MARCO ANTONIO LEONEL CAETANO

PYTHON E MERCADO FINANCEIRO

Programação para estudantes, investidores e analistas

2ª edição
revista e ampliada

Blucher

Marco Antonio Leonel Caetano

PYTHON E MERCADO FINANCEIRO

Programação para estudantes, investidores e analistas

2ª edição

Python e mercado financeiro: programação para estudantes, investidores e analistas, 2. ed.

Editora Edgard Blücher Ltda.

Publisher Edgard Blücher
Editor Eduardo Blücher
Coordenação editorial Rafael Fulanetti
Coordenação de produção Ana Cristina Garcia
Diagramação Guilherme Salvador
Revisão de texto Maurício Katayama
Capa Juliana Midori Horie
Imagem da capa iStockphoto

Blucher

Rua Pedroso Alvarenga, 1245, 4º andar
04531-934 – São Paulo – SP – Brasil
Tel.: 55 11 3078-5366
contato@blucher.com.br
www.blucher.com.br

Segundo o Novo Acordo Ortográfico, conforme 6. ed. do Vocabulário Ortográfico da Língua Portuguesa, Academia Brasileira de Letras, julho de 2021.

Dados Internacionais de Catalogação na Publicação (CIP)
Heytor Diniz Teixeira, CRB-8/10570

Caetano, Marco Antonio Leonel

Python e mercado financeiro : programação para estudantes, investidores e analistas / Marco Antonio Leonel Caetano. – 2. ed. – São Paulo : Blucher, 2026.

664 p. : il.

Bibliografia

ISBN 978-85-212-2846-2 (Impresso)

ISBN 978-85-212-2847-9 (Eletrônico – Epub)

ISBN 978-85-212-2844-8 (Eletrônico – PDF)

1. Python (Linguagem de programação de computador). 2. Mercado financeiro. 3. Algoritmos. 4. Negociação algorítmica. 5. Negociações financeiras – Python. 6. Investimentos. I. Título.

CDU 004.43:336.76

Índice para catálogo sistemático:
1. Mercado financeiro 336.76

CONTEÚDO

Parte I

PROGRAMAÇÃO BÁSICA DE ALGORITMOS

CAPÍTULO 1

PRINCÍPIOS DE PROGRAMAÇÃO

1.1 INTRODUÇÃO

O Anaconda possui diversos ambientes de programação para a linguagem Python ser executada. Entre os mais usados estão Jupyter e Spyder. Este livro é desenvolvido todo no ambiente de programação Spyder, um dos aplicativos para os algoritmos, conforme mostrado na Figura 1.1 (acessado no site https://www.anaconda.com/download).

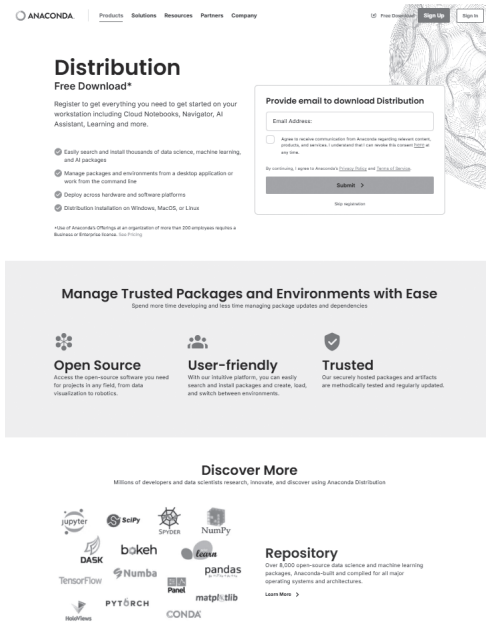


Figura 1.1 – Anaconda, *website* que disponibiliza o Python.

Passo a passo da instalação do Anaconda[1]

Quando se inicia o Spyder, observam-se três janelas diferentes para execução e programação dos algoritmos. A primeira janela à esquerda na Figura 1.2 é onde os algoritmos (ou script) são programados para rodar com diversos comandos ou funções. A janela acima e à direita é o explorador de arquivos ou variáveis. Essa janela serve para observar quais arquivos estão no diretório em que o programa está sendo executado. Outra função dessa janela é averiguar quais valores estão sendo adotados pelas variáveis, ou quais variáveis estão sendo corretamente utilizadas pelo programa. Possíveis erros podem ser detectados analisando as variáveis de um algoritmo com a utilização dessa janela.

A última janela abaixo e à direita é o chamado **Console**. É nele que os programas podem ser executados, as instalações das bibliotecas são feitas com o comando **pip install** ou são realizadas programações simples, como operações, leituras de arquivos, impressões em arquivos ou gráficos simples.

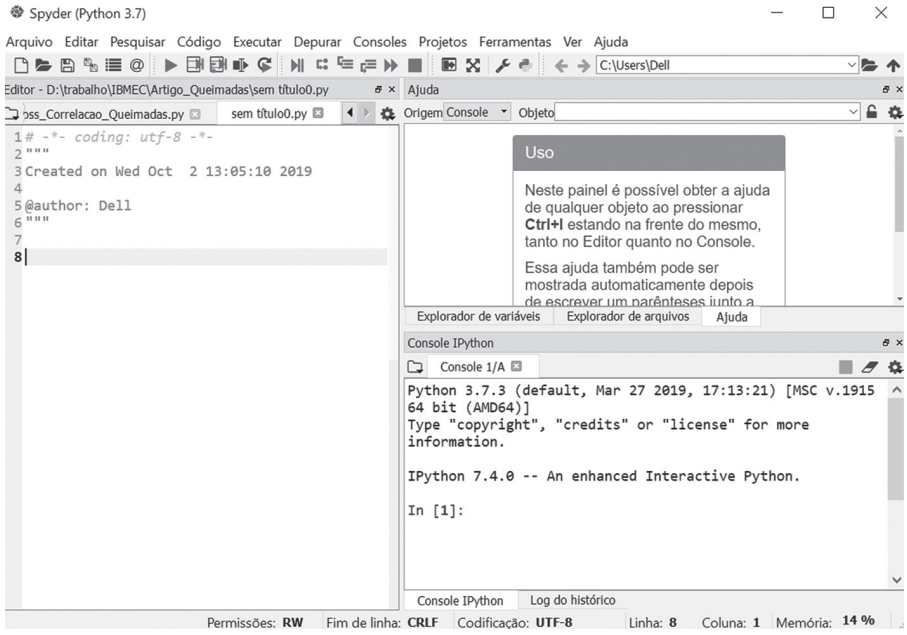


Figura 1.2 – Ambiente de programação do Python no Spyder.

1 Para acessar os vídeos, aponte a câmera do seu celular para os QR Codes. Em alguns aparelhos, o link aparecerá automaticamente na tela, bastando clicar sobre ele para ser direcionado ao vídeo. Em outros, pode ser necessário tirar uma foto do código e usar um aplicativo para o reconhecimento, como o Google Lens.

A barra de tarefas na parte superior é importante para abrir arquivos com algoritmos já digitados, salvar arquivos e executar os algoritmos. A execução de um algoritmo escrito no modo de script na janela à esquerda se dá pressionando o botão ▶. Outra maneira de executar um programa em Python é usar a aba com o conjunto de ações, na parte superior da barra de tarefas. Ao pressionar **Executar** ou a tecla **F5** do computador, como apresentado na Figura 1.3, uma série de ações podem ser realizadas, como a execução do programa.

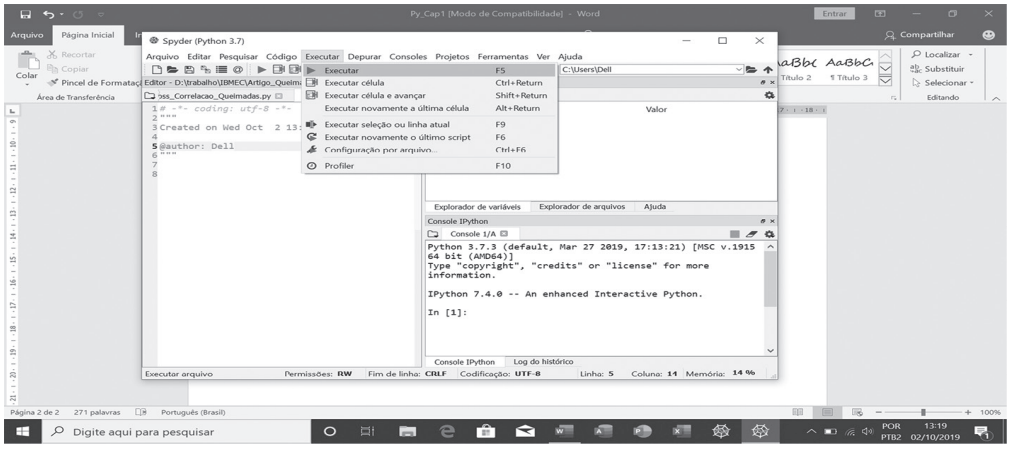


Figura 1.3 – Barra de tarefas para a execução de um programa.

Na utilização do **Console**, números com colchetes indicam quantas linhas de comando já foram utilizadas, como **In [1]:**, **In [2]:**, **In [3]:** etc. Quando o **Console** ficar com muitas linhas de programação, você pode limpá-lo escrevendo a palavra "clear" em qualquer linha. Outras vezes, muitas programações erradas podem travar o Python.

A utilização de **Ctrl+D** ajuda a reconfigurar todo o console, limpando e reiniciando automaticamente a área de programação. Outra maneira de limpar a memória é acessando a palavra **Consoles** na aba superior e clicando em **Reiniciar kernel**.

1.2 OPERAÇÕES SIMPLES NO CONSOLE

Operações simples e rápidas podem ser realizadas na área Console, janela inferior à direita na Figura 1.2. Somas, subtrações, divisões e chamadas de funções pré-programadas no Python podem rodar independentemente de qualquer algoritmo ou programa.

```
In [1]: 1+1
Out[1]: 2

In [2]: 3/2
Out[2]: 1.5

In [3]: 4-10
Out[3]: -6
```

A operação de potenciação deve usar duplo asterisco para obter corretamente o resultado. Assim, por exemplo, 2^3 torna-se "2**3". O resto da divisão entre dois números utiliza o símbolo **%**. Logo, o resto da divisão de 10 por 3 deve ser representado por "10 % 3". No Python, as operações ficam no **Console**, como apresentado a seguir.

```
In [4]: 2**3
Out[4]: 8

In [5]: 10 % 3
Out[5]: 1
```

Uma das noções mais importantes em computação tem relação com o armazenamento de dados em variáveis. Desde seu início, quando a computação ainda era apenas um ramo dentro da matemática, o uso de variáveis para salvar resultados, realizar operações na memória ou solucionar problemas é parte fundamental do processo de construção de um algoritmo. Então, por exemplo, para *x* e *y*:

```
In [6]: x=5

In [7]: y=10
```

Quando digitamos essas linhas assumindo os valores de 5 e 10 para *x* e *y*, ao pressionar **Enter** o resultado não aparece. Para visualizar todo resultado de variável ou de um conjunto de variáveis, deve-se utilizar o comando de impressão no **Console**, conhecido como **PRINT( )**.

```
In [8]: print(x)
5

In [9]: print(y)
10
```

Para ver o valor das duas variáveis ao mesmo tempo:

```
In [10]: print(x,y)
5 10
```

Ou pode-se usar a mensagem de texto para identificar *x* e *y*:

```
In [11]: print("x= ",x,"y= ",y)
x=  5 y=  10
```

Operações matemáticas podem ser realizadas dentro do comando de impressão **PRINT( )**, sem a necessidade da criação de novas variáveis. Pode-se ter, então, para algumas operações básicas os resultados diretamente no **Console**, como apresentado a seguir.

```
In [12]: print(x+y)
15

In [13]: print(x-y)
-5


In [14]: print(x/y)
0.5

In [15]: print(x**y)
9765625
```

Introdução ao Python com os primeiros comandos de linha no **Console**

Funções mais elaboradas como exponencial ou logaritmo, em matemática, necessitam de importação do módulo para seu funcionamento.

```
In [16]: import math
```

Entre as muitas funções no Python, pode-se destacar as funções logarítmicas ou exponenciais:

- math.exp(*x*): retorna *e* elevado a *x*.
- math.log2(*x*): retorna o logaritmo de *x* na base 2.
- math.log10(*x*): retorna o logaritmo de *x* na base 10.
- math.log(*x*): retorna o logaritmo natural de *x* (base *e*).
- math.log(*x*,*b*): retorna o logaritmo de *x* na base *b*, apenas quando *b* é diferente de 2 e 10.
- math.pow(*x*,*y*): retorna *x* elevado a *y*.
- math.sqrt(*x*): retorna a raiz quadrada de *x*.

Mas será que realmente é necessário usar a palavra "math" antes das funções? A resposta é sim, porque desse modo o Python identifica o módulo que vai entrar em operação. Caso não use "math", por exemplo, para e^2 tem-se o seguinte erro:

```
In [18]: exp(2)
Traceback (most recent call last):

  File "<ipython-input-18-840a487878a2>", line 1, in <module>
    exp(2)

NameError: name 'exp' is not defined
```

Alguns exemplos do uso das funções anteriores são:

```
In [19]: math.exp(1)
Out[19]: 2.718281828459045

In [20]: math.log(10)
Out[20]: 2.302585092994046

In [21]: math.log2(3)
Out[21]: 1.584962500721156

In [22]: math.pow(2,3)
Out[22]: 8.0

In [23]: math.sqrt(16)
Out[23]: 4.0
```

Também há as funções trigonométricas:

- math.cos(x): retorna o cosseno de x. math.sin(x): retorna o seno de x.
- math.tan(x): retorna a tangente de x.
- math.degrees(x): converte o ângulo x de radianos para graus. math.radians(x): converte o ângulo x de graus para radianos. math.acos(x): retorna o arco cosseno de x.
- math.asin(x): retorna o arco seno de x.
- math.atan(x): retorna o arco tangente de x.

Alguns exemplos com as funções trigonométricas são:

```
In [24]: math.sin(3)
Out[24]: 0.1411200080598672

In [25]: math.cos(3)
Out[25]: -0.9899924966004454

In [26]: math.degrees(3.14)
Out[26]: 179.9087476710785
```

A chamada ou importação de uma biblioteca do Python pode ser simplificada por sua renomeação com um "apelido" que denote a respectiva função. Por exemplo, a biblioteca de matemática math pode ser simplificada em sua chamada usando na frente de seu nome a denominação "as" seguida do apelido.

```
In [1]: import math as mt

In [2]: mt.cos(3)
Out[2]: -0.9899924966004454

In [3]: mt.log(2)
Out[3]: 0.6931471805599453
```

Assim, na importação, o nome da biblioteca foi simplificado para mt. Após essa renomeação, em vez de digitar "math" basta colocar o apelido "mt" e a função desejada (**sin**, **cos**, **log** etc.).

Primeiras operações com a biblioteca math

1.3 LISTAS NO CONSOLE

Lista, no Python, é uma sequência ordenada em que os elementos são sempre associados a um índice. Apesar de lembrar vetores ou matrizes de outras linguagens de programação, uma lista é mais simples e de fácil manipulação com dados numéricos ou *strings* (letras). Como será visto em outras partes do texto, o Python também possui estruturas como vetores ou matrizes, mas seu uso é mais sofisticado do que uma lista.

O uso de listas no **Console** é bem simples e intuitivo. Por exemplo, podemos denominar uma lista com os números "[10,5,1,1,2,2,3,5,10,2,2,1,3,4,4]" no Python desta maneira:

```
In [4]: dados=[10,5,1,1,2,2,3,5,10,2,2,1,3,4,4]
```

Como escrito antes, a lista é baseada em índices. Diferentemente de outras linguagens, no Python uma lista sempre começa com o índice zero, e não com o índice um.

- Primeiro elemento da lista

O primeiro elemento da variável *dados* anterior é "dados[0]". Para ver esse elemento, basta usar o comando **PRINT( )** para a variável em formato de lista "dados".

```
In [5]: print(dados[0])
10
```

- Último elemento da lista

Para ver o último elemento, basta usar "-1" dentro do nome da variável de lista, ou seja:

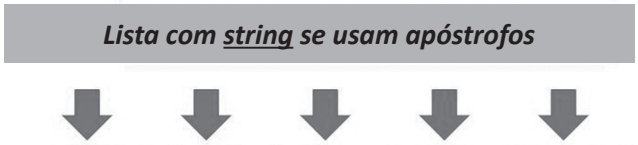
```
In [6]: print(dados[-1])
4
```

- Total de elementos da lista

Para o total de dados de uma lista, utiliza-se o comando **len** ou:

```
In [7]: print(len(dados))
15
```

em que o Python nos mostra que temos na lista "dados" quinze elementos no total. Listas com nomes ou *strings* devem ser compostas com aspas simples para que o Python entenda que está diante de texto.




```
In [6]: mercado=['ações','opções','futuro','dolar','ouro','criptomoedas']

In [7]: print(mercado)
['ações', 'opções', 'futuro', 'dolar', 'ouro', 'criptomoedas']
```

- Fatiamento de uma lista (*slice*)

O interessante de uma lista é seu fatiamento, ou seja, a extração de um elemento ou um conjunto de elementos para estudos em particular. Na lista de nomes "mercado", se é desejado extrair do primeiro ao terceiro elemento, devemos começar no índice "0" e terminar no "3". Esse fato pode gerar uma confusão, pois matematicamente 0,1,2,3 tem quatro elementos, e não os três primeiros.

O fato é que sempre, em uma lista no Python, o seu índice vai de zero até o elemento menos um, ou (n – 1). Assim, por exemplo, em uma lista com:

$$x = [d(0), d(1), d(2), d(3), d(4), ...,d(n)]$$

se desejamos os três primeiros elementos, temos de escolher de zero a três, pois (3 – 1) fornece o índice 2, que totaliza três elementos. Então, no exemplo da lista "mercado":

```
In [10]: print(mercado[0:3])
['ações', 'opções', 'futuro']
```

Os dois-pontos indicam que a lista vai do elemento de índice 0 até o elemento de índice (3 – 1) e por isso os colchetes ficam [0:3]. Assim, é possível fazer um fatiamento de diversas maneiras para separar trechos da lista para estudos em casos particulares. Pode-se, inclusive, fatiar termos e salvar em novas variáveis, criando listas alternativas à original.

Um fatiamento do terceiro ao quinto elemento da lista "mercado" é da seguinte forma:

```
In [11]: print(mercado[2:5])
['futuro', 'dolar', 'ouro']
```

- Buscas em lista (comando **in**)

Muitas vezes, é interessante percorrer uma lista para a verificação de um elemento ou a localização de um elemento particular. Para isso, o comando **in** é interessante, como pode ser visto a seguir. A reposta do **in** é sempre verdade (*True*) ou falso (*False*). Então,

se procuramos as palavras "futuro", "ouro" e "indice" (sem acento) na lista "mercado", o resultado é *True*, *True* e *False*, pois a palavra "indice" não pertence à lista.

```
In [12]: 'futuro' in mercado
Out[12]: True

In [13]: 'ouro' in mercado
Out[13]: True

In [14]: 'indice' in mercado
Out[14]: False
```

A busca por um elemento da lista pode ser salva em variável. Por exemplo, pode-se salvar o resultado (*True*, *False*) para uma variável, por exemplo, denominada de *fut*.

```
In [15]: fut='futuro' in mercado

In [16]: print(" 'futuro' está na lista? ", fut)
 'futuro' está na lista?  True
```

- Alterando elementos de uma lista

Como um elemento da lista é conhecido por seu índice, podemos então alterá-lo, se for o caso. Por exemplo, se desejamos alterar a palavra "futuro" pela palavra "commodity", basta substituir a lista "mercado" com o índice de onde se localiza "futuro". No caso "mercado[2]" troca as palavras desejadas.

```
In [17]: mercado[2]='commodity'

In [18]: print(mercado)
['ações', 'opções', 'commodity', 'dolar', 'ouro', 'criptomoedas']
```

Para alterar mais de uma palavra, usa-se a noção do fatiamento. Por exemplo, para alterar as duas primeiras palavras por "tesouro" e "títulos", o comando deve ser:

```
In [19]: mercado[0:2]=['tesouro','títulos']

In [20]: print(mercado)
['tesouro', 'títulos', 'commodity', 'dolar', 'ouro', 'criptomoedas']
```

O lado direito é uma sublista que é inserida na localização que começa em zero e vai até o índice (2 – 1).

- Adicionando um elemento à lista original (**append**)

Uma lista formada pode receber um ou mais elementos usando o comando **append** com um ponto ao final do nome da lista. Vamos supor que desejamos colocar a palavra "comprar" na lista "mercado".

```
In [21]: mercado.append('comprar')

In [22]: print(mercado)
['tesouro', 'títulos', 'commodity', 'dolar', 'ouro', 'criptomoedas', 'comprar']
```

- Contando repetições (**count**)

Listas muitas vezes podem conter elementos repetidos entre os diversos componentes. Assim, por exemplo, vamos primeiro adicionar a palavra "dolar" ao final da lista "mercado".

```
In [29]: print(mercado)
['tesouro', 'títulos', 'commodity', 'dolar', 'ouro', 'criptomoedas', 'comprar',
'dolar']
```

Agora, usando o comando **count**, podemos pedir ao Python que indique quantas vezes aparece a palavra "dolar" na lista.

```
In [30]: mercado.count('dolar')
Out[30]: 2
```

- Adicionando lista em lista (**extend**)

Em vez de uma palavra, muitas vezes desejamos adicionar outra lista à lista original. Para isso, é necessário usar o comando **extend**. Vamos supor que se deseja adicionar as palavras "Petrobras", "BB" e "Vale" à lista "mercado". Então, devemos proceder da seguinte forma:

```
In [31]: mercado.extend(['Petrobras','BB','Vale'])
```

E usando o comando **PRINT( )**, vemos o resultado final completo.

```
In [32]: print(mercado)
['tesouro', 'títulos', 'commodity', 'dolar', 'ouro', 'criptomoedas', 'comprar',
'dolar', 'Petrobras', 'BB', 'Vale']
```

- Ordenação de uma lista em ordem crescente (**sort**)

Colocar uma lista em ordem alfabética ou crescente é bem fácil, usando o comando **sort** ao final do nome da lista. No entanto, se temos nomes com letras maiúsculas, o Python leva em conta na separação entre letras o tamanho das letras e ordena separadamente. Como a lista adicionada tinha as palavras iniciadas por letras maiúsculas, a ordenação primeiro ordena todas as palavras com letras maiúsculas e, depois, as demais.

```
In [33]: mercado.sort()

In [34]: print(mercado)
['BB', 'Petrobras', 'Vale', 'commodity', 'comprar', 'criptomoedas', 'dolar',
'dolar', 'ouro', 'tesouro', 'títulos']
```

Se ainda assim desejamos colocar em ordem independentemente do tamanho da fonte ou palavra inicial, devemos usar uma chave para que o Python ignore o tamanho e ordene mesmo assim, sem levar em conta o tamanho da letra. A chave é **key**, que deve igualar ao comando de **string** válido para todos os casos de letras: **str.casefold**. Em nossa lista "mercado" com essa chave, a ordenação fica correta.

```
In [35]: mercado.sort(key=str.casefold)

In [36]: print(mercado)
['BB', 'commodity', 'comprar', 'criptomoedas', 'dolar', 'dolar', 'ouro',
'Petrobras', 'tesouro', 'títulos', 'Vale']
```

- Ordenando em ordem inversa (**reverse**)

Igual ao caso anterior, porém aqui se deseja ordenar a lista em ordem inversa. A palavra **reverse** deve estar nos parêntesis seguida de **True**.

```
In [12]: mercado.sort(reverse=True)

In [13]: print(mercado)
['títulos', 'tesouro', 'ouro', 'dolar', 'dolar', 'criptomoedas', 'comprar', 'commodity',
'Vale', 'Petroobras', 'BB']
```

Uma outra alternativa é primeiro fazer a ordenação e depois fazer uma inversão da lista. A inversão da lista não é uma ordenação inversa, mas apenas trocar de lugar os elementos, tornando os últimos como primeiros e os primeiros como últimos. Nesse caso usamos apenas a palavra reverse para a lista e fora dos parêntesis como visto a seguir.

```
In [37]: mercado.reverse()

In [38]: print(mercado)
['Vale', 'títulos', 'tesouro', 'Petrobras', 'ouro', 'dolar', 'dolar',
'criptomoedas', 'comprar', 'commodity', 'BB']
```

- Removendo elementos da lista (**remove**)

O comando **remove** deleta um ou mais elementos da lista. Por exemplo, para deletar a palavra "ouro", o uso do comando fica como a seguir.

```
In [39]: mercado.remove('ouro')

In [40]: print(mercado)
['Vale', 'títulos', 'tesouro', 'Petrobras', 'dolar', 'dolar', 'criptomoedas',
'comprar', 'commodity', 'BB']
```

- Descobrindo o índice dos elementos (**index**)

Já foi mencionado que uma lista é formada por elementos que possuem índices em sua formação. Como descobrir a localização de um elemento em particular? Nesse caso,

basta usar o comando **index**. Por exemplo, para encontrar a localização de "Petrobras" e "criptomoedas", a formação do comando deve ser como a seguir.

```
In [41]: mercado.index('Petrobras')
Out[41]: 3

In [42]: mercado.index('criptomoedas')
Out[42]: 6
```

- Inserindo elemento em uma posição desejada (**insert**)

De nossa lista "mercado" atual, podemos desejar inserir um elemento, por exemplo, na posição 3. Devemos lembrar que a posição 3 possui, na realidade, índice 2. Então, se usamos o comando **insert** com o índice 2, estamos inserindo uma nova palavra na posição 3. Por exemplo, para inserir uma nova palavra "Fundo de Investimento" na posição 3, devemos prosseguir com a linha a seguir.

```
In [18]: mercado.insert(2,'Fundo de Investimento')

In [19]: print(mercado)
['Vale', 'títulos', 'Fundo de Investimento', 'tesouro', 'Petrobras', 'dolar',
'dolar', 'criptomoedas', 'comprar', 'commodity', 'BB']
```

O primeiro número dos parênteses do **insert** é o índice a ser inserido e, a seguir, vem o texto a ser inserido. O que o Python faz é deslocar todos os índices para um valor a mais para poder inserir a nova palavra.

- Limpando a lista toda (**clear**)

Para limpar a lista toda e utilizar o mesmo nome para outras operações, o comando a ser utilizado é **clear**. Nesse caso, todos os elementos são apagados de forma simultânea. O resultado na impressão é sempre [ ].

```
In [43]: mercado.clear()

In [44]: print(mercado)
```

Quadro 1.1 – Resumo das operações com listas

***nome*.append(*x*)**	Adiciona o elemento x à lista *"nome".*
***nome*.clear()**	Remove todos os elementos da lista *"nome".*
***nome*.count(*x*)**	Conta o número de ocorrências de x na lista *"nome".*
***nome*.extend(*y*)**	Insere a lista y no final da lista *"nome".*
***nome*.index(*x*)**	Retorna o valor do índice do elemento x da lista *"nome".*
***nome*.insert(pos,*x*)**	Insere um elemento x na posição *"pos" da lista "nome".*

***nome*.remove(*x*)**	Remove um elemento x da lista *"nome".*
***nome*.reverse( )**	Ordena inversamente a lista *"nome".*
***nome*.sort( )**	Ordena em ordem crescente ou alfabética a lista *"nome".*

1.4 OPERAÇÕES COM ELEMENTOS DAS LISTAS NO CONSOLE

Na seção anterior, apresentamos algumas funções pré-programadas bastante úteis para se manipular elementos em uma lista. Selecionamento, extração ou fatiamento da lista podem ser realizados por seções ou "limites". Aprender a usar os índices ajuda bastante em operações para cálculos, gráficos e decisões em programações mais avançadas.

Vamos supor que agora temos uma lista formada pelas palavras "Ibov = [PETR4, BBAS3, USIM5, GGBR4, VALE3]", sendo estas os símbolos de algumas ações do Ibovespa. A criação dessa lista no Python deve proceder como visto nas seções anteriores.

```
In [20]: ibov=['PETR4','BBAS3','USIM5','GGBR4','VALE3']
```

- ibov[2:4]: do elemento de índice 2 ao índice 3

```
In [22]: ibov[2:4]
Out[22]: ['USIM5', 'GGBR4']
```

- ibov[1:]: do elemento de índice 1 até o último

```
In [23]: ibov[1:]
Out[23]: ['BBAS3', 'USIM5', 'GGBR4', 'VALE3']
```

- ibov[:3]: do elemento inicial (índice zero) ao elemento de índice 2 (pois é 3 – 1)

```
In [24]: ibov[:3]
Out[24]: ['PETR4', 'BBAS3', 'USIM5']
```

- ibov[0:5:2]: do elemento inicial ao último saltando de 2 em 2

```
In [25]: ibov[0:5:2]
Out[25]: ['PETR4', 'USIM5', 'VALE3']
```

- ibov[−3:]: seleciona os 3 últimos elementos da lista

```
In [26]: ibov[-3:]
Out[26]: ['USIM5', 'GGBR4', 'VALE3']
```

- ibov[:−3]: seleciona os 2 primeiros elementos da lista (pois é −(3 − 1))

```
In [27]: ibov[:-3]
Out[27]: ['PETR4', 'BBAS3']
```

Apresentação e demonstração das operações com listas no Python

1.5 ESTATÍSTICAS BÁSICAS NO CONSOLE

Estatísticas simples podem ser realizadas no **Console** para uma lista de números, sendo necessário para isso importar a biblioteca de estatística. Como visto antes, podemos importar e dar um apelido para não precisar carregar a cada comando o nome completo da biblioteca. No caso, a biblioteca se chama statistics, e vamos considerar a importação com a lista de preços de uma ação, tomada minuto a minuto no Ibovespa.

```
In [1]: import statistics as st

In [2]: prec=[10,11,11,10,10,10,8,8,9,7,11,12,13,8,9]

In [3]: print(prec)
[10, 11, 11, 10, 10, 10, 8, 8, 9, 7, 11, 12, 13, 8, 9]
```

Adotamos o nome st para a biblioteca e, com ele, agora podemos chamar as funções de estatística (média, mediana, desvio-padrão etc.) para os cálculos.

- Média (**mean**)

Retorna o elemento médio de uma lista de números.

```
In [4]: st.mean(prec)
Out[4]: 9.8
```

- Mediana (**median**)

Retorna o elemento mediano de uma lista, primeiro ordenando em ordem crescente e, então, escolhendo aquele em que, abaixo e acima dele, a quantidade de dados é a mesma, ou seja, metade dos elementos de toda a lista.

```
In [5]: st.median(prec)
Out[5]: 10
```

- Moda (mode)

Retorna o elemento que é mais frequente na lista de números.

```
In [6]: st.mode(prec)
Out[6]: 10
```

- Desvio-padrão amostral (**stdev**)

Calcula o desvio-padrão amostral dos elementos de uma lista.

```
In [7]: st.stdev(prec)
Out[7]: 1.65615734242165
```

- Desvio-padrão populacional (**pstdev**)

Calcula o desvio-padrão populacional de uma lista.

```
In [8]: st.pstdev(prec)
Out[8]: 1.6
```

1.6 BIBLIOTECAS CIENTÍFICAS NO CONSOLE

Três bibliotecas tornam-se essenciais no desenvolvimento de cálculos mais avançados e colaboram com a ampliação do espectro de análises já proporcionada pela biblioteca de estatística.

A biblioteca numpy (*numerical Python*) é fundamental na computação científica em Python, aumentando as funcionalidades para o uso de **array** (vetores e matrizes) e auxiliando em muitas atividades. As bibliotecas scipy (*scientific Python*), pandas (*Python data analysis library*) e matplotlib (*plotting library*) completam o pacote essencial para gráficos, análises de tabelas, como a tabela dinâmica no Excel (*pivot table*) chamada **DataFrame**, e algoritmos para cálculos com vetores e matrizes. Essas quatro bibliotecas, mais matemática e estatística vistas anteriormente, fecham o pacote da maioria das atividades que todo programador ou analista precisa para entender uma série de dados.

Para fazer um gráfico no **Console**, precisamos importar as bibliotecas matplo-tlib e numpy para criar um vetor de pontos e para a plotagem do gráfico. Na função $y = 3x - 3$, é necessário criar o vetor x para alimentar o vetor y. Ambos os vetores são chamados pela função **plot** para executar o gráfico.

```
In [1]: import matplotlib.pyplot as fig

In [2]: import numpy as ny

In [3]: x=ny.arange(1,20)

In [4]: y=3*x-3

In [5]: fig.plot(x,y)
```

As duas primeiras linhas do **Console** estão importando as bibliotecas para fazer o gráfico com **pyplot** e criar arrays ou vetores (*x* e *y*). A linha [3] está criando um vetor com vinte pontos em sequência (1,2,3,4 …,20). A linha [4] está criando o vetor *y* com base na equação fornecida como regra. A última linha faz o gráfico colocando o vetor *x* no eixo horizontal e *y* no eixo vertical, como na Figura 1.4.

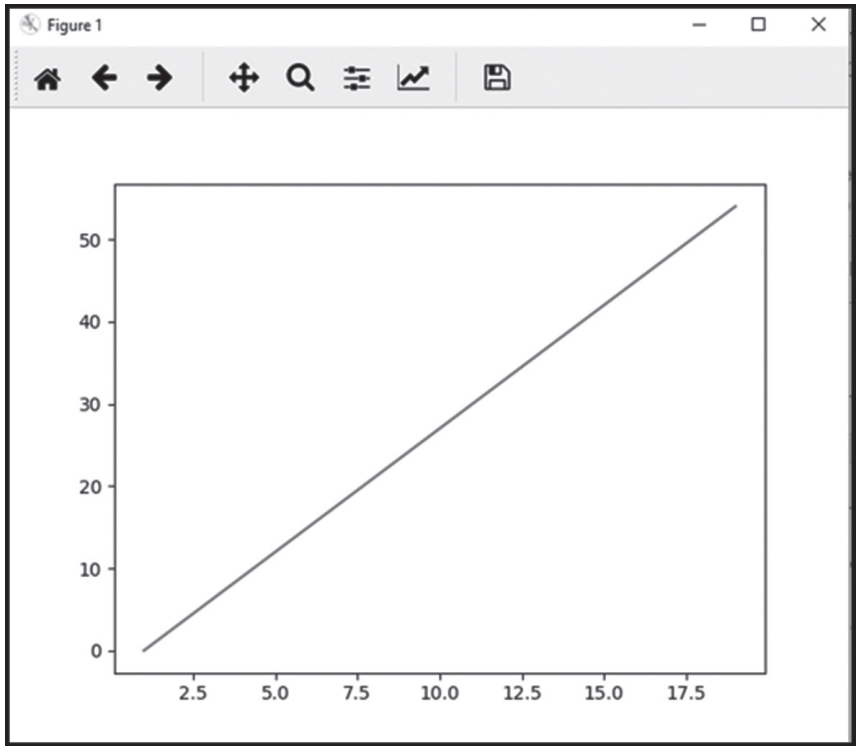


Figura 1.4 – Gráfico da equação *y* = 3*x* − 3 usando **pyplot**.

Muitas formatações especiais para gráficos são realizadas ao longo deste livro, mas algumas para o uso no **Console** podem ser construídas rapidamente. Por exemplo, no mesmo comando de linha, pode-se fazer o gráfico, colocar título e nomes nos eixos, usando **plot**, **title**, **xlabel** e **ylabel**, como visto a seguir.

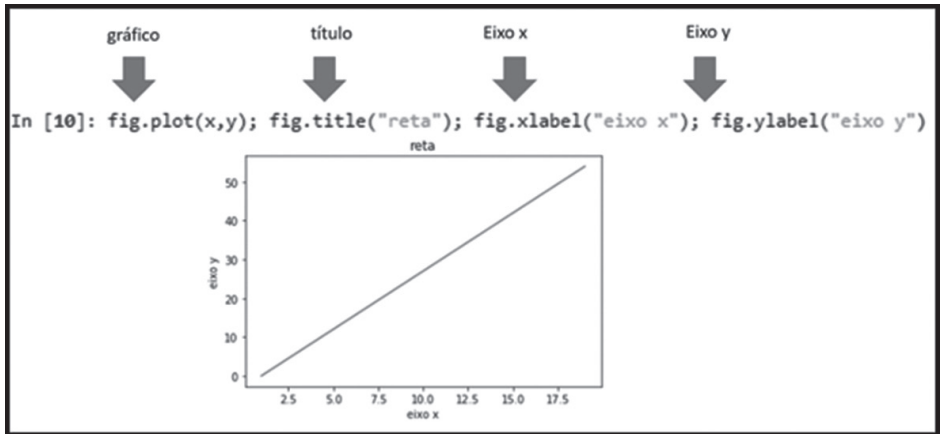


Figura 1.5 – Formatando o gráfico no **pyplot**.

Sempre que se instala o Python, a janela de gráfico está formatada para aparecer no **Console**, ou seja, nos comandos de linha. Apesar de prático, a qualidade e a visualização do gráfico não são boas. É interessante alterar a configuração para que a visualização fique em separado ao **Console**.

Para alterar a configuração, é interessante ir até a barra de tarefas do Spyder e escolher **Preferências**. Como apresentado na Figura 1.6, nessa área, escolhe-se **Console IPython**. Ao lado, abre uma janela de configurações possíveis, como cor da letra, dos alertas etc. Na barra superior está a aba **Gráficos**, que deve ser escolhida.

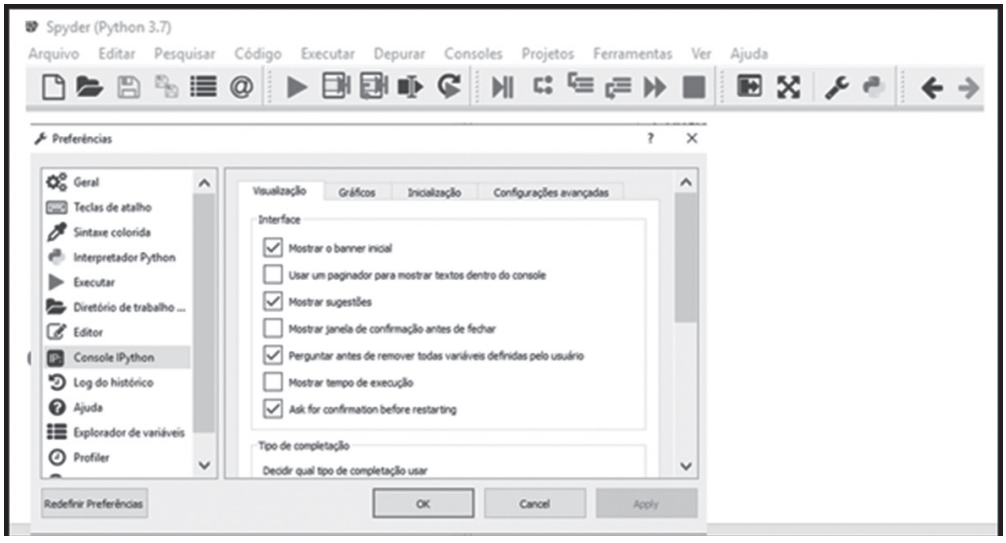


Figura 1.6 – Preferências para alterar o gráfico.

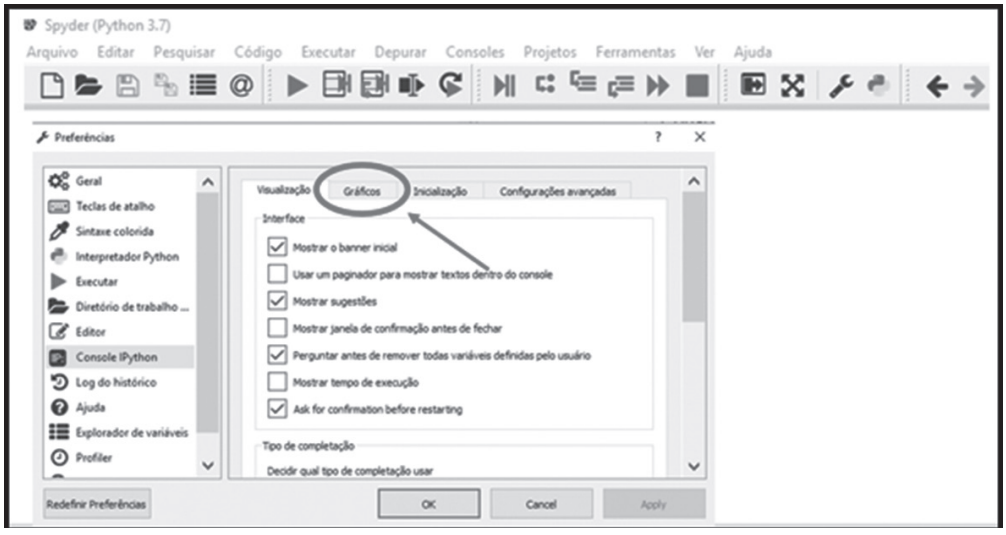


Figura 1.7 – Escolhendo a aba **Gráficos** para alteração do padrão.

Conforme pode ser visualizado na Figura 1.7, uma vez escolhida a aba **Gráficos**, outra janela de informações se abre (Figura 1.8). E nessa janela podemos escolher em **Saída** a caixa para trocar de **Linha** para **Automático**.

A partir de então, todos os gráficos são executados em uma janela separada e não mais no comando de linha. A vantagem da janela é que possui botões para ajuste de cores, nomes dos eixos, limites dos eixos; assim, não é preciso voltar ao comando de linha para reformatar o gráfico.

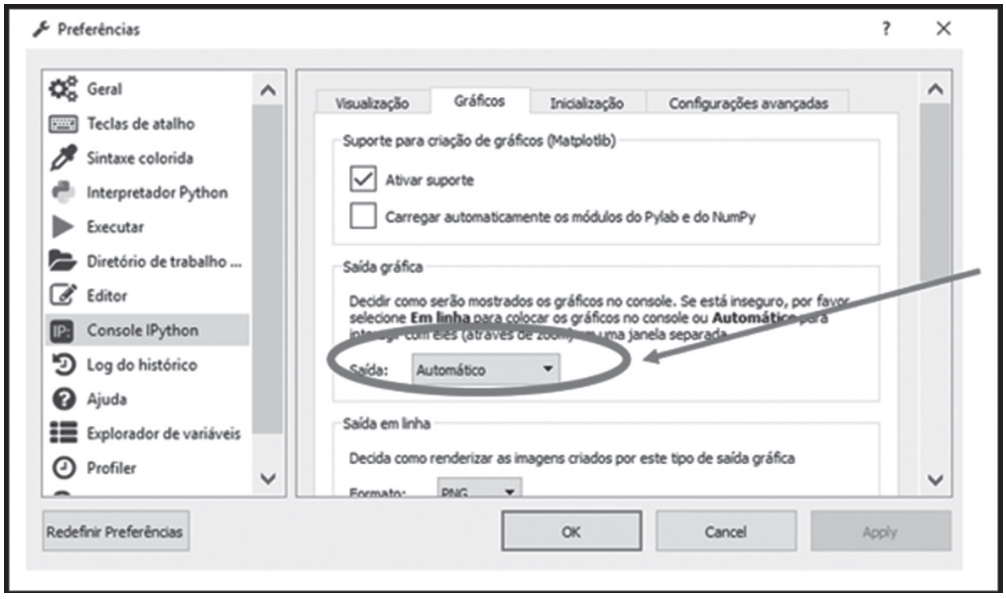


Figura 1.8 – Alterando o padrão de gráfico de **Linha** para **Automático**.

Ainda assim, no comando de linha, o gráfico pode ser salvo em qualquer formato tradicional para ser importado por outros *softwares*. Usando a extensão *.savefig no **pyplot**, é possível escolher em qual formato se deseja que o gráfico seja salvo. Por exemplo, para o formato mais popular *.jpeg, pode-se ver o resultado na Figura 1.9.

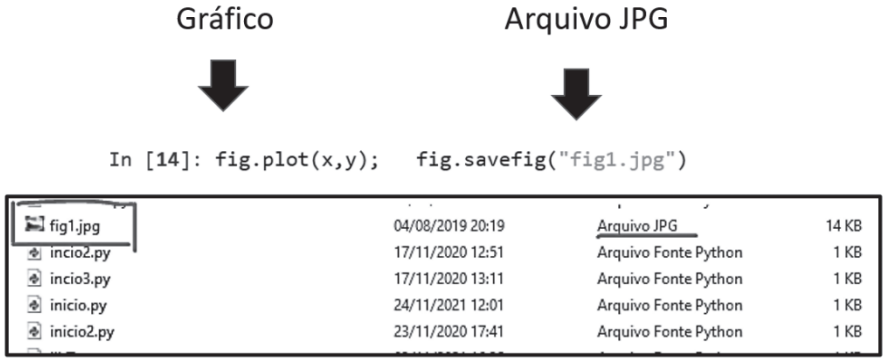


Figura 1.9 – Salvando um gráfico do **pyplot** como *.jpeg.

Quando as últimas linhas são rodadas, o gráfico fica minimizado na barra de tarefas. Isso pode ser resolvido apenas clicando no símbolo do Spyder. No entanto, caso queira que o gráfico apareça assim que a última linha seja executada, basta colocar uma linha a mais com o comando **show( )**, como visto a seguir. Pode não parecer nada muito diferente, mas esse comando será importante nos capítulos posteriores, quando dados serão adicionados de forma dinâmica da internet ou mesmo de um banco de dados e, ao mesmo tempo, deseja-se visualizar os resultados.

```
In [1]: import matplotlib.pyplot as fig

In [2]: import numpy as ny

In [3]: x = ny.arange(1,20)

In [4]: y = 3*x - 3

In [5]: fig.plot(x,y)
Out[5]: [<matplotlib.lines.Line2D at 0x1a6b0578b38>]

In [6]: fig.show()
```

Exemplos gráficos e sua construção no **Console** *do Python com as bibliotecas numpy e matplotlib.pyplot*

1.7 ARQUIVOS DE TIPO TEXTO NO CONSOLE

Abrir e fechar arquivos é o que faz o contato de um ambiente de programação com outros programas e dados. Importar dados gerados em outros programas é o que torna uma linguagem mais flexível e amigável. O tipo de arquivo mais antigo em termos de programação são os arquivos do tipo texto, que muitas vezes são usados com extensão *.txt, mas que podem ter qualquer tipo de extensão, desde que especificada em sua geração que é do tipo texto.

No **Console** do Python, é possível importar e exportar qualquer tipo de dados para todos os formatos conhecidos. Como exemplo, os arquivos do tipo texto são exportados e importados com a técnica bastante conhecida há décadas como **open** e **write**.

Variável = **open**('*nome do arquivo*', '*tipo de procedimento*')

Depois do comando **open**, o primeiro elemento do parêntese é o nome do arquivo que se deseja criar ou ler do diretório. O segundo elemento deve ser completado com **w** para imprimir um resultado no arquivo (*write*) ou **r** para ler um conjunto de dados já existente (*read*). Vamos supor que desejamos criar um arquivo de texto com nome dad.txt e que nele desejamos exportar a lista *x* = [3,2,1]. Os comandos de linha são os seguintes.

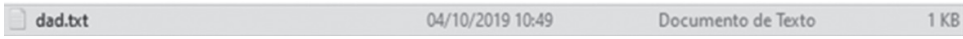
```
In [1]: x=[3,2,1]

In [2]: f=open('dad.txt','w')

In [3]: f.write('%d %d %d\n' % (x[0],x[1],x[2]))
Out[3]: 6

In [4]: f.close()
```

Na ordem das linhas, primeiro se cria a lista, depois abre-se o programa dad.txt com operador **w**, indicando que o arquivo vai ser usado para impressão. A seguir, tem-se o comando **write**. O nome da variável que se criou para abrir o arquivo foi *f*, por isso o comando recebe o nome da variável com ponto decimal e comando de impressão **write**.

A formatação do arquivo precisa ser elemento a elemento, respeitando a tabela de impressão para arquivos tipo texto. Como são três elementos na lista, cada um deve ser especificado. Utiliza-se sempre o símbolo **%** com aspas simples para indicar a formatação do número. Para o caso de número inteiro, o formato é **%d**. O símbolo **\n** indica que, após o último elemento, o programa muda de linha, permitindo em uma próxima impressão a utilização da linha de baixo. O símbolo **%** após os comandos dentro das aspas simples indica que os dados vão ser fornecidos a seguir. Como temos mais de um elemento na lista, deve-se usar **( )** e indicar cada elemento a ser impresso.

Por fim, é obrigatório o fechamento do arquivo usando o comando **close** com o nome da variável em ponto decimal. No caso do exemplo, **f.close( )** fecha o arquivo, e os parênteses do comando **close** devem estar vazios, como visto anteriormente.

O resultado no diretório vai ser o nome aparecendo com a data da criação do arquivo, o tipo de documento (texto nesse caso) e o tamanho em *bytes*.



Para visualizar o arquivo, basta abrir com qualquer programa habilitado para ler dados do tipo texto, como o Bloco de Notas do Windows. Quando se abre o arquivo, observa-se a lista que foi digitada no **Console**.

Para ler dados de um arquivo tipo texto, o procedimento é parecido; na abertura, a tipificação do comando e a criação de uma variável recebem o resultado dentro do **Console**.

```
In [5]: f = open('dad.txt', 'r')

In [6]: y = f.read()

In [7]: print(y)
3 2 1


In [8]: f.close()
```

Novamente o arquivo é aberto com **open**, e agora utiliza-se **read** para ler o arquivo. Nesse caso, uma variável precisa receber os números do arquivo, que nesse exemplo chamamos de *y*. Ao imprimir *y*, recuperamos a lista original que estava em *x*, mas que agora foi importada do arquivo dad.txt. Finalmente, deve-se fechar o arquivo usando **close**.

Quais são os formatos permitidos no Python quando se exporta dados para um arquivo? Como exemplo, a linha a seguir imprime alguns dos formatos permitidos.

```
In [12]: print('%s %5.2f %10.4e %d' % ('texto', 7.3, 1e-10, 34) )
texto  7.30 1.0000e-10 34
```

Quadro 1.2 – Formatação dos números para impressão de dados

Formato	Descrição
%s	Formato *string* é usado para textos.
%a.b*f*	Formato em ponto flutuante é utilizado para números reais que possuem casas decimais. Nesse caso, "a" é o tamanho de dígitos reservados para o número, "b" é a quantidade de casas decimais e "f" é a indicação de que é ponto flutuante (número real). Nesse caso, %5.2f significa um campo total de 5 dígitos e 2 casas decimais.
%a.b*e*	Formato em notação científica, com "a" dígitos, "b" casas decimais com "e" representando 10 elevado a um número. Assim, %10.4 e significa 10 dígitos (inclusive espaço em branco) e 4 casas decimais; a letra "e" indica a representação da potência de 10.
%d	Formato usado para números inteiros.
\n	Mudar de linha no arquivo.
str(x)	Transforma um número em texto para impressão em que "str" significa *string*.

Processo de abertura e fechamento de arquivos e demonstração do uso da biblioteca statistics

1.8 ARQUIVOS EXCEL NO CONSOLE

Além dos arquivos tipo texto, outra maneira frequente de arquivos com dados a serem importados são aqueles armazenados no formato do Excel (*.xls, *.xlsx). A importação desse tipo de arquivo para o Python está presente em diversas bibliotecas. Por exemplo, vamos supor que exista um arquivo chamado Dados.xlsx no diretório de trabalho, cujos dados estão representados na coluna A do Excel.

	A	B
1	10	
2	-2	
3	5	
4		

Uma biblioteca de importação é a openpyxl (*open Python* com formato X L). A letra final é *L,* e não o número 1. Com o **load**, abre-se o arquivo em Excel e depois indica-se em qual planilha estão os dados. O valor do dado pode ser armazenado em uma variável do **Console** e ser impresso como mostrado a seguir.

```
In [4]: import openpyxl

In [5]: arq = openpyxl.load_workbook('Dados.xlsx')

In [6]: plan=arq['Planilha1']

In [7]: x = plan['A1'].value

In [8]: print(x)
10
```

Outra biblioteca que auxilia na importação de dados do Excel é a biblioteca Pandas, à qual daremos destaque ao longo do livro. Ela é uma das mais importantes para análise de dados e cruzamentos de informações. A biblioteca possui funções especiais para trabalhar com arquivos em Excel. Ao criar um arquivo no Excel, no entanto, precisamos deixar a primeira célula com um título para os dados que serão importados para o Python. Vejamos o mesmo arquivo anterior, mas agora com o título "valores".

	A	B
1	valores	
2	10	
3	-2	
4	5	

O primeiro passo é deixar o arquivo do Excel na mesma pasta onde o programa em Python será executado. Caso contrário o Python não conseguirá localizar o arquivo *.xlsx do Excel. A função para importar do Excel é "read_excel", onde nos parêntesis colocamos o nome do arquivo em Excel entre aspas ou apóstrofos, como visto a seguir.

Na linha 3 podemos ver que a importação do Excel vem num formato conhecido como DataFrame, que aprenderemos em detalhes em capítulos posteriores. Esse formato é uma tabela onde na primeira coluna temos as linhas e na segunda coluna os valores que estão no Excel.

```
In [1]: import pandas as pd

In [2]: x = pd.read_excel('Dados.xlsx')

In [3]: print(x)
   valores
0       10
1       -2
2        5

In [4]: x=x['valores'].values

In [5]: x[0]
Out[5]: 10
```

Ao invés de usarmos esse formato, podemos extrair os valores numéricos colocando-os em formato de vetor (x[0], x[1], x[2]). Para tanto, na linha 4, ao colocarmos a extensão "values" (x['valores'].values), transformamos o que era uma tabela em vetor. Devemos observar que o título da coluna é "valores" no Excel, por isso usamos x['valores'] no Pyhton. Caso fosse outro nome, colocaríamos esse nome do tipo x['nome'].values.

1.9 ENTRADA DE DADOS COM INPUT NO CONSOLE

Quando não se deseja entrar com valores fixos nas variáveis, um programa pode solicitar ao usuário que ele próprio alimente uma variável. Esse tipo de entrada é modulado pelo comando conhecido como **input** no Python. O comando pode ou não ser seguido de um texto usando aspas duplas (" ") ou simples (' ').

Uma entrada como a seguinte não é tão boa como se pode observar. O número digitado foi "2", mas está colado ao lado da palavra "número", isso porque dentro dos parênteses não foi deixado espaço em branco suficiente para o usuário entrar com o número dois.

```
In [1]: var1=input("entre com o número")

entre com o número2
```

A entrada correta deve ser como a seguinte, em que um sinal de = foi digitado para chamar a atenção do usuário e um espaço em branco foi utilizado para não colar o número digitado ao texto.

```
In [2]: var1=input("entre com o número = ")

entre com o número = 2

In [3]: print(var1)
2
```

Quando a *var1* é impressa no comando **PRINT( )**, o número que o usuário digitou e estava armazenado na memória aparece no **Console**.

1.10 CÁLCULOS COM IMPORTAÇÃO DE DADOS NO CONSOLE

Em seções anteriores, tratamos de operações básicas com listas no **Console** do Python. Outras funções podem ser adicionadas às funções já descritas, como soma, valores máximos, valores mínimos. Também se pode começar a agregar essas funções com as funções estatísticas na biblioteca statistics.

Como exemplo, vamos novamente importar dados que estão no arquivo do Excel já descrito na seção 1.8. Nosso arquivo Dados.xlsx (10,–2,5) é importado para o uso de algumas medidas. A importação repete o processo anterior com o uso da biblioteca xlrd.

```
In [1]: import pandas as pd

In [2]: x = pd.read_excel('Dados.xlsx')

In [3]: x=x['valores'].values
```

De posse da lista x = [10,–2,5], podemos encontrar o máximo, o mínimo e a soma total com comandos de linha bem simples, como os apresentados a seguir.

```
In [4]: max(x)
Out[4]: 10

In [5]: min(x)
Out[5]: -2

In [6]: sum(x)
Out[6]: 13
```

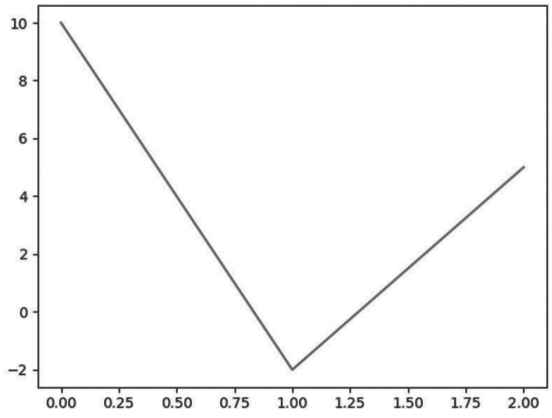
Já para a média e o desvio-padrão, podemos usar as funções de estatísticas que já existem no formato de vetor, bastando para tanto colocar ".mean()" para a média e ".std()" para o desvio-padrão. Então a média e o desvio-padrão populacional de nossos dados que estão no Excel são encontrados como segue.

```
In [30]: import pandas as pd

In [31]: x = pd.read_excel('Dados.xlsx')

In [32]: x=x['valores'].values

In [33]: x.mean()
Out[33]: 4.333333333333333

In [34]: x.std()
Out[34]: 4.921607686744467
```

Com um vetor ou lista importada do Excel, pode-se fazer o gráfico para estudos sobre retornos, investimentos, previsões, análises e tudo o que a estatística e a matemática podem ajudar na construção dos cenários econômicos. Para o gráfico, devemos novamente seguir os passos desde a importação das bibliotecas matplotlib e numpy até a configuração final da janela gráfica.

A última linha do processo a seguir mostra o que consta na memória do Python após a execução do gráfico. Para a visualização do resultado, é preciso se lembrar de clicar na janela minimizada que fica abaixo, na barra de fixação do Windows, com o símbolo do Spyder, nas versões antigas do Python e um símbolo parecido com página em branco nas versões mais recentes.

```
In [26]: import matplotlib.pyplot as fig

In [27]: import numpy as ny

In [28]: t=ny.arange(0,2)

In [29]: import matplotlib.pyplot as fig

In [30]: import numpy as ny

In [31]: t=ny.arange(0,3)

In [32]: fig.plot(t,x)
Out[32]: [<matplotlib.lines.Line2D at 0x22c8af08a58>]
```



1.11 TRANSFORMANDO UMA LISTA EM ARRAY E ARRAY EM LISTA

No exemplo anterior da importação de dados, nossos números estão no Python no formato de array advindos da importação da biblioteca Pandas para o arquivo em Excel.

```
In [35]: x
Out[35]: array([10, -2,  5], dtype=int64)

In [36]: print(x)
[10 -2  5]
```

Para transformar um array em lista, basta usar a função "list" como demonstrado a seguir, com o nome do array dentro dos parêntesis.

```
In [37]: x=list(x)

In [38]: x
Out[38]: [10, -2, 5]
```

A diferença entre lista e array é que na lista temos vírgulas separando os valores, e no array não. A transformação anterior alterou "x" para lista e podemos fatiar normalmente, como foi visto antes.

```
In [53]: x[0:2]
Out[53]: [10.0, -2.0]

In [54]: x[1:3]
Out[54]: [-2.0, 5.0]
```

Mas uma lista não aceita operações de soma, subtração ou divisão para ela toda de uma única vez. Por exemplo, não é permitido fazer a subtração elemento a elemento na forma de $x[1:3] - x[0:2]$. Quando se tenta isso, o resultado é o seguinte erro.

```
In [55]: x[1:3]-x[0:2]
Traceback (most recent call last):

  File "<ipython-input-55-67ead4716ae9>", line 1, in <module>
    x[1:3]-x[0:2]

TypeError: unsupported operand type(s) for -: 'list' and 'list'
```

Existem diversas alternativas para operar esse erro na lista, que será visto mais adiante neste livro. No momento, podemos fazer algo mais fácil: a transformação de uma lista em um **array** (vetor). Um **array** é uma lista que aceita uma complexidade bem maior de operações e utilizações em diversos programas no Python. A transformação da lista *x* em **array** é feita na própria biblioteca numpy. Uma vez importada a biblioteca numpy, como ny, a transformação segue apenas usando o comando **ny.array(x)**.

```
In [56]: vetor=ny.array(x)

In [57]: vetor
Out[57]: array([10., -2.,  5.])

In [58]: print(vetor)
[10. -2.  5.]
```

Quando a variável *vetor* assume a transformação, o vetor aparece com a palavra **array** e com um formato um pouco diferente da lista. Aparece, por exemplo, um parêntese que não existe quando exibimos uma lista. No entanto, quando se usa **PRINT( )**, a visualização para o usuário é a mesma da lista, pois o **array** é uma modificação apenas visível internamente na memória do Python.

A vantagem fica evidente nos comandos de linha a seguir. Agora o vetor que substitui a lista *x* aceita a operação que antes era negada na lista, ou seja, o que não era possível fazer, *x*[1:3] – *x*[0:2], torna-se possível com *vetor*[1:3] – *vetor*[0:2].

```
In [60]: resp=vetor[1:3]-vetor[0:2]

In [61]: print(resp)
[-12.   7.]
```

Como pode ser notado, agora a variável *resp* também é um vetor, e gráficos ou qualquer tipo de cálculo são possíveis.

EXEMPLO 1.1

Quando se trabalha ou se investe no mercado financeiro, uma das operações mais frequentes é o cálculo do retorno financeiro. É o retorno que vai dizer se uma ação tem, em média, uma rentabilidade melhor que outra. Ou é o retorno que vai dizer se a volatilidade (desvio-padrão) é maior em um ativo em comparação com outro tipo de ativo.

Vinte dados de uma opção da Petrobras foram adquiridos a cada 5 minutos de um terminal e armazenados em uma planilha do Excel. O nome desse arquivo é Petrobras.xlsx, como pode ser visto a seguir. Os dados estão na **Planilha1** desse arquivo. Pretende-se calcular o retorno financeiro dessa opção. O retorno financeiro pode ser calculado como:

$$ret = \frac{preço\,(i) - preço\,(i-1)}{preço\,(i-1)}$$

	A
1	preço
2	1,31
3	1,34
4	1,42
5	1,4
6	1,42
7	1,4
8	1,47
9	1,45
10	1,48
11	1,42
12	1,34
13	1,34
14	1,35
15	1,35
16	1,36
17	1,32
18	1,24
19	1,22
20	1,27
21	1,26

```
In [1]: import pandas as pd

In [2]: import numpy as np

In [3]: import matplotlib.pyplot as fig

In [4]: plan = pd.read_excel('Petrobras.xlsx')
```

Nesse passo, repetimos exatamente o processo de importação dos dados que estão na **Planilha1** do arquivo chamado Petrobras.xlsx com a primeira célula tendo o título "preço". Utiliza-se a biblioteca Pandas para a importação do arquivo com os preços da Petrobras. O próximo passo é transformar em vetor para que o cálculo do retorno financeiro seja possível. Percebe-se a seguir que o vetor (**array**) de preços está completo no **Console** do Python com 20 elementos no total.

```
In [7]: vetor = plan['preço'].values

In [8]: print(vetor)
[1.31 1.34 1.42 1.4  1.42 1.4  1.47 1.45 1.48 1.42 1.34 1.34 1.35 1.35
 1.36 1.32 1.24 1.22 1.27 1.26]
```

O cálculo do retorno dos preços no Python é:

```
In [9]: retorno= (vetor[1:20]-vetor[0:19])/vetor[0:19]
```

Fornecendo como resultado o **array** de retorno:

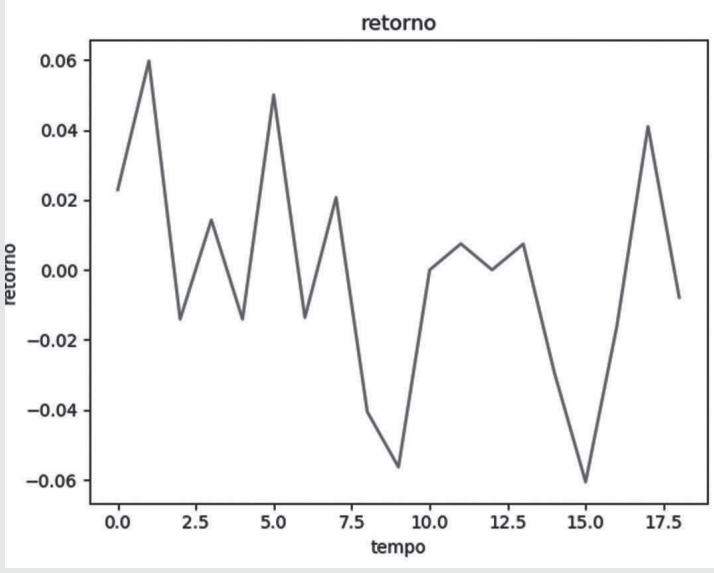
```
In [10]: print(retorno)
[ 0.02290076  0.05970149 -0.01408451  0.01428571 -0.01408451  0.05
 -0.01360544  0.02068966 -0.04054054 -0.05633803  0.          0.00746269
  0.          0.00740741 -0.02941176 -0.06060606 -0.01612903  0.04098361
 -0.00787402]
```

Para fazer o gráfico, precisamos nos lembrar de primeiro criar um vetor do eixo horizontal, como *t*, usando o comando **arange**. Após isso, chamamos **plot** e colocamos os títulos no gráfico, nos eixos horizontal e vertical, como já apresentados antes. O resultado fica como mostrado a seguir.

```
In [17]: t = np.arange(0,19)

In [18]: fig.plot(t,retorno)
Out[18]: [<matplotlib.lines.Line2D at 0x1b7523df160>]

In [19]: fig.title('retorno')
Out[19]: Text(0.5, 1.0, 'retorno')
```

```
In [20]: fig.xlabel('tempo')
Out[20]: Text(0.5, 23.52222222222222, 'tempo')

In [21]: fig.ylabel('retorno')
Out[21]: Text(22.347222222222214, 0.5, 'retorno')
```

O gráfico é apresentado com os títulos, e a oscilação verificada no eixo vertical pode ser inclusive transformada para o formato de porcentagem, uma vez que o retorno financeiro é sempre em termos de porcentagem. Para tanto, basta apenas multiplicar o vetor de retornos por 100 e indicar isso no título do eixo vertical.



■

Passo a passo da construção dos retornos de um vetor

EXEMPLO 1.2

Utilizando o vetor anterior de retorno, podemos criar os cenários otimista e pessimista para essa opção da Petrobras. Os cenários otimista e pessimista são construídos com base na fórmula originada da estatística sob o intervalo de confiança. Para 95% de confiança na estimativa do retorno médio, as fórmulas são as seguintes:

$$Cenário\ otimista = retorno_{médio} + \frac{2.desvio\text{-}padrão}{\sqrt{n}}$$

$$Cenário\ pessimista = retorno_{médio} - \frac{2.desvio\text{-}padrão}{\sqrt{n}}$$

Então, considerando que todo processo de importação dos dados é realizado como no exemplo anterior, primeiro deve-se calcular a média dos retornos e o desvio-padrão. Aqui precisamos importar, além da biblioteca statistics, também a biblioteca math para o **Console** com a função matemática da raiz quadrada dos *n* dados amostrados, que nesse caso é *n* = 19 dados de retorno.

```
In [31]: import math

In [32]: import statistics as st

In [33]: media = st.mean(retorno)

In [34]: desvio=st.pstdev(retorno)

In [35]: otimista = media+2*desvio/math.sqrt(19)

In [36]: pessimista = media-2*desvio/math.sqrt(19)

In [37]: print(otimista,pessimista)
0.013056375555366216 -0.016134541069746648
```

O resultado é que, no cenário otimista, se espera um retorno para essa opção de 1.3% e, para o cenário pessimista, um retorno financeiro de –1.6%.

■

No exemplo anterior, poderíamos não utilizar a biblioteca statistics? Sim, basta lembrar que podemos usar as funções do array "mean( )" e "std( )" para média e desvio-padrão. Nesse caso a alteração seria a seguinte:

```
In [31]: media = retorno.mean()

In [32]: desvio = retorno.std()

In [33]: otimista=media+2*desvio/math.sqrt(19)

In [34]: pessimista=media-2*desvio/math.sqrt(19)
```

EXEMPLO 1.3

Podemos aproveitar os dados de importação para ver, na mesma figura, os dados dos preços da opção da Petrobras e seu retorno. Uma técnica para isso é dividir a figura em duas partes, usando para isso o comando **subplot( )**. Esse comando divide a tela da figura em formato de matriz, cuja identificação segue o formato a seguir. Para uma tela com dois gráficos, um embaixo do outro, o primeiro gráfico deve ficar na área **subplot(211)**. Os dois primeiros números indicam as posições dos gráficos na figura; nesse caso (2,1) são dois gráficos em duas linhas diferentes e uma única coluna. O segundo gráfico deve receber **subplot(212)**, ou seja, no total da tela são dois gráficos e uma única coluna, e deseja-se que o computador faça o segundo gráfico.

subplot(211)
subplot(212)

Para fazer quatro gráficos no formato de matriz, os gráficos seriam **subplot(221)**, **subplot(222)**, **subplot(223)** e **subplot(224)**.

subplot(221)	subplot(222)
subplot(223)	subplot(224)

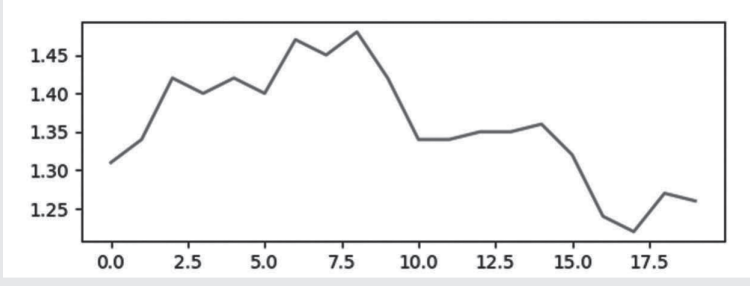
Deve-se antes lembrar que o tamanho do vetor de preços é diferente do vetor de retornos. Enquanto temos 20 preços, teremos apenas 19 retornos. Logo, o eixo horizontal para os preços deve ser formado à parte. Por exemplo, podemos criar o vetor como uma nova variável *tempo = np.arange(0,20)*. Então, a programação no **Console** será como mostrada a seguir.

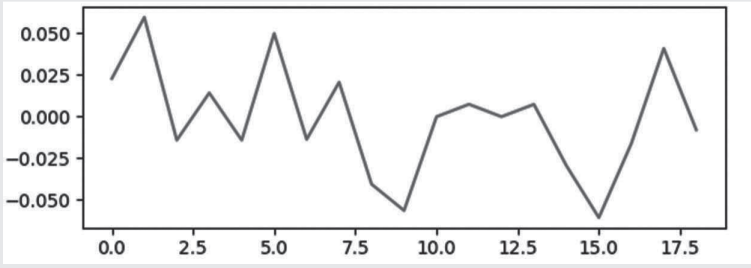
```
In [35]: tempo = np.arange(0,20)
```

Uma vez criado o eixo horizontal dos preços, estamos prontos para a plotagem dos dois gráficos, sendo o superior dos preços e o inferior dos retornos. Podemos, inclusive, colocar esses comandos na mesma linha separados por ponto e vírgula no **Console**.

Então **fig.subplot(211)** habilita a janela gráfica para o primeiro gráfico dos dois gráficos a serem feitos. E, na mesma linha, temos o comando de plotagem normal para a variável *tempo* e *vetor*. A linha **out** é apenas a saída da memória do Python, indicando que a área gráfica foi criada. O **fig.subplot(212)** habilita a segunda parte para a janela gráfica fazer o gráfico da variável *t* para o retorno.

```
In [50]: fig.subplot(211); fig.plot(tempo,vetor)
Out[50]: [<matplotlib.lines.Line2D at 0x26db7c78710>]

In [51]: fig.subplot(212); fig.plot(t,retorno)
Out[51]: [<matplotlib.lines.Line2D at 0x26dbaacf048>]
```





1.12 REINICIANDO O CONSOLE E AUMENTANDO A FONTE

Muitas vezes, é interessante reiniciar o **Console** para limpar da memória dados ou partes de programações anteriores. Para isso, segurando a tecla de controle **CTRL** e apertando a tecla **D**, o console reinicializa e limpa a memória. Então, **CTRL+D** apresenta a seguinte tela para o **Console**. Como mencionado no início do capítulo, outra maneira de limpar a memória é acessando a palavra **Consoles** na aba superior e clicando em **Reiniciar kernel**.

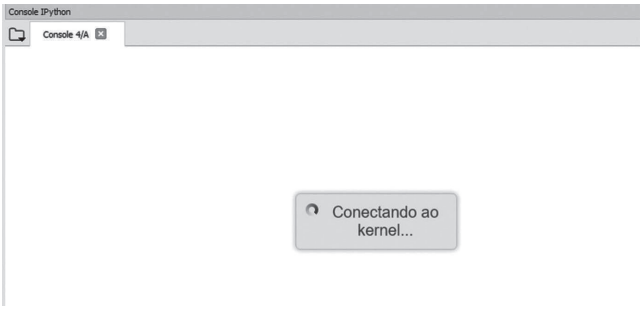


Figura 1.10 – Reiniciando o **Console**.

Às vezes, alguns computadores usam como padrão para a programação uma fonte pequena. Isso pode ser alterado em preferências ou, de maneira mais rápida e temporária, pode-se utilizar as teclas **CTRL+SHIFT(+)** para aumentar a fonte.

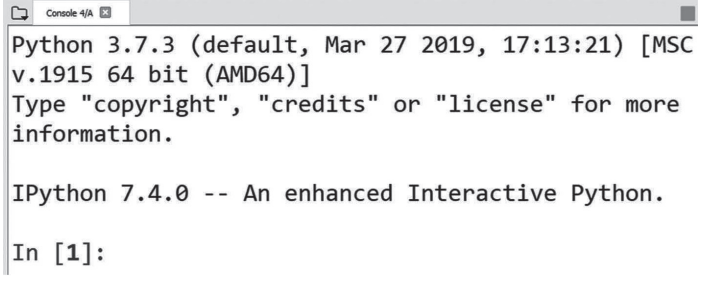


Figura 1.11 – Fonte aumentada no **Console**.

Para diminuir a fonte, basta usar os comandos **CTRL(–)** e ela ficará com tamanho menor. Se continuar usando essas combinações de teclas, a fonte vai diminuir gradativamente até o tamanho desejado.

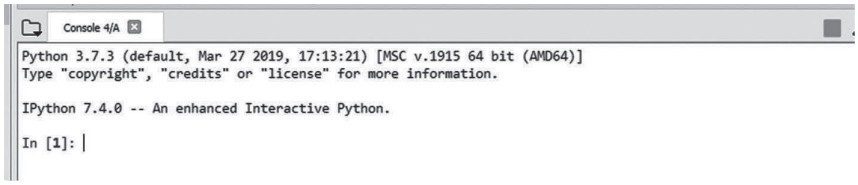


Figura 1.12 – Fonte diminuída no **Console**.

1.13 EXERCÍCIOS

1. Usar a biblioteca correta e as funções matemáticas do Python, quando necessárias, para resolver as seguintes expressões:
   (a) $y = 43 - 22$
   (b) $x = sen(2) - cos(4.2)$
   (c) $z = cos(sen(3.7) - tan(1.3))$
   (d) Resto da divisão de 26 por 4.
   (e) Converter $x = 46.2°$ para *radianos*.
   (f) Converter $y = 3.1$ rad para *graus*.

2. Assumir como constante no comando de linha do Python $x = 3$ e $y = 6$ e imprimir usando **PRINT( )** o resultado das equações seguintes:
   (a) $w = e^x - ln(y)$
   (b) $z = x*y^2 + y*cos(x)$
   (c) $s = \sqrt{\frac{x}{y} + \ln(x+y) + \tan(x)}$

3. Criar a lista de números *num*=[3, 3, 4, 1, 2, 1, 1, 2, 3, 4, 4, 1, 1, 5, 2] e fatiá-la conforme os itens a seguir:
   (a) Fatiar do elemento de índice 2 ao de índice 3.
   (b) Fatiar do quinto elemento ao nono elemento.
   (c) Fatiar do elemento de índice 1 ao último.
   (d) Fatiar do primeiro elemento ao último.
   (e) Fatiar do primeiro elemento ao último saltando de três em três elementos.
   (f) Selecionar o último elemento da lista.
   (g) Selecionar os três últimos elementos da lista.

(h) Selecionar os quatro primeiros elementos da lista.

(i) Contar o número de elementos da lista.

(j) Contar quantas vezes aparece o número 1 na lista.

4. Criar a lista com nomes das bolsas de valores do mundo: Bolsas = [‘dow’, ‘ibov’, ‘ftse’, ‘dax’, ‘nasdaq’, ‘cac’]. Fatiá-la conforme os itens a seguir.

   (a) Selecionar as três primeiras.

   (b) Incluir a sublista Bs = [‘hong kong’, ‘merval’] na lista anterior.

   (c) Descobrir qual o índice da ‘nasdaq’.

   (d) Remover ‘cac’ da lista.

   (e) Inserir “sp&500” como índice 2 na lista de bolsas, mas sem excluir nenhum elemento já inscrito.

5. Abrir um arquivo chamado bov.txt e salvar os dados das siglas das ações e seus valores na seguinte ordem: ‘petr4’, ‘vale3’, ‘ggbr4’, 28.4, 31.3, 15.76.

6. Abrir o arquivo bov.txt do exercício anterior no **Console** e imprimir o resultado dos elementos existentes nele.

7. Observar a seguinte lista de dados, *lista*= [2, 2, 3, 3, 3, −1, −1, −2, 0, 0, 0, 2, 4, 5, 1, 2, 2, 0, 0, 0,2 ,1, 5, 5, 7, 6, 5, 0, 0]. Programar o **Console** para encontrar as seguintes medidas estatísticas:

   (a) Soma de todos os elementos.

   (b) Máximo elemento da lista.

   (c) Mínimo elemento da lista.

   (d) Média dos elementos da lista.

   (e) Mediana dos elementos da lista.

   (f) Moda dos elementos da lista.

   (g) Desvio-padrão amostral.

   (h) Desvio-padrão populacional.

   (i) Contar o número de vezes que aparece o número 0.

   (j) Contar o número de vezes que aparece o número 5.

   (k) Ordenar a lista em ordem crescente.

   (l) Ordenar a lista em ordem decrescente.

8. Os dados a seguir representam os preços diários de fechamentos no pregão da Bovespa entre os meses de setembro e outubro de 2019. A coluna A do Excel se refere a VALE3 (Vale do Rio Doce) e a coluna B indica GGBR4 (Gerdau). Os dados estão em um arquivo Excel na planilha denominada **Planilha1**.

	A	B
1	Vale	Gerdau
2	46,01	12,66
3	45,52	12,58
4	46,52	12,58
5	46,52	12,67
6	46,45	12,43
7	47,89	13,11
8	48,24	13,52
9	47,95	13,23
10	49,69	13,61
11	49,79	13,56
12	48,59	13,5
13	48,9	13,58
14	48,4	13,43
15	48,32	13,4
16	48,42	13,25
17	48,1	13,21
18	46,93	12,95
19	47,86	13,11
20	47,86	13,13
21	47,66	13,03
22	47,75	13,16
23	47,71	13,09
24	45,1	12,6
25	45,44	12,78
26	46,59	13,03
27	46,04	12,78
28	45,32	12,55
29	45,67	12,52

Deseja-se, então, que esses dados sejam importados para o Python com a biblioteca Pandas e que se responda aos itens a seguir.

(a) Importar os dados do Excel e transformar a coluna A em uma variável que represente a Vale e a coluna B em outra variável que represente a coluna B.

(b) Transformar as variáveis em vetores (array).

(c) Fazer os dois gráficos dos preços da Vale e da Gerdau usando **subplot**. Colocar a Vale na parte superior da figura e a Gerdau ao seu lado.

(d) Calcular os retornos das duas empresas e plotar os quatro gráficos (preço da Vale e seu retorno; preço da Gerdau e seu retorno) no formato de uma matriz com 2×2 elementos.

9. Leia no console a variável x = 0.1 e y = 0.5. Resolva com a biblioteca adequada a fórmula a seguir:

$$z = \sqrt{\frac{sen(x)^2 + \sqrt{y}}{2x + 3y}}$$

(a) Qual comando de linha para a solução?

(b) Qual o valor encontrado?

10. Leia no console a variável x = 0.1 e y = 0.5. Resolva com a biblioteca adequada a fórmula a seguir:

$$z = \sqrt{\frac{sen(x)^2 + \sqrt{y}}{2x + 3y}}$$

(a) Qual comando de linha para a solução?

(b) Qual o valor encontrado?

11. Colocar o comando para descobrir o valor de $y = \sqrt{\cos(x) + e^x}$ considerando no console do Python que x = 0.5

    Qual o comando?

    Qual o valor encontrado (4 casas decimais)?

12. Colocar o comando para descobrir o valor de $z = \sqrt{x + e^y + sen(y)}$ considerando no console do Python que x = 0.5 e y = 0.2.

    (a) Comando?

    (b) Resposta do console.

    Considerar a seguinte lista **dad=[15, 12, 1, -30, -4, 50, 4, 8]**

13. Qual o comando para fatiar a lista e formar nova lista do número -30 ao 4 incluindo ambos?

14. Indicar o comando para formar uma nova lista com os últimos 3 elementos da lista dad.

15. Indicar o comando para remover o número 50 da lista.

CAPÍTULO 2

ITERAÇÃO E DECISÃO

2.1 INTRODUÇÃO

O Anaconda – Spyder possui uma área reservada para a programação em arquivos, também conhecida como **scripts**. No capítulo anterior, foi apresentada a programação de listas e **array** para a resolução de equações, medidas de estatística e utilização de algumas bibliotecas auxiliares.

Apesar de interessante e rápida, a maneira de programar no **Console** torna-se cansativa e não eficiente quando temos muitos processos para serem efetivados, com cálculos demasiadamente extensos.

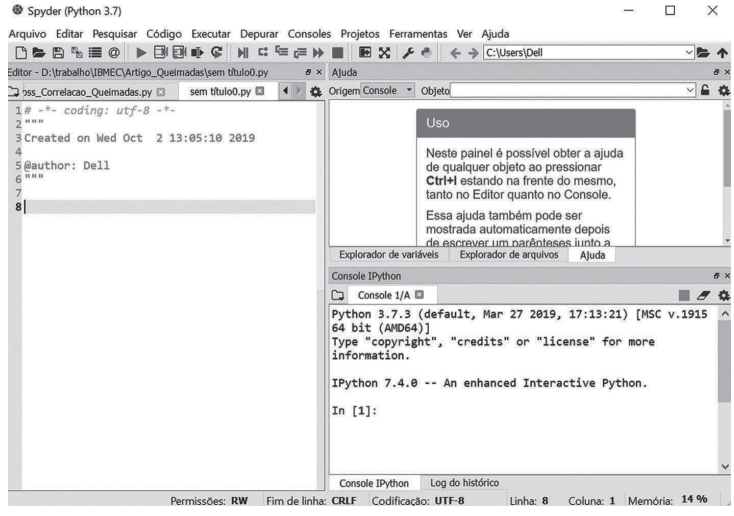


Figura 2.1 – Ambiente de programação para **script**.

Essa área de programação à esquerda na Figura 2.1 (em destaque) é onde os programas são elaborados, os algoritmos são definidos, as avaliações do compilador do Python são executadas e o programa de fato roda. Uma vez o algoritmo programado, a execução deve ser realizada pressionando ▶. Outra forma de rodar o algoritmo é apertando a tecla de função **F5** do teclado. Tudo o que pode ser feito no **Console** pode ser realizado na área de **script**, com a vantagem de não precisar reescrever todos os comandos.

Na Figura 2.2, é possível ver um exemplo clássico de programação, em que o usuário deve entrar no **input** com sua idade e o programa descobre o ano de nascimento.

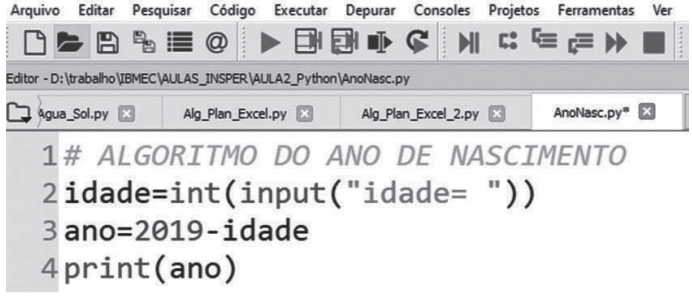


Figura 2.2 – Programação do algoritmo da idade.

Logo na primeira linha pode-se perceber o símbolo **#**. Toda vez ou em qualquer lugar que esse símbolo aparece indica que, a seguir, o que se tem é apenas um comentário. Essa linha não é processada como um comando, mas apenas uma anotação ou explicação do que uma linha ou o programa inteiro realiza.

O lado esquerdo do sinal de igual nunca pode ter uma operação, mas apenas uma variável para receber dados, informações, equações, impressões, entradas ou saídas de dados. Na figura em questão, as variáveis do problema são *idade* e *ano*.

Antes do **input** existe uma indicação de número inteiro, em que se usa **int**. Mesmo que o usuário digite um número real com casa decimal, com o comando **int**, o número é arredondado para o mais próximo inteiro. Sabendo-se a idade que foi informada pelo usuário, na linha a seguir, subtrai-se esse valor do ano. Por exemplo, para o ano de 2019, subtrai-se esse ano da idade e se obtém o ano de nascimento.

Para a visualização do resultado, como visto no capítulo anterior, utiliza-se o comando de impressão que aparece no **Console**, o **PRINT( )**. O resultado é visualizado no lado direito da Figura 2.1. Ao rodar esse algoritmo, o resultado é o apresentado a seguir.

```
In [2]: runfile('D:/trabalho/IBMEC/AULAS_INSPER/AULA2_Python/AnoNasc.py', wdir='D:/trabalho/IBMEC/AULAS_INSPER/AULA2_Python')

idade= 54
1965
```

No lado esquerdo, sempre aparece a linha do **Console**, no caso **In [2]:**, pois algum outro comando já foi digitado antes. Essas linhas são contínuas e a cada novo programa ou comando vão se sucedendo em ordem crescente.

Nessa linha, o Python indica a localidade do programa na máquina em que está rodando, o diretório e o nome do programa. Logo abaixo, começa o programa em si, perguntando ao usuário a idade. Neste exemplo, preenchemos com "54"; o programa entendeu e, na linha seguinte, subtraiu esse valor do ano 2019.

Como na última linha da programação existia o comando **PRINT( )**, observamos abaixo da entrada o número 1965, que veio da operação de subtração entre 2019 e a idade, valor este salvo na variável *ano*.

```
"""++++++++++++++++++++++++++++++
   ALGORITMO DO ANO DE NASCIMENTO
   ++++++++++++++++++++++++++++++
"""
idade=int(input("idade= "))
ano=2019-idade
print(ano)
```

Figura 2.3 – Outra forma de comentar um programa usando aspas.

Quando se deseja fazer muitas anotações de forma contínua em diversas linhas, outra forma de anotar é utilizar três aspas seguidas (" " "). Quando o comentário termina, fecha-se a região comentada com outras três aspas seguidas (" " "). Uma vez salvo o programa, a extensão a ser procurada no diretório é *.py. Assim, se salvamos o programa anterior como AnoNasc.py, podemos rodar esse programa diversas vezes e ele sempre vai estar salvo no diretório.

2.2 ALGORITMOS SIMPLES

Todos os comandos discutidos e construídos no **Console** podem ser aproveitados e expandidos para resoluções mais complexas na área de **script**. Os primeiros passos de toda linguagem de programação são a relação de entrada e saída de dados e a comunicação com o usuário.

EXEMPLO 2.1

Fazer um algoritmo para ler o raio de um círculo e imprimir sua área. Para programar esse simples algoritmo, o usuário precisa conhecer qual fórmula matemática permite descobrir a área de um círculo. A fórmula para a área do círculo é:

$$área = \pi\ raio^2$$

Assim como em toda linguagem de programação, também em Python o produto entre números e variáveis é representado por um asterisco (*). Na fórmula, o valor de π é constante, enquanto o raio do círculo é variável. Para cada novo valor do raio, a área é outra. Logo, a entrada dessa variável (*raio*) é necessária para que se efetue o cálculo da área.

O valor de π = 3.14... é conhecido pelo Python e está armazenado na biblioteca de matemática. Logo, é necessário para esse programa a importação de math.

```
# Cálculo da área de círculo
import math

raio=float(input("raio = "))
area=math.pi*raio**2
print("área = ",area)

```

A entrada desse programa no **input** deve ser um número real, que pode ou não ter casa decimal. Nesse caso, devemos informar ao Python que a entrada é com ponto flutuante, o que está representado no algoritmo como *float*. A elevação a segunda potência $raio^2$ é representada no Python por duplo asterisco (**), por isso no algoritmo temos *raio***2. O valor de π vem da biblioteca de matemática, por isso aparece como math.pi. No final desse algoritmo, para ver o resultado, usamos o comando **PRINT( )**. O resultado no **Console** é o seguinte:

```
raio = 3.5
área =  38.48451000647496
```

Entrando no **input** com o valor do raio = 3.5, o cálculo nos fornece como área do círculo o valor 38.48.

■

EXEMPLO 2.2

Fazer um algoritmo para ler o valor de *x* e de *y* com **input** e imprimir o resultado da equação:

$$z = \sqrt{x^2 + y^2} - \ln(x) + e^y$$

Como no exemplo anterior, precisamos entrar com números nas variáveis de entrada *x* e *y*. Para a raiz quadrada, logaritmo natural e número neperiano, devemos importar a biblioteca de matemática. O algoritmo é descrito a seguir.

```
# Exemplo
import math

x=float(input("x = "))
y=float(input("y = "))
z=math.sqrt(x**2+y**2)-math.log(x)+math.exp(y)

print("valor de z = " , z)
```

Entrando com x e y como ponto flutuante, a entrada é colocada na equação z, usando as denominações da biblioteca math para cada um dos operadores matemáticos. Para o caso de $x = 2$ e $y = 3$, o resultado é 22.99, como visto no **Console**.

```
x = 2

y = 3
valor de z =  22.997941018091712
```

■

EXEMPLO 2.3

A estimativa do retorno de uma carteira de investimentos pode ser encontrada ao se tomar retornos individuais. Assim, esses retornos são multiplicados pela probabilidade de ocorrerem e todos os valores são somados. Em termos de fórmula matemática, um retorno médio é dado por:

$$\overline{r} = \sum_{i=1}^{n} r_i * p_i$$

ou seja,

$$\overline{r} = r_1 * p_1 + r_2 * p_2 + r_3 * p_3 + \ldots r_n * p_n$$

Observando a tabela seguinte de retornos e suas probabilidades, pode-se fazer um programa em Python que apresente, ao final, o retorno esperado.

Retornos (R$)	Probabilidade
100	*0.2*
90	*0.7*
120	*0.1*

Nesse algoritmo, precisamos de seis entradas com o comando **input**, sendo três para os retornos financeiros e três para as probabilidades. A operação em si é muito simples, sendo apenas uma soma dos produtos.

```
1 # Exemplo
2
3 r1=float(input("retorno 1 = "))
4 r2=float(input("retorno 2 = "))
5 r3=float(input("retorno 3 = "))
6
7 p1=float(input("probabilidade 1 = "))
8 p2=float(input("probabilidade 2 = "))
9 p3=float(input("probabilidade 3 = "))
10
11 ret_medio = r1*p1 + r2*p2 + r3*p3
12
13 print("retorno médio = " , ret_medio)
```

Após a programação, ao rodar o algoritmo primeiro, ele pede ao usuário que apresente todos os retornos e depois todas as probabilidades. Ao final, o comando **PRINT( )** mostra o resultado do retorno médio. No caso do exemplo, o resultado para a tabela fornecida é R$ 95.

```
retorno 1 = 100

retorno 2 = 90

retorno 3 = 120

probabilidade 1 = 0.2

probabilidade 2 = 0.7

probabilidade 3 = 0.1
retorno médio =  95.0
```

■

Criação de programas simples na área de script do Python

2.3 LÓGICA CONDICIONAL (IF)

Uma das grandes colaborações de Alan Turing foi a imaginação do desenvolvimento de uma máquina "pensante". A interação entre máquina e homem foi um sonho movido por Turing, e seu pensamento foi expresso em toda sua vida de desenvolvimento de máquinas e algoritmos. O que diferencia um computador de uma máquina de calcular é que o computador pode ser programado para tomar decisões.

Toda linguagem de programação possui a estrutura conhecida como **se( )**, uma lógica condicional que desvia o caminho seguido pelo computador para uma regra colocada pelo programador. Na imaginação popular, diz-se que o computador está tomando a decisão sobre a finalização de um evento, mas na realidade o computador está seguindo um algoritmo colocado por um humano, dizendo a ele que caminho deve seguir se determinada regra for verdade ou que outros caminhos deve seguir se essa regra é falsa.

No Python, como em todas as linguagens de programação, a estrutura lógica é definida pelo comando conhecido como **if**, que necessita de uma condição e verifica se essa condição é verdadeira (*True*) ou falsa (*False*).

A estrutura do **if** no Python é a seguinte:

if < condição > :
    Comando-1
    Comando-2
    Comando-3
    .....
    Comando-n
else :
    Comando-1
    Comando-2
    Comando-3
    .....
    Comando-n

Diferentemente de outras linguagens, o condicional **if** do Python não possui **end** ou **end if** ou chaves para determinar que a lógica condicional terminou. O que marca o final da tomada de decisão é o alinhamento (conforme ilustrado pela linha tracejada no esquema anterior). Quando a lógica termina, o programador deve iniciar a continuação do programa fora do alinhamento do comando **if**, para que o Python saiba que aquela linha não faz parte da decisão.

Outra diferença é que o Python não possui **then**, geralmente usado por outras linguagens para indicar o que o computador deve fazer. Em vez disso, **then** é substituído por dois-pontos (**:**).

Os sinais utilizados no Python para as expressões lógicas nas decisões são:

Operador	Descrição
>	*maior que*
<	*menor que*
>=	*maior ou igual*
<=	*menor ou igual*
= =	*igual a*
!=	*diferente*

Em relação a outras linguagens, a expressão dentro do **if** para verificar se uma condição é igual a outra é "= =". Para dizer ao Python que as duas condições são não iguais, usa-se a expressão de diferença: a exclamação seguida de sinal de igual.

EXEMPLO 2.4

Ler o preço de dois resultados de uma operação financeira. A primeira entrada é o resultado do primeiro dia e a segunda entrada é o resultado do segundo dia. Se a diferença entre o segundo e o primeiro dia for *positiva*, deve aparecer no **Console** a palavra "lucro", caso contrário aparece a palavra "prejuízo".

Este é o algoritmo em Python:

```
# Calculo retono

p1=float(input("resultado do primeiro dia = "))
p2=float(input("resultado do segundo dia = "))
retorno=p2-p1
if retorno>=0:
    print("lucro")
else:
    print("prejuizo")
```

Como se percebe, a leitura do resultado é um **input** com ponto flutuante, e a operação de subtração dos dois valores vai determinar o lucro ou prejuízo. A "<condição>" aqui é a expressão do retorno que se aplica ao se perguntar se é maior ou igual a zero (> = 0). Os dois-pontos ao final da linha do **if** indicam que a condição terminou e a linha logo abaixo é sempre a decisão a ser tomada em caso verdadeiro (*True*). Então, se o retorno é positivo ou igual a zero, a linha logo abaixo é o que o computador deve fazer. Nesse caso, imprimir no **Console** a palavra "lucro".

A palavra **else** e os dois-pontos indicam que, se o retorno não é positivo (*False*), o computador deve seguir a instrução logo abaixo dessa palavra. Nesse caso, imprimir a palavra "prejuízo".

Supondo que *p1* = 10 e *p2* = 20, temos a seguinte resposta no **Console**:

```
resultado do primeiro dia = 10

resultado do segundo dia = 20
lucro
```

Supondo que *p1* = 10 e *p2* = 5, o resultado é a palavra "prejuízo".

```
resultado do primeiro dia = 10

resultado do segundo dia = 5
prejuizo
```

O alinhamento dos comandos dentro do **if** é importante. Por exemplo, vamos supor que desejamos que, além de imprimir "lucro", o computador imprima também logo abaixo "fiquei feliz". E caso o resultado seja negativo imprima, além de "prejuízo", a frase "fiquei triste". Nesse caso, o algoritmo correto fica como apresentado a seguir.

```
# Calculo retono

p1=float(input("resultado do primeiro dia = "))
p2=float(input("resultado do segundo dia = "))
retorno=p2-p1
if retorno>=0:
    print("lucro")
    print("fiquei feliz")
else:
    print("prejuizo")
    print("fiquei triste")
```

Para os valores *p1* = 10 e *p2* = 20, temos o resultado:

```
resultado do primeiro dia = 10

resultado do segundo dia = 20
lucro
fiquei feliz
```

No entanto, vamos supor que por descuido não colocamos o último *print* alinhado ao comando **if**. Como pode ser visto a seguir, colocamos propositadamente a impressão de "fiquei triste" desalinhado ao comando **if**, alinhado exatamente à margem esquerda da janela de script.

```
# Calculo retono

p1=float(input("resultado do primeiro dia = "))
p2=float(input("resultado do segundo dia = "))
retorno=p2-p1
if retorno>=0:
    print("lucro")
    print("fiquei feliz")
else:
    print("prejuizo")
print("fiquei triste")
```

Ao rodar esse algoritmo, novamente repetimos os valores *p1* = 10 e *p2* = 20 e temos como resultado:

```
resultado do primeiro dia = 10

resultado do segundo dia = 20
lucro
fiquei feliz
fiquei triste
```

Podemos observar que o resultado do retorno foi positivo e que a palavra "lucro" e a frase "fiquei feliz" estão corretas. Mas logo abaixo apareceu também a frase "fiquei triste". Isso porque, uma vez desalinhado do **if**, o comando de impressão (**print**) vai aparecer sendo o retorno positivo ou negativo. Fora do alinhamento, o computador é obrigado a colocar forçadamente a palavra "fiquei triste". E, claramente, o algoritmo está errado.

Muitas vezes, apenas o caso *True* ou *False* não resolve a lógica para a tomada de decisão. Por exemplo, uma vez falsa uma afirmação, temos outra pergunta para direcionar a lógica para posições de decisões mais refinadas ou particulares. Nesse sentido, precisamos de um comando **caso-contrário-se** ou **senão**, muito utilizado em diversas linguagens.

No Python, essa estrutura se chama **if-elif-else**. O **elif** é na realidade uma pergunta dentro do ramo falso da pergunta anterior. E, por ser uma nova pergunta, necessita de uma condição para analisar a verdade ou falsidade da afirmação e, então, tomar a decisão.

Outro fato é que, como se tem sempre uma pergunta dentro de outra pergunta encadeada, também necessita estar alinhado ao primeiro **if** e ter ao final o sinal de **:**. No esquema a seguir, é possível ver que os alinhamentos continuam obrigatórios para todos os **elif** e também para o **else**.

if < condição > :

Comando-1

Comando-2

Comando-3

.....

Comando-n

elif < condição > :

Comando-1

Comando-2

Comando-3

.....

Comando-n

elif < condição > :

Comando-1

Comando-2

Comando-3

.....

Comando-n

else :

Comando-1

.....

Comando-n

Comando-1

Comando-2

Comando-3

.....

Comando-n

■

EXEMPLO 2.5

Fazer um algoritmo para ler o preço de um produto. Se o preço for menor que R$ 18, deve imprimir "barato". Se o preço for maior que R$ 18, mas abaixo de R$ 25, deve imprimir "adequado". Se o preço for maior que R$ 25 e menor que R$ 32, deve imprimir "caro". Se o preço for maior que R$ 32, deve imprimir "extremamente caro".

```
1 # Calculo faixa de preco
2
3 p1=float(input("preco 1 = "))
4
5 if p1<=18:
6     print("barato")
7 elif (p1>18) and (p1<=25):
8     print("adequado")
9 elif (p1>25) and (p1<=32):
10    print("caro")
11 else:
12    print("extremamente caro")
```

Como pode ser observado, o algoritmo trabalha com *True* e *False* encadeando os preços para as faixas com as frases correspondentes. Se o preço é menor que R$ 18, então temos uma verdade (*True*), o que leva na linha logo abaixo o comando de impressão "barato".

Nesse caso, o computador vai pular todos os outros **elif** e seguir o programa para depois da linha 12. Mas se o preço for maior que R$ 18 e menor que R$ 25, o computador pula a primeira linha (linha 6) e vai para a linha 8, imprimindo "adequado".

Também se pode fazer uso de união e intersecção de condições usando **and** e **or** para agrupar duas ou mais condições. As combinações de **and** e **or** tornam a lógica sempre interessante para direcionar o computador na tomada de decisão correta.

Novamente deve-se observar o alinhamento, sempre em relação ao primeiro **if**, que indica a primeira condição lógica para a decisão.

Vamos supor que o preço do produto foi *p1* = R$ 30. A resposta do algoritmo é a seguinte no **Console**.

```
preco 1 = 30
caro
```

Se o preço for R$ 40, temos a seguinte resposta.

```
preco 1 = 40
extremamente caro
```

■

EXEMPLO 2.6

Fazer um algoritmo para ler um número inteiro *n* e decidir se ele é par ou ímpar.

```
1 # Decide se um num é par ou ímpar
2
3 n=int(input("entre com o número = "))
4
5 resto=n % 2
6 if (resto==0):
7     print("par")
8 else:
9     print("ímpar")
10
```

Esse é um problema clássico para tomada de decisão baseada apenas em uma informação e na definição matemática de par e ímpar. No caso do Python, como em todas as linguagens, um comando interessante é **n % m**. O comando **%** é conhecido como módulo ou resto da divisão entre *n* e *m*. Se fazemos **n/m**, estamos encontrando o quociente. Assim, 23/4 fornece quociente 5 e resto 3. O comando **%** nos retorna exatamente 3 como resultado.

No caso desse algoritmo, forçamos a entrada de um número inteiro, colocando no **input** o **int**, para obrigar o número digitado a ser inteiro. Assim, uma vez lido *n*, se o resto da divisão desse número por 2 é zero, ele é par, caso contrário, é ímpar. O comando de impressão dentro do **if** decide o que imprimir dependendo da condição verdadeira ou falsa.

Por exemplo, entrando com *n* = 113, a resposta no **Console** é a seguinte.

```
entre com o número = 113
ímpar
```

■

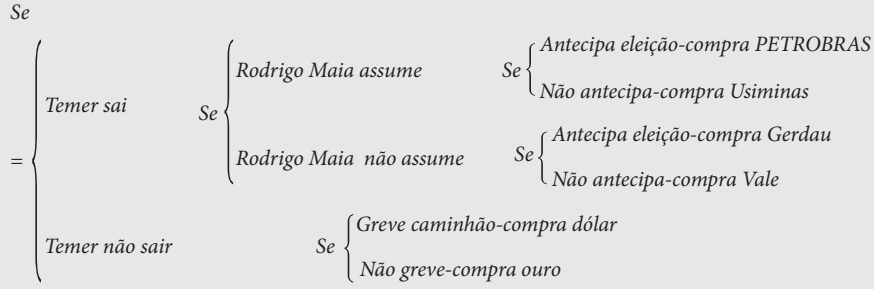
EXEMPLO 2.7

Em 2018, tivemos uma grande greve de caminhoneiros por conta da atualização da tarifa de transportes que recebiam. O então presidente Michel Temer sofria ameaça de abertura de processo de *impeachment* e, se assim ocorresse, Rodrigo Maia, presidente da Câmara dos Deputados, teria de assumir o cargo de presidente.

Um grande cliente estava preocupado com a política nacional e desejava um programa em Python em que ele poderia assumir riscos de investimentos conforme os desdobramentos da tal greve.

Desejava-se então um algoritmo para a tomada de decisão envolvendo o jogo de xadrez político e para concluir qual o melhor tipo de investimento. Perguntas devem ser feitas no **input** para passar ao ramo seguinte da decisão. A decisão final deve ser emitida via **print** no **Console** para cada tipo de investimento, dizendo se o usuário deve comprar ou não cada um dos ativos do último ramo.

Observe que as variáveis de entrada na leitura das perguntas são do tipo *string*.



Os ativos que o cliente desejava comprar eram as ações da Petrobras, da Usiminas, da Gerdau ou da Vale. Em outro ramo, a compra deve ser de dólar ou ouro. A solução deve ser **if** encadeado, como pode ser observado a seguir.

```
#Encadeamento de If
perg1=str(input("Temer sai?"))
if perg1=="s":
    perg2=str(input("Maia assume a presidência?"))
    if perg2=="s":
         perg3=str(input("Antecipa a eleição?"))
         if perg3=="s":
             print("comprar Petrobras")
         else :
                print("comprar Usiminas")
    else :
           perg3=str(input("Antecipa a eleição?"))
           if perg3=="s":
                print("comprar Gerdau")
           else :
                print("comprar Vale")
else :
    perg2=str(input("Greve de caminhoneiros?"))
    if perg2=="s":
        print("comprar dolar")
    else :
        print("comprar ouro")
```

Vamos supor que o processo contra o presidente Michel Temer fosse aprovado e ele fosse removido do cargo e que Rodrigo Maia assumisse a presidência e não antecipasse a eleição. A decisão do computador teria sido "comprar Usiminas".

```
Temer sai?s

Maia assume a presidência?s

Antecipa a eleição?n
comprar Usiminas
```

2.4 ITERAÇÃO COM WHILE

O que transformou o computador em uma máquina fundamental em nossas vidas foi a capacidade de realizar cálculos extremamente rápidos e eficientes baseados em regras lógicas. Para que o computador repita uma tarefa com precisão e eficiência quantas vezes forem necessárias, essa repetição precisa ser programada em comandos que obriguem o algoritmo a todo instante retornar ao ponto de partida. Essa repetição ou retorno ao ponto inicial de uma lógica para decisão é conhecida como *loop*.

$$S = 1+2+3+4+5+6+...$$

Vamos supor que desejamos fazer a soma de certa quantidade n de números inteiros e imprimir o resultado final. A ideia de variável nos remete ao exemplo simples que retrata uma variável como uma caixa de memória que recebe valores mais recentes e apaga os anteriores. Isso permite que a variável sempre necessite de pouco espaço de memória, mas seja possível fazer cálculos com grandes quantidades de valores.

Em computação, a noção de igualdade é apenas uma simplificação do que realmente significa uma variável receber um valor. Quando uma variável recebe uma constante, um número ou letra, o símbolo correto deve ser uma seta, pois uma posição de memória física dentro do computador está sendo alimentada por número. Vamos supor que desejamos que a variável s receba o número 1. A nomenclatura correta é:

$$s \leftarrow 1$$

Do ponto de vista de computação, as linguagens adotam o sinal de igual para representar essa alimentação na memória. Então escrevemos:

$$s = 1$$

Isso não parece ser muito problema, mas é para o entendimento de iteração. Quando desejamos adicionar um valor de memória a outro que foi lido pelo programa, dizemos

ao computador que deve somar o novo número e colocar na caixa de memória anterior, onde o antigo número estava armazenado. Esse fato é o mesmo que:

$$s \leftarrow s + 2$$

Nesse caso, como antes, o armazenamento tinha como valor de *s* o número 1; após receber o número 2, o novo valor é 3, pois obrigamos o computador a somar 2 ao *s* e, no final, substituir (por isso a seta) o valor anterior. Na linguagem de computador escrevemos:

$$s = s + 2$$

A cada passo um novo número deve ser adicionado, ou seja, temos de forma repetitiva (iterativa) a sequência:

$$s \leftarrow s + 3$$
$$s \leftarrow s + 4$$
$$s \leftarrow s + 5$$
.....

Os termos novos que são adicionados a cada repetição (*loop*) são geralmente chamados de contadores, pois servem como referência de quantas vezes a referida soma já foi repetida. Como visto na Figura 2.4, a cada novo valor salvo na caixa de memória, o valor que está dentro no passo anterior é aproveitado para somar ao novo valor que está chegando à caixa. Um comando que auxilia essa repetição, iteração ou *loop* é o comando **while**.

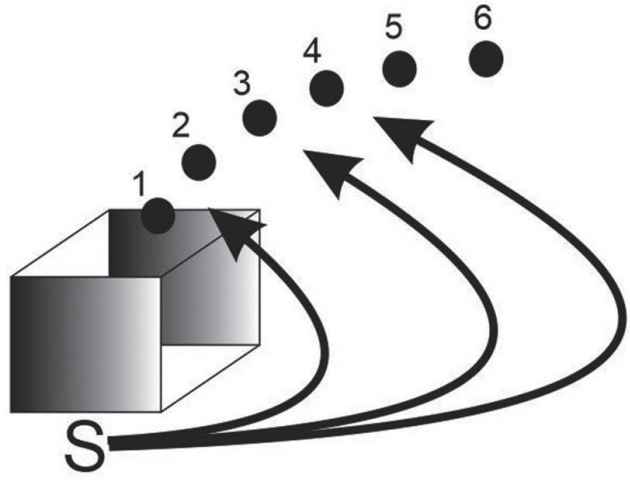


Figura 2.4 – Iteração com *loop* em variável.

No Python, temos o seguinte esquema para o funcionamento do **while**:

< contador >

while < condição de parada >:

comando 1

comando 2

comando 3

...

comando *n*

Nesse esquema, o "contador" pode ou não ser usado pelo **while** e serve para contar o número de *loops* que já foram realizados. A "condição de parada" serve para delimitar o número de voltas ou mesmo outra condição lógica em que o computador deve permanecer "preso" à ordem de repetição. Assim, "comando 1", "comando 2" etc. são ordens que devem ser repetidas a cada nova volta (*loop*) do **while**. Esses comandos podem ser impressões de resultados no **Console**, mas também podem ser operações matemáticas, construções de gráficos, salvamentos em arquivos, abertura de arquivos etc.

Então, vamos voltar à soma de *n* números inteiros 1, 2, 3, … *n*. O algoritmo em Python é:

```
1    # Soma de n números
2    n=int(input(" qtde de números = "))
3    cont=1
4    s=0
5    while cont<=n:
6        s=s+cont
7        cont=cont+1
8        print(s)
```

O computador pede ao usuário a quantidade de números inteiros que deseja somar, começando por 1. A variável *cont* começa com o número 1, pois será o primeiro *loop* que acontece no **while**. A variável *s* inicia com zero, pois a caixa de memória deve começar vazia antes de o **while** começar a rodar. Finalmente a condição de parada nesse caso é quando o contador de voltas (*loop*) chegar ao número de termos *n* desejado para finalizar a soma.

Como o contador modifica seu valor? Sempre tomando o valor do passo anterior e somando o número 1, pois a definição-padrão do **while** é de uma em uma volta. O valor da caixa de memória *s* é atualizado pelo novo valor do contador e seu resultado é, passo a passo, impresso no **Console**. Ao final, o que se vê no **Console** para os seis primeiros números inteiros é:

```
 qtde de números = 6
1
3
6
10
15
21
```

O que acontece se o comando **PRINT( )** não está alinhado com o **while**?

```
1    # Soma de n números
2    n=int(input(" qtde de números = "))
3    cont=1
4    s=0
5    while cont<=n:
6            s=s+cont
7            cont=cont+1
8    print(s)
```

O resultado no **Console** nesse caso é:

```
 qtde de números = 6
21
```

Como se pode ver, a soma não apresenta erro e o resultado é o mesmo do anterior, mas, como o **PRINT( )** não estava alinhado, o computador entendeu que era para imprimir apenas o resultado final, e não o passo a passo.

EXEMPLO 2.8

Fazer um algoritmo para encontrar os múltiplos de 5 até 20 (inclusive). Podemos fazer uma soma de 5 em 5, ou aproveitar o contador de loops para multiplicar por 5. Também se pode notar que nem sempre é necessário usar como parada o critério envolvendo o contador. Esse é um caso em que a condição de parada é o número 20.

```
1    # Gera multiplos de 5 a 20
2
3    cont=1
4    x=0
5    while x<20:
6            x=cont*5
7            cont=cont+1
8            print(x)
```

O resultado no **Console** é:

```
5
10
15
20
```

■

EXEMPLO 2.9

Fazer um algoritmo para encontrar a soma dos primeiros números *n* pares, pode ser feito desta forma:

```
1    # Soma de n números pares
2    n=int(input(" qtde de números = "))
3    cont=0
4    s=0
5    while cont<=n:
6        if cont % 2 ==0:
7            s=s+cont
8            print(s)
```

O resultado no **Console** para *n* = 10 é:

```
 qtde de números = 10
0
2
6
12
20
30
```

■

Passo a passo do uso dos comandos **if**, **while** *e* **for**

EXEMPLO 2.10

Fazer um algoritmo para imprimir uma lista *sem nomes repetidos*. Esse é um exemplo para relembrar que uma lista pode conter *strings* (letras) e ser do tipo texto. Ainda assim, pode-se usar o **while** para selecionar palavras específicas para serem impressas no **Console**. Vamos supor que queremos imprimir todos os nomes diferentes do nome "mário" da seguinte lista.

```
1    # Imprimir parte de uma lista
2    nomes=['mário','josé','mário','maria','carlos','ana','mário']
3    n=len(nomes)
4    cont=1
5    while cont<=n-1:
6          if nomes[cont] != 'mário':
7                print( nomes[cont] )
8          cont=cont+1
```

No **Console**:

```
josé
maria
carlos
ana
```

■

2.5 INTERRUPÇÃO NO WHILE (BREAK)

Muitas vezes, desejamos que a repetição seja interrompida por algum motivo, como a localização já alcançada de um termo, um número de avaliações já atingido ou uma precisão de operação matemática que chegou a seu limite. Nesse caso, não queremos que o computador continue sua tarefa e podemos "estourar" o contador para que a iteração cesse. Para isso, no Python, pode-se fazer uso do comando **break**. Vamos ver o exemplo a seguir.

EXEMPLO 2.11

Fazer um algoritmo para somar n números inteiros.

$$S = 1 + 2 + 3 + 4 + \ldots$$

Depois, deve-se interromper a soma quando o primeiro resultado par aparecer.

```
1    # Soma de n números
2
3    n=int(input(" qtde de números = "))
4    cont=1
5    s=0
6    while cont<=n:
7          s=s+cont
8          cont=cont+1
9          if s % 2 == 0: break
10         print(s)
```

No **Console**:

```
 qtde de números = 10
1
3
```

O oposto do comando **break** é o comando **continue**. Nesse caso, exigimos que o computador continue sua execução caso alguma condição seja verdadeira ou falsa.

■

EXEMPLO 2.12

Fazer um algoritmo para somar *n* números inteiros. Usamos:

$$S = 1 + 2 + 3 + 4 + \ldots$$

Depois, deve-se imprimir as somas parciais cujo resultado é ímpar.

```
1     # Soma de n números
2
3     n=int(input(" qtde de números = "))
4     cont=1
5     s=0
6     while cont<=n:
7           s=s+cont
8           cont=cont+1
9           if s % 2 == 0: continue
10          print(s)
```

No **Console**:

```
 qtde de números = 10
1
3
15
21
45
55
```

■

Passo a passo do uso dos comandos **break** *e* **continue**

2.6 ITERAÇÃO COM FOR E RANGE

O segundo modo de controle de repetição ou iteração é por meio do comando do Python conhecido como **for**. Sua função é a mesma do **while**, mas sua estrutura tem uso diferente. No **while**, desejamos que uma tarefa seja feita enquanto a condição lógica é verdade. No caso do **for**, o computador realiza a tarefa de qualquer maneira, pois não existe condição lógica de parada. Apenas indicamos na linguagem seu início e fim como visto a seguir.

For *<contador>* ***in*** *range (início, fim, passo)* ***:***

Como nos casos de **while** e **if**, os comandos abaixo do **for** devem seguir o alinhamento para que o computador entenda que estão dentro das repetições a serem realizadas. Por exemplo, para imprimir no **Console** dez vezes a palavra "bovespa", podemos programar no Python usando **for** da seguinte maneira.

```
1    # imprimindo nome
2    n=int(input('total de repetições = '))
3
4    for i in range(0,n):
5        print('bovespa')
```

No **Console**:

```
total de repetições = 10
bovespa
bovespa
bovespa
bovespa
bovespa
bovespa
bovespa
bovespa
bovespa
bovespa
```

Para imprimir números iniciando em 5 e indo até 10, devemos programar o Python para a seguinte rotina.

```
1    # imprimindo numeros
2
3    for i in range(5,11):
4        print(i)
```

Devemos observar que, para a impressão do número 10, precisamos colocar na finalização do **for** o número 11. Isso porque, como na lista, a parada é sempre $(n - 1)$ termos.

```
5
6
7
8
9
10
```

O passo da iteração é exatamente de quantos em quantos termos o **for** deve ser repetido. Vamos supor que, no exemplo anterior, desejamos imprimir saltado de 2 em 2.

```
1    # imprimindo numeros
2
3    for i in range(5,11,2):
4        print(i)
```

No **Console**:

```
5
7
9
```

EXEMPLO 2.13

Fazer um algoritmo para somar *n* números reais quaisquer; basta o usuário entrar com os valores usando **input**.

A diferença entre esse algoritmo e os anteriores é que, neste exemplo, os números não estão mais em sequência, mas são escolhidos pelo usuário. Pelo **input**, o usuário alimenta um novo número, passo a passo, a cada iteração. Para cada novo valor lido, o **float** transforma o número em representação real com ponto flutuante na memória. Conforme mostrado a seguir, o resultado é uma soma parcial que está dentro do **for** e, ao fim de tudo, a soma total.

```
1    # Imprimindo numeros
2
3    n=int(input("total de números = "))
4    s=0
5
6    for i in range(0,n):
7        x=float(input("número = "))
8        s=s+x
9        print(s)
10   print("soma total = ",s)
```

No **Console**, para o total *n* = 5 números [–2, 4, 5, 7, –1]:

```
total de números = 5

número = -2
-2.0

número = 4
2.0

número = 5
7.0

número = 7
14.0

número = -1
13.0
soma total =  13.0
```

2.7 TRABALHANDO COM FOR EM LISTAS

Como **while**, o **for** pode ser usado para fatiar uma lista, escolher elementos, separar, filtrar, enfim, agilizar diversos processos complexos envolvendo listas. Podemos usar o **for** para imprimir palavras de uma lista. Por exemplo, na lista de nomes do algoritmo a seguir, obrigamos o **for** a começar na posição inicial (índice 0) e terminar no último elemento. Para generalizar, pode-se usar o comando **len** para descobrir o total de elementos da lista. E, como a lista começa com índice 0, devemos lembrar que a iteração vai até (n – 1) elementos da lista.

Então, para a lista_nomes = [‘um’,‘dois’,‘três’,‘quatro’,‘cinco’], o primeiro elemento é lista_nomes[0] = ‘um’ e o último é lista_nomes[4] = ‘cinco’.

```
1    # Imprimindo palavras
2
3    lista_nomes = [‘um’, ‘dois’, ‘três’ ,’quatro’ ,’cinco’]
4
5    for i in range(0, len(lista_nomes)):
6            print( lista_nomes[i] )
```

No **Console** a impressão será:

```
um
dois
três
quatro
cinco
```

Outra aplicação de listas usando **for** é para adicionar elementos e criar uma nova lista. Para isso, primeiro devemos iniciar uma lista vazia, usando [ ]. Com a iteração, colocamos novos elementos na lista vazia usando o comando de lista já mencionado antes como **append**.

```
1    # Imprimindo palavras
2
3    lista_nomes=[‘um’,’dois’,’três’,’quatro’,’cinco’]
4    nova=[ ]
5    for i in range(0, len(lista_nomes)):
6            nova.append(lista_nomes[i])
7            print(nova)
```

O **append** vai incluindo os elementos da lista_nomes passo a passo na nova lista que está vazia. No **Console**, a impressão passo a passo é a seguinte.

```
['um']
['um', 'dois']
['um', 'dois', 'três']
['um', 'dois', 'três', 'quatro']
['um', 'dois', 'três', 'quatro', 'cinco']
```

Se o **PRINT( )** não está alinhado com o **for**, vamos observar apenas a nova lista já preenchida.

```
1    # Imprimindo palavras
2
3    lista_nomes=['um','dois','três','quatro','cinco']
4    nova=[ ]
5    for i in range(0, len(lista_nomes)):
6            nova.append(lista_nomes[i])
7    print(nova)
```

Resultado no **Console**:

```
['um', 'dois', 'três', 'quatro', 'cinco']
```

EXEMPLO 2.14

Fazer um algoritmo para salvar a lista_nomes = ['um','dois','três','quatro','cinco'] em ordem inversa usando o **for**.

Lembrando que o comando **for** sempre parte de um **range** com início, fim e o passo, devemos começar a rodar a iteração começando do último elemento, mas usando passo negativo. O **append** recebe os nomes na ordem inversa, usando o contador *i* subtraído do valor total da lista dentro dos índices. O algoritmo será:

```
1    # Imprimindo palavras
2
3    lista_nomes=['um','dois','três','quatro','cinco']
4    nova=[ ]
5    for i in range(len(lista_nomes)-1,-1,-1):
6            nova.append(lista_nomes[i-len(lista_nomes)])
7    print(nova)
```

No **Console**:

```
['um', 'dois', 'três', 'quatro', 'cinco']
```

■

EXEMPLO 2.15

Fazer um algoritmo para fatiar uma lista em que os elementos são também listas de letras. Por exemplo, para a lista de letras: [['a','b','c'], ['a','d','f','a'] ,['b','b','d','c']], deseja-se criar uma nova lista apenas com as letras, e não com as listas.

Muitas vezes, listas são obtidas do Excel ou de banco de dados e formadas por outras listas com palavras, bairros, cidades, países, CNPJ etc. Os elementos dessa importação original são listas. E, nesse caso, estudos estatísticos não podem ser realizados, a menos que tratemos os elementos de forma isolada.

Para fatiar esses elementos, podemos lembrar e fazer uso do **append**, visto anteriormente, para que cada elemento seja separado e adicionado em lista vazia. A estratégia então é iniciar uma lista vazia **[ ]** e depois, com o uso do comando **for**, percorrer a lista original.

Esse é um caso em que o contador do **for** não é índice, mas um elemento que percorre automaticamente uma lista. Adotando a variável *p* para esse primeiro **for**, a parada do **for** não é um número, mas o final da lista. Nesse primeiro **for** o resultado para *p* será o seguinte:

p[0] = [‘a’,‘b’,‘c’]

p[1] = [‘a’,‘d’,‘f’,‘a’]

p[2] = [‘b’,‘b’,‘d’,‘c’]

Então, para separar os elementos dentro dessas listas que são elementos de *p*, precisamos de outro **for**; para isso o **append** coloca cada elemento de *p* de forma separada na lista que estava vazia.

```
1    # Exemplo para fatiar uma lista com for
2
3    letras=[[‘a’,’b’,’c’],[‘a’,’d’,’f’,’a’],[‘b’,’b’,’d’,’c’]]
4    i=1
5    letras_nova=[ ]
6    for p in letras:
7            for x in p:
8            letras_nova.append(x)
9
10   print(‘++++++++++++ Nova Lista ++++++++++++’)
11   print(letras_nova)
```

O resultado no **Console** do Python é:

```
+++++++++ Nova Lista +++++++++
['a', 'b', 'c', 'a', 'd', 'f', 'a', 'b', 'b', 'd', 'c']
```

■

Programação do Exemplo 2.15 com comentários sobre os passos e os possíveis erros que podem ocorrer nesse procedimento

2.8 USANDO FOR EM SÉRIES MATEMÁTICAS

Séries e sequências matemáticas sempre fizeram parte dos testes no início das linguagens de programação, por oferecerem, além da velocidade no cálculo de funções mais elaboradas, a capacidade de verificar a convergência da série ou mesmo a velocidade com que os algoritmos podem ser otimizados.

EXEMPLO 2.16

Fazer um algoritmo ler com **input** o valor de *n* inteiro e, ao final, imprimir o fatorial desse número. O fatorial de um número é representado em matemática como:

$$n! = 1 * 2 * 3 * 4 * \ldots * (n-1) * n$$

Como nos algoritmos que envolvem a soma, no caso de um produto, o elemento neutro é a unidade. Iniciamos a variável *fat* desse exemplo como *fat* = 1. Então, dentro do **for**, obrigamos o algoritmo a tomar o valor anterior a cada iteração e multiplicar pelo novo contador.

Pode-se observar que a parada agora é (*n* + 1), exatamente porque o Python sempre termina no valor (*n* – 1).

```
1    # fatorial
2
3    n=int(input("n = "))
4    fat=1
5    for i in range(1,n+1):
6            fat=fat*i
7    print("fatorial = ",fat)
```

No **Console** para *n* = 3:

```
n = 3
fatorial =  6
```

Como observação, temos de lembrar que esse é apenas um exemplo de como programar usando **for**, visto que no Python a função fatorial já existe. Para tanto, precisa-se importar a biblioteca math e usar a função **factorial**, como a seguir.

```
In [24]: import math

In [25]: math.factorial(3)
Out[25]: 6
```

■

EXEMPLO 2.17

Fazer um algoritmo ler com **input** o valor n de termos e o valor x, para calcular a soma de:

$$S = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!}$$

Esse algoritmo é uma composição dos algoritmos iniciais usando a soma dos números em sequência e o algoritmo do fatorial anterior. Necessita-se, nesse caso, de dois comandos **input**, um para a quantidade de termos n a ser somada e outro para o valor de x. Ambos são valores externos e fornecidos pelo usuário.

```
1    # Algoritmo do Exponencial
2
3    n=int(input("n = "))
4    x=float(input("x = "))
5    fat=1
6    s=1
7    for i in range(1,n+1):
8            fat=fat*i
9            s=s+x**i/fat
10   print("Soma = ",s)
```

Para n = 10 termos e x = 1, temos no **Console** a resposta:

```
n = 10

x = 1
Soma =  2.7182818011463845
```

Essa soma é bastante conhecida em matemática e representa a função exponencial, que também já está programada na biblioteca math do Python com o nome de **exp**. Usando o mesmo valor x = 1 e chamando a função do math, temos:

```
In [28]: math.exp(1)
Out[28]: 2.718281828459045
```

Para melhorar a precisão do algoritmo e aproximar a resposta da função da biblioteca, basta aumentar o número de termos. Os resultados estarão cada vez mais próximos.

■

2.9 EXERCÍCIOS

1. Fazer algoritmo e programa em Python: o usuário entra com um número usando **input** e o programa imprime com **PRINT( )**, se o número é positivo ou negativo.

2. Dados três lados de um triângulo qualquer, fazer um programa para descobrir se o triângulo é equilátero, isósceles ou escaleno. Entrar com os valores com **input**.

3. Um *website* de compras deseja deixar um link para os franqueados entrarem com o valor de compra de um produto no **input** e o valor de venda no **input** e descobrir se o lucro foi menor que 10%, entre 10% e 20%, ou superior a 20%. Para tanto, deve-se fazer um programa em Python que imprima a mensagem com **PRINT( )** dizendo em qual faixa a mercadoria do comerciante se localiza.

4. O "suporte" de uma ação é calculado como 30% do intervalo histórico de uma ação (máximo subtraído do mínimo). A "resistência" é calculada como o valor de 60% do intervalo histórico. Veja o exemplo:

   Baixa histórica: 1.50

   Alta histórica: 2.20

   Suporte: 1.5 + (2.20 – 1.50) * 0.3

   Resistência: 1.5 + (2.20 – 1.5) * 0.6

   Fazer um programa em Python em que o usuário forneça com **input** o valor mais baixo historicamente e o valor mais alto. O programa pede o valor atual da ação também com **input** e diz com **PRINT( )** qual é o suporte, qual é a resistência e se o preço da ação está dentro da faixa de suporte-resistência ou fora dela.

5. Considere uma equação do segundo grau:

$$Ax^2 + Bx + C = 0$$

   Utilizando-se $\Delta = B^2 - 4AC$, escrever um programa em Python que calcule as raízes da equação tal que:

   (i) Se não há raízes ($\Delta < 0$), o programa retorna um **PRINT** "não existem raízes reais".

   (ii) Se há uma única raiz ($\Delta = 0$), o programa mostra ao usuário a única raiz calculada como: $x1 = -B / (2*A)$.

   (iii) Se há duas raízes, mostre ao usuário calculando-as desta forma:

$$x_1 = \frac{-B + \sqrt{\Delta}}{2A} \qquad x_2 = \frac{-B - \sqrt{\Delta}}{2A}$$

6. A tabela a seguir fornece os descontos de uma compra. Faça um programa em Python que leia o valor de uma compra, determine o desconto a ser aplicado e calcule o valor a ser pago pelo cliente. Imprimir o valor a ser pago com **PRINT( )**.

Tabela 1 – Descontos

Valor da compra (R$)	Desconto (%)
Entre 0 e 20	5
Entre 21 e 50	10
Entre 51 e 100	15
Entre 101 e 1.000	20
Maior que 1.000	30

7. Uma loja de departamentos de roupas de cama e banho percebeu que, se separar seu servidor de vendas em pedidos pares e ímpares, o atendimento fica mais ágil. Fazer um algoritmo em Python em que se deve ler o número que representa o total de vendas *n* de uma sequência de números naturais e, posteriormente, ler a própria sequência com **input**. Armazenados os números da sequência, o programa deve mostrar com **PRINT( )** o valor da soma dos números pares (SP) e o valor da soma dos números ímpares (SI).

8. O financiamento de um automóvel foi contratado respeitando a série matemática a seguir para 70 meses. Criar um algoritmo em Python para calcular a soma seguinte até o termo *n* que o usuário desejar, seguindo a lógica a seguir.

$$S = \frac{70}{7} + \frac{69}{14} + \frac{68}{21} + \frac{67}{28} + \ldots$$

9. Fazer um algoritmo em Python que leia uma quantidade *n* de valores numéricos fornecidos pelo usuário por meio do **input**. O programa deve contar quantos pares e quantos ímpares existem e imprimir a soma com o **PRINT( )**.

10. A área abaixo de uma função matemática fornece alguns resultados importantes em termos de demanda, custo, lucro marginal, entre outros resultados na área de finanças. A integral seguinte não pode ser resolvida utilizando as técnicas usuais de cálculo diferencial e integral. Mas, pela aproximação da integral em *n* termos da série seguinte, tem-se uma boa aproximação do seu valor exato.

$$\int_0^x e^{-u^2}\,du = x - \frac{x^3}{3*1!} + \frac{x^5}{5*2!} - \frac{x^7}{7*3!} + \cdots$$

Assim, construir um programa em Python em que o usuário entre com a variável *n* (que representa a quantidade de termos) com o comando **input** e *x* (que representa

o limite de integração) com o comando **input**. Ao final, o programa imprime com **PRINT( )** o resultado da integral (a soma do lado direito do sinal de igualdade).

11. O valor de π pode ser encontrado com a seguinte série matemática:

$$s = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \ldots$$

Para encontrar o valor mais próximo possível conhecido, é necessário o comando **while**. Com o **while**, é possível escolher a precisão ou acurácia da resposta dessa soma. Fazer um programa em Python para somar a série anterior até que a precisão da soma seja de duas casas decimais, informando quantos termos da série foram utilizados. Por exemplo, o resultado para essa precisão necessita de algo próximo a mil termos com **while**.

```
s =  3.140592653839794  diferença =  0.000999999749998981  termos =  1000
```

12. Fazer um algoritmo usando **while** para encontrar a soma de *n* termos para a série a seguir:

$$s = 1 + \frac{3}{2} + \frac{5}{3} + \frac{7}{4} + \frac{9}{5} + \ldots$$

O usuário deve fornecer o número de termos desejado usando **input**, e o resultado deve ser impresso no **Console** com **PRINT( )**.

13. Fazer um algoritmo em Python para percorrer uma lista e imprimir com **PRINT( )** apenas ações do mesmo setor. Lembrar e usar o comando **continue** para essa tarefa. Por exemplo, na lista a seguir, excluir com o uso do **while** a sigla da Petrobras (petr4) do setor de petróleo, visto que a lista possui bancos.

```
lista=['bbdc4','itub4','petr4','petr4','bbas3','petr4','sanb4','petr4','bpac3','petr4']
```

```
Ação de Banco :  bbdc4
Ação de Banco :  itub4
Ação de Banco :  bbas3
Ação de Banco :  sanb4
Ação de Banco :  bpac3
+++ fim da impressão +++
```

14. Fazer um algoritmo em Python usando **while** para percorrer uma lista e parar a busca por um elemento assim que encontrar sua aparição pela primeira vez. Imprimir a posição (índice que ocupa na lista) em que esse elemento se encontra. Por exemplo, para a lista a seguir:

```
lista=['bbdc4','itub4','petr4','petr4','bbas3','petr4','sanb4','petr4','bpac3','petr4']
```

a solução a ser impressa é:

```
Petrobras aparece pela primeira vez no índice :  2
+++ fim da impressão +++
```

15. Uma lista tem elementos repetidos. Fazer um algoritmo em Python usando **while** para percorrer essa lista e parar a busca por um elemento assim que ele aparecer pela segunda vez. Imprimir a posição (índice que ocupa na lista) desse elemento. Por exemplo, para a lista a seguir, se procuramos a sigla petr4:

```
lista=['bbdc4','itub4','petr4','petr4','bbas3','petr4','sanb4','petr4','bpac3','petr4']
```

a impressão da resposta é:

```
Petrobras aparece a segunda vez no índice :  5
+++ fim da impressão +++
```

16. É bastante comum se ter uma lista cujos elementos também são listas. Por exemplo: Palavras = [[‘comprar’,‘vender’],[‘manter’,‘alertar’,‘indicar’],[‘tendencia’,‘crash’,‘lucro’]].

    Nessa lista, o primeiro elemento da lista Palavras é [‘comprar’,‘vender’], o segundo é [‘manter’,‘alertar’,‘indicar’] e o terceiro, [‘tendencia’,‘crash’,‘lucro’]. Utilizar a noção de **for** para imprimir esses elementos-listas de forma separada.

17. Na lista do exercício anterior: Palavras = [[‘comprar’,‘vender’],[‘manter’,‘alertar’, ‘indicar’],[‘tendencia’,‘crash’,‘lucro’]], fazer um algoritmo em Python para imprimir no **Console** os elementos das listas que são elementos da lista original Palavras. Ou seja, devem aparecer na impressão os elementos todos sem os colchetes que representam listas. Nesse caso da lista Palavras, a impressão deve ser:

```
comprar
vender
manter
alertar
indicar
tendencia
crash
lucro
```

18. Muitas vezes, temos listas cujos elementos são listas e desejamos que os elementos desses elementos formem uma nova lista. Fazer um algoritmo em Python que tome uma lista de listas, por exemplo: Lista = [[1,2,−1] , [3,−1,4,5] , [0,0,1,2,−1] , [−1,−1,2,2,−1,2,−1] , [3,2,0] , [1,1,−1,0,2]], e forme uma nova lista usando **for** e **append**.

    Lista_nova = [1, 2, −1, 3, −1, 4, 5, 0, 0, 1, 2, −1, −1, −1, 2, 2, −1, 2, −1, 3, 2, 0, 1, 1, −1, 0, 2].

Com essa nova lista fatiada da lista original, fazer com que seja apresentado no **Console** usando o comando **PRINT( )**:

(a) A nova lista.

(b) A soma dos elementos dessa nova lista.

(c) O maior elemento dessa nova lista.

(d) O menor elemento dessa nova lista.

(e) A média dos elementos dessa nova lista.

(f) A moda dos elementos dessa nova lista.

(g) O desvio-padrão populacional dessa nova lista.

19. Fazer uma nova lista a partir de uma lista original composta de palavras.

    Lista = [['ontem','hoje','amanhã'],['sp','rj','mg','ce'],['são paulo','rio','santos','cuiabá']]

    A nova lista deve ser construída usando **for** e **append**.

    Lista_nova = ['ontem','hoje','amanhã','sp','rj','mg','ce','são paulo','rio','santos','cuiabá']

    Após isso, deve-se:

    (a) Estender a lista original com a nova lista ['férias','negócios'].

    (b) Colocar a nova lista em ordem alfabética.

20. Uma lista está repleta de letras repetidas: let = ['a','b','c','a','d','f','a','b','b','d','c']. Fazer um algoritmo em Python para criar uma nova lista apenas com elementos não repetidos da lista anterior. A resposta deve ser ['a','b','c','d','f'].

21. Fazer um programa para ler duas variáveis inteiras x e y e calcular o resto da divisão de x por 10 e o resto da divisão de y por 5. O programa deverá somar os dois restos e ao final decidir: Se a soma dos restos for maior ou igual a 1, imprimir "maior ou igual a 1", se for menor do que 1 imprimir "menor do que 1".

22. Ler duas variáveis reais x e y. Se x for maior do que y ao quadrado, imprimir ("x é maior"). Se x for igual a y ao quadrado imprimir ("x é igual"). Se x for menor do que y ao quadrado imprimir ("x é menor").

23. Fazer um programa para ler duas variáveis inteiras x e y e calcular o resto da divisão de x por y. Se o resto da divisão for maior do que 1 o programa deverá imprimir "maior do que 1", caso contrário imprimir ("menor do que 1").

24. Fazer um algoritmo para ler um número "n" com "input". Se o número for divisível por 5, elevá-lo ao quadrado, caso contrário multiplicar por 10. Ao final imprimir no console o novo número.

25. Ler duas variáveis reais x e y. Se x for maior do que y, imprimir ("x é maior"). Se x for igual a y imprimir ("x é igual"). Se x for menor do que y imprimir ("x é menor").

26. Fazer um algoritmo para ler um número "n" com "input" e imprimir se ele é múltiplo de 5, múltiplo de 7 ou múltiplo de 11 usando "if". Se não for múltiplo de nenhum deles, deverá ser impressa uma mensagem dizendo isso.

27. Fazer um algoritmo para ler com "input" dois números, "x" e "y", e, usando "if", imprimir mensagem com o resultado da variável quando x>=y e o resultado caso contrário.

28. Ler a quantidade "n" de termos e imprimir usando **While** termo a termo a sequência: {2, 4, 6, 8, 10, ...}

29. Ler a quantidade "n' de termos e imprimir usando **While** termo a termo a sequência: {2, 4, 6, 8, 10, ...}

30. Ler a quantidade "n' de termos com input e gerar a sequência

$$\left[1, \frac{1}{3}, \frac{1}{5}, \frac{1}{7}, \frac{1}{9}, \ldots\right]$$

salvando em LISTA, usando **While**, termo a termo. Após isso a lista deverá virar vetor. Com o vetor, imprima com print soma total dos termos.

31. Fazer um algoritmo que imprima a sequência: 1, 8, 27, 64, ... usando **while**. Leia a quantidade "n" de termos por input.

32. Fazer um algoritmo que imprima a sequência de números: 1,5,9,13,17,... usando **while**. Leia a quantidade "n" de termos usando input.

33. Fazer um algoritmo que faça a soma:

$$s = 1 + \frac{1}{\sqrt{4}} + \frac{1}{\sqrt{7}} + \frac{1}{\sqrt{10}} + \frac{1}{\sqrt{13}} + \cdots$$

com **while**. Leia a quantidade de termos "n" com input e imprima todos os termos da série até seu último termo.

34. Fazer um algoritmo que faça a soma:
s=1+4+9+16+... com **while**. Leia a quantidade de termos "n" com input e ao final imprima somente a soma total s.

35. Fazer um algoritmo que leia um número "real" "x" com input. Depois leia com input o número "n" de termos para fazer a soma de todos os termos que respeite a regra:

$$s = 1 + \frac{x}{4} + \frac{x}{9} + \frac{x}{16} \ldots$$

Construir o algoritmo usando "FOR". Não pode usar "while" nesse algoritmo, O algoritmo deve imprimir apenas a soma total "s".

36. Fazer um algoritmo que gera e imprima termo a termo no console a sequência:

$$\sqrt{1}, \sqrt{2}, \sqrt{3}, \sqrt{4}, \sqrt{5}, \ldots \sqrt{n}$$

usando **while** termo a termo na iteração, sem vetor e sem lista. Leia a quantidade "n" por input.

37. Observe esta lista S de listas.

S = [[3,2], [5,7,2], [3,9,5,8], [10,1]]

Fazer um algoritmo com "For" para separar todos os números na lista nova

x = [3, 2, 5, 7, 2, 3, 9, 5, 8, 10, 1] .

O programa deverá imprimir x.

38. Observe esta lista S de listas.

S = [[5,1], [0,1], [3,9], [10,1,8]]

Pode supor que a lista S já está digitada no programa.
Fazer um algoritmo com "For" para separar todos os números na lista nova

x = [5, 1, 0, 3, 9, 10, 1, 8]

O programa deverá imprimir x.

Esta obra não está relacionada apenas com o ensino de uma linguagem, no caso, Python. Nesta 2ª edição, o conceito de programação de algoritmos para o público do mercado financeiro continua robusto e avançado, como na anterior. Todas as bibliotecas do Python foram revisadas e os exemplos atualizados, permitindo o mesmo ritmo de estudo e aplicações em problemas reais.

O texto continua com a divisão em duas partes, sendo a primeira composta de capítulos que representam uma evolução, da instalação do ambiente de programação do Anaconda-Spyder até problemas relacionados com array, functions e DataFrames. A principal atualização foi na importação e exportação de dados para o Excel e aquisição de dados reais do mercado financeiro com a atualização de diversas funcionalidades da biblioteca Pandas.

A segunda parte do livro aborda problemas mais avançados e reais, relacionados ao mercado de títulos, derivativos, ações e demais produtos financeiros, construindo soluções com as bibliotecas mais avançadas do Python. Foi adicionado um novo capítulo, que trata de técnicas de Inteligência Artificial baseadas em classificadores e Machine Learning para problemas em mercado financeiro.

Nesta edição, os vídeos introdutórios foram mantidos no mesmo formato, com as mesmas explicações. Além do conteúdo dos vídeos, foram acrescentados novos projetos e mais exercícios para que o leitor possa reforçar o aprendizado do conteúdo.

ISBN 978-85-212-2846-2

www.blucher.com.br

Blucher

Clique aqui e:

VEJA NA LOJA

Python e mercado financeiro

Marco Antonio Leonel Caetano

ISBN: 9788521228462
Páginas: 664
Formato: 17 x 24 cm
Ano de Publicação: 2026