MARCO ANTONIO LEONEL CAETANO

PANDAS E O DEMÔNIO DOS DADOS

Algoritmos em Python para Sistemas de Informação

Blucher

Marco Antonio Leonel Caetano

PANDAS E O DEMÔNIO DOS DADOS

Algoritmos em Python para Sistemas de Informação

Pandas e o demônio dos dados: algoritmos em Python para Sistemas de Informação

Editora Edgard Blücher Ltda.

Publisher Edgard Blücher
Editor Eduardo Blücher
Coordenador editorial Rafael Fulanetti
Coordenação editorial Ana Cristina Garcia
Produção editorial Kedma Marques
Preparação do texto Ana Lúcia dos Santos
Diagramação Roberta Pereira de Paula
Revisão de texto Ariana Corrêa
Capa Juliana Midori Horie
Imagem da capa Equipe Editora Blucher

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)
Angélica Ilacqua CRB-8/7057

Caetano, Marco Antonio Leonel
Pandas e o demônio dos dados : algoritmos em Python para Sistemas de Informação / Marco Antonio Leonel Caetano. – São Paulo : Blucher, 2025.
470 p. : il.

ISBN 978-85-212-2515-7

1. Python (Linguagem de programação de computador)
I. Título

25-1039 CDD 004.68

Índice para catálogo sistemático:
1. Python (Linguagem de programação de computador)

Conteúdo

Prefácio

No início, todo livro vai se formando por páginas em branco, apenas com rascunhos de ideias do autor. Pensamos no tema geral, vamos desenvolvendo as conexões, colocando os temas e seus desdobramentos, e, então, o livro toma conta e praticamente se desenvolve sozinho. O autor é apenas um transmissor do que o livro solicita, e quase exige, que seja colocado no papel.

Foi assim com os meus outros livros, sempre um rascunho de ideias após algumas aulas e sentar no computador para escrever, esperando as conexões se formarem. Muitas vezes, o barulho da chuva, o silêncio da noite mais profunda, o canto dos pássaros nos galhos do meu ipê colaborava para isso se desenrolar mais rápido.

Com esta obra foi diferente. Foi se delineando automaticamente nos últimos anos, à medida que a linguagem de computação Python foi sendo aprofundada em meus cursos de programação. Diversas vezes – e não foram poucas –, programas quase idênticos, seguindo algoritmos bem conhecidos e simples, desperdiçavam longos minutos com problemas estranhos.

Os erros mais esquisitos e mais absurdos ocorriam e ainda ocorrem em programação de algoritmos simples em Python, seja por distração, cansaço, pressa em resolver problemas, ou mesmo por compilador mal-estruturado. É comum os programadores se depararem com erros absurdos. E esses erros, depois de resolvidos, fazem nos perguntarmos como cometemos erros tão bobos.

Os deslizes que cometemos em programação, não são um privilégio ou uma consequência apenas da linguagem Python. Linhas mal-escritas de programa, esquecimento de fechar parêntesis e colchetes ou vírgulas erradas, sempre ocorrem em qualquer linguagem.

O mais impressionante de tudo, em todos esses anos em que trabalho com ciência da computação, é que a maioria dos livros coloca ao leitor os algoritmos que funcionam, cujos programas rodam perfeitamente e "nunca" erram. É difícil encontrar livros em que, antes de colocar a solução final, o autor diga: "eu sofri muito para isso dar certo". Ao leitor, fica sempre parecendo que o erro acontece com ele, pois não estudou direito, não escreveu direito, não resolveu os exercícios como os textos lhe informaram.

Quem programa em linguagens de computação sabe que um algoritmo que roda em segundos pode ter levado semanas até ter sua concepção final. Lembro-me de quando era estudante de graduação, quando fazia iniciação científica, aos 20 anos, que um programa me tomou mais de um mês, por causa de um ponto e vírgula! Sim, a linguagem dos anos 1980 era o Turbo Pascal e tinha como separadores de linha pontos e vírgulas.

Como o programa tinha mais de 500 linhas, mesmo imprimindo em impressora matricial barulhenta, mesmo olhando linha por linha, só fui descobrir o erro um mês depois, quando estava já cansado e apenas olhando para o papel. Eis que desponta o erro da falta do ponto e vírgula!

Nos últimos anos, o ensino e a programação em Python para meus alunos me fizeram reviver esses erros. Tanto quando os alunos cometem erros e vêm tirar dúvidas quanto em meus programas, quando preparo aula, apesar de saber como se faz, sempre um erro aparece.

E, de novo, como antigamente, o erro é sempre de colchetes ou de estrutura malformada para um banco de dados ou, ainda, achar que um mesmo comando que funciona para um procedimento vai funcionar para outro semelhante. Mas não é assim.

A biblioteca Pandas, do Python, é impressionante e revolucionária, tenho quase 100% de certeza de que foi uma das responsáveis por desenvolver e promover a área de ciência dos dados. A informação é tudo em nossas vidas e, na ciência da computação, é a alma dos algoritmos. Com o advento da biblioteca Pandas, muitos problemas, que exigiam uma quantidade absurda de linhas para se trabalhar com matrizes, foram substituídos por funções rápidas e criativas.

Quando os alunos se deparam com a biblioteca Pandas, ouve-se nas salas de aula um "Ooohhhh!" por parte deles quando rodam um exemplo. É que, realmente, para quem programou em outras linguagens e lembra-se do conceito de iteração

(repetição) ou *loops* por parte de conceitos de resolução de algoritmos com matrizes, deparando-se com a Pandas, são de espantar a facilidade e a agilidade da estrutura.

O problema é que, quando começamos a trabalhar com uma quantidade grande de dados – dados não estruturados em websites ou formatos do tipo texto para números ou datas –, grandes problemas aparecem. E as soluções, quando aparecem, são sempre ridículas, como uma simples troca de ordem de parâmetros, chamada de funções específicas ou apenas uma troca de linhas por colunas.

Antigamente, precisávamos ficar conversando com colegas, chamando para tomar um café, fazendo seminários, ou mesmo, escrevendo artigos em conjunto para descobrirmos as soluções. Nos dias atuais, principalmente em Python, as soluções estão soltas e caminhando pelas palavras de milhões de programadores nos fóruns de computação.

Muitas pessoas não conhecem ou não sabem como procurar essas informações, pois não têm habilidade para usar palavras-chave específicas e deparar-se com maravilhosas e criativas soluções.

Mesmo quando se mudam as versões do Python, não estamos livres de problemas que nos tomam horas e dias para fazer um programa funcionar de maneira adequada. Por exemplo, no Capítulo 8, quando estava tudo terminado, as distribuições de probabilidade estavam funcionando de maneira adequada numa versão mais antiga do Python. No entanto, ao comparar algumas soluções com versões mais recentes do Python (no momento em que escrevo, a mais atual é a Python 3.12), o Capítulo 8, pregou-me uma grande peça.

Os valores das estimativas dos parâmetros para o exemplo das chuvas em São Sebastião (SP), começaram a dar errado, e, muitas vezes, nem rodaram. Depois de uma tarde inteira, descobri que o detalhe estava na versão mais recente, o formato de data deveria vir adicionado à função *.date( ) com parêntesis no final. Sim, apenas um parêntesis tomou, deste autor, quatro horas para adequar um programa antigo na versão mais recente do Python.

Pensando nisso, longe de ser um livro que resume todos os problemas, esse texto toma uma parte de alguns problemas que diariamente podem ocorrer com todos os programadores e tenta mostrar como solucioná-los. O leitor poderá achar muitas soluções mais sofisticadas na internet, mas para quem não tem tempo de pesquisar com calma, este livro tenta alertar sobre alguns erros básicos que nós cometemos na pressa do dia a dia ou, como alerto, provocados pelo "diabo escondido atrás dos erros".

Como explico no Capítulo 4, o provérbio atribuído a Nietzsche diz que "O diabo mora nos detalhes". O saudoso e fascinante Carl Sagan também escreveu um livro sobre o "diabo" e como ele se relaciona com a vida humana. Carl Sagan, com sua

elegante maneira de explicar, nos diz em seu livro *O mundo assombrado pelos demônios* (1996), o significado da palavra "demônio": "podem assumir qualquer forma, e sabem muitas coisas – 'demônio' significa 'conhecimento' em grego […]".

Carl explica que, em grego, "demônio" significa "conhecimento" e "ciência" significa "conhecimento" em latim. Logo, como conhecimento é tudo em nossa vida, durante muitos séculos, a população foi exposta ao conceito de associar demônio ao mal, ao perverso, aquilo que tudo de ruim pode acontecer, advindo de um espaço longe da vida terrena. E, no entanto, essa palavra se associa ao conhecimento, ao prazer da descoberta na língua grega.

Logo, se demônio significa conhecimento em grego e se Nietzsche está correto, mesmo morando nos detalhes, o demônio é o conhecimento de como as coisas funcionam e, por isso, precisamos aprender com os detalhes dos erros. E em termos de Python, o que mais temos são detalhes quando estamos programando os algoritmos.

Para chegar aos erros, precisamos antes conhecer por que eles acontecem, por que o tal diabo está nos afrontando quando estamos programando algoritmos. Pensando nisso, elaborei este livro, cujos três primeiros capítulos revisam as formas básicas de programação em Python e apresentam o funcionamento da biblioteca Pandas. Espero, com isso, deixar o leitor confortável e conectado com a linguagem, para entender os erros que serão apresentados nos capítulos posteriores.

Alguns capítulos trazem exercícios retirados de listas ou avaliações dos meus cursos, com o intuito de que, após treinar com sucesso com algoritmos simples, o leitor e programador possa perceber que detalhes importantes sempre passam despercebidos em programas mais avançados.

Pequenos desenhos de um diabinho correndo de um ursinho panda sempre aparecerão em partes de algum programa, para o leitor perceber que existe um erro que poderá ser apontado ou não pelo compilador em Python. Toda vez que o leitor vir o desenho, significará que, naquela parte do algoritmo, há algum problema não observado ou um problema futuro por conta de comandos impróprios para as linhas programadas.

O Capítulo 1 apresenta os comandos básicos e as bibliotecas mais comuns em Python para quem está aprendendo a programar. São trazidas funções para se construírem gráficos, listas, array, além de comandos de iterações e estruturas para lógica de decisão. Exemplos de estatística e seu uso adequado também preenchem esse capítulo.

O Capítulo 2 é dedicado inteiramente à biblioteca Pandas e ao seu uso adequado, com exemplos simples para que o leitor perceba e compare as simplificações que ela possibilita em programas que precisavam de muitos comandos. Gráficos,

agrupamentos, localizações de informações, enfim, toda a parte básica da criação de DataFrame é apresentada nesse capítulo.

No Capítulo 3, aprofundamos o uso da parte gráfica da Pandas. São apresentadas ferramentas sensacionais que facilitam a interpretação de dados por meio de gráficos, também de maneira bastante simples, para que o leitor possa ir escrevendo seu próprio programa e acompanhando os resultados quando executado. Os tipos e formulações dos gráficos em Pandas encantam todos pela sua facilidade e pela qualidade de informações que passa para qualquer banco de dados.

É no Capítulo 4 que os problemas começam a se aprofundar. Problemas com dados mal estruturados, ou mesmo formas erradas de tratar as funções da biblioteca Pandas, são apresentados nesse capítulo. O leitor poderá reparar que qualquer distração na programação que envolva banco de dados poderá acarretar erros absurdos, e, na primeira vez, erros que não deveriam ocorrer. E, claro, sempre mostramos como a Pandas soluciona esses erros.

Sempre quando baixamos dados de websites, ou quando algum sistema muito antigo de armazenamento de dados salva números ou dados com formatos antiquados, problemas enormes aparecem. Um problema muito comum é que, às vezes, linhas que estão num arquivo, não estão em outros, e precisamos juntar e agrupar essas informações sem perdermos a noção espacial e temporal dos acontecimentos. Por exemplo, muitas vezes, os dados estão em datas desconexas entre os arquivos, e precisamos agrupar essas informações. No Capítulo 5, apresentamos e comparamos quatro formas distintas de se juntarem os dados e os problemas que ocorrem quando isso é feito.

No Capítulo 6, é apresentada a mágica função Lambda do Python. É feita uma conexão histórica com sua criação e invenção por grandes nomes da computação e da filosofia lógica do pensamento. Com essa função, um conjunto de linhas inteiras pode ser substituído por apenas uma linha, tornando a programação mais limpa e de fácil compreensão. Mas os erros são ainda mais fáceis de aparecer, e mostramos isso com exemplos.

Quando estamos com problemas, nada melhor do que contar com um bom amigo para nos socorrer. Uma grande amiga da Pandas é a biblioteca Seaborn, que ajuda a Pandas a dispor de gráficos e de interpretações maravilhosas sobre os DataFrames. É o que mostramos no Capítulo 7, expondo a Seaborn a exemplos simples e demonstrando onde os cuidados são mais necessários para se evitar a aparição do diabo nos erros.

O que mais acontece no mundo da computação são falhas. O que mais ocorre em nossas vidas são falhas, todos os dias, em todos os momentos. Mas há algumas falhas

que nos levam a catástrofes. Começamos o Capítulo 8 apresentando a distribuição de probabilidades das falhas com a densidade de probabilidade de Weibull e seu uso com os DataFrames na Pandas. Uma ligação de conceitos deve ser feita nesse capítulo entre a Pandas e também com outra biblioteca importante, conhecida como SciPy, no Python. Apenas observar resultados gráficos não melhora nosso sentido de entendimento, e precisamos de informações quantitativas por meio de resultados estatísticos advindos da SciPy conectados com a Pandas.

Dados pluviométricos reais e de grande porte são usados no Capítulo 8 para demonstrar os erros que acontecem quando programamos em Pandas sem o devido cuidado. Esses dados são baixados e disponibilizados todos os dias pelo Centro Nacional de Monitoramento e Alertas de Desastres Naturais (Cemaden), localizado em São José dos Campos – SP.

Toda discussão política e econômica deve se basear em dados. Isso, ultimamente, não tem ocorrido ao redor do mundo, tornando uma porção de informação em falácia ideológica para prejudicar um governo ou ajudar outro. No caso do Brasil, uma boa fonte de informação econômica se encontra no Banco Central do Brasil, que disponibiliza um histórico de décadas de dados econômicos, que podem ser adquiridos pelo Python, e o mais interessante: todos os dados estão no formato da Pandas.

Exemplificamos o uso da Pandas com os dados do Banco Central no Capítulo 9, apresentando inúmeros erros que podem ocorrer no tratamento errôneo ou na filtragem errada das bases de dados quando estas são baixadas do website do banco. Nesse capítulo, todos os conceitos abordados anteriormente são utilizados, como agrupamento, filtragem, gráficos, concatenação, formatação de datas, enfim, tudo o que foi visto antes, apresentando os erros que aparecem e suas soluções.

Sendo uma das maiores fontes de dados, o mercado financeiro não poderia ficar de fora deste texto. No Capítulo 10, apresentamos uma quantidade enorme de exemplos e possíveis erros que ocorrem quando utilizamos dados de empresas, dados armazenados na biblioteca do Python ligada ao Yahoo! Finance, diretamente dentro dos códigos. Partes dos programas e programas completos são apresentados, mostrando quais tipos de erros surgem nos DataFrames quando nos esquecemos de detalhes que o "diabo" insiste em nos esconder. Além disso, esse capítulo também mostra algumas técnicas que utilizam sensor de compra e venda de ações, como média móvel e *candles*, apresentando erros em sua montagem e que podem ocorrer nas interpretações.

No Capítulo 11, tratamos de eventos que estão sempre em nosso inconsciente coletivo e que são motivo de muitas teorias da conspiração sobre a destruição do mundo, sobre a NASA esconder dados importantes, entre tantas outras *fakenews*. A NASA libera há décadas arquivos para baixar dados sobre asteroides descobertos,

suas órbitas, elementos de estrutura, como tamanhos, energia, luz refletida, velocidade etc. Nos anos mais recentes, a agência espacial norte-americana criou uma biblioteca em formato Pandas, com DataFrame, pronta para ser usada para diversas medidas, diversas abordagens estatísticas sobre impactos, potência de destruição e outros cálculos. Nesse capítulo, baixamos alguns dados da NASA e mostramos que não basta apenas fazer isso, mas que precisamos estruturar os DataFrames para que as informações importantes sejam visualizadas na Pandas e na Seaborn.

O Capítulo 12 apresenta outro ambiente interessante para programar DataFrames da Pandas, com certa liberdade na hora de rodar os códigos. O ambiente Jupyter é apresentado de forma simples e básica para que o leitor que não tem intimidade com o Python possa se deleitar com tamanha facilidade de programação e visualização de resultados da biblioteca Pandas.

Este livro serve para o leitor aprender com programas simples como funciona a biblioteca Pandas e sua estrutura para obter informações de dados. Apesar de alguns exemplos simples, oferecemos outros mais complexos e complicados, como os de precipitação, econômicos e dos asteroides. Essas abordagens foram colocadas neste livro, pois a grande dificuldade dos programadores no dia a dia é de como resolver rapidamente problemas sem precisar procurar na internet por soluções que não contemplam a resolução eficaz.

Claro que, com o advento da inteligência artificial, muitos leitores desatentos poderão pensar que basta usar o ChatGPT para conseguir um programa fácil e rápido para seus propósitos. Muitos erros ocorrem com esse pensamento. Primeiro, o leitor não estará aprendendo nada. Segundo, os programas de IA são sempre mais rebuscados e complexos para resolver, muitas vezes, problemas bem simples. Terceiro, ao mudar para um outro problema semelhante, o leitor terá de buscar novamente outro programa no ChatGPT, que, outra vez, será uma "caixa-preta", e ele não vai entender boa parte da resolução.

Assim, apesar de mais trabalhoso, aprender sem usar programas prontos é sempre melhor e prazeroso, e liberta da escravidão de pensamento realizado por uma máquina. Se o leitor quer aprender e crescer intelectualmente com os erros provocados pelo Demônio de que Carl Sagan comentou, é bem-vindo ao maravilhoso mundo da biblioteca Pandas.

O autor.

CAPÍTULO 1
Revisitando o Python

1.1 INTRODUÇÃO

A linguagem Python foi desenvolvida em dezembro de 1989 por Guido van Rossum. Conhecedor de linguagens como ALGOL e Pascal, Guido procurava por um interpretador de comandos mais fácil e poderoso do que os existentes à época. Devemos nos lembrar de que as linguagens mais populares e utilizadas na época eram BASIC, FORTRAN, COBOL e ALGOL. Apesar disso, somente em 1999, o Python estava com uma estrutura mais adaptada. Graças ao seu estilo de linguagem aberta, Guido ganhou um prêmio em 2002 por avanços em software livre.

Como toda linguagem, a premissa adotada é de que as operações básicas devem ser apoiadas por variáveis que são compostas por números reais, por estruturas unidimensionais, conhecidas como array (ou vetor), e bidimensionais (matrizes). A inovação, no caso do Python, é que bibliotecas externas aos programas (também chamados de script) podem ser importadas ou não, dependendo da natureza do problema envolvido.

Com isso, a democratização veio no sentido de que pessoas com base de programação mais simples podem atuar com essa linguagem usando conhecimento de outras linguagens. Programadores por meio de pesquisas mais avançadas também

podem usar a mesma linguagem, baixando e importando, para isso, bibliotecas específicas e direcionadas para sua área de estudo.

Uma das grandes novidades foi a construção da biblioteca NumPy (*Numerical Python*) para trabalhar com as estruturas de array de maneira mais rápida e amigável do que, por exemplo, o FORTRAN da época. Outra inovação veio com a biblioteca Pandas, que revolucionou o modo de resolução de problemas que envolviam grandes estruturas de dados. Isso ajudou a difundir a biblioteca e a linguagem para áreas, que foram se consolidando a partir dos anos 2000, condensando-se e integrando-se no que conhecemos hoje como *ciência dos dados*.

1.2 OPERAÇÕES BÁSICAS

Se tivermos duas variáveis x e y, as operações básicas são facilmente desenvolvidas, como apresentadas a seguir. Soma, subtração, multiplicação e divisão seguem o padrão de outras linguagens com os mesmos símbolos. As operações aqui apresentadas estão ocorrendo no ambiente do *Spyder* – Anaconda, dentro do console, e, assim, temos os símbolos *In* [1], *In* [2] etc. para as entradas; e *Out* [1], *Out* [2] etc. para os resultados das operações. Os mesmos comandos trabalham da mesma maneira em outros ambientes, como o editor de texto do *Spyder* (no caso da Anaconda) ou do Jupyter.

```
In [1]: x=2

In [2]: y=3

In [3]: x+y
Out[3]: 5

In [4]: x-y
Out[4]: -1

In [5]: x*y
Out[5]: 6

In [6]: x/y
Out[6]: 0.6666666666666666
```

Uma operação bastante utilizada é descobrir o resto da divisão entre dois números, por exemplo, x e y. O símbolo, nesse caso, é o "%", que não significa operação de cálculo de porcentagem, mas o acionamento do programa para se calcular o

resto da divisão entre x e y. Por exemplo, se x = 10 e y = 3, sabemos que o resto da divisão x/y será 1. Em Python, escrevemos da seguinte forma:

```
In [7]: x=10

In [8]: y=3

In [9]: x % y
Out[9]: 1
```

1.3 BIBLIOTECA MATEMÁTICA

Talvez a biblioteca mais simples do Python seja a Math. Ela é necessária quando desejamos utilizar expressões muito comuns, como exponencial e raiz quadrada ou expressões de logaritmos e trigonometria para números ou variáveis unidimensionais.

Para se utilizar das funções existentes na biblioteca Math, é necessário antes importá-las, para então começar a utilizar os recursos existentes. Para a raiz quadrada, a função é *sqrt*; para a função exponencial, usamos *exp*; e para o logaritmo natural, usamos *log*. Pode-se perceber que devemos usar o nome da biblioteca antes do nome da função para termos o resultado das operações. Por isso, temos math.sqrt(4) para calcular a raiz quadrada de 4.

```
In [10]: import math

In [11]: math.sqrt(4)
Out[11]: 2.0

In [12]: math.exp(0.1)
Out[12]: 1.1051709180756477

In [13]: math.log(2)
Out[13]: 0.6931471805599453
```

Claro que, além de usarmos a palavra com o nome da biblioteca, podemos colocar um apelido nela para simplificar o uso das funções. Por exemplo, se chamarmos a importação, substituindo o nome *math* por *mt*, as funções devem ser chamadas agora como mt.sqrt(4), e não como math.sqrt(4).

```
In [14]: import math as mt

In [15]: mt.sqrt(4)
Out[15]: 2.0

In [16]: mt.exp(0.1)
Out[16]: 1.1051709180756477

In [17]: mt.log(2)
Out[17]: 0.6931471805599453
```

Para as funções trigonométricas, fazemos uso da biblioteca da mesma maneira, como demonstrado para o cálculo de seno, cosseno e tangente.

```
In [18]: mt.sin(0.5)
Out[18]: 0.479425538604203

In [19]: mt.cos(0.5)
Out[19]: 0.8775825618903728

In [20]: mt.tan(2)
Out[20]: -2.185039863261519
```

1.4 BIBLIOTECA ESTATÍSTICA

Para medidas estatísticas simples, podemos fazer uso da biblioteca estatística, cujo nome é *statistics*. Uma das entradas de dados para a utilização das estatísticas básicas que conhecemos, como média, desvio-padrão populacional e desvio-padrão amostral, é representar a amostra como um tupla (um conjunto de dados separados por vírgula).

```
In [21]: x=2,3,4,5

In [22]: import statistics as st

In [23]: st.mean(x)
Out[23]: 3.5

In [24]: st.pstdev(x)
Out[24]: 1.118033988749895

In [25]: st.stdev(x)
Out[25]: 1.2909944487358056
```

No caso em questão, *pstdev* se refere ao desvio-padrão populacional, cuja fórmula tradicional em estatística é:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}\left(x_i - \overline{x}\right)^2}{n}}$$

em que $\overline{x}$ é a média da amostra. Já para o desvio-padrão amostral *stdev*, temos:

$$s = \sqrt{\frac{\sum_{i=1}^{n}\left(x_i - \overline{x}\right)^2}{n-1}}$$

Valor máximo e valor mínimo de um conjunto de dados não precisam de biblioteca, basta colocar apenas as iniciais *max* e *min* na frente da variável. Para o caso deste exemplo:

```
In [26]: max(x)
Out[26]: 5

In [27]: min(x)
Out[27]: 2
```

1.5 O QUE SÃO LISTAS?

Lista é uma modalidade de armazenamento de dados bastante especial do Python, na qual os dados são colocados numa variável, separados por vírgula e dentro de colchetes para a identificação de que estamos armazenando os dados para serem usados em diversas análises. É uma modalidade que, apesar de limitada, principalmente no que se refere a operações entre os dados, é bastante útil no auxílio de desenvolvimento de diversas partes de um programa.

Uma lista também é separada por vírgulas, mas utilizam-se colchetes para sua representação. Ela é baseada em índices para identificação da localização dos dados. Diferentemente de outras linguagens, no Python, uma lista sempre começa com o índice zero, não com o índice um. Vamos supor que tenhamos a seguinte lista "x" no console do Python:

```
In [1]: x = [10,4,3,5,8,9]
```

- **primeiro elemento da lista:** o primeiro elemento da variável "x" anterior é x[0]. Para ver esse elemento, basta usar x[0] no console e, em seguida, aparecerá o valor desse elemento:

```
In [1]: x = [10,4,3,5,8,9]

In [2]: x[0]
Out[2]: 10
```

- **último elemento da lista:** para ver o último elemento, basta usar -1 dentro do nome da variável lista, ou seja:

```
In [3]: x[-1]
Out[3]: 9
```

- **total de elementos da lista:** para o total de dados de uma lista, utiliza-se o comando "len", ou

```
In [4]: len(x)
Out[4]: 6
```

em que o Python nos mostra que temos na lista "x" seis elementos no total.

- **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 *x*, se deserjamos extrair do primeiro ao terceiro elemento, devemos começar com o índice *0* e terminar com índice *3*. Esse fato pode gerar uma confusão, pois, matematicamente, 0, 1, 2, 3 têm 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 representado como (n-1). Assim, por exemplo, em uma lista com

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

se desejarmos 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 *x*,

```
In [5]: x[0:3]
Out[5]: [10, 4, 3]
```

Nesse caso, x[0] = 10, x[1] = 4 e x[2] = 3. Podemos, ainda, usar o comando *print* para imprimirmos o resultado no console, o que é bastante útil quando estamos programando no editor de programação, e não no console do Python.

```
In [6]: print(x)
[10, 4, 3, 5, 8, 9]

In [7]: print(x[0:3])
[10, 4, 3]
```

- **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 desejemos colocar o número 100 na lista *x* anterior:

```
In [8]: x.append(100)

In [9]: x
Out[9]: [10, 4, 3, 5, 8, 9, 100]
```

- **Ordenação de uma lista**

Colocar em ordem alfabética ou crescente uma lista é bem fácil, usando *sort* ao final do nome da lista.

```
In [10]: x.sort()

In [11]: x
Out[11]: [3, 4, 5, 8, 9, 10, 100]
```

Para se colocar em ordem decrescente os elementos de uma lista, basta inserir a palavra *reverse* dentro dos parêntesis, com a afirmação *True*, como apresentado a seguir.

```
In [4]: x.sort(reverse=True)

In [5]: x
Out[5]: [100, 10, 9, 8, 5, 4, 3]
```

Mais comandos de operações com lista podem ser verificados na Tabela 1.1 a seguir. É apenas um pequeno resumo das muitas funções existentes para manipulação de dados em listas.

Tabela 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 crescente ou alfabeticamente a lista "nome".*

As listas permitem os fatiamentos dos elementos para tratamentos especiais ou de separação para alguns cálculos mais simples. Esses fatiamentos são conhecidos como *slice* e correspondem a comandos bem interessantes para amostragens dos elementos.

Vamos supor que tenhamos listas com nomes de frutas:

frutas = ['limão','laranja','abacate','abacaxi','banana','mamão']

Para selecionar do segundo elemento ('laranja') ao quarto elemento ('abacaxi'), o uso dos ":" indicam a localização dos dados que deverão ser cortados para a seleção. Nesse caso, o segundo elemento da lista frutas tem índice 1, e o quarto elemento ('abacaxi') terá índice 3. Mas, na escolha do fatiamento em listas, o último elemento colocado no colchete obriga a seleção a escolher sempre até um antes do fim. Então, nesse caso, como o quarto elemento é abacaxi, seu índice é 3; porém, devemos colocar o número 4, pois a amostragem vai parar no índice 3.

```
In [7]: frutas[1:4]
Out[7]: ['laranja', 'abacate', 'abacaxi']
```

Como visto no comando anterior, frutas[1] é laranja, e frutas[4] é banana. Mas, quando colocamos frutas[1:4], o último elemento não é banana, mas um índice antes, abacaxi, cuja representação na lista é frutas[3].

Em outro exemplo, se quisermos os dois primeiros elementos, devemos fatiar a lista começando em 0 e terminando em 2, visto que frutas[0] = 'limão' e frutas[1] = 'laranja'.

```
In [8]: frutas[0:2]
Out[8]: ['limão', 'laranja']
```

Outra forma de representar os dois primeiros elementos da lista de frutas é não colocar o índice zero, que automaticamente o Python vai subentender que desejamos o primeiro elemento, nesse caso, é limão. Então, usar frutas[0:2] ou frutas[:2] dará o mesmo resultado.

```
In [8]: frutas[0:2]
Out[8]: ['limão', 'laranja']

In [9]: frutas[:2]
Out[9]: ['limão', 'laranja']
```

Da primeira fruta até a última, saltando de duas em duas, usamos a seguinte notação [início : fim : salto], ou, para essa lista:

```
In [10]: frutas[0:6:2]
Out[10]: ['limão', 'abacate', 'banana']
```

Para selecionarmos os três últimos elementos da lista, fazemos uso desta operação:

```
In [11]: frutas[-3:]
Out[11]: ['abacaxi', 'banana', 'mamão']
```

E, para selecionarmos todos os elementos, com exceção do último,

```
In [12]: frutas[:-1]
Out[12]: ['limão', 'laranja', 'abacate', 'abacaxi', 'banana']
```

1.6 OPERAÇÕES COM ARRAY (VETORES)

Listas são formas de se armazenarem dados, mas são bastante limitadas. Por exemplo, não conseguimos fazer operações complexas entre os elementos de uma lista. Mesmo em operações simples, como dividir os elementos de uma lista por outros de outra lista, isso não é possível. Vamos supor que tenhamos duas listas "x" e "y":

```
In [13]: x=[10,2,6]

In [14]: y=[1,2,3]

In [15]: x/y
```

Quando tentarmos realizar a operação de divisão x/y, o diabo escondido nos erros começa a rir sobre nossos ombros, pois ele sabe que isso vai ocasionar um erro para o programador que ainda está iniciando seu aprendizado em Python.

Teremos a seguinte resposta de erro:

```
  File "C:\Users\marcoalc1\AppData\Local\Temp\ipykernel_1736\1719814938.py", line 1, in <module>
    x/y

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

Então, nas mais importantes operações entre elementos de listas, o primeiro passo é transformar as listas no formato de array (vetor). Para isso, precisamos importar a biblioteca NumPy, em que as funções são preparadas para operações com array. Após a importação, transformamos as duas listas em array com a NumPy, e, então, isso nos permite a divisão elemento a elemento de cada lista. O resultado de cada passo é apresentado a seguir. A função array( ) transforma lista em array (ou vetor).

```
In [16]: import numpy as np

In [17]: x=np.array(x)

In [18]: y=np.array(y)

In [19]: x/y
Out[19]: array([10.,  1.,  2.])
```

Também nesse caso, parece que o formato é o mesmo das listas anteriores; mas, se usarmos o *print*, veremos que as vírgulas desaparecem nesse formato, permitindo todo tipo de operação com os elementos:

```
In [20]: print(x/y)
[10.  1.  2.]
```

Assim como nas listas, com arrays, o acesso aos índices se dá por meio dos colchetes, ou seja, o primeiro elemento terá índice zero, o segundo, índice 1, e assim por diante.

```
In [21]: x[0]
Out[21]: 10

In [22]: x[1]
Out[22]: 2

In [23]: x[2]
Out[23]: 6
```

1.7 A BIBLIOTECA NUMPY

A biblioteca NumPy (*Numerical Python*) é uma das mais antigas e importantes do Python, servindo de base e auxiliando as demais bibliotecas conhecidas e as mais recentes. Ela ajuda nas operações necessárias entre vetores ou entre elementos de vetores.

A NumPy engloba tudo que existe nas bibliotecas Math e Statistics, aplicando as funções no formato de array. Por exemplo, se temos que x = 4 e desejamos a raiz quadrada de x, vimos anteriormente que basta importar a biblioteca Math e usar a função *sqrt* como math.sqrt( ). Logo:

```
In [24]: import math

In [25]: x=4

In [26]: math.sqrt(x)
Out[26]: 2.0
```

Mas vamos imaginar que tenhamos o vetor (ou lista)

x = [2, 4, 7, 8, 10, 15]

e desejamos transformar esses elementos em

$$y = \left[\sqrt{2}, \sqrt{4}, \sqrt{7}, \sqrt{8}, \sqrt{10}, \sqrt{15} \right]$$

Nesse caso, não adianta usar a biblioteca Math, pois ela não consegue trabalhar com vetores (ou array). O leitor poderá ver o erro que aparece a seguir, nele, novamente, o diabo contido nos dados espreita sua face.

```
In [27]: x = [2, 4, 7, 8, 10, 15]

In [28]: math.sqrt(x)
Traceback (most recent call last):

  File "C:\Users\marcoalc1\AppData\Local\
    math.sqrt(x)

TypeError: must be real number, not list
```

Para isso, devemos usar a biblioteca NumPy, que tem as mesmas funções da Math e da *Statistics*, no entanto, formulada para o uso em array. Podemos perceber que, ao aplicarmos a raiz quadrada em cada elemento de x, o resultado também será um vetor (palavra array na frente dos valores numéricos).

```
In [29]: import numpy as np

In [30]: x = [2, 4, 7, 8, 10, 15]

In [31]: np.sqrt(x)
Out[31]:
array([1.41421356, 2.        , 2.64575131, 2.82842712, 3.16227766,
       3.87298335])
```

Exemplo 1.1

Fazer um programa em Python que tenha dois vetores x e y, e, ao final, calcular e imprimir o vetor z, utilizando a fórmula:

$$z = \sqrt{e^{-x}\cos\left(x+y\right)}$$

Solução:

Nesse caso, percebemos que três funções matemáticas são necessárias para o cálculo no formato de array: raiz quadrada, exponencial e cosseno. Vamos supor que teremos x = [2, 3, 4] e y = [5, 2, 1]. A programação (agora no editor de texto do Python) será:

```
1   import numpy as np
2
3   x=np.array([2,3,4])
4
5   y=np.array([5,2,1])
6
7   z=np.sqrt(np.exp(-x)*np.cos(x+y))
8   print(z)
```

Observamos que, no caso do programa, apesar de a entrada estar em formato de lista para x e y, transformamos em vetor (array) para que seja possível o uso das funções matemáticas. Outra forma de usar a NumPy é, na hora da importação, buscar apenas pelas funções específicas que serão usadas, eliminando, assim, a importação de toda a NumPy. Basta, nesse caso, usar o comando "from", como a seguir.

```
1   from numpy import array, sqrt, exp, cos
2
3   x=array([2,3,4])
4   y=array([5,2,1])
5
6   z=sqrt(exp(-x)*cos(x+y))
7   print(z)
```

O resultado é:

```
[0.31942069 0.118839   0.0720795 ]
```

Exemplo 1.2

Fazer um programa para ler o vetor x = [2, 3, 7] e calcular as estatísticas: média, desvio-padrão e a soma total dos elementos. Imprimir o resultado no console.

```
1   import numpy as np
2
3   x=np.array([2,3,7])
4   media = x.mean()
5   desvio = x.std()
6   somat = x.sum()
7
8   print('**** resultado das estatisticas ****')
9   print('média = ',media)
10  print('desvio padrão = ',desvio)
11  print('soma total de x = ',somat)
12  print('************************************')
```

Uma das vantagens de se usar o formato de array é que muitas operações estatísticas já estão embutidas nele próprio, bastando colocar apenas o nome do array, indicando com um ponto qual a estatística desejada. Na resolução anterior, vimos que as medidas estatísticas são calculadas nas linhas 4, 5 e 6. O resultado para esse exemplo é:

```
**** resultado das estatisticas ****
média =  4.0
desvio padrão =  2.160246899469287
soma total de x =  12
************************************
```

Ainda assim, se desejarmos continuar usando a biblioteca NumPy, os comandos também são válidos, como apresentado a seguir.

```
1   import numpy as np
2
3   x=np.array([2,3,7])
4   media = np.mean(x)
5   desvio = np.std(x)
6   somat = np.sum(x)
7
8   print('**** resultado das estatisticas ****')
9   print('média = ',media)
10  print('desvio padrão = ',desvio)
11  print('soma total de x = ',somat)
12  print('************************************')
```

Exemplo 1.3

Fazer um algoritmo para calcular dez números aleatórios reais entre 0 e 1, e 10 números com distribuição de probabilidade normal com média 2 e desvio-padrão 3. Calcular a média e o desvio-padrão de cada um dos vetores.

Solução:

A biblioteca NumPy possui todas as distribuições de probabilidades importantes da estatística. No caso da função "random", retornarão números aleatórios entre 0 e 1, sendo necessário, nesse caso, apenas colocar a quantidade desejada de números a ser gerada pelo computador. Já para a distribuição normal, a função é "normal", mas devemos informar a média, o desvio-padrão e quantos números vamos gerar. A distribuição normal segue a fórmula da função gaussiana

$$f(x) = \frac{e^{\frac{-(x-\mu)^2}{2\sigma^2}}}{\sigma\sqrt{2\pi}},$$

em que μ é a média; e σ, o desvio-padrão. O programa do Python será:

```
1   import numpy as np
2
3   x=np.random.random(10)
4   y=np.random.normal(2,3,10)
5
6   media1=np.mean(x)
7   media2=np.mean(y)
8   desvio1=np.std(x)
9   desvio2=np.std(y)
10
11  print(x)
12  print(' ------------------------- ')
13  print(y)
14
15  print('######### estatisticas ######')
16  print('média aleatórios = ',media1)
17  print('média normal = ',media2)
18  print('desvio padrão aleatórios',desvio1)
19  print('desvio padrão normal', desvio2)
20  print('############################')
```

A solução nesse caso (veja que é aleatório; logo, quem rodar esse programa encontrará valores diferentes deste texto):

Números gerados para x e y

```
[0.28669552 0.72794918 0.62117001 0.60708563 0.78364714 0.06456026
 0.48292847 0.67124778 0.34346646 0.30147663]
 -------------------------
[ 2.36922803  0.58402867  5.30147158  4.03968223  4.84442716 -1.18335742
 -3.95025658  2.04796322  1.07479296  6.59921153]
```

Estatísticas para x e y

```
######### estatisticas ######
média aleatórios =  0.48902270808092235
média normal =  2.172719137931792
desvio padrão aleatórios 0.22045387496042823
desvio padrão normal 3.0421068723275315
############################
```

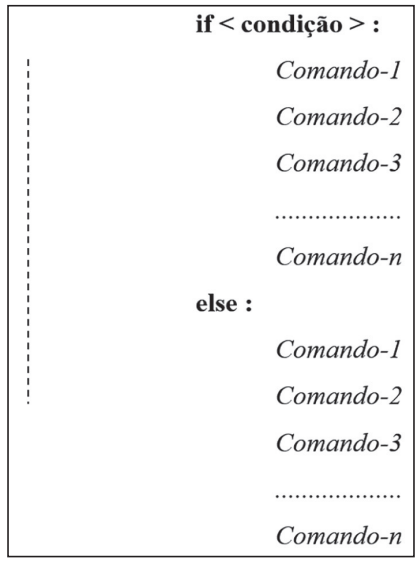
1.8 *FOR, WHILE* E O USO DA LÓGICA CONDICIONAL COM IF

Como todas as outras linguagens, o Python possui os comandos de *loop* para que suas estruturas rodem de forma automatizada para muitas operações repetitivas e comandos de lógica, como o if.

Vamos, primeiro, avaliar as decisões pelo computador com a utilização de lógicas baseadas no comando "if" ("se"). 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.

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 esta é verdadeira (*true*) ou falsa (*false*). A forma genérica da chamada para as decisões usando o if no Python é apresentada a seguir.

A estrutura do if no Python é:



Diferente 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.

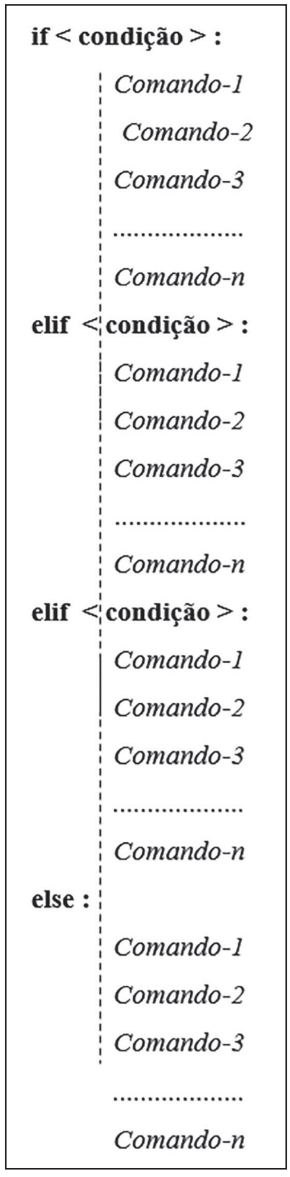
Exemplo 1.4

Fazer um algoritmo ler dois números reais "x" e "y" e dizer qual é o maior entre eles.

```
1    import numpy as np
2
3    x=float(input('x = '))
4    y=float(input('y = '))
5
6    if x>y:
7      print(x,' é o maior número')
8    else:
9      print(y, 'é o maior número')
```

Muitas vezes, apenas o caso *true* ou *false* não resolve a lógica para a tomada de decisão. Por exemplo, sendo 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 a falsidade da afirmação e, então, tomar a decisão. Outro fato é que, como se tem sempre uma pergunta dentro de outra pergunta encadeada, também é necessário que esse comando esteja alinhado ao primeiro if e tenha, ao final, o sinal de ":" (dois pontos). No esquema a seguir, é possível ver que os alinhamentos continuam obrigatórios para todos os elif e também para o else.



Junto com as tomadas de decisão, as questões lógicas, muitas vezes, precisam 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. Os sinais utilizados no Python para as expressões lógicas nas decisões são os seguintes:

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 " = =" (dois sinais de igual). A expressão de diferença é a exclamação seguida do sinal de igual, para dizer ao Python que as duas condições são "não iguais".

Exemplo 1.5

Aproveitando o exemplo anterior, agora, modificar o algoritmo para ler três números reais "x", "y" e "z" e dizer qual é o maior entre eles.

```
1   import numpy as np
2
3   x=float(input('x = '))
4   y=float(input('y = '))
5   z=float(input('z = '))
6
7   if x>y and x>z:
8     print(x,' é o maior número')
9   elif y>x and y>z:
10    print(y, 'é o maior número')
11  else:
12    print(z,'é o maior número')
```

O uso da lógica condicional "if" parece extremamente fácil, mas seu uso prático e para problemas que envolvam decisões complexas torna as colocações das perguntas algo fundamental para o programador. Por exemplo, tudo fica ainda mais difícil quando precisamos usar "if" dentro de *loops*, por meio dos comandos de iteração "while" e "for".

O comando que auxilia essa repetição ou iteração, também conhecido como *loop*, é o comando while. O esquema de repetição é apresentado na Figura 1.1, na qual o programa é representado como uma caixa que possui uma variável "S", que, a cada instante, recebe um valor diferente S = 1, 2, 3, …

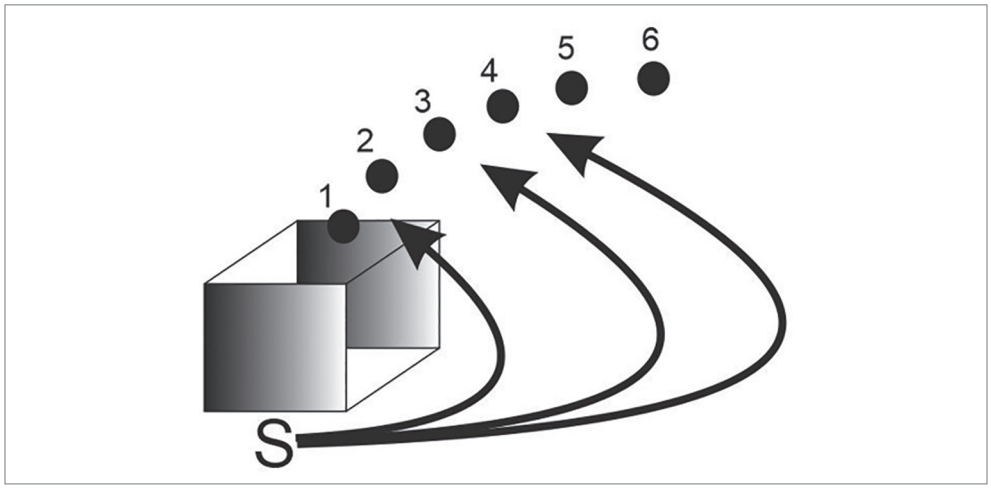


Figura 1.1 Iteração com *loop* em variável.

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 uma outra condição lógica em que o computador deve permanecer "preso" à ordem de repetição. Os comandos "*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, operações matemáticas, construções de gráficos, salvamentos em arquivos, abertura de arquivos etc.

Exemplo 1.6

Fazer um algoritmo para ler "n" números reais e contar quantos estão abaixo de zero, quantos estão entre 0 e 3 (não incluindo o 3) e quantos estão acima de 3 ou iguais a 3.

O que se deseja agora é uma ampliação da escolha das variáveis usando "if" para separação lógica, como há muitos termos e não sabemos quantos termos o usuário digitará, não faz sentido solicitar variáveis como x, y, ou z. Não sabemos quantos números são do interesse do usuário. Por isso, pedimos, inicialmente, que ele informe a quantidade de números que digitará para só então começamos a pedir os números usando "input" na linha 8 do script a seguir.

```
1   n=int(input('n = '))
2
3   i=1
4   cont1=0
5   cont2=0
6   cont3=0
7   while i<=n:
8     x=float(input('entre com x = '))
9     if x<0:
10      cont1=cont1+1
11    elif (x>=0) and (x<3):
12      cont2=cont2+1
13    else:
14      cont3=cont3+1
15    i=i+1
16
17  print('contagem de números abaixo de 0: ',cont1)
18  print('contagem de números entre 0 e 3: ',cont2)
19  print('contagem de números maiores ou iguais a 3: ',cont3)
```

Nesse exemplo, percebemos, que ao mesmo tempo que lemos os números, precisamos que o algoritmo separe e conte os números para cada faixa de intervalo. Enquanto o algoritmo lê um número específico digitado pelo usuário ("x") na linha 8,

o condicional, usando o "if", separa e salva a contagem nas variáveis *cont1*, *cont2* e *cont3*, que, inicialmente, são nulas, antes do *loop* do while.

Tudo o que está entre as linhas 7 e 15 é repetido até a contagem de "i" estourar seu limite "n" na linha 7. Por exemplo, se rodarmos o programa anterior para dez valores (n = 10) com os números {1, 2, 3.2, 2.7, 1.5, -3, -2, -8, 8.1, 4.7}, teremos três números negativos, quatro entre 0 e 3, e três números acima de 3.

A seguir, mostramos como entrar com os números no console do Python, que vai solicitando número por número até atingir n = 10 no comando de while da linha 7 do script anterior. É claro que o while faz muito mais do que apenas solicitar números, letras e separação delas. Seu poder está na automatização do código e na repetição de operações matemáticas importantes ou de cálculos exaustivos que são repetitivos.

```
n = 10

entre com x = 1

entre com x = 2

entre com x = 3.2

entre com x = 2.7

entre com x = 1.5

entre com x = -3

entre com x = -2

entre com x = -8

entre com x = 8.1

entre com x = 4.7
contagem de números abaixo de 0:  3
contagem de números entre 0 e 3:  4
contagem de números maiores ou iguais a 3:  3
```

Além do while, outro comando que tem o mesmo sentido de *loop* ou repetição é o "for". No Python seu formato é:

For i in range(0,n):
<commando>
<commando>
<commando>
<commando>

Nesse caso, "i" é o contador, e "range" indica qual o intervalo que esse contador varia ao longo do *loop*. Para o "for", não se inicializa a variável "i", pois, dentro do range, o primeiro número indica o início, e, o segundo indica o final nas repetições, iniciando em zero e terminando em (n-1). No caso do "for", apesar de o range ser (0,n), a iteração sempre terminará um ponto antes do indicado no intervalo. Por exemplo, repetições com range(0,5) começarão com i = 0 e terminarão com i = 4.

Outro formato possível é quando não se deseja percorrer os dados de 1 em 1, mas de 3 em 3. Ou seja, temos um step = 3 (passo = 3). Por exemplo, o comando for seria *for i in range(0,n,3)*.

Exemplo 1.7

Fazer um algoritmo para ler "n" números reais e imprimir x = 2*i + 1.

```
1    n=int(input('n = '))
2
3    for i in range(0,n):
4        x= 2*i + 1
5        print(i,'  ',x)
```

O resultado, nesse caso, para n = 5, é:

```
n = 5
0    1
1    3
2    5
3    7
4    9
```

No comando "print", o primeiro termo é o próprio contador, que começa em 0 e vai até 4. O leitor deve observar que "n" é a quantidade de elementos, mas, como o range começa com zero, seu final termina com um número a menos, porém, tendo n = 5 termos desejados.

Uma forma interessante de treinar o uso do "for" é com séries e sequências matemáticas.

Exemplo 1.8

Fazer um algoritmo para ler "n" (números de termos desejados para a série a seguir) e imprimir a soma total desses termos.

$$S = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \ldots$$

O primeiro passo nesse tipo de programa é encontrar a regra existente entre um termo e o outro. Devemos sempre, como num jogo de quebra-cabeça, encontrar onde colocamos o contador "i" da iteração. Não é difícil perceber que, para i = 1, 2, 3, …, o termo necessário a ser programado é 1/i na série anterior. Então, o programa será o seguinte.

```
1    n=int(input('n = '))
2    s=0
3    for i in range(1,n+1):
4        s = s + 1/i
5        print(i,' ',1/i,' ',s)
6
7    print('soma total dos termos = ',s)
```

Vamos supor que o usuário tenha escolhido apenas os quatro primeiros termos para a soma, ou seja, n = 4. Como o range termina um ponto antes, se deixarmos "n" no range, ele apenas fará a soma de 3 termos. Para obtermos a soma dos quatro primeiros termos, colocamos "n + 1" na linha 3, a fim de que o algoritmo calcule os quatro termos. Assim, como n = 4, o range será (1,5), mas, como ele termina antes, a soma irá de 1 a 4. Veja como aparece no console o resultado do programa anterior para n = 4.

O primeiro termo é 1, o segundo é 0.5, o terceiro é 0.33, e o último é 0.25. A soma é o acúmulo dos termos, ou seja, quando i = 1, temos s = 1; quando i = 2, temos a soma de 1 + 0.5, que nos fornece 1.5, e, assim, sucessivamente.

```
n = 4
1   1.0   1.0
2   0.5   1.5
3   0.3333333333333333   1.8333333333333333
4   0.25   2.083333333333333
soma total dos termos =  2.083333333333333
```

1.9 INSERÇÕES EM LISTAS E VETORES COM *LOOP*

Não são poucas as vezes que temos de incluir numa lista, ou num vetor, cálculos ou palavras decorrentes de algumas escolhas ou operações matemáticas. Para o caso de uma lista, devemos nos lembrar de que a inserção de termos em sua composição deve ser feita com a função "append". Veja o exemplo a seguir.

Exemplo 1.9

Fazer um algoritmo para ler "n" (números de termos desejados), os quais, respeitando-se a regra, devem ser colocados em uma lista como a que segue.

$$x = \left[\sqrt{2}, \sqrt{4}, \sqrt{6}, \ldots \right]$$

```
1    import math as mt
2    n=int(input('n = '))
3    x=[ ]
4    i=1
5    while i<=n:
6      y=mt.sqrt(i*2)
7      x.append(y)
8      i=i+1
9    print(x)
```

Resultado para n = 4:

```
n = 4
[1.4142135623730951, 2.0, 2.449489742783178, 2.8284271247461903]
```

No exemplo em questão, a operação matemática de cada termo era a raiz quadrada dos números pares. Basta apenas perceber que os números pares são gerados a partir do produto do contador por 2, ou seja, 2*i. A raiz quadrada faz parte da biblioteca Math, e a inserção de cada termo deve ser via append. Uma observação importante, que não pode ser dispensada, é que, no caso de uma lista, devemos sempre iniciá-la e avisar o Python de que sua composição inicial está vazia.

Por isso, a linha 3 possui x = [ ], pois, sem essa inicialização, o Python não tem condições de saber se estamos querendo alimentar uma lista, um vetor ou apenas uma variável simples e comum. Quando ele lê a linha 3, reconhece que, para todo o programa, deverá trabalhar x com as limitações que possui uma lista.

No caso de um "for", o procedimento não é muito diferente, tendo apenas que adequar o problema a forma de programação ao range no "for", como mencionado anteriormente com o uso de n + 1. O algoritmo é o mesmo do anterior, como visto a seguir.

```
1    import math as mt
2    n=int(input('n = '))
3    x=[]
4    for i in range(1,n+1):
5      y=mt.sqrt(i*2)
6      x.append(y)
7    print(x)
```

Mas a inserção para vetores é diferente. O vetor ou array precisa necessariamente ter alocadas as posições da memória que será usada. Não pode ser como no caso da lista, informando que ela começa vazia. Assim, o que podemos fazer é preencher as posições da memória do array com zeros para serem substituídos após o início dos cálculos.

Logo, o uso da função do NumPy, chamada de "zeros", indica que as "n" posições terão inicialmente zeros, e, depois, esses valores serão corrigidos pelos valores dos termos da raiz quadrada. A inserção também é diferente e não usa colchetes com lista vazia, mas, sim, o indicativo de quanto vale cada valor do array com seu índice respectivo x[i], como pode ser visto a seguir, na linha 7.

```
1    import math as mt
2    import numpy as np
3
4    n=int(input('n = '))
5    x=np.zeros(n)
6    for i in range(0,n):
7      x[i]=mt.sqrt((i+1)*2)
8
9    print(x)
```

O resultado não muda, e têm-se os mesmos valores da lista, mas, agora, no formato de array. A diferença, aqui, para n = 4 é que a saída é do tipo array, ou seja, é possível usar o array para operações matemáticas. No console, teremos a solução seguinte, sem as vírgulas separando os números.

```
n = 4
[1.41421356 2.         2.44948974 2.82842712]
```

Ou, então, se digitarmos "x" no console, é possível ver a palavra array na variável, indicando a solução como um vetor.

```
x
array([1.41421356, 2.        , 2.44948974, 2.82842712])
```

É claro que, para esse exemplo simples, não precisamos usar os *loops* com "for" ou com "while", visto que conseguimos o mesmo resultado de maneira bem mais simples usando apenas a biblioteca NumPy, como apresentado a seguir.

```
np.sqrt(np.arange(2,10,2))
array([1.41421356, 2.        , 2.44948974, 2.82842712])
```

1.10 ESTATÍSTICAS RÁPIDAS COM ARRAY (VETORES)

Como vimos anteriormente, estatísticas podem ser realizadas com dados usando a biblioteca Statistics. Também já foi mencionado como usar a NumPy para trabalhar com estatísticas ou geração de números aleatórios. Precisamos alertar o leitor sobre um fato importante de comparação entre bibliotecas.

A Statistics é específica para medidas estatísticas que estão em listas. Se, por descuido, um pesquisador ou programador esquecer disso, poderá usar a biblioteca errada, produzindo resultados que, a princípio, não indicam erro de compilação, mas que produzem erros numéricos.

Vamos supor que tenhamos uma lista x = [1, 2, 3] e desejamos calcular a média e o desvio-padrão populacional de x.

```
In [5]: import statistics as st

In [6]: x=[1,2,3]

In [7]: st.mean(x)
Out[7]: 2

In [8]: st.pstdev(x)
Out[8]: 0.816496580927726
```

Observamos que, nesse caso, a média da lista x é 2, e seu desvio-padrão calculado via biblioteca Statistics é 0.816. Mas se x fosse um array (vetor)? Vamos transformar a lista x anterior em vetor e usar as mesmas medidas via biblioteca Statistics.

```
In [9]: import numpy as np

In [10]: x=np.array(x)

In [11]: st.mean(x)
Out[11]: 2

In [12]: st.pstdev(x)
Out[12]: 0.0
```

O primeiro passo que fizemos foi chamar a biblioteca NumPy, na linha 9, e transformar a lista x em array, na linha 10. Na linha 11, usamos a média pela biblioteca Statistics, que forneceu o resultado correto. Mas se o leitor observar com calma, perceberá que o desvio-padrão está errado!

A Statistics voltou com valor zero para os mesmos números que antes estavam no formato de lista, e cujo desvio-padrão é 0.816, não zero, como podemos notar na linha 12. Quando calculamos as estatísticas pela biblioteca NumPy, que é própria para array, o resultado coincide com o resultado obtido na lista.

```
In [13]: np.mean(x)
Out[13]: 2.0

In [14]: np.std(x,ddof=0)
Out[14]: 0.816496580927726
```

Com uma notação diferente, o desvio-padrão populacional da biblioteca NumPy é std( ), usando ddof = 0. E, para o desvio-padrão amostral, deve-se usar ddof = 1. Mas o que significa isso? Vamos observar a fórmula do desvio-padrão populacional.

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}\left(x_i - \overline{x}\right)^2}{n}}$$

em que $\overline{x}$ é a média da amostra. Já para o desvio-padrão amostral, temos:

$$s = \sqrt{\frac{\sum_{i=1}^{n}\left(x_i - \overline{x}\right)^2}{n-1}}$$

Podemos observar que a única diferença entre os dois tipos de desvio-padrão é que o termo da divisão dentro da raiz é "n", no caso populacional, e "n – 1", no caso amostral. Ou seja, no caso populacional, temos "n – 0", e no caso amostral, temos "n – 1". Então, o termo ddof significa em inglês *delta degree of freedom*, que conhecemos como grau de liberdade. Quando desejamos grau de liberdade zero (ddof = 0), queremos calcular o desvio-padrão populacional; e quando colocamos grau de liberdade 1 (ddof = 1), desejamos o desvio-padrão amostral. A fórmula geral para o desvio-padrão na biblioteca NumPy é:

$$std(\ ) = \sqrt{\frac{\sum_{i=1}^{n}\left(x_i - \overline{x}\right)^2}{n - ddof}}$$

Qual a diferença entre os valores da amostra x? Ao compararmos, vemos que a diferença é pequena, mas é seu sentido estatístico que se torna significativo, pois se expressa sobre a população toda ou sobre uma pequena parcela escolhida aleatoriamente, a qual chamamos de amostra. Para o array x = [1 2 3], temos:

```
In [14]: np.std(x,ddof=0)
Out[14]: 0.816496580927726

In [15]: np.std(x,ddof=1)
Out[15]: 1.0
```

Qual é o padrão da NumPy? Quando não colocamos nada sobre ddof, o padrão é sempre voltar o cálculo do desvio-padrão populacional, como apresentado no comando a seguir. Sempre que desejarmos usar o desvio-padrão populacional, não precisaremos colocar ddof = 0; a NumPy já está programada para usar esse fator. Para o nosso array x, vemos que o resultado a seguir é o mesmo que o encontrado com ddof = 0.

```
np.std(x)
0.816496580927726
```

Já mostramos isso na definição da biblioteca NumPy, mas, aqui, podemos reforçar que uma maneira mais simples de usar as estatísticas no array é colocando a função desejada ao final do nome.

```
In [17]: x.mean()
Out[17]: 2.0

In [18]: x.std()
Out[18]: 0.816496580927726

In [19]: x.max()
Out[19]: 3

In [20]: x.min()
Out[20]: 1
```

1.11 GRÁFICOS BÁSICOS COM MATPLOTLIB.PYPLOT

A biblioteca mais simples e rápida de gráfico no Python é a Matplotlib.pyplot, cuja configuração básica trata-se apenas de colocar o comando de plotagem e o nome da variável que se deseja o gráfico. Vamos ver alguns exemplos.

Exemplo 1.10

Fazer um algoritmo para gerar o gráfico da equação de segundo grau

$$y = x^2 - 5x + 6$$

usando a biblioteca Matplotlib.

"O diabo mora nos detalhes", frase atribuída a Nietzsche, é dita sempre que algo dá errado. Quando programamos códigos no computador estamos sujeitos a erros que, a princípio, não deveriam ocorrer. Tragédias, em boa parte, ocorrem por descuido ou detalhes quase imperceptíveis.

Após anos de programação e a vivência de muitos erros em códigos, neste livro o autor buscou apresentar programas em linguagem Python que podem levar a erros quando se trabalha com dados não estruturados. Erros de programação, erros de lógica, erros de comandos e chamadas de bibliotecas que mudaram de configuração são disponibilizados no texto como alertas de erros. Soluções são apresentadas via biblioteca Pandas do Python. A obra oferece abordagens computacionais para iniciantes e amantes do assunto, com códigos em Python.

ISBN 978-85-212-2515-7

www.blucher.com.br

Blucher

Clique aqui e:

VEJA NA LOJA

Pandas e o demônio dos dados

Algoritmos em Python para Sistemas de Informação

Marco Antonio Leonel Caetano

ISBN: 9788521225157

Páginas: 470

Formato: 17 x 24 cm

Ano de Publicação: 2025
