investimento = 2
      K = 10 # Estoque de capital em t = 0
```

```

for t in range(6): #L1
    print(K)        # L2
    K = K + investimento # L3

```

[out]: 10

12

14

16

18

20

2. (Repetição do exercício 1 do capítulo 5 “Operações matemáticas básicas”)

Dados:

$L = 100$ , o número de trabalhadores de um determinado país no tempo  $t = 0$

$g = 3\%$ , o crescimento do número de trabalhadores a cada período

Compute o número de trabalhadores deste país a cada período, de  $t = 0$  a  $t = 5$ .

Solução sugerida 1: loop com uma lista

```

[in]: L = 100 # número inicial de trabalhadores em t = 0
      g = 0.03 # taxa de crescimento de número de trabalhadores por período: 3%
      trabalho = [ ] # gera uma lista vazia
      tempo = [0, 1, 2, 3, 4, 5]
      for t in tempo:
          trabalho.append(L)
          L = L*(1+g)
      print(trabalho)

```

[out]: [100, 103.0, 106.09, 109.2727, 112.550881, 115.92740743]

Solução sugerida 2: loop com a função range( )

[in]:  $L = 100$  # número inicial de trabalhadores em  $t = 0$

$g = 0.03$  # taxa de crescimento de número de trabalhadores por período: 3%

```

trabalho = [ ] # gera uma lista vazia
for t in range(6):
    trabalho.append(L)
    L = L*(1+g)
print(trabalho)
[out]: [100, 103.0, 106.09, 109.2727, 112.550881, 115.92740743]

```

### 3. (Repetição do exercício 2 do capítulo 5 “Operações matemáticas básicas”)

Considere uma economia com as seguintes características:

- $PIB = Y = F(K, L) = (K * L)^{\alpha}$
- $\alpha = 0.5$
- $K = 100$  (Estoque de capital da economia em unidades reais)
- $L = 100$  (Quantidade de trabalho em  $t = 0$ )
- $g = 0.03$  (taxa de crescimento de número de trabalhadores por período: 3%)

A partir dos dados acima, compute o PIB do país de  $t = 0$  a  $t = 5$

Solução sugerida:

```

[in]: pib = [ ] # gera uma lista vazia para armazenar o PIB a cada período
alpha = 0.5
K = 100 # Estoque de capital da economia
g = 0.03 # crescimento do número de trabalhadores na economia a cada período
L = 100 # número inicial de trabalhadores em t = 0
tempo = [0, 1, 2, 3, 4, 5]

for t in tempo:
    Y = (K*L)**alpha
    pib.append(Y)
    L = L*(1+g)

print(f'PIB = {pib}')
[out]: PIB = [100.0, 101.49, 103.0, 104.53, 106.09, 107.67]

```

4. Vamos supor que queiramos computar os gastos do governo (G) do período  $t = 0$  a  $t = 10$ . De  $t = 0$  a  $t = 4$ , temos  $G = 20$ , mas em  $t = 5$  o governo promove um aumento permanente de gastos de três unidades ( $\Delta G = +3,00$ ). Gere uma lista que tenha como elementos os gastos do governo no período.

Solução sugerida:

```
[in]: G = 20.0 # gasto do governo inicial
      gastos = [] # gera uma lista vazia para computar os Gastos do governo
      tempo = list(range(11)) # gera a lista [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
      G_variacao = +3.00
      t_variacao = 5 # tempo em que ocorre a variação permanente de G
      for t in tempo:
          if t == t_variacao:
              G = G + G_variacao
              gastos.append(G)
      print(gastos)
[out]: [20.0, 20.0, 20.0, 20.0, 20.0, 23.0, 23.0, 23.0, 23.0, 23.0, 23.0]
```

### 8.3 Laços do tipo “while”

Assim como os laços do tipo “for \_\_\_\_ in \_\_\_\_” permitem repetir a execução de um bloco determinado, também podemos fazer isto com os laços do tipo “while” através da estrutura geral abaixo:

```
while condição:
    bloco de repetição
```

Nesta estrutura, em “condição” devemos escolher uma condição qualquer de nosso interesse. Sempre que “condição” for verdadeira (*True*), Python irá executar “bloco de repetição”, retomando a execução deste enquanto a “condição” manter-se como *True*. Quando a “condição” tornar-se “*False*”, Python interrompe a repetição e passa para o restante do programa. Note que o “bloco de repetição” deve possuir indentação para que Python identifique as linhas pertencentes ao bloco. Vejamos um exemplo simples como ilustração:

## Exemplo 8.3:

```

[in]: t = 1                # L1
      while t <= 10:        # L2: Condição: t <= 10
          print(t)          # L3: Primeira linha do bloco de repetição
          t = t + 1          # L4: Segunda linha do bloco de repetição
      print("Fim do Programa") # L5: não pertence ao bloco de repetição

[out]: 1
       2
       3
       4
       5
       6
       7
       8
       9
      10
      Fim do Programa

```

No programa acima, temos a condição como “ $t \leq 10$ ”. Com isso, sempre que  $t$  for menor do que ou igual a 10, o “bloco de repetição” (L3 e L4) será executado repetidamente. Note que tanto L3 quanto L4 estão indentados, permitindo que Python possa identificar o bloco a ser repetido. Quando  $t$  passa a valer 11, a condição “ $t \leq 10$ ” passa a ser “*False*” e Python interrompe as repetições, passando para a execução do restante do programa (em nosso exemplo, apenas a linha L5).

Antes de prosseguirmos, vamos ver com um pouco mais de cuidado o que fizemos neste exemplo 8.3. Na primeira linha atribuímos à “ $t$ ” o valor 1. Quando Python lê a segunda linha do programa (`while t <= 10:`), a condição “ $t \leq 10$ ” é considerada verdadeira (*True*) e por isso o bloco de repetição é executado (L3 e L4). Em L3, Python imprime como saída 1, pois  $t = 1$ . Já em L4, adicionamos 1 à  $t$ , e assim  $t$  passa a ser igual a 2. Com o fim das linhas do “bloco de repetição”, Python retorna para o início da estrutura de repetição (L2) e avalia novamente a condição “ $t \leq 10$ ”. Como  $t = 2$ , a condição é avaliada como *True* e Python executa novamente o bloco de repetição, imprimindo 2 como saída em L3 e somando 1 à  $t$  em L4 (agora,  $t$  passa a valer 3). Python, então, repete este *loop* até que  $t$  assumo o valor 11. Quando isto ocorrer, a

condição “ $t \leq 10$ ” é avaliada como *False* e o bloco de repetição deixa de ser executado. A partir daí, Python passa para o restante do programa (neste caso, apenas L5). Note que se não tivéssemos colocado a linha L4, a variável  $t$  seria sempre 1 e a condição “ $t \leq 10$ ” seria sempre verdadeira. Neste caso, Python continuaria executando o *loop* indefinidamente.

#### 8.4 Exercícios Resolvidos

5. (Repetição do exercício 1 do capítulo 5 “Operações matemáticas básicas”)

Dados:

$L = 100$ , o número de trabalhadores de um determinado país no tempo  $t = 0$

$g = 3\%$ , o crescimento do número de trabalhadores a cada período

Utilizando o laço de repetição do tipo “*while*”, compute o número de trabalhadores deste país a cada período, de  $t = 0$  a  $t = 5$ .

Solução sugerida:

```
[in]: L = 100 # número inicial de trabalhadores em t = 0
      g = 0.03 # taxa de crescimento de número de trabalhadores por período
      trabalho = [ ] # Gera uma lista vazia
      tempo = 0
      while tempo <= 5:
          trabalho.append(L)
          L = L*(1+g)
          tempo = tempo + 1
      print(trabalho)
```

[out]: [100, 103.0, 106.09, 109.2727, 112.550881, 115.92740743]

## Capítulo 9 Importação de bibliotecas

Ao escrever um programa com alguma aplicação real, dificilmente podemos fazê-lo apenas com funções nativas. Na maior parte das vezes (na realidade, acreditamos que em todas as vezes), precisamos importar bibliotecas em nossos programas para utilizar suas funções e ferramentas.

Python possui uma vasta quantidade de bibliotecas para aplicações específicas. Boa parte da aprendizagem desta linguagem envolve adquirir o domínio de bibliotecas que possibilitam fazermos o que queremos fazer. Neste capítulo, veremos como importar essas bibliotecas para nosso programa, o que torna suas funções disponíveis para uso. Também veremos como instalar bibliotecas caso estas ainda não estejam prontas para a importação.

### 9.1 Bibliotecas já instaladas

Ao instalar o Anacondas, o pacote já vem com muitas bibliotecas importantes previamente instaladas. Para utilização destas, apenas precisamos de alguns comandos de importação em nosso programa.

Para saber se uma determinada biblioteca já está pré-instalada, basta escrever os comandos de importação. Caso Python não apresente erro na execução, a biblioteca já está instalada. Caso ocorra erro na execução da importação, será necessário realizar sua instalação (veremos no item 9.2).

Em todos os programas que quisermos utilizar uma determinada biblioteca, esta deverá ser importada, independentemente se já tivermos eventualmente feito sua importação em outro programa escrito anteriormente.

Existem diferentes maneiras para importar bibliotecas em um programa. Iremos utilizar a biblioteca `numpy` e uma de suas funções (função `zeros()`) para exemplificar como isso pode ser feito. Não estaremos interessados em saber o que a função `zero()` de `numpy` faz, apenas a utilizaremos para mostrar como importamos bibliotecas e como podemos acessar suas funções. Veremos três formas alternativas de importação e suas peculiaridades.

#### 9.1.1 `import nome_biblioteca`

Podemos utilizar o código com a estrutura “`import nome_biblioteca`” (sem as aspas) para acessar as ferramentas de uma determinada biblioteca. Vejamos abaixo como isto é feito com a biblioteca `numpy` como exemplo:



### Exemplo 9.1

```
[in]: import numpy    # L1
      x = numpy.zeros((2,2))    # L2
      print(x)    #L3
[out]:
array([[0., 0.],
       [0., 0.]])
```

Em L1 importamos a biblioteca numpy para o nosso programa. A partir disto, podemos dispor de suas funções. Em L2, utilizamos a função “zeros” de numpy, passando como argumento (2,2). Note que devemos colocar o nome da biblioteca antes da função, separando-os por um ponto. Isto informa para Python que “zeros( )” é uma função da biblioteca numpy. Em L3 apenas imprimimos como saída do programa a variável x gerada em L2<sup>18</sup>.

#### 9.1.2 import *nome\_biblioteca as apelido*

Alternativamente, podemos utilizar a estrutura “import *nome\_biblioteca as apelido*”. Vejamos um exemplo:

### Exemplo 9.2

```
[in]: import numpy as np    # L1
      x = np.zeros((2,2))    # L2
      print(x)
[out]:
array([[0., 0.],
       [0., 0.]])
```

Em L1, fazemos a importação da biblioteca numpy e estabelecemos um “apelido” (ou *alias*) para ela. Note que a diferença em relação à forma anterior de importação é que, desta vez, incluímos a expressão “as np”, sendo np o apelido dado à numpy.

---

<sup>18</sup> Lembre-se: aqui, não nos preocupamos com o que faz a função zeros( ) do numpy. Apenas queremos fornecer um exemplo de como importar uma biblioteca para uso de suas funções.

Quando utilizamos a estrutura de importação de bibliotecas vista acima temos uma outra diferença importante em relação ao visto no item anterior: agora, para chamar uma função da biblioteca que importamos, devemos escrever o nome da função precedido do apelido que demos para a biblioteca, separados por um ponto. Podemos ver isto em L2: o código para chamarmos a função “zeros” do numpy é na forma “np.zeros( )”, pois “np” é o apelido dado ao importarmos a biblioteca na linha L1.

### 9.1.3 *from nome\_biblioteca import função*

Também podemos usar a estrutura “from *nome\_biblioteca* import *função*”, como abaixo:

#### Exemplo 9.3

```
[in]: from numpy import zeros      # L1
      x = zeros((2,2))  #L2
      print(x)
[out]:
array([[0., 0.],
       [0., 0.]])
```

Com esta estrutura, importamos da biblioteca numpy apenas a função que especificamos (no exemplo, a função zeros). Dessa forma, não podemos utilizar outras funções da biblioteca, a menos que seja realizada também sua importação<sup>19</sup>.

Destacamos abaixo duas importantes diferenças em relação às importações de bibliotecas vistas nos itens anteriores:

- 1) Com esta estrutura de importação (“from *nome\_biblioteca* import *função*”) importamos apenas uma determinada função. Desta forma, naturalmente, já devemos selecionar previamente as funções da biblioteca que deverão ser utilizadas ao longo do programa;
- 2) Ao chamarmos a função da biblioteca importada, não precisamos escrever o nome da biblioteca anteriormente à função, como podemos ver na linha L2. Apesar disso nos permitir

---

<sup>19</sup> Para ter acesso a mais de uma função da biblioteca em um programa, deveríamos especificar todas através da estrutura “from *nome\_biblioteca* import *função 1*, *função 2*, ..., *função n*”, substituindo *nome\_biblioteca* pela biblioteca e *função 1*, *função 2*, ..., *função n* pelos nomes das funções de interesse.

economizar um pouco na escrita, também pode causar confusão quando duas ou mais bibliotecas utilizadas em um programa possuem funções com o mesmo nome.

#### 9.1.4 Nota

É importante notarmos as especificidades no uso de cada forma de importação das bibliotecas. Comparando os três exemplos, podemos ver: quando utilizamos as duas primeiras formas de importação da biblioteca, antes da função devemos colocar o nome da biblioteca à qual ela pertence (na primeira forma temos `numpy.zeros()` e, na segunda, `np.zeros()`). Já na terceira forma de importação, usamos a função sem que ela seja antecedida por qualquer referência à biblioteca. A vantagem das duas primeiras formas de importação é que nosso programa apresentará explicitamente as bibliotecas às quais pertencem as funções utilizadas, diminuindo tanto as dificuldades de leitura do programa quanto as possibilidades de erros.

## 9.2 Bibliotecas ainda não instaladas

Ao tentar importar alguma biblioteca para uso em nosso programa, podemos obter como retorno um erro. Caso o código de importação esteja correto, provavelmente o erro decorra da tentativa de importar uma biblioteca que ainda não está instalada. Neste caso, devemos instalá-la. Lembre-se que o pacote Anaconda que sugerimos para uso do Python já vem com muitas bibliotecas previamente instaladas, mas certamente não todas.

No sistema operacional Windows<sup>20</sup>, a instalação de uma biblioteca é feita pelos seguintes passos (para isto, o Anaconda deve estar instalado):

- 1) Clique no botão iniciar do Windows e encontre a pasta Anaconda;
- 2) Abra o “Anaconda Powershell Prompt”;
- 3) No prompt, digite a seguinte estrutura:

```
pip install nome_biblioteca
```

em que, no lugar de *nome\_biblioteca*, devemos escrever o nome da biblioteca que estamos interessados. Após a instalação, a biblioteca estará disponível para uso e o comando de importação deverá funcionar corretamente.

---

<sup>20</sup> Para outros sistemas operacionais, como o Linux e o macOS da Apple, os comandos para instalação das bibliotecas podem ser facilmente encontrados em buscas na internet.

Caso desejarmos visualizar as bibliotecas já instaladas, seguimos os 2 primeiros passos acima e digitamos no prompt o seguinte código:

```
conda list
```

Com isto, obteremos a lista das bibliotecas que já estão instaladas e suas respectivas versões.

Por fim, também é possível atualizar a versão das bibliotecas para a mais recente. Para isto, siga os passos 1 e 2 para abrir o “Anaconda Powershell Prompt” e digite no prompt a seguinte estrutura:

```
pip install -U nome_biblioteca
```

em que, novamente, devemos substituir na estrutura acima *nome\_biblioteca* pelo nome da biblioteca de interesse. Após isto, a atualização da biblioteca será executada.

## Capítulo 10 Introdução à elaboração de gráficos em Python

Neste último capítulo, vamos apresentar uma breve introdução à geração de gráficos em Python. Incluímos este assunto pois os gráficos são instrumentos importantes no estudo teórico e empírico da economia.

Existem diversas bibliotecas específicas para produção de gráficos em Python, dentre elas temos:

- Matplotlib
- Seaborn
- Plotly
- Bokeh
- Altair
- ggplot
- Vpython
- networkx

O Matplotlib é a biblioteca pela qual, geralmente, o assunto sobre gráficos em Python é introduzido. Na realidade, um submódulo do matplotlib, o pyplot. Este possui diversas funções que nos permitem criar e modificar gráficos. A sua importação é feita utilizando os códigos vistos no capítulo anterior.

Iremos apresentar a criação de gráficos em Python transformando as respostas de alguns exercícios que fizemos ao longo do livro para forma gráfica. Por isso, vamos direto aos exercícios resolvidos, os quais utilizaremos para descrever como alguns comandos podem gerar gráficos e, também, formatá-los.

### 10.1 Exercícios Resolvidos

1. (Repetição do exercício 1 do capítulo 5 “Operações matemáticas básicas”)

Dados:

$L = 100$ , o número de trabalhadores de um determinado país no tempo  $t = 0$

$g = 3\%$ , o crescimento do número de trabalhadores a cada período

Compute o número de trabalhadores deste país a cada período, de  $t = 0$  a  $t = 5$ .

Solução sugerida 1:

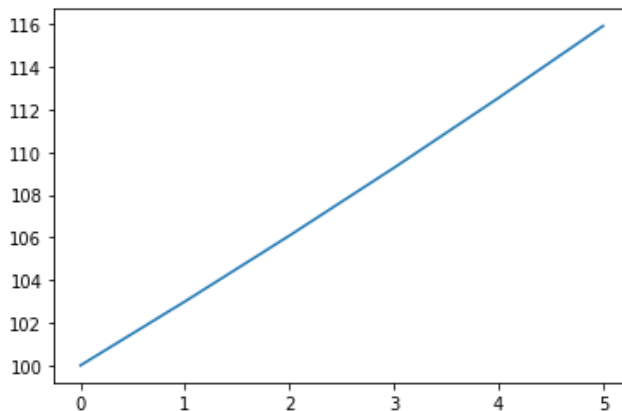
[in]: import matplotlib.pyplot as plt # L1: importa biblioteca para trabalhar com gráfico

```
trabalho = [ ] # L2: gera uma lista vazia
g = 0.03 # L3
L = 100 # L4: número inicial de trabalhadores em t = 0
tempo = [0, 1, 2, 3, 4, 5] # L5

for t in tempo: # L6
    trabalho.append(L) # L7
    L = L*(1 + g) # L8: número inicial de trabalhadores em t = 1

plt.plot(tempo, trabalho) # L9
```

[out]:



Em L1, importamos um submódulo da biblioteca matplotlib, o pyplot (por isso matplotlib.pyplot). Os códigos utilizados de L2 a L8 já são familiares e geram uma lista com a quantidade de trabalhadores do país em cada ponto do tempo (de  $t = 0$  a  $t = 5$ ). O código em L9 gera o gráfico da saída. Nele usamos a função plot do pyplot, o submódulo da biblioteca matplotlib que importamos em L1. Note que passamos dois argumentos para a função plot: primeiro a lista tempo e, segundo, a lista trabalho. Isto fará com que os valores da lista “tempo” constem no eixo horizontal do gráfico e os valores da lista “trabalho” no eixo vertical.

Solução sugerida 2: Incluindo algumas personalizações no gráfico

[in]: import matplotlib.pyplot as plt # L1: importa biblioteca para trabalhar com gráfico

trabalho = [ ] # L2: gera uma lista vazia

g = 0.03 # L3

L = 100 # L4: número inicial de trabalhadores em  $t = 0$

tempo = [0, 1, 2, 3, 4, 5] # L5: Poderíamos usar: list(range(6))

for t in tempo: # L6

trabalho.append(L) # L7

L = L\*(1 + g) # L8: número inicial de trabalhadores em  $t = 1$

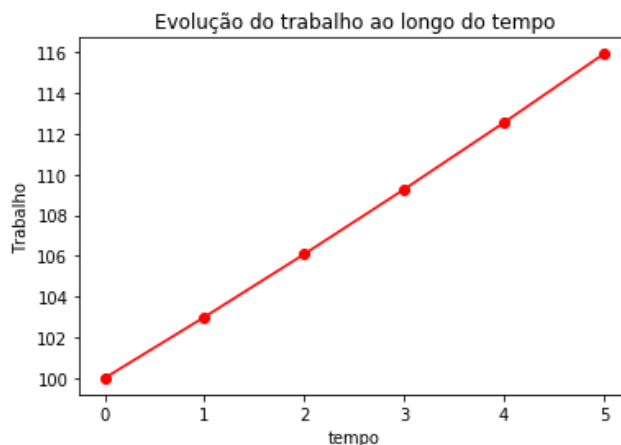
plt.plot(tempo, trabalho, color='red', marker='o') # L9

plt.xlabel("tempo") # L10

plt.ylabel("Trabalho") # L11

plt.title('Evolução do Trabalho ao Longo do Tempo') # L12

[out]:



Primeiramente, note que de L1 a L8 temos os mesmos códigos da “Solução sugerida 1”. No entanto, a partir de L9 incluímos alguns comandos que possibilitam a personalização do gráfico. Vejamos abaixo o que fizemos:

Linha L9:

Incluimos dois argumentos opcionais, “color” e “marker”. Para color, passamos a *string* “red”, o que torna a linha do gráfico vermelha. Em “marker”, passamos “o”, o que acrescenta uma “bola” nos pontos dos dados.

Linha L10:

Utilizamos a função `xlabel()` do `pyplot`. A *string* “tempo” passada como argumento para essa função aparece abaixo do eixo horizontal do gráfico.

Linha L11:

Utilizamos a função `ylabel()` do `pyplot`. A *string* “Trabalho” passada como argumento para essa função aparece ao lado do eixo vertical do gráfico.

Linha L12:

Utilizamos a função `title()` do `pyplot`. A *string* “Evolução do Trabalho ao Longo do Tempo” passada como argumento para essa função aparece acima do gráfico.

Existem muitas outras características gráficas que podemos modificar ou acrescentar. Para conhece-las, recomendamos ler a documentação do `matplotlib.pyplot`<sup>21</sup>.

## 2. (Repetição do exercício 2 do capítulo 5 “Operações matemáticas básicas”)

Considere uma economia com as seguintes características:

$$\rightarrow \text{PIB} = Y = F(K, L) = (K * L)^{\alpha}$$

$$\rightarrow \alpha = 0.5$$

$$\rightarrow K = 100 \quad (\text{Estoque de capital da economia em unidades reais})$$

$$\rightarrow L = 100 \quad (\text{Quantidade de trabalho em } t = 0)$$

$$\rightarrow g = 0.03 \quad (\text{taxa de crescimento de número de trabalhadores por período: 3\%})$$

A partir dos dados acima, compute o PIB do país de  $t = 0$  a  $t = 5$

Solução sugerida:

---

<sup>21</sup> Para encontrar a documentação completa, basta pesquisar no Google por “matplotlib.pyplot”.



```

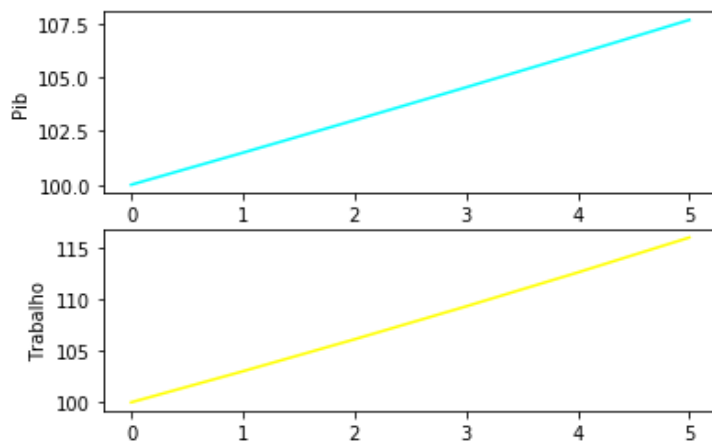
[in]: import matplotlib.pyplot as plt # L1: importa biblioteca para trabalhar com gráfico
      trabalho = [ ]                  # L2: gera uma lista vazia
      pib = [ ]                      # L3: gera uma lista vazia
      alpha = 0.5                    # L4
      K = 100                        # L5: Estoque de capital da economia
      g = 0.03                       # L6: crescimento do nº de trabalhadores a cada período
      L = 100                        # L7: número inicial de trabalhadores em t = 0
      tempo = [0, 1, 2, 3, 4, 5]    # L8

      for t in tempo:                # L9
          Y = (K*L)**alpha           # L10
          pib.append(Y)              # L11
          trabalho.append(L)         # L12
          L = L*(1+g)                # L13

      figura, (graf1, graf2) = plt.subplots(nrows=2, ncols=1) # L14
      graf1.plot(tempo, pib, color= "cyan")                    # L15
      graf1.set_ylabel('Pib')                                  # L16
      graf2.plot(tempo, trabalho, color= "yellow")             # L17
      graf2.set_ylabel('Trabalho')                             # L18

```

[out]:



Neste exercício, utilizamos a função `subplots()` do `pyplot` (submódulo do `matplotlib`). Com ela, podemos criar uma figura composta por vários gráficos.

Em nosso programa, as linhas L1 a L13 possuem códigos já conhecidos. A partir da linha L14 até L18, começamos a construir os dois gráficos que compõem a saída do programa. Vejamos o que faz cada uma dessas linhas.

Linha L14:

Nesta linha, utilizamos a função `subplots` do `matplotlib.pyplot` para gerar uma figura com dois gráficos. Essa função tem como retorno uma tupla de dois elementos, sendo que o primeiro chamamos de “figura” e o segundo de “(graf1, graf2)”. Fazemos aqui o desempacotamento de tuplas, como visto no capítulo 4 (item 4.3.2 “Desempacotando” uma tupla).

Observe que passamos dois argumentos para a função `subplots`, o `nrows=2` e o `ncols=1`. Isto indica que queremos duas linhas e uma coluna. Assim, com este código, Python gera uma figura composta por dois gráficos alinhados em duas linhas e uma coluna.

Linha L15:

Em L15 aplicamos o método `plot()` à “graf1” passando três argumentos: `tempo`, `pib` e `color="cyan"`. O primeiro informa para Python que queremos que a lista `tempo` seja colocada no eixo horizontal, o segundo que a lista `pib` seja colocada no eixo vertical e o terceiro que a linha do gráfico seja da cor ciano. Note que “graf1” diz respeito ao gráfico superior da figura.

Linha L16:

Em L16 aplicamos o método `set_ylabel()` à “graf1”. A *string* passada como argumento informa o texto que queremos no eixo vertical do gráfico.

Linha L17:

Em L17 aplicamos o método `plot()` à “graf2”. Os argumentos seguem o que foi visto na linha L15. Repare que “graf2” se refere ao gráfico que está na parte inferior da figura.

Linha L18:

Em L18 aplicamos o método `set_ylabel()` à “graf2”. Assim como em L16, o argumento informa o texto que queremos no eixo vertical do gráfico.

3. Considere uma economia que se comporta de acordo com o seguinte modelo (modelo keynesiano básico):

**Variáveis exógenas:**

- $c_0$ : Consumo autônomo
- $c_1$ : propensão marginal a consumir ( $0 < c_1 < 1$ )
- $\bar{I}$ : Investimento
- $\bar{G}$ : Gastos do Governo
- $\bar{T}$ : Impostos (líquido de transferências)

**Composição do Modelo:**

$$[1] Y = \frac{1}{(1-c_1)} * (c_0 + \bar{I} + \bar{G} - c_1 * \bar{T}) \quad (\text{Equilíbrio no mercado de bens: Relação IS})$$

$$[2] C = c_0 + c_1 * (Y - \bar{T}) \quad c_0 > 0 ; 0 < c_1 < 1 \quad (\text{Função Consumo})$$

$$[3] I = \bar{I} \quad (\text{Investimento Privado})$$

$$[4] G = \bar{G} \quad (\text{Gastos do Governo})$$

$$[5] T = \bar{T} \quad (\text{Impostos})$$

$$[6] S = Y - \bar{T} - C \quad (\text{Poupança Privada})$$

$$[7] S_G = \bar{T} - \bar{G} \quad (\text{Poupança Pública})$$

$$[8] S_T = S + S_G \quad (\text{Poupança Total para economias fechadas})$$

$$[9] \bar{I} = S_T = S + S_G$$

$$k = \frac{1}{(1-c_1)} \quad (\text{Multiplicador Keynesiano})$$

**Parâmetros da economia:**

$$c_0 = 100,00 \quad (\text{consumo autônomo})$$

$$c_1 = 0,6 \quad (\text{propensão marginal a consumir})$$

$$\bar{I} = 30,00$$

$$\bar{G} = 50,00$$

$$\bar{T} = 50,00$$

Considere que em  $t = 5$  o governo promova um aumento permanente de gastos em 1 unidade (ou seja,  $\Delta G = 1,0$ ). Compute o valor dos seguintes componentes do modelo no período de  $t = 0$  a  $t = 10$ :

- Produto
- Consumo
- Investimento
- Gastos do Governo
- Poupança Privada
- Poupança Pública

**Solução sugerida:**

```
[in]: import matplotlib.pyplot as plt
```

```
c0 = 100.0
```

```
c1 = 0.6
```

```
Ibarra = 30.0
```

```
Gbarra = 50.0
```

```
Tbarra = 50.0
```

```
choque_governo = + 1.0
```

```
Produto=[]
```

```
Consumo=[]
```

```
Investimento=[]
```

```
Governo=[]
```

```
Imposto=[]
```

```
Poupanca_Privada=[]
```

```
Poupanca_Publica=[]
```

```
tempo = list(range(11)) # Tempo de 0 a 10
```

```
for t in tempo:
```

```
    if t == 5:
```

```
        Gbarra = Gbarra + choque_governo
```

```

Y = (1/(1-c1))*(c0+Ibarra+Gbarra-c1*Tbarra)
C = c0+c1*(Y-Tbarra)
S = Y-Tbarra-C      #poupança privada
Sg = Tbarra-Gbarra   #poupança pública
Produto.append(Y)
Consumo.append(C)
Investimento.append(Ibarra)
Governo.append(Gbarra)
Imposto.append(Tbarra)
Poupanca_Privada.append(S)
Poupanca_Publica.append(Sg)

fig, (graf1, graf2, graf3, graf4, graf5, graf6, graf7) = plt.subplots(nrows=7,
                                                                    ncols=1,
                                                                    figsize=(7.0, 20.0))

graf1.plot(tempo, Produto)
graf1.set_ylabel("Produto")
graf2.plot(tempo, Consumo)
graf2.set_ylabel("Consumo")
graf3.plot(tempo, Investimento)
graf3.set_ylabel("Investimento")
graf4.plot(tempo, Governo)
graf4.set_ylabel("Gastos do Governo")
graf5.plot(tempo, Imposto)
graf5.set_ylabel("Imposto")
graf6.plot(tempo, Poupanca_Privada)
graf6.set_ylabel("Poupança Privada")
graf7.plot(tempo, Poupanca_Publica)
graf7.set_ylabel("Poupança Publica")

print(f'Produto: {Produto}')
print(f'Consumo: {Consumo}')
print(f'Investimento: {Investimento}')
print(f'Gastos do Governo: {Governo}')

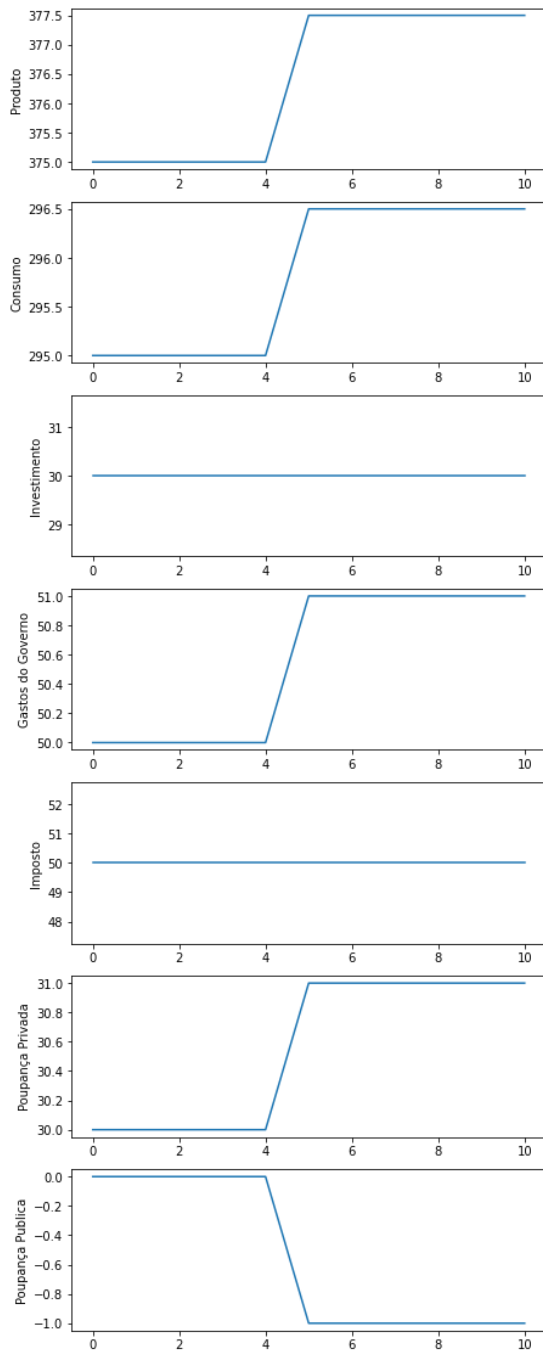
```

```

print(f"Impostos: {Imposto}")
print(f"Poupança Privada: {Poupanca_Privada}")
print(f"Poupança Pública: {Poupanca_Publica}")

```

[out]:



```

Produto: [375.0, 375.0, 375.0, 375.0, 375.0, 377.5, 377.5, 377.5, 377.5, 377.5, 377.5]
Consumo: [295.0, 295.0, 295.0, 295.0, 295.0, 296.5, 296.5, 296.5, 296.5, 296.5, 296.5]
Investimento: [30.0, 30.0, 30.0, 30.0, 30.0, 30.0, 30.0, 30.0, 30.0, 30.0, 30.0]
Gastos do Governo: [50.0, 50.0, 50.0, 50.0, 50.0, 51.0, 51.0, 51.0, 51.0, 51.0, 51.0]
Impostos: [50.0, 50.0, 50.0, 50.0, 50.0, 50.0, 50.0, 50.0, 50.0, 50.0, 50.0]
Poupança Privada: [30.0, 30.0, 30.0, 30.0, 30.0, 31.0, 31.0, 31.0, 31.0, 31.0, 31.0]
Poupança Pública: [0.0, 0.0, 0.0, 0.0, 0.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0]

```

Neste exercício aplicamos grande parte do que vimos ao longo de todo o livro. Como não há novidades<sup>22</sup>, não incluiremos explicações pois consideramos que todos os comandos utilizados na sua resolução já sejam conhecimento adquirido. Apenas observamos que não há uma única forma de escrever um programa para atingir um determinado objetivo e poderíamos reescrever o código acima utilizando as diferentes ferramentas que aprendemos. Conforme vamos avançando nos estudos percebemos que algumas características como eficiência, formatação e clareza dos códigos são importantes atributos e devem balizar nossas escolhas de como escrever um programa da melhor forma possível.

---

<sup>22</sup> Na realidade, existe uma única novidade. Ao chamar a função `subplots` do `matplotlib.pyplot`, utilizamos três argumentos. Os dois primeiros (`nrows` e `ncols`) já foram vistos. O terceiro (`figsize`) foi incluído para modificar o tamanho da figura. Note que atribuímos à `figsize` uma tupla composta de dois elementos do tipo *float*. De forma geral temos `figsize=(largura, altura)`, com largura e altura em polegadas.

## Posfácio

Neste livro apresentamos uma introdução à linguagem de programação Python tão direta quanto achamos possível para que o estudante da área de economia tenha uma base para aplicação imediata em seus estudos e para que tenha condições de prosseguir para tópicos mais avançados.

No que se refere ao estudo da linguagem Python em si, recomendamos que os próximos estudos envolvam aprofundamentos nos tópicos relacionados às Funções e Classes. Para esses assuntos, a leitura dos capítulos 8 (Funções) e 9 (Classes) do livro de Matthes (2016) é bastante proveitosa. Também é interessante que o estudante se habitue a ler as documentações das bibliotecas que tem interesse. Elas são facilmente encontradas na internet. Nessas documentações podemos encontrar os detalhes para o melhor uso e aproveitamento dos recursos que as bibliotecas oferecem.

Já no que diz respeito à economia, existem diversas aplicações atraentes e vantajosas de Python, tanto na área teórica como aplicada (em finanças, por exemplo, mas não só). Certamente, a análise de dados é a aplicação mais imediata e geralmente é a porta de entrada para maiores aprofundamentos no uso de Python por economistas. Dentro deste campo (de análise de dados), podemos utilizar Python, por exemplo, em econometria (regressão e testes estatísticos) e séries temporais (testes de raiz unitária, diferenciação de séries, etc). Para isto é imprescindível o conhecimento da biblioteca Pandas, que possui muitas ferramentas para análise de dados<sup>23</sup>. Também as bibliotecas para geração de gráficos, como Matplotlib e outras destacadas na introdução deste livro, são básicas para qualquer tipo de aplicação. Assim, estudar estas bibliotecas, em paralelo com os tópicos que recomendamos no parágrafo anterior, é um bom caminho para prosseguir nos estudos.

---

<sup>23</sup> Pandas nos possibilita, por exemplo, acessar dados de programas como o excel e também da internet. Isto nos permite trabalhar com imensas quantidades de dados para análise.



**Bibliografia e cursos consultados**

MATTHES, Eric. Curso Intensivo de Python: Uma Introdução Prática e Baseada em Projetos. São Paulo: Novatec Editora Ltda: 2016.

MCKINNEY, Wes. Python para Análise de Dados. São Paulo: Novatec Editora Ltda, 2018.

MENEZES, N. N. C. Introdução à Programação com Python: Algoritmos e Lógica de Programação para Iniciantes. São Paulo: Novatec Editora Ltda, 2014.

SANTOS, Rafael. Python – Aprenda os Fundamentos. Curso disponível em: [www.udemy.com](http://www.udemy.com)

SWEIGART, Al. Automatize Tarefas Maçantes com Python: Programação Prática para Verdadeiros Iniciantes. São Paulo: Novatec Editora Ltda: 2015.