MARCO ANTONIO LEONEL CAETANO

PYTHON E MERCADO FINANCEIRO

Programação para estudantes, investidores e analistas

Blucher

Marco Antonio Leonel Caetano

PYTHON E MERCADO FINANCEIRO

Programação para estudantes, investidores e analistas

Python e mercado financeiro: programação para estudantes, investidores e analistas

Editora Edgard Blücher Ltda.

Imagem de capa: iStockphoto

Publisher Edgard Blücher

Editor Eduardo Blücher

Coordenação editorial Jonatas Eliakim

Produção editorial Bárbara Waida

Preparação de texto Cátia de Almeida

Diagramação Villa d'Artes

Revisão de texto Karen Daikuzono

Capa Leandro Cunha

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 Novo Acordo Ortográfico, conforme 5. ed. do *Vocabulário Ortográfico da Língua Portuguesa*, Academia Brasileira de Letras, março de 2009.

Dados Internacionais de Catalogação na Publicação (CIP)
Angélica Ilacqua CRB-8/7057

Caetano, Marco Antonio Leonel
Python e mercado financeiro : programação para estudantes, investidores e analistas / Marco Antonio Leonel Caetano. -- São Paulo : Blucher, 2021.
532 p. : il

Bibliografia
ISBN 978-65-5506-240-3 (impresso)
ISBN 978-65-5506-241-0 (eletrônico)

1. Python (Linguagem de programação de computador) 2. Mercado financeiro 3. Bolsa de valores 4. Algoritmos I. Título

21-1140 CDD 005.133

Índice para catálogo sistemático:
1. Linguagem de programação

CONTEÚDO

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 Júpiter e Spyder. Este livro é desenvolvido todo no ambiente de programação Spyder, um dos aplicativos para os algoritmos, conforme mostrado na Figura 1.1.

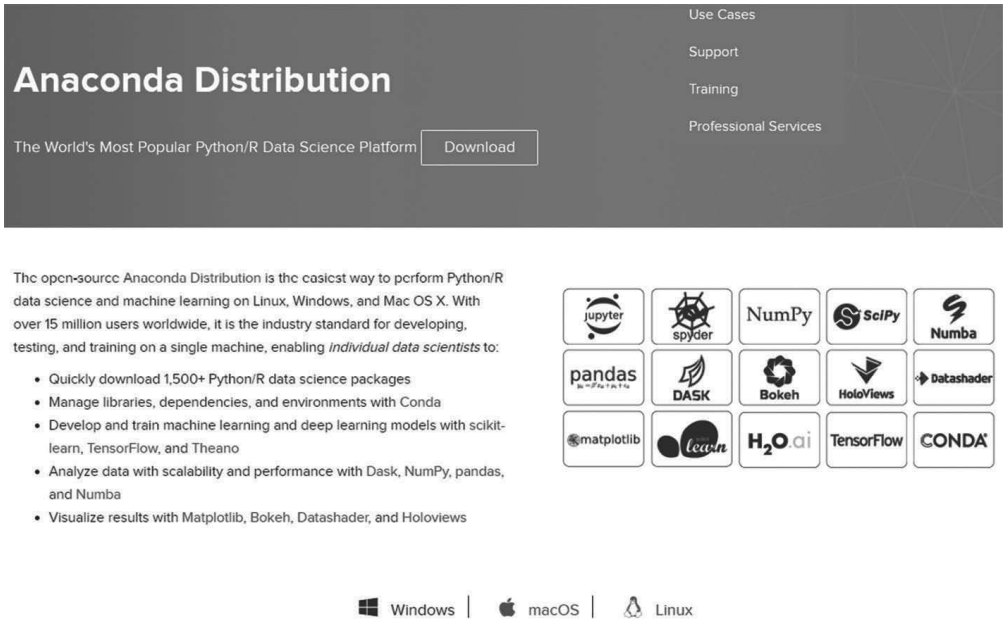


Figura 1.1 – Anaconda, *website* que disponibiliza o Python.

Passo a passo da instalação do Anaconda[1]

Quando se inicia o Spyder, observa-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.

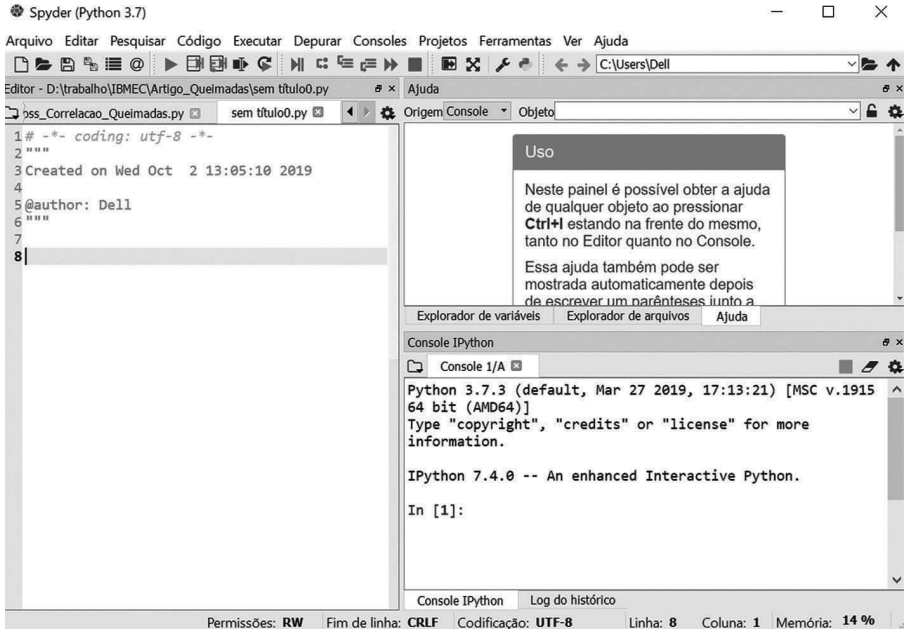


Figura 1.2 – Ambiente de programação do Python no Spyder.

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

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.

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.

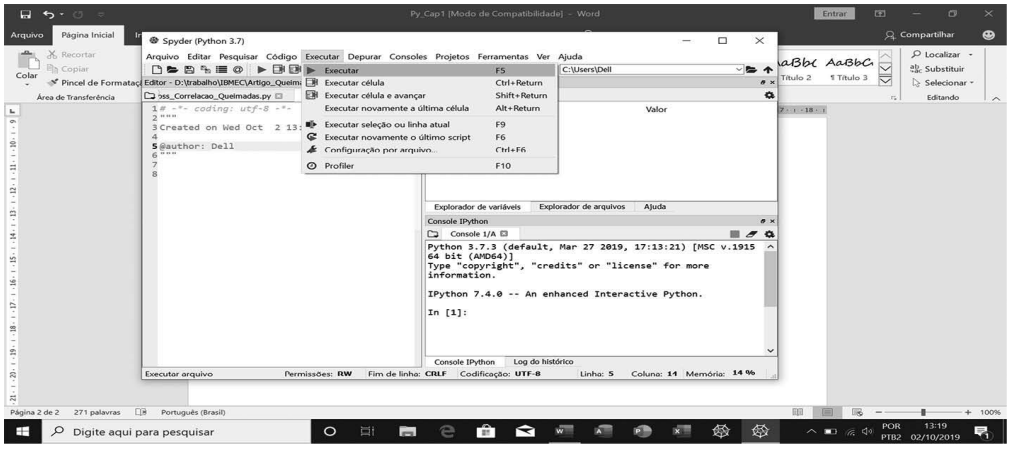


Figura 1.3 – Barra de tarefas para a execução de um programa.

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:

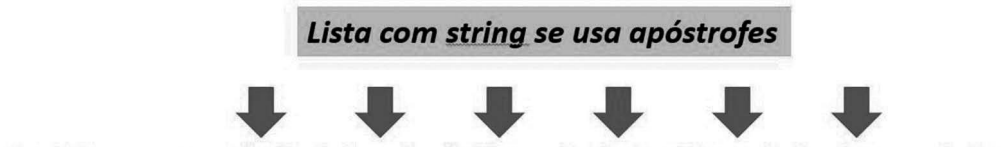
```
In [6]: print(dados[-1])
4
```

- 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 uma lista no Python 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']
```

O dois-pontos indica que a lista vai do elemento de índice 0 até o elemento de índice (3 – 1). 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 dever 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.

```
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 pode ser realizado 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 se 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 matplotlib 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.

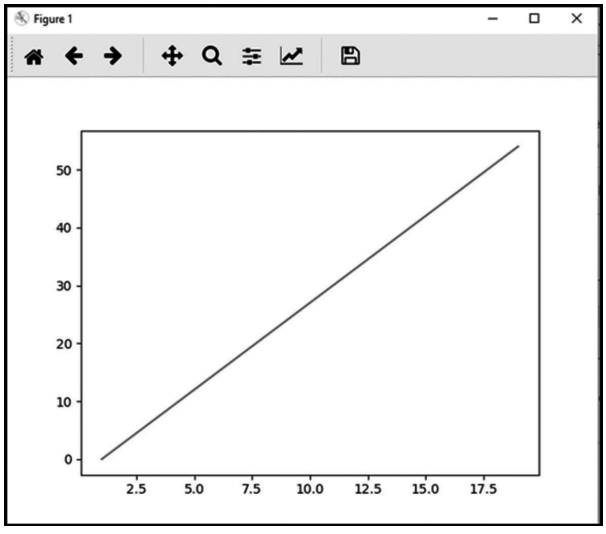


Figura 1.4 – Gráfico da equação $y = 3x - 3$ usando **pyplot**.

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.

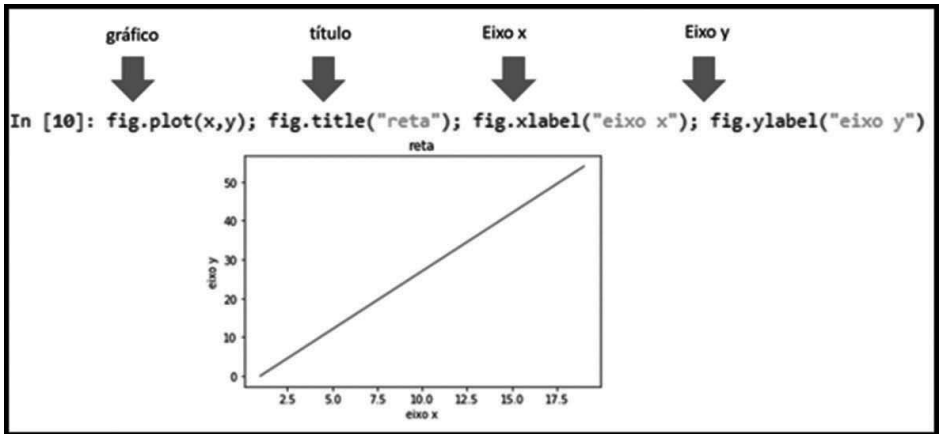


Figura 1.5 – Formatando o gráfico no **pyplot**.

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.

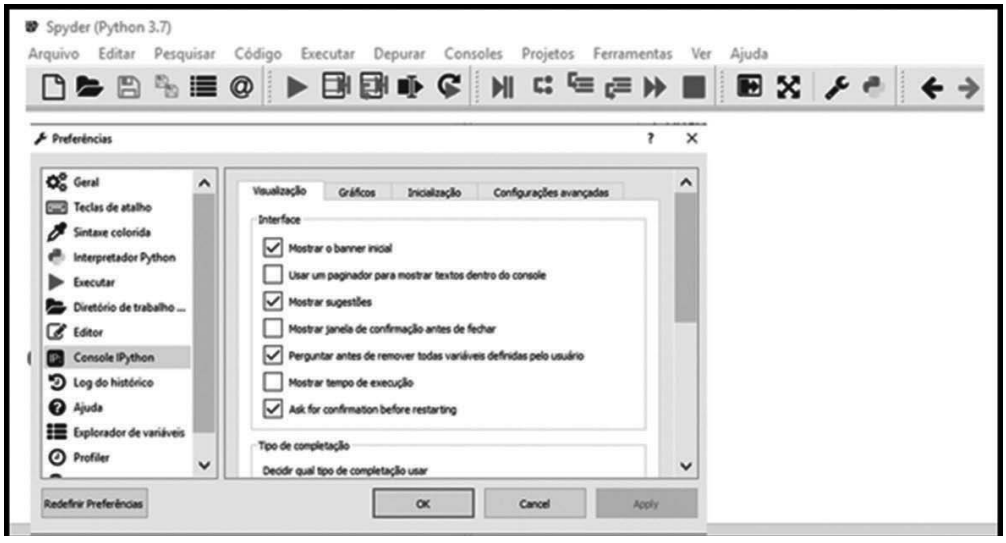


Figura 1.6 – Preferências para alterar o gráfico.

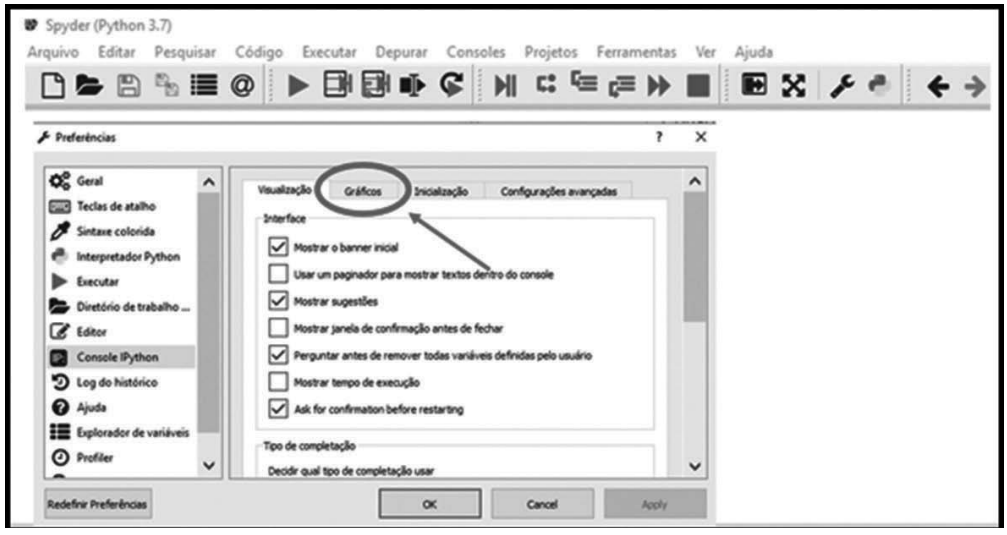


Figura 1.7 – Escolhendo a aba **Gráficos** para alteração do padrão.

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.

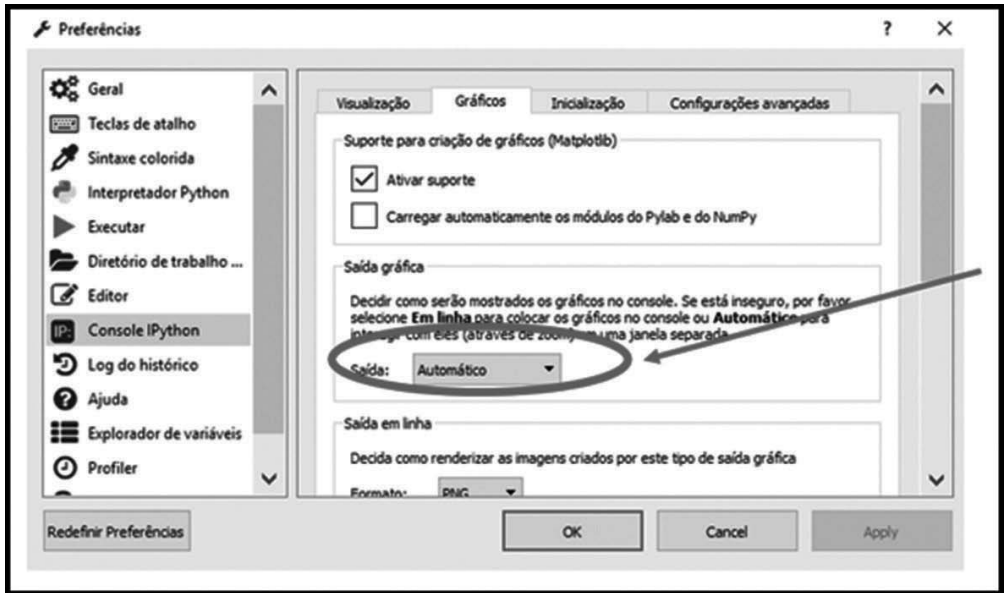


Figura 1.8 – Alterando o padrão de gráfico de **Linha** para **Automático**.

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 0x1f921122748>]

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.

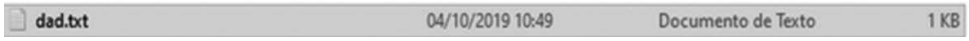
```
In [1]: x=[3,2,1]

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 [6]: f=open("dad.txt","r")

In [7]: y=f.read()

In [8]: print(y)
3 2 1


In [9]: 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 [10]: 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, %10.4e 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 xlrd, que deve ser importada para o **Console**. Depois da importação da biblioteca, deve-se especificar a planilha e usar os comandos **row** e **col** para denominação de linhas e colunas; seus números podem ser usados e colocados em listas no Python.

```
In [1]: import xlrd

In [2]: wb = xlrd.open_workbook('Dados.xlsx')

In [3]: plan = wb.sheet_by_name('Planilha1')

In [4]: plan.col_values(0)
Out[4]: [10.0, -2.0, 5.0]

In [5]: x=plan.col_values(0)

In [6]: x[0]
Out[6]: 10.0
```

A variável *wb* serve apenas para receber o nome do arquivo *.xlsx. Esse nome pode ser qualquer um. A variável *plan* recebe os dados da planilha no formato de lista em que a célula A1 é na verdade a linha zero e a coluna zero. Por isso é necessário usar:

Variável.col_values(0)

pois **col_values(0)** retorna todos os dados da coluna A para o Python. Ao salvar todos os dados na lista *x*, todos os métodos anteriores podem ser trabalhados com *x* dentro do Python. O que acontece se fazemos o mesmo para linhas?

Variável.row_values(0)

Como os dados do arquivo estão dispostos em coluna, quando se coloca o comando anterior, esse retorna apenas um dado, o que está armazenado na célula A1 do Excel.

```
In [10]: plan.row_values(0)
Out[10]: [10.0]
```

Caso se continue chamando **row_values( )** para outras linhas, sempre retornará apenas um valor, pois as demais células das outras colunas estão vazias.

```
In [12]: plan.row_values(0)
Out[12]: [10.0]

In [13]: plan.row_values(1)
Out[13]: [-2.0]

In [14]: plan.row_values(2)
Out[14]: [5.0]
```

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 **à**s 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 xlrd

In [2]: wb=xlrd.open_workbook('Dados.xlsx')

In [3]: plan=wb.sheet_by_name('Planilha1')

In [4]: x=plan.col_values(0)
```

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 [5]: max(x)
Out[5]: 10.0

In [6]: min(x)
Out[6]: -2.0

In [7]: sum(x)
Out[7]: 13.0
```

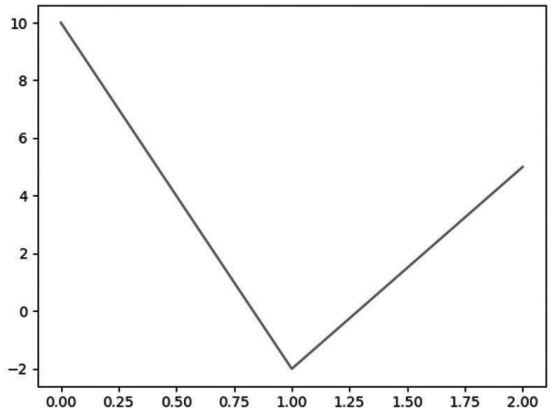
Já para a média e o desvio-padrão, temos de importar novamente a biblioteca de estatística, pois essas funções não estão nas operações básicas do **Console**. Então a média e o desvio-padrão populacional de nossos dados que estão no Excel são encontrados como segue.

```
In [11]: import statistics as st

In [12]: st.mean(x)
Out[12]: 4.333333333333333
```

Com a 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.

```
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

No exemplo anterior da importação de dados, nossos números estão no Python no formato de lista.

```
In [52]: x
Out[52]: [10.0, -2.0, 5.0]
```

É possível com esse formato realizar muitas operações, como somar elementos, subtrair dividir. Podemos ainda fatiar uma lista, como a seguir.

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

```
In [1]: import xlrd

In [2]: import numpy as ny

In [3]: import matplotlib.pyplot as fig

In [4]: wb=xlrd.open_workbook('Petrobras.xlsx')

In [5]: plan=wb.sheet_by_name('Planilha1')
```

Nesse passo, repetimos exatamente o processo de importação dos dados que estão na **Planilha1** do arquivo chamado Petrobras.xlsx. O próximo passo é transformar a lista 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 [6]: x=plan.col_values(0)

In [7]: vetor=ny.array(x)

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:

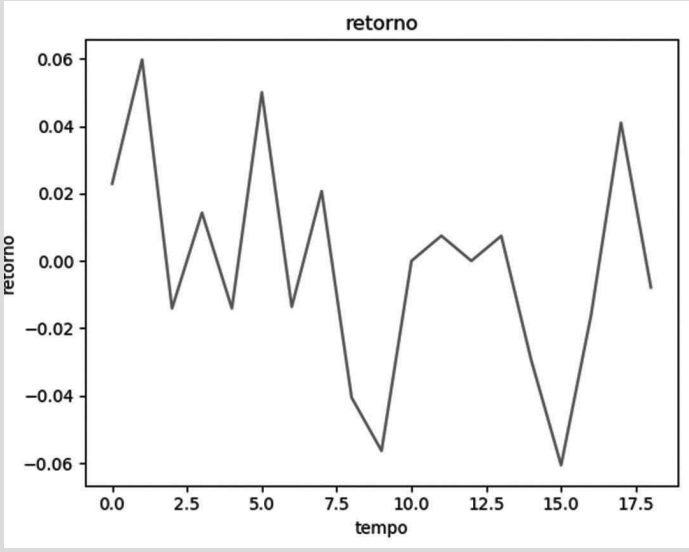
```
In [10]: print(retorno)
[ 0.02290076  0.05970149 -0.01408451  0.01428571 -0.01408451  0.05
 -0.01360544  0.02068966 -0.04054054 -0.05633803  0.          0.00746269
  0.          0.00740741 -0.02941176 -0.06060606 -0.01612903  0.04098361
 -0.00787402]
```

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 [14]: t=ny.arange(0,19)

In [15]: fig.plot(t,retorno)
Out[15]: [<matplotlib.lines.Line2D at 0x26db86aaef0>]

In [16]: fig.title('retorno')
Out[16]: Text(0.5, 1.0, 'retorno')
```

```
In [17]: fig.xlabel('tempo')
Out[17]: Text(0.5, 23.52222222222222, 'tempo')

In [18]: fig.ylabel('retorno')
Out[18]: 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.

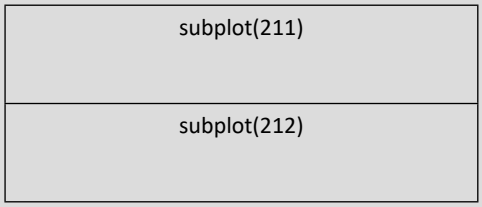
```
In [31]: import math

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)
0.01305637555366216 -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%.

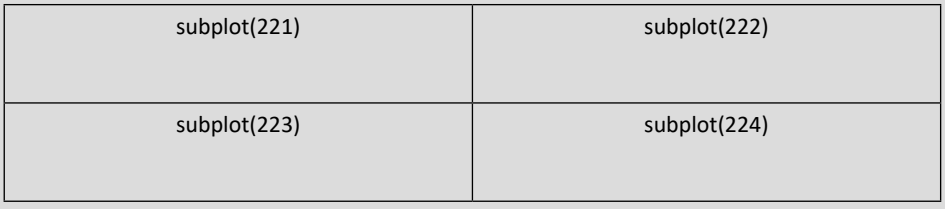
■

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.



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



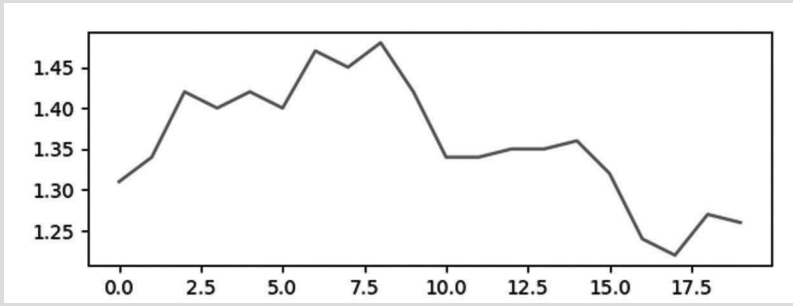
Deve-se antes lembrar que o tamanho do vetor de preços é diferente do vetor de retornos. Enquanto temos 20 preços, teremos apenas 19 retornos. Logo, o eixo horizontal para os preços deve ser formado à parte. Por exemplo, podemos criar o vetor como uma nova variável *tempo = ny.arange(0,20)*. Então, a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

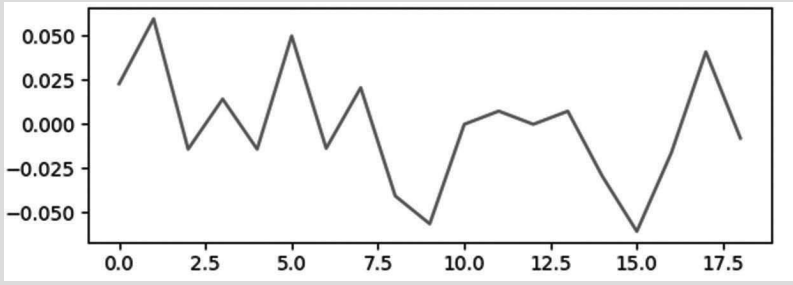
```
In [44]: tempo=ny.arange(0,20)
```

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.**

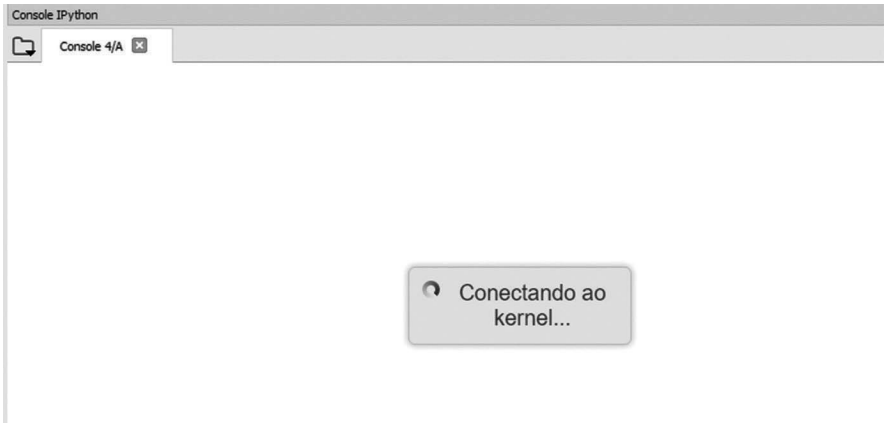


Figura 1.10 – Reiniciando o **Console**.

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.

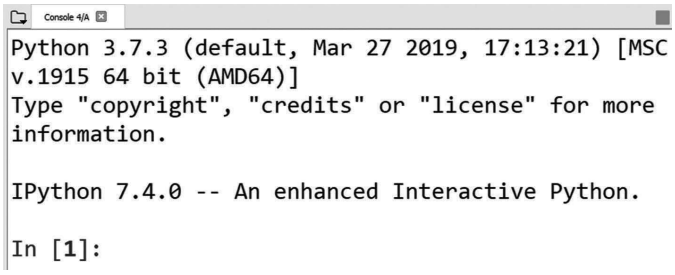


Figura 1.11 – Fonte aumentada no **Console**.

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.

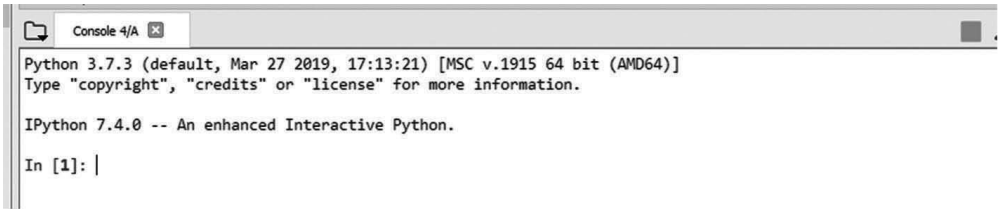


Figura 1.12 – Fonte diminuída no **Console**.

1.13 EXERCÍCIOS

1. Usar a biblioteca correta e as funções matemáticas do Python, quando necessárias, para resolver as seguintes expressões:

   (a) $y = 4^3 - 2^2$

   (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 **Plan1**.

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

Deseja-se, então, que esses dados sejam importados para o Python com a biblioteca xlrd 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 usando a biblioteca numpy.

(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 na parte inferior.

(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.

Esta obra não está relacionada apenas com o ensino de uma linguagem, no caso, Python. Ela tem foco no aprendizado de algoritmos voltados para o mercado financeiro. As aplicações são dispostas no livro com a utilização de dados reais das bolsas de valores e de produtos financeiros, que são tratados usando ferramentas nas áreas de finanças, matemática, estatística, ciência da computação e ciência dos dados.

O texto se divide 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**. As principais bibliotecas do Python são usadas para automatizar cálculos de tendência de mercado, risco, *value at risk*, otimização de carteiras e simulação de Monte Carlo. Os capítulos são contemplados com exemplos e exercícios, todos com soluções.

A segunda parte do livro apresenta 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.

Também fazem parte dos capítulos introdutórios QR Codes que levam a vídeos do próprio autor explicando os exemplos de cada seção, disponíveis em seu canal no YouTube.

ISBN 978-65-5506-240-3

www.blucher.com.br

Blucher

Clique aqui e:

VEJA NA LOJA

Python e Mercado Financeiro

Programação para estudantes, investidores e analistas

Marco Antonio Leonel Caetano

ISBN: 9786555062403
Páginas: 532
Formato: 17 x 24 cm
Ano de Publicação: 2021
