Programação para estudantes de engenharia, usando Python

Marcio Delamaro Stevão Andrade Misael Costa Júnior
Claudinei Brito Júnior

Sumário

Prefácio

Na primeira parte deste texto são apresentados alguns métodos numéricos e os algoritmos para a sua implementação. Na segunda parte é apresentada a linguagem Python. As implementações dos métodos numéricos são apresentadas na segunda parte do texto, e servem como motivação para o estudante.

Parte I

Algoritmos

Capítulo 1

Introdução – o que são algoritmos

Algoritmos são sequências de passos que nos levam à solução de um problema. Vamos escolher, então um problema exemplo, e tentar mostrar, de forma prática o que seria um algoritmo para sua resolução.

O problema que usaremos é bem conhecido e bastante simples. Queremos resolver uma equação de segundo grau, ou seja, dada a equação $ax^2 + bx + c$, queremos saber quais são as suas raízes reais, se elas existirem.

Vamos, então, descrever uma sequência de passos para solucionar esse problema. A primeira questão é: qual o nível de detalhes que precisamos especificar nesses passos? Por exemplo, podemos simplesmente escrever:

> "Resolva a equação $ax^2 + bx + c$."

Mas isso não nos ajuda muito. O que precisamos é uma sequência de operações simples, que saibamos fazer, e que nos levem às raízes desejadas. Então, antes de mais nada, precisamos saber quais são as operações "conhecidas" que estão à nossa disposição, para que possamos descrever o algoritmo. No caso do nosso problema, o algoritmo de Bhaskara, descrito a seguir, supõe que sabemos efetuar operações aritméticas como soma, subtração, divisão, multiplicação e extrair a raiz quadrada.

Usando como exemplo a equação $2x^2 + 2x - 6$, seguimos os seguintes passos:

1. Identifique na equação, os coeficientes a, b, e c.

   No caso, $a = 2$, $b = 2$ e $c = -6$.

2. Se $a = 0$ então a equação não é de segundo grau, e o algoritmo não pode ser usado. Caso contrário, continue.

   No nosso caso, continuamos pois $a \neq 0$

3. Calcule o valor do discriminante $\Delta = b^2 - 4 \times a \times c$;

   Temos $\Delta = 2^2 - 4 \times 2 \times -6 = 4 + 48 = 52$

4. Se o valor de Δ for negativo, então a equação não tem raízes reais, e o algoritmo termina aqui. Caso contrário, prossiga.

   Continuamos, pois nosso Δ é positivo.

5. Compute o valor da primeira raiz, x_1, que é dada pela expressão

$$x_1 = \frac{-b + \sqrt{\Delta}}{2 \times a}$$

$x_1 = (-2 + \sqrt{52})/4 = 1,3027756377319946$

6. Compute o valor da segunda raiz, x_2, que é dada pela expressão

$$x_2 = \frac{-b - \sqrt{\Delta}}{2 \times a}$$

$x_2 = (-2 - \sqrt{52})/4 = -2,302775637731995$

7. Se $x_1 = x_2$ é porque o valor do discriminante é zero, então temos uma única raiz para essa equação.

   No nosso exemplo, não se aplica pois $\Delta = 52$ e portanto $x_1 \neq x_2$

Fim dos trabalhos. Seguindo esses passos, podemos tomar uma equação de segundo grau e ao final olhar os valores de x_1 e x_2 para conhecermos as suas soluções.

Capítulo 2

Raízes de funções

2.1 O método da bisseção

O método de Bhaskara, visto no capítulo anterior serve para computarmos as raízes de um tipo específico de função: as quadráticas. Se tivermos outros tipos de funções (contínuas), precisamos de outros métodos para calcular as suas raízes. Neste capítulo veremos um método simples para isso.

Vamos tomar como exemplo a função $x^3 - x^2 - 13x + 8$. Queremos achar os valores de x para os quais o valor da função é zero. Olhando o gráfico dessa função, vemos que ela tem três raízes, conforme mostra a Figura 2.1. Para aplicarmos o método da bisseção, precisamos conhecer o intervalo em que essas raízes estão. Por exemplo, vemos que a raiz mais negativa está em algum ponto no intervalo (-4, -3).

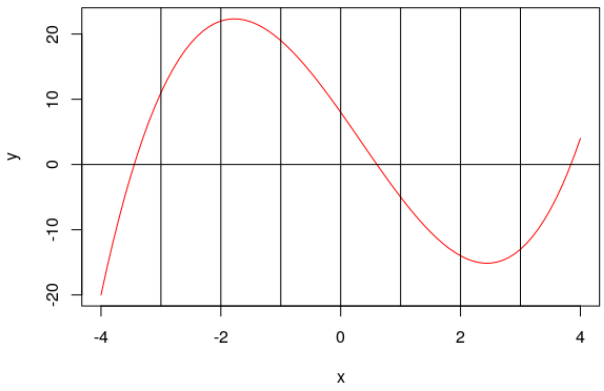

Figura 2.1: Gráfico da função $f(x) = x^3 - x^2 - 13x + 8$

Para tentar achar o ponto em que a curva cruza o eixo x, vamos chutar um valor entre os valores desse intervalo. Nada mais justo que escolher o ponto bem no meio dos dois. Ou seja, se consideramos o intervalo (a, b), vamos calcular um valor médio $c = (a + b)/2$. No caso do nosso exemplo, seria $c = -3,5$. Então

computamos o valor de f nesse ponto. Obtemos $f(-3.5) = -1,625$. Se o valor computado for zero, então achamos o valor da raiz. Caso contrário, conseguimos diminuir o tamanho do intervalo no qual devemos procurar a raiz.

Note, na Figura 2.2 que o valor da função no ponto médio $-3,5$ está abaixo do eixo x, ou seja, é negativo. Então, não adianta tentar procurar a solução que queremos no intervalo (a, c). Temos que procurar no outro subintervalo, ou seja, (c, b). Em outras palavras, devemos escolher esse intervalo pois verificamos que $f(c) \times f(b) < 0$, ou seja, $f(c)$ e $f(b)$ têm sinais opostos.

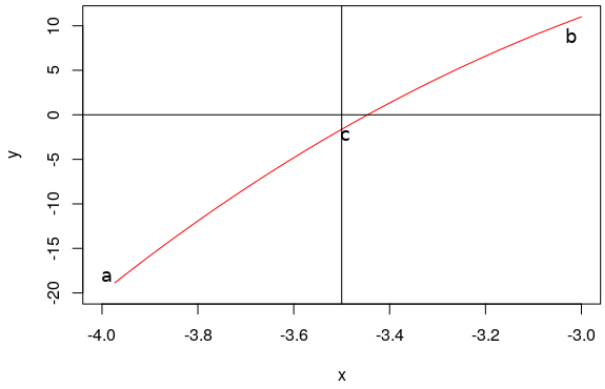

Figura 2.2: Dividindo o intervalo com o método da bisseção

Vamos repetir então as mesmas operações, mas agora considerando um intervalo menor. A cada iteração, o tamanho desse intervalo vai diminuindo e mesmo que nunca consigamos achar o valor exato da raiz, vamos cada vez nos aproximando mais dela. Ou, como costuma-se dizer, vamos convergindo para a solução.

Precisamos saber, também, qual é a tolerância, ou o erro que podemos admitir na nossa resposta. Por exemplo, se usarmos uma tolerância de $0,001$, isso significa que estamos satisfeitos com uma resposta que esteja a essa distância da raiz. Ou, olhando para o nosso algoritmo, quando o tamanho do intervalo for menor do que esse valor, podemos parar de dividir e assumir que o valor médio, c, é a resposta que desejamos.

Uma descrição, passo a passo do método pode ser a seguinte:

1. iniciamos com o intervalo (a, b);
2. calculamos o ponto médio do intervalo, $c = (a + b)/2$;
3. se $f(c) = 0$ ou se o $b - a <$tolerância, então, c é a resposta;
4. se $f(c) * f(a) < 0$ reduzimos o intervalo de busca para (a, c);
5. caso contrário o próximo intervalo de busca será (c, b);
6. repetimos o processo considerando esse novo intervalo.

A Tabela 2.1 mostra a sequência de valores que são utilizados em cada iteração do método, para a computar a primeira raiz do nosso exemplo. É bom notar que utilizamos uma tolerância bastante alta para podermos mostrar todos os valores calculados pois o método da bisseção converge de forma relativamente lenta. Se utilizássemos, por exemplo, 0.00001 como tolerância, precisaríamos de 17 iterações para achar o valor desejado.

Tabela 2.1: Sequência de valores no método da bisseção para $f(x) = x^3 - x^2 - 13x + 8$

Iteração	a	b	c	Erro
1	-4	-3	-3.5	0.5
2	-3.5	-3	-3.25	0.25
3	-3.5	-3.25	-3.375	0.125
4	-3.5	-3.375	-3.4375	0.0625
5	-3.5	-3.4375	-3.46875	0.03125
6	-3.46875	-3.4375	-3.453125	0.015625
7	-3.453125	-3.4375	-3.4453125	0.0078125
8	-3.453125	-3.4453125	-3.44921875	0.00390625
9	-3.44921875	-3.4453125	-3.447265625	0.001953125
10	-3.447265625	-3.4453125	-3.4462890625	0.0009765625

2.2 O método de Newton-Raphson

Este é um outro método iterativo para computarem-se raízes de funções. Assim como no método da bisseção, precisamos ter uma ideia de onde se encontra a raiz que desejamos computar. Mas nesse caso não precisamos definir um intervalo, e sim um "chute" inicial. Precisamos também conhecer a primeira derivada da função que desejamos computar e, ainda, um valor de tolerância que representa o erro máximo que aceitamos na nossa resposta.

A partir do nosso chute inicial, x_0, computamos a próxima aproximação da raiz fazendo:

$$x_i = x_{i-1} - \frac{f(x_{i-1})}{f'(x_{i-1})}$$

E assim, continuamos até que a diferença entre x_{i-1} e x_i seja menor que a nossa tolerância.

Olhando na Figura 2.3, podemos entender o que o método faz. Usando o nossa função como exemplo, e iniciando com um chute inicial de -3, achamos a reta tangente que passa por $f(-3)$ e cuja inclinação é dada por $f'(-3)$. Traçando essa reta, vamos encontrar o ponto onde ela cruza o eixo x, que é $-3,55$. Veja que esse ponto está mais próximo da raiz, e é justamente o próximo valor que usaremos como chute. Repetimos o processo nesse ponto, até chegarmos a uma aproximação aceitável.

Os primeiros passos, para o nosso exemplo, seriam os seguintes, considerando a primeira derivada da função $f'(x) = 3x^2 - 2x - 13$:

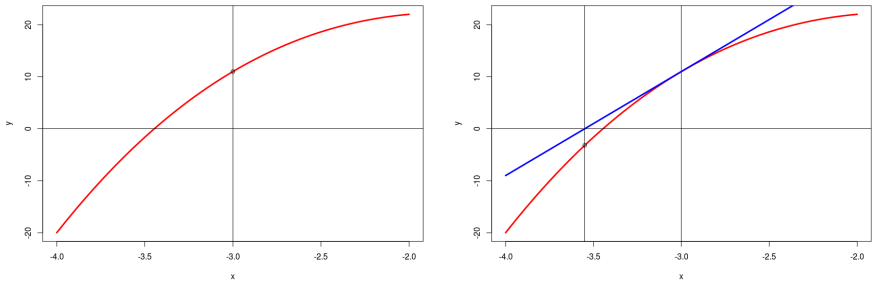

Figura 2.3: O método de Newton-Raphson para $f(x) = x^3 - x^2 - 13x + 8$

1) Chute inicial $x_0 = -3$
2) $x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} = -3 - \frac{11}{20} = -3.55$
3) $x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} = -3.55 - \frac{-3,1913749999999936}{31,9075} = -3,44998041212881$

Para acharmos a raiz mostrada na Figura 2.3, teríamos a sequência de pontos apresentada na Tabela 2.2. Note que, em relação ao método da bisseção, precisamos bem menos iterações para chegar na aproximação desejada. Na verdade, com essas 4 iterações chegamos a um erro inferior a 10^{-5}. Se utilizarmos uma aproximação de 10^{-9}, apenas uma iteração a mais é necessária. Realmente, uma das características deste método é que ele converge para a solução mais rapidamente.

Tabela 2.2: Sequência de valores no método de Newton-Raphson para $f(x) = x^3 - x^2 - 13x + 8$

Iteração	x_i	x_{i+1}	Erro
1	-3	-3,55	0,55
2	-3,55	-3,44998041212881	0,10001958787119
3	-3,44998041212881	-3,4460777931392697	0,003902618989540
4	-3,4460777931392697	-3,446071939012778	5,854126E-06

Por outro lado, dependendo da função que usamos e do chute inicial que damos, é possível que o método não convirja para a resposta correta. Por isso, é bom incluir na execução do algoritmo, um limite para o número de iterações. Se após esse determinado número de iterações não obtivermos o resultado desejado, o método falhou. Na Figura 2.4 mostramos o exemplo da função $f(x) = xe^{-x^2}$, para o qual escolhemos o valor inicial $x_0 = 1$. Como vemos com as duas primeiras iterações, os valores escolhidos vão se afastando da solução desejada ao invés de se aproximar. Para esse caso, se escolhermos um valor no intervalo aberto $(-0,5, 0,5)$, obteremos a resposta correta.

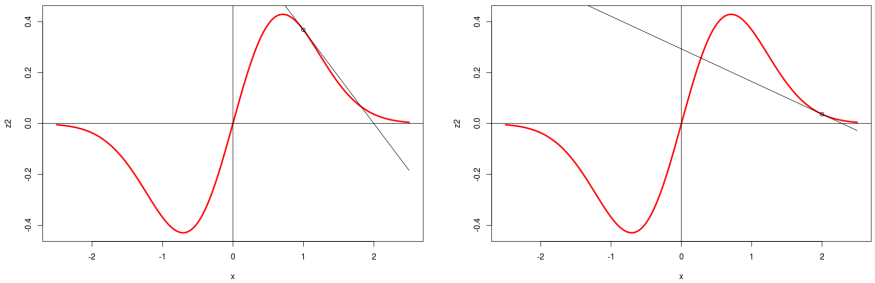

Figura 2.4: O método de Newton-Raphson para $f(x) = xe^{-x^2}$

Capítulo 3

Integração numérica

Os alunos de engenharia certamente estão familiarizados com conceitos de cálculo como limite, derivada e integral. Vamos falar um pouco sobre o cálculo de integrais, como:

$$\int_0^2 x^3 + 8 \; dx$$

Aprendemos que para computar o valor dessa integral, basta fazermos

$$\int_a^b f(x) \; dx = F(b) - F(a)$$

onde $F(x)$ corresponde à antiderivada de $f(x)$. Ou no nosso caso,

$$F(x) = \frac{1}{4}x^4 + 8x$$

$$\int_0^2 x^3 + 8dx = F(2) - F(0) = 20,0$$

Nesse caso bastou-nos computar a antiderivada da função, e conseguimos calcular o valor exato da integral. Porém, nem sempre é fácil computar a antiderivada e nesses casos precisamos apelar para métodos numéricos para achar o valor da integral.

Sabemos, também, que computar essa derivada é o mesmo que achar a área sob a curva, no intervalo que desejamos. Na Figura 3.1 vemos a curva da função $f(x) = x^3 + 8$ e, no intervalo [0,2], a área sob a curva, preenchida em cinza, corresponde à integral $\int_0^2 x^3 + 8dx$.

Então, a solução é tentar computar, de forma aproximada, a área sob a curva, e daí obter o valor da integral. Existem vários métodos para fazermos isso. Nas duas próximas seções vamos estudar dois deles.

3.1 O método dos trapézios

Nesse método, buscamos achar uma reta que interpole, ou represente da melhor maneira possível, a curva entre os dois pontos, inicial e final do intervalo de interesse. Como conhecemos apenas dois pontos, essa reta passa por esses dois

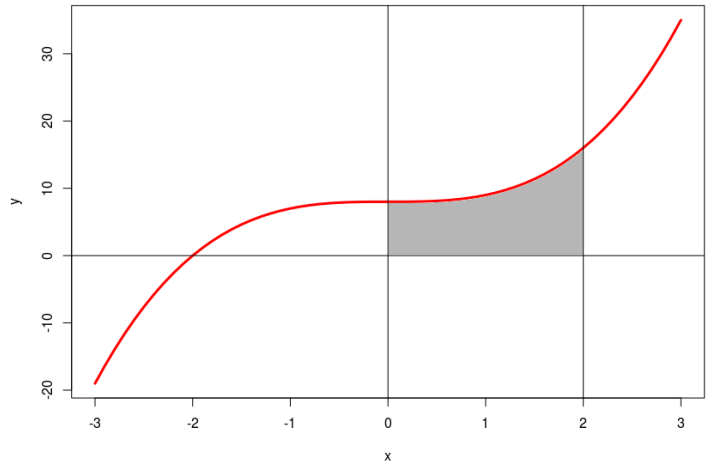

Figura 3.1: Representação da área sob a curva de $x^3 + 8$

pontos, como mostrado na Figura 3.2a. Computamos, então, a área sob a reta, que como indica o nome do método, forma um trapézio. O tamanho de uma das bases do trapézio é dado por $f(a)$ e o da outra por $f(b)$. A área, então, é $\frac{f(a)+f(b)}{2} \times (b - a)$, já que a altura do trapézio é $b - a$.

Essa aproximação não é nada boa. O valor exato para a integral do nosso exemplo, como vimos, é 20.0 e o valor aproximado computado pelo método é 24. Porém, podemos melhorar esse resultado dividindo o intervalo $[a, b]$ em subintervalos menores e calcular a área de cada um deles, usando o mesmo método dos trapézios. A Figura 3.2b mostra o intervalo dividido em quatro subintervalos, e os trapézios por eles determinados. Computando as áreas de cada um deles, obtém-se uma soma de 20.25, mais próximo do resultado esperado. Quanto mais subintervalos tivermos, maior será a precisão da resposta. Por exemplo, com 40 intervalos, temos a aproximação de 20.0025. Com 500 intervalos teremos 20.00001600000001.

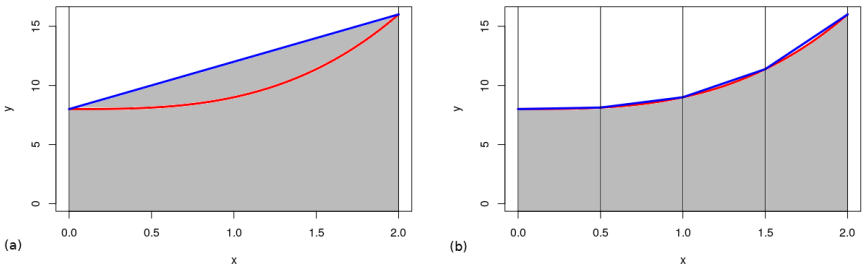

Figura 3.2: O método de quadratura dos trapézios para $f(x) = x^3 + 8$

Então, podemos descrever o algoritmo dos trapézios, informalmente, da se-

guinte forma, considerando $f(x)$ e o intervalo $[a, b]$:

1. determinamos n, o número de subintervalos que queremos usar;
2. computamos o tamanho de cada subintervalo, $h = (b - a)/n$;
3. computamos a área de cada subintervalo A_i;
4. computamos a área total $A = \sum_{i=1}^{n} A_i$.

3.2 O método de Simpson

Este método busca usar um polinômio de grau dois para interpolar três pontos da função que queremos computar a integral. No caso, os pontos a, b e o ponto médio. Da mesma forma, utilizando um único intervalo pode não gerar bons resultados, assim, dividimos o intervalo em subintervalos e então aplicamos o método.

Se dividirmos o intervalo em n subintervalos, obtemos uma sequência de valores $a, x_1, , x_2, ..., x_{n-1}, b$, que utilizaremos para computar a integral. Com um pouco de manipulação matemática, que não faremos aqui, obtém-se uma maneira bastante simples de computar a integral:

1. determinamos n, o número de subintervalos que queremos usar. n deve ser par;
2. computamos o valor $h = (b - a)/n$;
3. para cada x_i, em que i é impar, adicionamos $4f(x_i)$ ao valor da integral;
4. para cada x_i, em que i é par, adicionamos $2f(x_i)$ ao valor da integral;
5. adicionamos $f(a)$ e $f(b)$ ao valor da integral;
6. multiplicamos esse valor por $h/3$, que nos dá a resposta que queremos.

Para polinômios de grau até três, o método apresenta resultado exato com apenas dois subintervalos. Por isso, vamos utilizar como exemplo a função da Figura 2.4, $f(x) = xe^{-x^2}$. No intervalo $(0, 2)$, obtemos, com 40 subintervalos, o valor 0.4908423652668696 e no intervalo $(-2, 2)$ obtemos $2.868076146948321e-17$, valor muito próximo de zero, que seria o valor exato esperado.

Capítulo 4

Estatística descritiva

Não é fácil descrever o que seja estatística, ou tudo o que ela compreende. Mas, podemos vê-la como um conjunto de técnicas que permitem colher dados, organizá-los e analisá-los, para que possamos deles extrair algum tipo de conhecimento que nos interesse.

Nesse capítulo vamos apresentar uma pequena fração dessas técnicas. A estatística descritiva, segundo dizem os especialistas, é a etapa inicial que tomamos para sumarizar e tentar entender os dados. Note que nosso objetivo aqui não é ensinar estatística ou como utilizá-la, mas sim mostrar como podem ser calculas algumas medidas que estão associadas à estatística descritiva.

Vamos supor, então, que temos uma população e que dela queremos estimar uma variável. Por exemplo, queremos saber qual a altura da população masculina no Brasil. Ou quantos litros de cerveja bebem as estudantes de engenharia da USP, por semana. Infelizmente, não podemos coletar esses dados de todos os elementos que compõem essas populações. Por isso, fazemos uma amostragem, por exemplo, entrevistando ou medindo um subconjunto pequeno desses elementos.

Assim, temos um conjunto de dados, x_1, x_2, x_3, ..., x_n, que contém os valores medidos para a variável de interesse, para n elementos que compõem a nossa amostra. Por exemplo, podemos entrevistar dez estudantes de engenharia e descobrir que, por semana, elas bebem a quantidade de cerveja mostrada na Tabela 4.1.

Tabela 4.1: Quantidade de cerveja ingerida pelas estudantes de engenharia da USP

Estudante	1	2	3	4	5	6	7	8	9	10
Litros	1,5	2	2	4	0	2,5	3,5	2	6	3,5

A primeira medida, talvez a mais conhecida, é a média desses dados. Ela é uma das medidas que chamamos de medidas de tendência central ou medidas de posição. Como o nome sugere, elas servem que tenhamos uma ideia da localização dos dados, dentro da escala em que foram medidos. A média pode ser calculada da seguinte maneira:

$$média = \frac{x_1 + x_2 + ... + x_n}{n}$$

Ou seja, o cálculo da média é feita somando-se todos os valores da amostra e dividindo a sua soma pelo número de elementos na amostra, n. No caso das meninas que bebem cerveja, se somarmos os valores das medidas de cada uma delas, obtemos 27 litros de cerveja por semana. Ou, a média de 2,7 litros.

Outra medida de posição é a mediana. Ela é definida como o valor central das medidas. Ou seja, metade dos valores da amostra é maior do que a mediana e metade é menor.

Para calcular a mediana fica mais fácil ordenar a nossa amostra. Se o tamanho da amostra for ímpar, a mediana corresponde ao valor central da sequência de valores ordenados. Se for par, fazemos a média dos dois elementos centrais. Por exemplo, ordenando os valores da nossa amostra exemplo, obtemos a sequência a seguir.

$$0 \quad 1{,}5 \quad 2 \quad 2 \quad 2 \;\big|\; 2{,}5 \quad 3{,}5 \quad 3{,}5 \quad 4 \quad 6$$

Como o tamanho da amostra é par, não existe um elemento central, então pegamos o quinto e o sexto elementos e calculamos a média deles. Assim, obtemos a mediana: $(2 + 2,5)/2 = 2,25$.

A moda corresponde ao valor que aparece mais vezes ou que é mais comum na amostra. Por exemplo, entre as estudantes de engenharia, o valor mais comum para consumo semanal de cerveja é de dois litros. Esse valor aparece três vezes, na amostra coletada. Para calcular a moda, precisamos saber qual é a "frequência" de cada valor da amostra, ou seja, quantas vezes cada valor apareceu. Como na Tabela 4.2, na qual vemos quantas vezes apareceu cada valor da amostra. O valor mais frequente é o 2 que, portanto, é a moda da mostra.

Tabela 4.2: Frequência de ocorrências de cada valor da Tabela 4.1

Valor	0	1,5	2	2,5	3,5	4	6
Frequência	1	1	3	1	2	1	1

Note que é possível que uma amostra tenha mais do que uma moda. Se, por exemplo, o valor 6 aparecesse, também, três vezes na amostra, teríamos dois valores para a moda: 2 e 6. Então, precisamos olhar na tabela de frequências, todos os valores da amostra que correspondem ao maior valor de frequência para escolher a moda.

Outras medidas, chamadas de dispersão, servem para verificar o quanto os dados variam. No nosso exemplo, se tivéssemos uma amostra em que todas as moças tomam exatamente 2,7 litros de cerveja por semana, continuaríamos com a mesma média mas com uma dispersão menor. Ou sejam, os dados variam menos do que na amostra original.

A primeira medida é a amplitude. Ela é calculada pela diferença entre o maior valor e o menor valor da amostra. Se a amostra estiver ordenada, basta subtrair o primeiro do último valor. No nosso exemplo, a amplitude é $6 - 0 = 6$.

Já o cálculo da variância é um pouco mais trabalhoso. A variância avalia a dispersão total medindo o quanto cada ponto se afasta da medida central da média. Em outras palavras, calculamos a variância como:

$$variância = \frac{(x_1 - média)^2 + (x_2 - média)^2 + ... + (x_n - média)^2}{n - 1}$$

Ou seja, para cada valor da amostra, computamos a sua diferença com a média e a elevamos ao quadrado. Somamos todos esses valores e dividimos a soma pelo tamanho da amostra menos um. O desvio padrão é uma outra medida de dispersão, definida apenas como a raiz quadrada da variância.

Capítulo 5

Resolução de sistemas de equações lineares

Muitos problemas de engenharia requerem a solução de um sistema de equações lineares. Ou seja, se tivermos várias variáveis cujo valor precisamos calcular e o mesmo número de equações envolvendo essas variáveis, queremos calcular os valores das variáveis que satisfaçam todas as equações.

Por exemplo, vamos considerar um sistema com quatro incógnitas e quatro variáveis. Queremos no sistema a seguir, descobrir valores para x_1, x_2, x_3 e x_4 que satisfaçam as equações.

$$\begin{cases} 3x_1 + 6x_2 - x_3 & = & 25 \\ -2x_1 + 3x_2 + x_3 + x_4 & = & 6 \\ x_1 - 4x_2 + 2x_3 + 2x_4 & = & 2 \\ -2x_1 - 2x_2 + 2x_4 & = & 0 \end{cases}$$

As técnicas para resolver esse tipo de sistema baseiam-se na manipulação de matrizes. Uma matriz quadrada é usada para representar os coeficientes das variáveis e uma matriz coluna para representar o lado direto das equações. Temos então algo como:

$$\begin{bmatrix} 3 & 6 & -1 & 0 \\ -2 & 3 & 1 & 1 \\ 1 & -4 & 2 & 2 \\ -2 & -2 & 0 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 25 \\ 6 \\ 2 \\ 0 \end{bmatrix}$$

Ou, podemos ter uma matriz estendida, em que juntamos os dois lados da equação em uma única matriz.

$$\begin{bmatrix} 3 & 6 & -1 & 0 & 25 \\ -2 & 3 & 1 & 1 & 6 \\ 1 & -4 & 2 & 2 & 2 \\ -2 & -2 & 0 & 2 & 0 \end{bmatrix}$$

5.1 Regra de Cramer

Para resolver esse sistema de equações utilizando a Regra de Cramer (RC), precisamos, antes de mais saber como calcular o determinante de uma matriz quadrada. Nós conhecemos algumas regras para calcular o determinante de matrizes pequenas como as 3×3 mas quando temos matrizes maiores, precisamos de alguma outra alternativa. Uma das formas de fazê-lo é utilizando o método de Laplace.

Definimos o cofator de um elemento $a_{i,j}$ como sendo um valor $A_{i,j} = (-1)^{i+j}.D_{i,j}$. O valor $D_{i,j}$ é o determinante da matriz quadrada que sobra se tirarmos a linha i e a coluna j da matriz original. Usando a matriz do exemplo anterior, os cofatores de $a_{1,1}$ e de $a1,2$ podem ser computados da seguinte forma:

$$A_{1,1} = (-1)^2 \cdot \begin{vmatrix} 3 & 1 & 1 \\ -4 & 2 & 2 \\ -2 & 0 & 2 \end{vmatrix}$$

$$A_{1,2} = (-1)^3 \cdot \begin{vmatrix} -2 & 1 & 1 \\ 1 & 2 & 2 \\ -2 & 0 & 2 \end{vmatrix}$$

Duas coisas devem ser notadas nessa definição. Primeiro, os índices de uma posição da matriz, somados, resultam em um valor ímpar, então o cofator é $-D_{i,j}$ e se resultam em valor par, é $D_{i,j}$. Segundo, para calcular o cofator de uma posição temos, ainda, que computar o determinante de uma matriz. Porém, essa matriz é menor que a matriz original. Assim, podemos ir diminuindo o tamanho da matriz até que saibamos como calcular o determinante. No exemplo acima, temos:

$$A_{1,1} = 1^2.20 = 20$$

$$A_{1,2} = -1^3. - 10 = 10$$

No método de Laplace, devemos escolher uma linha ou uma coluna da matriz. Computamos os cofatores de todos os elementos dessa linha ou coluna, multiplicamos os cofatores pelos valores dos elementos a que eles correspondem e somamos todos eles. Assim obtemos o determinante da matriz.

Vamos escolher da matriz M do nosso exemplo, a primeira linha. Então, temos que calcular o cofator de todos os elementos da linha e o valor do determinante é dado por:

$$\begin{array}{rcccccccc} det(M) = & a_{1,1}A_{1,1} & + & a_{1,2}A_{1,2} & + & a_{1,3}A_{1,3} & + & a_{1,4}A_{1,4} \\ & 3 \times 20 & + & 6 \times 10 & + & -1 \times -20 & + & 0 \\ = & 140 & & & & & & \end{array}$$

Agora que sabemos computar o determinante, podemos utilizar a regra de Cramer. Para computar o valor de x_i no sistema de equações, devemos:

a) pegamos a matriz coluna da direita da igualdade no sistema de equações;

b) substituímos, na matriz de coeficientes, a coluna correspondente a x_i pela matriz coluna;

c) calculamos o determinante dessa nova matriz; e

d) dividimos esse valor pelo determinante da matriz de coeficientes original.

O valor calculado dessa forma nos fornece o valor de x_i. è importante notar que essa regra só pode ser aplicada de o determinante da matriz de coeficientes é diferente de zero. Essa, na verdade, é uma propriedade conhecida, ou seja, um sistema de equações só possui solução se o determinante da sua matriz de coeficientes for diferente de zero.

Sabemos que o determinante da matriz de coeficientes é 140. Para calcularmos o valor de cada x_i no nosso exemplo, temos os seguintes cálculos:

$$x_1 = \frac{\begin{vmatrix} 25 & 6 & -1 & 0 \\ 6 & 3 & 1 & 1 \\ 2 & -4 & 2 & 2 \\ 0 & -2 & 0 & 2 \end{vmatrix}}{140} = \frac{420}{140} = 3$$

$$x_2 = \frac{\begin{vmatrix} 3 & 25 & -1 & 0 \\ -2 & 6 & 1 & 1 \\ 1 & 2 & 2 & 2 \\ -2 & 0 & 0 & 2 \end{vmatrix}}{140} = \frac{350}{140} = 2,5$$

$$x_3 = \frac{\begin{vmatrix} 3 & 6 & 25 & 0 \\ -2 & 3 & 6 & 1 \\ 1 & -4 & 2 & 2 \\ -2 & -2 & 0 & 2 \end{vmatrix}}{140} = \frac{-140}{140} = -1$$

$$x_4 = \frac{\begin{vmatrix} 3 & 6 & -1 & 25 \\ -2 & 3 & 1 & 6 \\ 1 & -4 & 2 & 2 \\ -2 & -2 & 0 & 0 \end{vmatrix}}{140} = \frac{770}{140} = 5,5$$

A regra de Cramer parece o método perfeito para resolvermos sistemas de equações lineares, certo? Ele é simples e, como veremos na segunda parte do livro, fácil de implementar. É verdade, mas ela tem um problema: ela é computacionalmente muito cara. Isso quer dizer que para computar a solução usando o método de Cramer, o número de operações que temos que fazer é muito grande. Na verdade, para computar o determinante de uma matriz precisamos realizar muitas multiplicações. E isso, se tivermos uma matriz muito grande, faz com que a execução do programa demore muito tempo para executar, ou que não termine nunca de executar.

Assim, se tivermos um sistema de equações relativamente grande – dez ou mais equações e variáveis – fica impossível usar esse método. Por isso precisamos de outro método.

5.2 Método da eliminação de Gauss

Esse método tem como objetivo transformar a matriz de coeficientes em uma matriz triangular superior equivalente à original.

Para conseguir isso, vamos aplicar algumas operações básicas que vão modificando a matriz mas que mantêm a mesma solução para o sistema de equações. Primeiro, vamos zerar os elementos da primeira coluna da matriz, um de cada vez, começando com a segunda linha. O elemento $a_{1,1}$ é o pivô, que vai ser usado nas substituições.

Para zerar um elemento da primeira coluna, $a_{i,1}$, calculamos inicialmente o valor $d = -(ai, 1/a_{1,1})$. Então, aplicamos a operação de multiplicar a primeira linha por d e somá-la na linha i. Por exemplo, para zerar $a_{2,1}$, temos $d = -(-2/3) = 2/3$. Multiplicamos cada elemento da linha 1 e somamos na linha 2, inclusive na última coluna.

$$\begin{bmatrix} 3 & 6 & -1 & 0 & 25 \\ 3\times\frac{2}{3}+(-2) & 6\times\frac{2}{3}+3 & -1\times\frac{2}{3}+1 & 0\times\frac{2}{3}+1 & 25\times\frac{2}{3}+6 \\ 1 & -4 & 2 & 2 & 2 \\ -2 & -2 & 0 & 2 & 0 \end{bmatrix} =$$

$$\begin{bmatrix} 3 & 6 & -1 & 0 & 25 \\ 0 & 7 & 0,3333 & 1 & 22,6667 \\ 1 & -4 & 2 & 2 & 2 \\ -2 & -2 & 0 & 2 & 0 \end{bmatrix}$$

Em seguida, ainda usando $a_{1,1}$ como pivô repetimos o processo para zerar $a_{3,1}$ e $a_{4,1}$. temos como resultado:

$$\begin{bmatrix} 3 & 6 & -1 & 0 & 25 \\ 0 & 7 & 0,3333 & 1 & 22,6667 \\ 0 & -6 & 2,3333 & 2 & -6,3333 \\ 0 & 2 & -0,6667 & 2 & 16.6667 \end{bmatrix}$$

Terminamos de tratar a primeira coluna. Em seguida, vamos usar $a_{2,2}$ como pivô e zeramos os elementos abaixo dele, ou seja, $a_{3,2}$ e $a_{4,2}$. Obtemos a matriz a seguir. É bom notar que o pivô não pode ser zero. Se isso acontecer, devemos procurar uma outra linha, abaixo dessa, que tenha um valor não zero na coluna do pivô e trocar as duas linhas de posição.

$$\begin{bmatrix} 3 & 6 & -1 & 0 & 25 \\ 0 & 7 & 0,3333 & 1 & 22,6667 \\ 0 & 0 & 2,6190 & 2,8571 & 13,0952 \\ 0 & 0 & 0,7619 & 1,7143 & 10,1905 \end{bmatrix}$$

Por último, usamos como pivô o elemento $a_{3,3}$ para zera $a_{4,3}$. E obtemos a matriz de coeficientes que é triangular superior.

$$\begin{bmatrix} 3 & 6 & -1 & 0 & 25 \\ 0 & 7 & 0,3333 & 1 & 22,6667 \\ 0 & 0 & 2,6190 & 2,8571 & 13,0952 \\ 0 & 0 & 0 & 2,5454 & 14 \end{bmatrix}$$

Nesse ponto, podemos computar o valor de x_4 pois na última linha da matriz sobrou uma única variável. Temos então:

$$\begin{aligned} 2,5455x_4 &= 14 \\ x_4 &= 5,5 \end{aligned}$$

Devemos usar a terceira linha da matriz para computar o valor de x_3, uma vez que já conhecemos x_4. A equação que temos é:

$$\begin{aligned} 2,6190x_3 + 2,8571x_4 &= 13,0952 \\ 2,6190x_3 + 2,8571(5,5) &= 13,0952 \\ x_3 &= -1 \end{aligned}$$

E assim, sucessivamente, podemos calcular x_2 usando a segunda equação e x_1 usando a primeira equação. Obtemos a solução $x_1 = 3$, $x_2 = 2,5$, $x_3 = -1$ e $x_4 = 5,5$.

Capítulo 6

Cálculo do caminho mínimo

Nesse capítulo, vamos tratar do algoritmo, comum em vários problemas de engenharia, de acha o menor caminho entre dois nós de um grafo valorado. Se o leitor não entendeu o que isso significa, não se preocupe pois vamos, inicialmente, introduzir alguns conceitos importantes.

6.1 Grafos

Um grafo é uma estrutura que serve para representar um relacionamento entre os elementos de um conjunto. Por exemplo, vamos tomar o conjunto de alunos de uma determinada classe e queremos representar quais pares de alunos são amigos. Temos então

$$V = \{Ana, Beto, Carlos, Dani, Enio, Francisco, Guto, Helena\}$$

o conjunto de alunos.

E para representar quem é amigo de quem, usamos um conjunto de pares, não ordenados como:

$$E = \{(Ana, Beto), (Ana, Dani), (Ana, Francisco), (Beto, Carlos), (Beto, Guto),$$
$$(Carlos, Dani), (Carlos, Francisco), (Carlos, Guto), (Carlos, Helena),$$
$$(Dani, Francisco), (Enio, Guto), (Enio, Helena), (Francisco, Guto)\}$$

Isso quer dizer, por exemplo, que Beto e Ana são amigos mas não o são Ana e Guto.

Isso é um grafo, que denotamos $G = (V, E)$. O conjunto V é dito conjunto de vértices e E é o conjunto de arestas do grafo. Dois vértices ligados por uma aresta são ditos vizinhos ou adjacentes.

Uma representação muito usada para grafos é na forma de um diagrama, no qual os vértices são apresentados como círculos e as arestas como arcos, que unem aquelas arestas que estão relacionadas. Por exemplo, o grafo de amizades é apresentado na Figura 6.1. Essa representação facilita muito a visualização das relações entre os vértices.

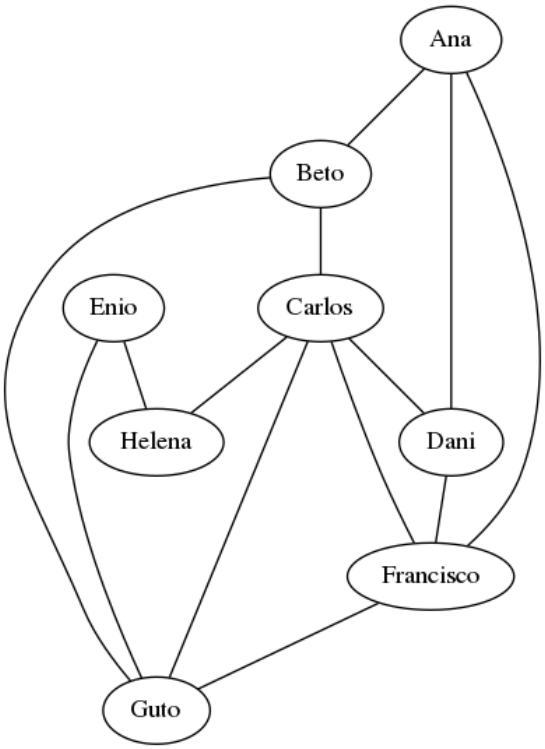

Figura 6.1: Representação do grafo de amizade entre estudantes

A relação de amizade, pelo menos no nosso grafo, é considerada simétrica, ou seja, se Ana é amiga de Beto, então Beto é amigo de Ana. Nem toda relação é assim. Por exemplo, a relação "é pai de" é unidirecional; se Arnaldo é pai de Bernardo, isso não significa que Bernardo é pai de Arnaldo. Podemos, também, expressar esse tipo de relação em grafos ditos "dirigidos". Nesse caso, as arestas são definidas por pares ordenados. E no caso da representação por diagrama, os arcos são setas, indicando a direção em que o relacionamento é válido. Um exemplo:

$$V = \{Arnaldo, Bernardo, Camila, Dino, Ernesto\}$$

$$E = \{(Arnaldo, Bernardo), (Arnaldo, Camila), (Bernardo, Dino)\}$$

Esse grafo é apresentado na Figura 6.1. Ele indica que Arnaldo é pai de Bernardo e de Camila. Bernardo é pai de Dino e Ernesto não tem relação com nenhum deles.

Um para encerrar por enquanto, vamos falar um pouco sobre grafos valorados. Nesses grafos, associam-se às arestas, números que representam o custo ou valor daquela aresta, dependendo da relação que se deseja exprimir e do problema que se deseja resolver.

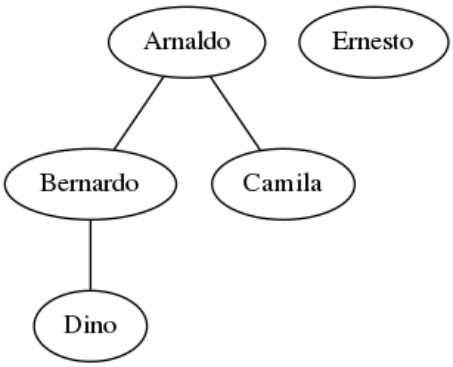

Figura 6.2: Representação do grafo entre pais e filhos

Vamos tomar como exemplo algumas cidades do interior de São Paulo e as rodovias que as servem. Podemos representar as distâncias diretas entre as cidades por meio do grafo da Figura 6.1. Vemos por exemplo, que existe uma forma de ir de São Carlos a Rio Claro diretamente, sem passar por nenhuma outra cidade, e a distância a ser percorrida é de 65 km. Já para ir de São Carlos a Campinas, por exemplo, não existe uma rota direta, sem passar por alguma outra cidade. E nesse caso, várias rotas são possíveis.

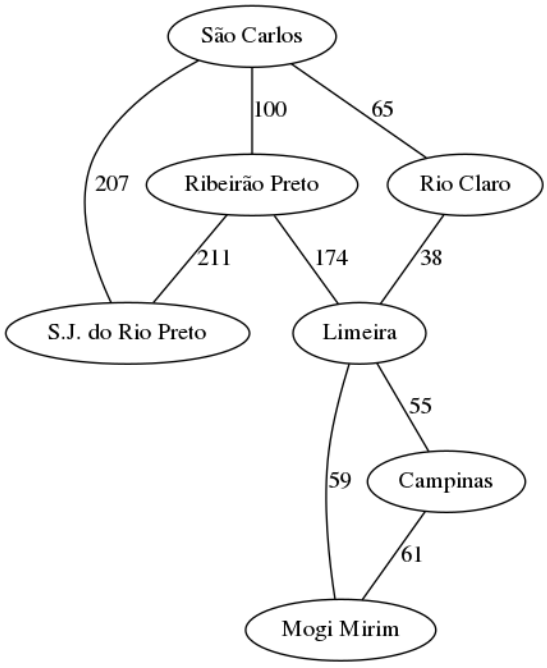

Figura 6.3: Grafo que exprime as distâncias diretas entre algumas cidades

6.2 O algoritmo de Dijkstra

O problema que queremos resolver serve para o grafo da Figura 6.1 e também para qualquer grafo valorado. Dado um vértice de início, queremos saber qual é o menor custo para se chegar a um determinado vértice final. No caso do nosso grafo das cidades queremos saber qual é a menor distância a percorrer saindo de uma cidade (S.J. do Rio Preto, por exemplo) para se alcançar outra cidade (Campinas, por exemplo).

Esse problema parece trivial pois olhando o nosso grafo dá pra se calcular com certa facilidade a solução. Mas imagine um grafo com todas as cidades do Brasil. Qual seria a rota mais curta entre Belém e Porto Alegre? Além disso, outros custos podem ser associados a esse mesmo grafo – por exemplo, preço dos pedágios – e o algoritmo para achar a solução ainda é o mesmo.

O algoritmo de Dijkstra é um dos que solucionam esse problema. Na verdade, ele acha, a partir de um determinado vértice A, o menor caminho para todos os outros vértices. Vamos tentar descrever como funciona o algoritmo. Suponha o grafo $G = (V, E)$, $V = \{v_1, v_2, ..., v_n\}$ e v_{init} o vértice inicial.

1. Criamos um vetor D que representa a estimativa de distância de v_{init} até cada um dos outros vértices. Inicialmente atribuimos a $D[init] = 0$ e $D[i] = \infty$ para todo $i \neq init$;

2. criamos também, um vetor A que indica quem é o antecessor de cada um dos vértices. Ou seja, indica como chegar a cada vértice pelo caminho mínimo.

3. achamos o vértice, não visitado, que tem a menor distância até v_{init}. Isso é feito olhando o vetor D e achando seu menor valor. Chamamos esse vértice de k

4. para cada vizinho do vértice k, ou seja, para cada vértice i adjacente a k, calculamos sua distância até v_{init}. Isso é feito da seguinte forma:

   - se $D[i] > D[k] + custo\ da\ aresta(k, i)$, atualizamos a estimativa de distância $D[i] = D[k] + custo\ da\ aresta(k, i)$. Além disso, registramos que k é o antecessor de i fazendo $A[i] = k$;
   - caso contrário, $D[i]$ e $A[i]$ permanecem como estão.

5. o vértice v_k é marcado como visitado

6. se existe algum nó que ainda não foi visitado, retornamos ao passo 3

7. caso contrário, D tem a distância mínima de cada nó até v_{init} e A mostra qual é o caminho mínimo.

Vamos utilizar o grafo das cidades como exemplo para ver como funciona o algoritmo de Dijkstra. Para isso, vamos numerar os vértices, conforme a Figura 6.2.

Inicialmente, temos os seguintes valores, considerando que queremos a cidade de São José do Rio Preto como início:

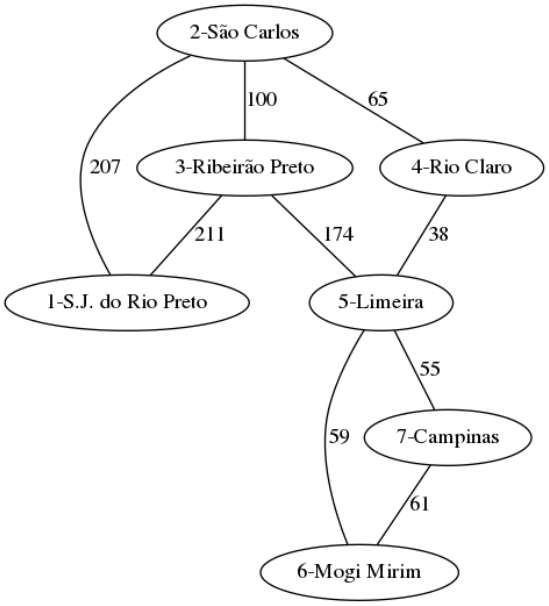

Figura 6.4: Grafo numerado que exprime as distâncias diretas entre algumas cidades

	1	2	3	4	5	6	7
D	**0**	∞	∞	∞	∞	∞	∞
E							
Visitado							

Executando a primeira iteração do algoritmos, selecionamos o vértice 1 como sendo a menor distância, ou seja, temos que $k = 1$. Os vizinhos do vértice 1 são o 2 e o 3. No passo 4, computamos os valores de $D[2]$ e $D[3]$. Como esses valores inicialmente tem um valor muito alto (∞), eles vão ser calculados como o valor de $D[1]$ somado com o peso das aresta $(1, 2)$ e $(1, 3)$, respectivamente. Além disso, o antecessor desse dois vizinhos fica sendo o vértice 1 que, por sua vez, é marcado como visitado.

	1	2	3	4	5	6	7
D	0	**207**	211	∞	∞	∞	∞
A		1	1				
Visitado	**X**						

Continuando, selecionamos $k = 2$, e seus vizinhos são 1, 3 e 4. O valor de $D[1]$ é menor do que o valor de $D[2] + custo(1, 2)$ e por isso, não vai ser modificado. O mesmo acontece com $D[3]$ que atualmente vale 211. Se tomarmos $D[2] + custo(2, 3)$ obtemos $207 + 100 > 211$ e portanto, não alteramos $D[3]$. Diferentemente, precisamos ajustar $D[4]$ que passa a valer $207 + 65 = 272$ e o antecessor do vértice 4 é o vértice 2.

	1	2	3	4	5	6	7
D	0	207	**211**	272	∞	∞	∞
A		1	1	2			
Visitado	**X**	**X**					

Continuando a executar, temos a seguinte sequência de valores para as nossas variáveis.

	1	2	3	4	5	6	7
D	0	207	211	**272**	385	∞	∞
A		1	1	2	3		
Visitado	**X**	**X**	**X**				

Note que nos dados abaixo vamos recalcular o valor do vértice 5, pois descobrimos que chegar até ele pelo vértice 4 é mais econômico do que pelo vértice 3. Por isso, mudamos também o seu antecessor.

	1	2	3	4	5	6	7
D	0	207	211	272	**310**	∞	∞
A		1	1	2	4		
Visitado	**X**	**X**	**X**	**X**			

	1	2	3	4	5	6	7
D	0	207	211	272	310	369	**365**
A		1	1	2	4	5	5
Visitado	**X**	**X**	**X**	**X**	**X**		

	1	2	3	4	5	6	7
D	0	207	211	272	310	**369**	365
A		1	1	2	4	5	5
Visitado	**X**	**X**	**X**	**X**	**X**		**X**

	1	2	3	4	5	6	7
D	0	207	211	272	310	369	365
A		1	1	2	4	5	5
Visitado	**X**	**X**	**X**	**X**	**X**	**X**	**X**

Olhando o quadro final, podemos identificar qual é a menor distância a ser percorrida de São José do Rio Preto até cada uma das outras cidades. Por exemplo, para chegar a Campinas são necessários 365 km. Além disso, olhando a linha dos antecessores, podemos saber qual é o caminho mínimo para chegar até uma dada cidade. No caso de Campinas, vértice 7, temos que o vértice 5 o antecede. Depois, 4 antecede o 5, 2 antecede o 4 e 1 antecede o 2. Portanto, nosso caminho mínimo seria $(1, 2, 4, 5, 7)$.

Parte II

A linguagem Python

Capítulo 7

Introdução – o que é Python

Monty Python foi um grupo cômico britânico muito bom. Eu (o mais experiente dos autores) particularmente gostava muito dos seus filmes, pelo menos aqueles que chegavam ao Brasil como:

- Monty Python e o cálice sagrado;
- A vida de Brian; e
- O sentido da vida.

Python é, também, um gênero de cobras. Mas, a origem do nome da linguagem de programação é mesmo uma homenagem ao grupo inglês.

Python é uma linguagem de programação. É fácil de usar, mesmo para os principiantes. E para aqueles que se dedicam um pouco mais ao seu aprendizado, revela estruturas poderosas e flexíveis. Por isso, embora não seja uma linguagem nova (foi crida na década de 1980), tem ganhado popularidade rapidamente.

Além das qualidades relacionadas especificamente à linguagem, o uso de Python é alavancado pelo fato de sua implementação ser bastante eficiente, ou seja, programas escritos em Python são rápidos para executar. Além disso, existe uma quantidade enorme de "pacotes" desenvolvidos para Python.

Esses pacotes (ou bibliotecas, como costumamos chamar) são programas que já implementam a solução para alguns problemas "populares" e que outros programadores podem usar nos seus próprios programas. Por exemplo, alguns pacotes muito utilizados pelos usuários de Python são:

- **NumPy**: biblioteca desenvolvida para matemáticos, cientistas e engenheiros. Implementa principalmente funções para manipulação de matrizes;
- **SciPy**: acrescenta recursos à biblioteca NumPy, com funções para cálculos científicos como cálculo numérico de integrais, otimização e resolução de equações diferenciais;
- **matplotlib**: biblioteca para a geração de gráficos para representação de dados como histogramas, gráfico de pizza e barras, entre muitos outros;
- **Pygame**: biblioteca para criação de jogos.

Existe um site (que chamamos de repositório de software) no qual podemos encontrar quase todos os pacotes de Python, em quase qualquer área. No repositório `https://pypi.org/` encontramos mais de 140 mil projetos, que podem ser utilizados na construção dos nossos programas.

7.1 Como instalar

Sistemas operacionais Linux, em geral, já possuem em sua distribuição normal o Python instalado. Windows, em geral, não. De qualquer forma, é possível fazer o download da versão que você deseja e instalá-la no seu computador.

Para isso, acesse a página `https://www.python.org/downloads/` e escolha o seu sistema operacional e a versão que deseja instalar. Em princípio, não existe nenhum motivo para não baixar a versão mais recente. No caso da Figura 7.1, seria **Python 3.6.5**. Baixe o instalador e execute-o no seu computador, como mostra a Figura 7.2. Não esqueça de marcar a opção "Add Python to PATH".

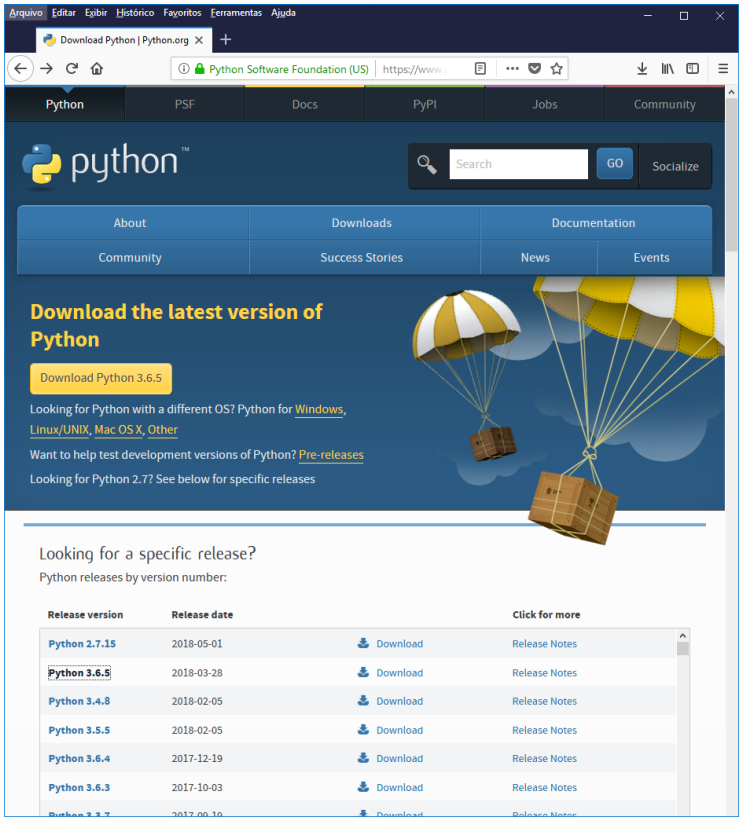

Figura 7.1: Baixando a versão mais recente de Python

Figura 7.2: Instalador Python para Windows

7.2 Como executar

Para verificar que o Python foi instalado corretamente, você pode abrir o seu "Prompt de comando" do Windows e digitar "python" na linha de comandos. Você deve ver as mensagens exibidas na Figura 7.3. Essas mensagens indicam que você estará interagindo com o interpretador de Python, sobre o qual falaremos no capítulo seguinte. Para sair do ambiente do interpretador, você pode fornecer o comando "exit()" e então estará de volta ao "Prompt de comandos" do Windows.

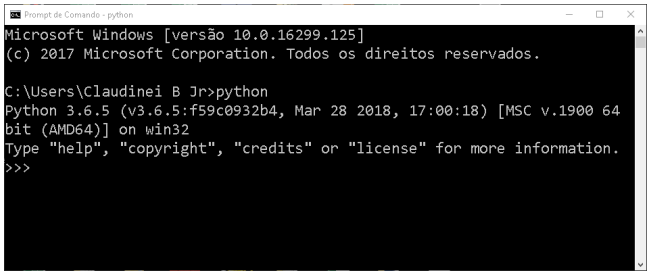

Figura 7.3: Executando Python no Prompt de comandos do Windows

No caso do Linux, o procedimento para executar o interpretador é o mesmo. Ou seja, abrimos um console e digitamos "python". Em alguns casos, é possível termos instalados duas versões diferentes da linguagem, no caso, o Pyhton 3 e o Python 2.7. Então, devemos, para invocar a versão mais recente do interpretador, executar "python3".

Uma outra forma de verificar a sua instalação e executar o interpretador Python é procurando nos programas do Windows pelo "IDLE Python..." (Figura 7.4). Executando esse programa, abre-se uma janela que já é o interpretador, o mesmo que executamos na linha de comandos. Essa janela, mostrada na Figura 7.5 contém na parte superior uma série de opções para criar e executar programas. Ou seja, além do interpretador, o "Shell" de Python provê recursos

como um editor de texto para criarmos nossos programas e um depurador [1]

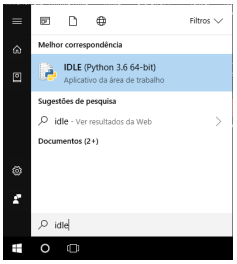

Figura 7.4: Executando o shell Python no windows

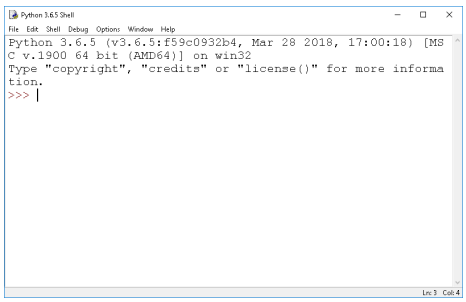

Figura 7.5: Janela do shell Python no Windows

No Linux também podemos executar o interpretador por meio da aplicação "ipython" ou "ipython3" (que poderá ser instalada executando o comando "sudo apt install ipython" ou "sudo apt install ipython3"), que cria um novo console e já executa o interpretador nesse console.

No próximo capítulo iremos conhecer um pouco mais sobre o interpretador Python. Para evitar o uso excessivo de figuras, exibiremos as interações entre o programador e interpretador no formato exibido abaixo. O símbolo ">>>", mostrado na cor verde, é o *prompt* do interpretador, ou seja, é o sinal de que o interpretador está esperando o programador digitar um comando. Os comandos digitados pelo programador aparecem em preto e os resultados, exibidos pelo interpretador aparecem em azul.

```
>>> print("O valor esperado é: " + str(10))
O valor esperado é: 10
>>>
```

[1] Um depurador é um programa que permite ao programador executar o programa de forma controlada, por exemplo, comando por comando. É bastante útil para procurarmos erros nos nossos programas.

Capítulo 8

O interpretador Python

Como mencionado no capítulo de introdução, o ambiente para a execução do Python é baseado num interpretador de comandos. Esse interpretador é um programa que pode ser executado a partir da linha de comandos de um terminal, ou, como vimos no capítulo anterior, por meio de um programa do Windows ou Linux.

Uma vez que o interpretador esteja sendo executado, você, programador, pode fornecer comandos, e dependendo do que você mandou o interpretador fazer, ele lhe fornece a resposta correspondente.

8.1 Números

Inicialmente, vamos pensar no interpretador como uma simples calculadora, à qual fornecemos uma expressão e ela nos fornece o resultado. Por exemplo, ao digitar a expressão "2 + 2" seguida da tecla *Enter*, o interpretador Python responde com o resultado dessa soma.

```
>>> 2 + 2
4
>>>
```

E assim, podemos fornecer expressões e a nossa "calculadora" fornece o resultado. Vejamos mais alguns exemplos, adiantando que o asterisco é utilizado como operador de multiplicação:

```
>>> 2 * 3 + 5
11
>>> 2 + 3 * 5
17
>>>
```

Note, na segunda expressão do interpretador acima, que a expressão calculada segue as regras de precedência que nós já conhecemos, ou seja, primeiro fazemos a multiplicação e divisão, que tem maior precedência, e depois a soma e a subtração.

Para mudarmos a ordem de execução das operações, podemos utilizar pares de abre e fecha parênteses. Cada expressão pode ter um ou mais deles. Como nos exemplos abaixo:

```
>>> 2 * (3 + 5)
16
>>> (2 + 3) * 5
25
>>> (2 + 3) * (8 - 2)
30
>>> (2 + 3) * ((8 - 2) / (4 - 13))
-3.333333333333333
>>>
```

Se escrevermos alguma coisa que não seja válida, ou seja, que o interpretador não entenda como um comando válido, a resposta apresentada vai ser um mensagem de erro, tentando indicar o que fizemos de errado.

```
>>> 2 * 3 +
  File "<stdin>", line 1
    2 * 3 +
          ^
SyntaxError: invalid syntax
>>>
```

A primeira linha, por enquanto não nos interessa. As duas seguintes mostram o ponto do comando onde o erro foi identificado, por meio da "setinha" ^. E a última linha mostra que o que fizemos de errado foi não seguir as regras sintáticas para um expressão bem formada. No caso, o interpretador estava

esperando achar alguma coisa para usar como operando do operador +, mas não encontrou. Outros tipos de erros podem ocorrer, não relacionados com a sintaxe da expressão, mas com sua execução, como por exemplo, ao tentarmos dividir um número por zero.

```
>>> 3 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

8.2 Números não são todos iguais

Cada elemento que usaremos na linguagem Python tem um "tipo". Dessa forma, o que podemos fazer com um determinado elemento depende do seu tipo. Por exemplo, não faz sentido multiplicar duas coisas se elas não forem números.

Mas mesmo os números em Python se dividem em dois tipos diferentes. Os inteiros (que indicamos por `int`) e os de ponto flutuante (que indicamos por `float`) [1].

Os números `int` são aqueles que representam valores inteiros como 1, 2, 3 ou −1, −2 e −3. Ao contrário do que acontece com outras linguagens de programação, não existe um limite mínimo ou máximo para os números inteiros que podem ser representados em Python. Assim, podem-se utilizar números tão grandes – negativos e positivos – quanto se queira. Os números `float` representam frações, ou mais precisamente, o conjunto de números racionais. São números que possuem a parte inteira e um número finito de casas decimais como 2.718281828459045 ou 3.141592653589793 [2]. Para sabermos o tipo de um número, podemos usar o seguinte comando:

[1] Na verdade, existe suporte para outros tipos de números em Python. Mas por ora eles não nos interessam.

[2] Assim como na maioria das linguagens de programação, utiliza-se um ponto para separar a parte inteira da parte decimal, e não uma vírgula, como usamos por padrão na língua portuguesa.

```
>>> type(2)
<class 'int'>
>>> type(2.0)
<class 'float'>
>>> type(2 + 2.0)
<class 'float'>
>>> type(4 / 2)
<class 'float'>
>>> type(5 // 2)
<class 'int'>
>>>
```

A primeira coisa a notar deste exemplo é que a forma como escrevemos um número determina o seu tipo. O número 2 é um `int` e o 2.0 é um `float`. Quer dizer que não importa o valor que o número representa, mas sim a forma que ele se apresenta. Um número de ponto flutuante pode, também, ser escrito no formato de notação científica. Nele, o valor 2.0 pode ser representado como 2e0 ou seja 2×10^0, o 20.0 seria 2e1 (2×10^1), o 0.2 seria 2e-1 (2×10^{-1}) e o 0.0002 seria 2e-4 (2×10^{-4}).

```
>>> 2e0
2.0
>>> 2e1
20.0
>>> 2e-1
0.2
>>> 2e-4
0.0002
>>>
```

O segundo ponto importante é que o resultado de uma operação aritmética também é um número e, portanto, tem um tipo. A regra para saber o tipo do resultado de uma operação é: se os operandos são iguais, o resultado tem o mesmo tipo que os operandos usados na operação. Se eles forem inteiros, o resultado é inteiro e se foram `float`, o resultado é `float`. A exceção a essa regra é a divisão, representada pela barra. Como no exemplo anterior, "4/2" produz como resultado o float 2.0.

```
>>> type(2 + 2)
<class 'int'>
>>> type(2.0 + 2.0)
<class 'float'>
>>> type(4 / 2)
<class 'float'>
>>>
```

E quando os operadores são de tipos diferentes? Ou seja, um `int` e um `float`. Nesse caso, antes de fazer a operação, o interpretador vai transformar o número inteiro num numero `float` e então utilizar esse `float` na operação. Ou seja, ao executarmos a expressão $2.0 + 3 * 2$, o interpretador:

i) faz a multiplicação, obtendo o resultado 6 (`int`);

ii) transforma esse 6 em 6.0 (`float`);

iii) faz a soma, obtendo 8.0(`float`).

O operador // computa a divisão inteira de dois números. Mas ele respeita a regra de que o tipo dos operandos determina o tipo do resultado. Ele divide o primeiro operando pelo segundo e o resultado é esse valor, truncado, ou seja, apenas com a parte inteira. Se um dos operandos for `float`, o resultado é `float`, caso contrário o resultado é um `int`.

```
>>> 5 // 2
2
>>> -5 // 2
-3
>>>
```

Note que na segunda expressão do exemplo acima, o resultado não é exatamente o que esperávamos, ou seja, -2. Isso acontece porque, por definição, o interpretador Python calcula a divisão inteira da seguinte forma:

i) divide o primeiro operando pelo segundo, obtendo o quociente;

ii) pega o mais próximo valor inteiro, que seja menor do que o quociente;

iii) esse é o resultado da operação.

Então, no caso de -5 // 2, a divisão resulta -2.5 e o valor inteiro, menor do que -2.5 é o -3.

Esse comportamento está relacionado com a definição de outro operador, que é o resto da divisão inteira, indicado pelo %. No caso de números inteiros positivos, é bem simples entender o seu comportamento. Por exemplo, $13\%4 =$

1, ou seja, $13 \;//\; 4 = 3$ e resta 1, que é o resultado da operação. No caso de quaisquer números, incluindo números não inteiros e negativos, o que se espera é que, se tivermos dois números A e B, a seguinte relação seja válida:

$$(A \;//\; B) \;*\; B \;+\; A \;\%\; B = A$$

Ou seja, se pegarmos o quociente da divisão inteira, multiplicarmos pelo denominador da divisão e somarmos o resto, obtemos o numerador da divisão.

```
>>> -13 // 4.2
-4.0
>>> -13 % 4.2
3.8000000000000007
>>> (-13 // 4.2) * 4.2 + (-13 % 4.2)
-13.0
>>>
```

Além desses operadores vistos até agora, temos a exponenciação, indicada por um "$**$". Por exemplo, 3 ** 2 quer dizer 3^2, e é igual a 9. E temos ainda os operadores $+$ e $-$ unários, ou seja, que se aplicam a um único operando para determinar o seu sinal. Por exemplo:

```
>>> -3 + -8
-11
>>> -3 + +8
5
>>> - -3 + 8
11
>>> - -3 + +8
11
>>> ---3 + -8
-11
>>>
```

É importante conhecermos a precedência (ou prioridade) que têm esses operandos. A maior de todas é do operando de exponenciação. Ele tem precedência maior até que os operadores unários, de sinal. Portanto, tome cuidado ao escrever algo como

```
>>> -3 ** 2
-9
>>> (-3) ** 2
9
>>>
```

A primeira expressão, primeiro computa a potência e depois aplica o sinal ao resultado, produzindo -9. A segunda, eleva o valor -3 ao quadrado, resultando no valor 9.

Além disso, ao contrário das outras operações, a exponenciação tem o que chamamos "associatividade à direita". Ou seja, quando escrevemos 2 ** 2 ** 3 obtemos o valor 256 como resposta, pois o que estamos calculando é: 2 ** (2 ** 3). Se quisermos elevar dois ao quadrado e o resultado ao cubo, devemos utilizar parênteses.

```
>>> 2 ** 2 ** 3
256
>>> (2 ** 2) ** 3
64
>>>
```

Depois da exponenciação, a maior prioridade é dos operadores unários de sinal. Depois, os de multiplicação e divisão (*, /, // e %). fim os de soma e subtração. A Figura 8.1, resume a ordem de precedência dos operadores.

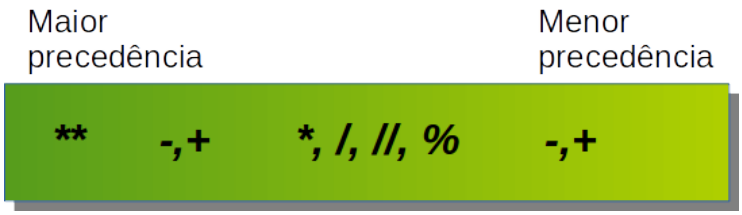

Figura 8.1: Precedência dos operadores aritméticos

8.2.1 Exercícios

1. Realize as operações abaixo no interpretador Python para praticar e fixar os operadores aritméticos disponíveis na linguagem de programação Python.

   » $10 + 5 = 15$

» 10 - 5 = 5

» 10 * 5 = 50

» 10 / 5 = 2.0

» 10 // 5 = 2

» 10 ** 5 = 100000

2. Agora que já sabemos realizar operações básicas no interpretador Python, vamos observar o resultado de algumas operações. Converta as seguintes operações para executar no interpretador Python e observe o resultado, destacando a procedência dos operadores aritméticos:

   » $10 - 2 \times 3 = 4$

   » $5 + 2 - 1 \times 6 - 4 / 2$ = -1.0

   » $10 / 2 - 1 \times 6$ = -1.0

   » $10 / (2 - 1) \times 6 = 60.0$

   » $10 \% 2 + 5 \times 3 - 2 = 13$

   » $5 - 4^2 / 1 + 3$ = -8

3. Suponha que um aluno obteve as seguintes notas durante um semestre letivo: 4, 6, 8 e 10. Portanto, a média final do aluno é igual a 7. Baseado neste exemplo, utilize o interpretador Python para calcular a média para as seguintes notas:

   » 7, 6, 4, 9

   » 5, 6, 2, 8

   » 10, 10, 9, 10

   » 8, 8, 9, 4

4. Realize a conversão de temperatura em graus Fahrenheit para graus Celcius. Utilize o interpretador Python para executar a fórmula C = (5 * (F-32) / 9) e realizar a conversão das seguintes temperaturas em graus Fahrenheit para graus Celcius:

   » 77.0 °F

   » 69.8 °F

   » 91.4 °F

   » 104.0 °F

5. O perímetro equivale a soma de todos os lados de uma figura geométrica. Tendo isso em vista, utilizando o interpretador Python, calcule o perímetro das seguintes figuras geométricas:

   » Triângulo escaleno com lados iguais a 15 cm, 12 cm e 20 cm

   » Retângulo com base de 20 cm e altura de 12 cm

   » Quadrado com lados iguais a 8 cm

6. Calcule o IMC (Índice de Massa Corporal) no interpretador Python. Substitua os valores de altura (em metros) e peso (em quilogramas) na fórmula do IMC, execute a fórmula no interpretador Python e observe o resultado:

   » IMC = peso / (altura * altura)

8.3 Variáveis

Todos conhecemos variáveis, desde o tempo do ensino básico. Por exemplo, nosso professor de matemática muitas vezes perguntou qual o valor da variável x em dada equação. Quando usamos o termo "variável" em programação, nos referimos a um conceito diferente desse que estávamos acostumados. Uma variável é um espaço na memória RAM do computador, que usamos para depositarmos um valor. Com a variável, podemos recuperar o valor e usá-lo, por exemplo, em expressões aritméticas.

No exemplo abaixo estamos criando duas variáveis, de nomes `x` e `j`. Na variável `x` estamos colocando o valor `float` 10.0 e na variável `j`, o valor inteiro 16. Note que a variável em si não tem um tipo pré-determinado, mas o valor que está na variável, tem. O sinal de igual usado abaixo, é para a linguagem Python, uma atribuição de valores, ou seja, indica que o valor à direita está sendo armazenado na variável à esquerda.

```
>>> x = 10.0
>>> j = 16
>>>
```

Uma vez armazenado o valor na variável, esta pode ser utilizada em qualquer lugar que aquele valor poderia ser usado. Por exemplo, em expressões aritméticas, como as que vimos anteriormente. Abaixo vemos alguns exemplos. Vemos também que o valor da variável não é imutável. Ele pode ser alterado quantas vezes quisermos, utilizando outros comandos de atribuição. Além disso, o que está à direita do sinal de atribuição não precisa ser um número. Pode ser um valor calculado utilizando números e variáveis. E o tipo de valor armazenado na variável também pode ser mudado. Por exemplo, a variável `x` que inicialmente tinha um valor `float`, pode passar a armazenar um `int`.

```
>>> x = 10.0
>>> j = 16
>>> x
10.0
>>> j
16
>>> x + j
26.0
>>> x * j
160.0
>>> x = j ** 2
>>> x
256
>>> j = 2 * j
>>> j
32
>>>
```

Uma expressão particularmente interessante é a j = 2 * j. Pode parecer estranho estarmos definindo o valor da variável `j`, em termos dela mesmo. Mas basta pensar da seguinte forma, e tudo fica mais simples: na atribuição, o interpretador computa o valor da expressão à direita e depois coloca o resultado na variável à esquerda. Nesse caso, computa o valor de 2 *16 (que é o valor em `j`) e atribui 32 para a variável `j`. Se quiséssemos computar, por exemplo, o cubo de `j`, e atribuí-lo à própria variável `j`, sem usar exponenciação, poderíamos escrever a seguinte expressão:

```
>>> j = 16
>>> j = j * j * j
>>> j
4096
>>>
```

Note que, assim como em uma operação de divisão ou de exponenciação, por exemplo, a posição dos elementos em um comando de atribuição é importante. Ou seja, a variável à qual se deseja atribuir o valor fica à esquerda do sinal de atribuição e a expressão que computa esse valor fica à direita. O contrário não funciona.

```
>>> 2 * j = j
  File "<stdin>", line 1
SyntaxError: can't assign to operator
>>> 2 = j
  File "<stdin>", line 1
SyntaxError: can't assign to literal
>>>
```

A linguagem Python aceita nomes de variáveis bastante variados. Mas, em geral, nomes com letras, dígitos e *underscore* (caractere _) são os mais populares. Usando apenas esses três tipos de caracteres, a única restrição é que o nome não pode começar com um dígito. A tabela 8.1 mostra alguns exemplos de nomes válidos e não válidos. Caracteres especiais como @ e # e o espaço em branco não podem ser usados no nome de variáveis.

Tabela 8.1: Nomes de variáveis válidos e não válidos

Válidos	Não válidos
X1	1X
x1	1x
idade	#x1
peso	p@eso
endereco	tamanho pe
credito	
_idade	
tamanho_pe	

Outra restrição importante e que deve ser levada em conta, é que o nome da variável não poderá ser uma palavra reservada. Dentro de qualquer linguagem de programação, existem palavras que já têm funções pré-determinadas. A linguagem Python tem 29 palavras reservadas, e a Tabela 8.2 mostra quais são elas.

Tabela 8.2: Palavras reservadas dentro da linguagem Python

and	def	exec	if	not	return
assert	del	finally	import	or	try
break	elif	for	in	pass	while
class	else	from	is	print	yield
continue	except	global	lambda	raise	

Costumamos usar nomes de variáveis que indiquem o que vai ser armazenado nelas durante a execução do nosso programa. Por exemplo, se você vai armazenar a idade de uma pessoa, uma variável com nome `idade` seria recomendável. Outros exemplos seriam `peso`, `altura`, `saldo_conta`.

Importante observar que a definição de variáveis no Python diferencia letras maiúsculas de minúsculas, ou seja, a variável `peso` é diferente da variável `Peso` e `x` é diferente de `X`.

```
>>> peso = 65
>>> Peso = 80
>>> peso
65
>>> Peso
80
>>> x = 1003.9
>>> X
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'X' is not defined
>>>
```

Nesse exemplo, ao tentarmos usar a variável `X` temos um erro pois ela não existe. Defininimos apenas `x`, que é uma outra variável.

8.3.1 Exercícios

1. Utilize o comando ***type***, visto anteriormente, para verificar qual o tipo das variáveis que definimos nesta seção. Observe que ao mudar o tipo do valor atribuído à variável, o resultado do comando ***type*** também será alterado.

2. Defina uma variável que corresponda ao valor do raio de um círculo e calcule a área do círculo no interpretador Python.

3. No exercício 3 da Seção 8.2.1, aprendemos a calcular a média das notas de um aluno no interpretador Python. Agora vamos definir, separadamente, um valor para cada nota do aluno e calcular a média a partir das notas definidas. A execução deve ser similar a exemplificada abaixo:

```
>>> media = (nota1 + nota2 + nota3 + nota4)/4
>>> media
7.5
>>>
```

8.4 Strings

Muitas vezes em nossos programas estamos interessados em processar não apenas números. Um outro tipo de dado que existe na linguagem Python é a cadeia de caracteres, ou string. Como o nome sugere, um string é simplesmente uma sequência de caracteres como letras, dígitos, pontuação, ou qualquer outra coisa que você consiga digitar para o interpretador. Para delimitar o início e final de um string usamos um par de aspas (caractere ") ou um par de apóstrofo (caractere '). Usamos strings para representar, por exemplo, o nome de uma pessoa, um endereço, um código postal, uma frase, etc.

```
>>> "José da Silva"
'José da Silva'
>>> 'José da Silva'
'José da Silva'
>>> 'Rua das Rosas, no. 5'
'Rua das Rosas, no. 5'
>>> '13564-869'
'13564-869'
>>> 'Python para engenheiros é muito bom.'
'Python para engenheiros é muito bom.'
>>>
```

Um string, como um valor numérico, pode ser guardado em uma variável, usando o mesmo comando de atribuição que vimos anteriormente.

```
>>> nome = "José da Silva"
>>> nome
'José da Silva'
>>>
```

Embora não seja tão versátil quantos os números, existem algumas operações interessantes que podemos fazer com as strings. A primeira delas é a concatenação, ou seja, juntar dois strings, uma em seguida da outra, formando um novo string. Essa operação pode ser realizada com o símbolo de soma.

```
>>> nome = "José da Silva"
>>> endereco = 'Rua das Rosas, no. 5'
>>> nome + ' mora em: ' + endereco
'José da Silva mora em: Rua das Rosas, no. 5'
>>>
```

No exemplo acima, atribuímos uma string para a variável `nome`, um outro para a variável `endereco` e depois concatenamos o nome, com o string " mora em: " e com o endereco, obtendo um novo string, que poderia, inclusive ter sido atribuído a outra variável, conforme exibido no exemplo abaixo.

```
>>> nome = "José da Silva"
>>> endereco = 'Rua das Rosas, no. 5'
>>> frase = nome + ' mora em: ' + endereco
>>> frase
'José da Silva mora em: Rua das Rosas, no. 5'
>>>
```

Cada um dos caracteres da string ocupa uma posição. Cada posição é numerada, iniciando-se pela posição zero, que corresponde ao primeiro caractere. Então, no caso do string "José da Silva", a letra 'J' ocupa a posição 0, a letra 'o' a posição 1, o 's', a posição 2, e assim por diante. Podemos acessar cada um dos caracteres da seguinte maneira:

```
>>> nome = "José da Silva"
>>> nome[0]
'J'
>>> nome[1]
'o'
>>> nome[12]
'a'
>>> nome[13]
'Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range'
>>>
```

Ao fornecemos uma expressão como `nome[0]`, o que o interpretador faz é criar uma nova string que possui uma única letra. vemos, também, no exemplo

acima, que se tentarmos acessar uma posição da string que não existe, obtemos uma mensagem indicando o erro. Por outro lado, índices negativos são utilizados pelo interpretador para acessar as posições da string, do fim para o começo. Então, a posição -1 corresponde ao último caractere da string, -2 corresponde à penúltima, e assim por diante.

```
>>> nome = "José da Silva"
>>> nome[-1]
'a'
>>> nome[-2]
'v'
>>> nome[-13]
'J'
>>> nome[-14]
'Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range'
>>>
```

Usando essa mesma notação, podemos pegar parte do string ou, como costumamos dizer, um substring. Para isso devemos indicar onde começa e onde termina o substring que queremos. Por exemplo:

```
>>> nome = "José da Silva"
>>> nome[0:4]
'José'
>>> nome[5:12]
'da Silv'
>>> nome[5:13]
'da Silva'
>>> nome[5:14]
'da Silva'
>>>
```

Algumas coisas importantes devem ser notadas na sequência de comandos acima. Primeiro, que ao utilizarmos na nossa expressão [0:4] estamos dizendo que queremos criar um substring que inicia na posição 0 e termina antes da posição 4, ou seja, na posição 4. Por isso, embora o nosso string `nome` tenha 13 posições, numeradas de 0 até 12, quando solicitamos a substring de 5 a 12, cortamos a última letra. A segunda coisa a notar é que nesse caso não faz mal utilizarmos índices que não existem, como o 13 ou o 14. O interpretador não vai reclamar, mas vai usar apenas aquela parte do string que realmente existe,

no caso, até a posição 12.

Podemos usar também índices negativos para acessar substrings. A ideia é a mesma, já que um índice negativo também indica uma posição do string. Vejamos alguns exemplos.

```
>>> nome = "José da Silva"
>>> nome[0:-1]
'José da Silv'
>>> nome[0:-9]
'José'
>>> nome[-13:-9]
'José'
>>> nome[-8:14]
'da Silva'
>>> nome[-30:30]
'José da Silva'
>>>
```

Para indicar que queremos um substring a partir da posição inicial ou até a posição final, podemos deixar um dos ou ambos os valores vazios.

```
>>> nome = "José da Silva"
>>> nome[:3]
'Jos'
>>> nome[:-3]
'José da Si'
>>> nome[5:]
'da Silva'
>>> nome[-5:]
'da Silva'
>>> nome[:]
'José da Silva'
>>>
```

Na primeira expressão acima, indicamos que queremos a substring a partir da posição inicial, até a 3ª posição. Na segunda expressão, indicamos que queremos a partir da posição inicial, retirando apenas as 3 últimas posições. Na terceira expressão, indicamos que queremos a substring que inicie na 5ª posição e vá até o final. Na quarta expressão, indicamos que queremos as últimas 5 posições. E na última expressão, queremos uma substring que inicia na primeira posição e vai até a última, ou seja, estamos criando um cópia do string original.

Para encerrar, por ora, a apresentação de strings, vamos falar sobre a string

vazia. Esse é um caso especial, em que a string não possui nenhum caractere. Ele é representado por um abre e fecha aspas, sem nada entre elas. Uma das propriedades do string vazio é que ele não tem qualquer influência no resultado, quando utilizado em uma operação de concatenação.

```
>>> '' + 'José da Silva' + ''
'José da Silva'
>>> str('') + str('José da Silva') + str('')
'José da Silva'
>>>
```

8.4.1 Exercícios

1. Utilize o operador de soma para concatenar strings. Exemplo: se a primeira string definida for "Bom dia, " e a segunda string for "moçada !", então o retorno deve ser "Bom dia, moçada !". Note que o resultado da concatenação deve ser impresso no interpretador.

2. Python possibilita de uma maneira prática referir-se a subpartes de uma string. Defina a string s = 'strings em python' e execute os comandos abaixo no interpretador Python para praticar e fixar os conceitos anteriormente apresentados.

s	t	r	i	n	g	s		e	m		p	y	t	h	o	n
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
-17	-16	-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

>>> s[1:4] – o string copiado inicia-se no índice 1 e vai até o índice 4, porém, o índice 4 não é incluído.

>>> s[1:] – o string copiado inicia-se no índice 1. Como o índice final foi omitido, o string é copiado até o ultimo carácter.

>>> s[:] – ao omitir os dois lados será produzida uma cópia igual ao string original.

>>> s[1:100] – se o índice final for maior que o tamanho do string, o mesmo será truncado ao tamanho limite do string.

>>> s[-1] – copia apenas o último carácter (o primeiro a partir do final).

>>> s[-4] – copia o quarto carácter a partir do final.

>>> s[:-3] – copia o string, porém omite os 3 últimos caracteres.

>>> s[-3:] – realiza a cópia contando do terceiro caractere do final do string até o final da mesma.

8.5 Funções

Funções são trechos de programas que servem para realizar alguma tarefa ou, em particular, calcular algum valor. Por exemplo, funções que conhecemos e que

já existem, como parte da linguagem Python, são as trigonométricas como seno, cosseno, tangente etc. A maioria das funções requer que forneçamos algum valor ou alguns valores para que seja calculado o resultado da função para aqueles valores. Por exemplo, no caso das trigonométricas devemos fornecer o valor do ângulo, em radianos, e a função computa o resultado correspondente. Aos valores que fornecemos à funções damos o nome de parâmetros ou argumentos.

Já utilizamos anteriormente, sem saber, a função `type` para nos mostrar qual o tipo de um determinado valor. Passamos como parâmetro um valor qualquer e a função retorna como resultado o tipo daquele valor.

Algumas das funções que queremos conhecer neste capítulo, embora façam parte da biblioteca padrão de Python, estão localizadas em "módulos" específicos, que servem para organizar as funções por domínios. Por exemplo, funções matemáticas estão organizadas em módulo chamado `math`. Para podermos utilizar funções de um módulo precisamos avisar antecipadamente o interpretador Python. Para isso, podemos importar todas as funções de um módulo utilizando o comando "import [nome do módulo]".

```
>>> math.sin(3.14/4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'math' is not defined
>>> import math
>>> math.sin(3.14/4)
0.706825181105366
>>>
```

Como a função seno (`sin`, seguindo o nome em inglês) está no módulo `math` precisamos fazer o *import* do módulo antes de usá-la e, depois, temos que nos referir a ela com o seu nome completo: `math.sin`.

Podemos também importar apenas um função de um determinado módulo, através do comando "from [nome do módulo] import [nome da função]".

```
>>> sin(3.14/4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sin' is not defined
>>> from math import sin
>>> sin(3.14/4)
0.706825181105366
>>>
```

A forma de passar parâmetros para as funções é utilizando os parênteses.

No caso do seno, dentro dos parênteses passamos o valor do ângulo do qual queremos computar o seno. A função retorna o valor corresponde. Podemos usar as funções no meio de expressões, de maneira semelhante à que fizemos com as variáveis.

```
>>> math.sin(3.14/4) ** 2 + math.cos(3.14/4) ** 2
1.0
>>> math.sin(math.pi/4) ** 2 + math.cos(math.pi/4) ** 2
1.0
>>>
```

No exemplo acima computamos a conhecida expressão $\sin^2(x) + \cos^2(x)$, cujo resultado sabemos é 1.0. Na segunda expressão utilizamos uma constante, definida no módulo `math` que nos dá o valor de π com uma certa precisão. Se quisermos ver qual o valor de `math.pi` podemos utilizar o interpretador.

```
>>> math.pi
3.141592653589793
>>>
```

Algumas outras funções do módulos `math` são apresentadas na Tabela 8.5. Quando a função requer mais do que um parâmetro, eles devem ser separados por vírgula, dentro dos parênteses. É o caso, por exemplo, do máximo divisor comum, que requer dois números como parâmetros. Utilizamos `gcd(a,b)`.

Funções em Python podem ter um número variável de parâmetros. Dependendo de quantos parâmetros passamos, mudo o comportamento da função. Um exemplo disso é a função `log`. Essa função pode receber apenas um parâmetro. Nesse caso, `log(x)` retorna o logaritmo natural (ln) de `x`. Se for usado um segundo parâmetro, `log(x, b)` a função retorna $\log_b x$.

```
>>> math.log(10)
2.302585092994046
>>> math.log(10,10)
1.0
>>>
```

As funções do módulo `math` obedecem os domínios definidos para as funções matemáticas e cada uma retorna valores de um determinado tipo. Por exem-

Tabela 8.3: Funções matemáticas do módulo `math`

Nome	Significado	Exemplo
ceil(x)	Teto de x	math.ceil(3.1) = 4 math.ceil(−3.1) = -3
cos(x)	Coseno de x	math.cos(1.5) = 0.0707372016677029
fabs(x)	Valor absoluto de x	math.fabs(−3.1) = 3.1
factorial(x)	Fatorial de x	math.factorial(7.0) = 5040
floor(x)	Piso de x	math.floor(3.1) = 3 math.floor(−3.1) = −4
gcd(x,y)	Máximo divisor comum de x e y	math.gcd(48,184) = 8
log2(x)	Logaritmo de x na base 2	math.log2(4050) = 11.98370619265935
log10(x)	Logaritmo de x na base 10	math.log10(4050) = 3.6074550232146683
pow(x, y)	x elevado a y. x^y	math.pow(4050,−12.3) = 4.249161271145364e-45
sin(x)	Seno de x	math.sin(1.5) = 0.9974949866040544
sqrt(x)	raiz quadrada de x	math.sqrt(40.5) = 6.363961030678928
tan(x)	Tangente de x	math.tan(1.5) = 14.101419947171719

plo, a função `sqrt` retorna sempre um número do tipo `float` e recebe como parâmetro um número não negativo, `int` ou `float`, não importa [3]. Se tentarmos aplicar a função a um número negativo, o interpretador acusa um erro. O mesmo acontece se tentarmos passar um número negativo ou que não seja "integral" (integral aqui quer dizer inteiro ou um `float` com a parte decimal igual a zero) para a a função `factorial`. Essa função retorna sempre um número do tipo `int`.

```
>>> math.sqrt(-2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
>>> math.factorial(3.0)
6
>>> math.factorial(3.1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: factorial() only accepts integral values
>>>
```

[3]Na verdade, um `int` passado como parâmetro é convertido para ponto flutuante (`float`).

Além das funções, o módulo `math` possui algumas constantes, além da constante `math.pi`, que já mencionamos. São elas:

1. `math.e`: Número de Euler. No interpretador Python assume o valor 2.718281828459045;
2. `math.tau`: Corresponde a $2 \times \pi$. Assume o valor 6.283185307179586 [4];
3. `math.inf`: Corresponde ao conceito de $+\infty$.

Algumas funções da biblioteca padrão são chamadas "built in". Elas não estão em nenhum módulo específico e por isso não requerem um comando import. Elas implementam funcionalidades variadas como, por exemplo, a função `len`, que retorna o tamanho de um string, ou seja, o número de caracteres do string.

```
>>> len('José' + ' da ' + 'Silva')
13
>>>
```

Outras funções nessa categoria servem para transformar um determinado valor, de um tipo para outro tipo. Por exemplo, se temos um valor `int` e queremos obter uma representação desse valor no formato de um string, utilizamos a função `str`.

```
>>> idade = 36
>>> 'A idade daquele senhor é ' + idade
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
>>> 'A idade daquele senhor é ' + str(idade)
'A idade daquele senhor é 36'
>>>
```

Note no exemplo acima que temos uma variável inteira e que precisa ser concatenada com um string, para formar a frase que indica a idade de alguém. Mas, não podemos fazer essa concatenação diretamente. Só podemos concatenar um string com outro string. Por isso, na segunda expressão, invocamos a função `str`, que retorna um string que, então pode ser concatenado. As funções `int` e `float` convertem valores de outros tipos para `int` e `float`, respectivamente.

[4] Na verdade, até a versão 3.5 esta constante não existe no módulo `math`. Deve ser incorporada na versão 3.6.

```
>>> int(3.3333)
3
>>> int('3.333')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10:
'3.33333'
>>> int('33')
33
>>> float('3.3333')
3.3333
>>> float(33)
33.0
>>>
```

Algumas funções em Python são associadas com um tipo específico e, por isso, são chamadas de um maneira um pouco diferente. É o caso de funções associadas ao tipo string, por exemplo `islower` e `isupper`, que verificam se um string é formado apenas por letras minúsculas ou maiúsculas, respectivamente. No quadro abaixo vemos alguns exemplos de como chamar essas funções. Utiliza-se o objeto sobre o qual se deseja aplicar a função (no caso um string) seguido de um "." e depois o nome da função. Consultando a documentação da linguagem o leitor identifica como uma função deve ser invocada.

```
>>> 'abc'.islower()
True
>>> c = 'abc'
>>> c.islower()
True
>>> c.isupper()
False
>>> 'ABC'.isupper()
True
>>>
```

O leitor deve acostumar-se a consultar a documentação sobre a linguagem e as funções da biblioteca padrão de Python. Ela pode ser encontrada em `https://docs.python.org/3/library/index.html`. As funções *built-in* são descritas em `https://docs.python.org/3/library/functions.html` e as matemáticas em `https://docs.python.org/3/library/math.html`.

Além de consultar a documentação da linguagem de programação, alguns ambientes de programação fornecem facilidades no momento de procurar por

uma função dentro da biblioteca. No IDLE, após ter importado um módulo, basta digitar o nome do módulo seguido por um "." e teclar `<tab>`, que a interface exibe quais são as funções daquele módulo. A Figura 8.2 exibe a utilização desse recurso.

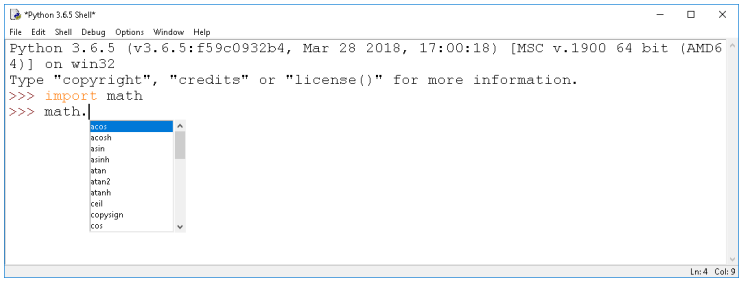

Figura 8.2: Exibição das funções do módulo `Math` no IDLE

No interpretador Python no Linux, após ter importado um módulo, basta digitar o nome do módulo seguido por um "." e teclar duas vezes `<tab>`, que serão exibidas as funções daquele módulo. A Figura 8.3 exibe a utilização desse recurso.

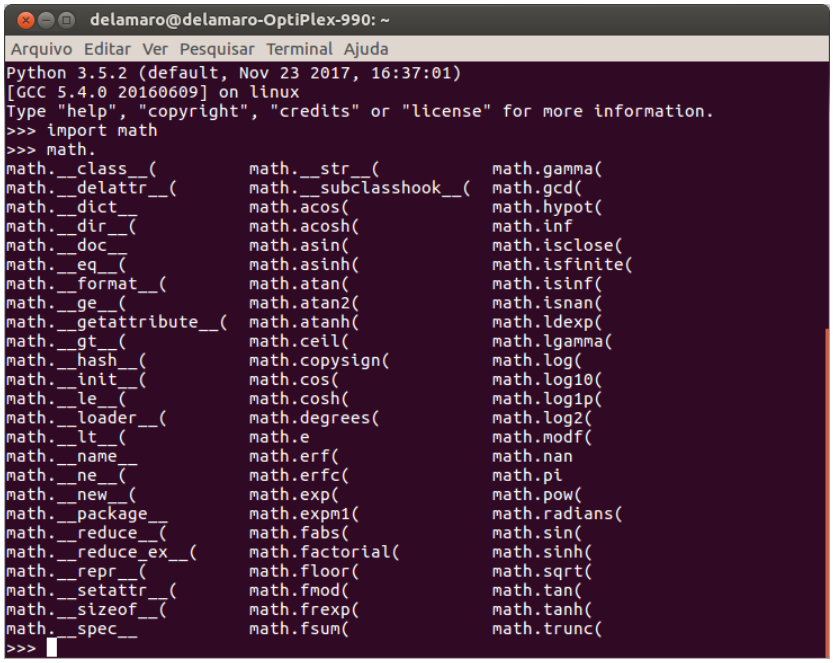

Figura 8.3: Exibição das funções do módulo `Math` no interpretador Python

8.5.1 Exercícios

1. Após aprender sobre strings e utilização de funções, defina uma variável que armazene seu nome e depois, exiba o número de caracteres contidos

dentro daquela variável.

2. Defina um número qualquer maior que 0 no interpretador Python, exiba este número, seu sucessor, seu antecessor, sua raiz quadrada, a raiz quadrada do seu sucessor e a raiz quadrada do seu antecessor. Exemplo: Se o número definido for 5, o retorno deve ser: "Numero escolhido: 5. Sucessor: 6. Antecessor: 4. Raiz: 2.23606797749979. Raiz Sucessor: 2.449489742783178. Raiz Antecessor: 2.0"

8.6 Aplicação: o método de Bhaskara

Com o que vimos até agora, podemos utilizar a nossa super calculadora para computar as raízes de uma equação de segundo grau, conforme discutimos no Capítulo 1. vamos calcular a solução para a equação $308x^2 + 113x - 1033 = 0$.

Iniciamos seguindo o primeiro passo do algoritmo e atribuindo os valores dos coeficientes para as variáveis `a`, `b` e `c`, no interpretador Python.

```
>>> a = 308
>>> b = 113
>>> c = -1033
>>>
```

Como o valor de `a` é diferente de zero, trata-se de uma equação do segundo grau e podemos então continuar para o próximo passo, que é computar o valor de Δ. Seguindo as restrições que temos para batizar as variáveis em Python, vamos definir uma variável `delta` que vai guardar o valor do discriminante.

```
>>> delta = b ** 2 - 4 * a * c
>>> delta
1285425
>>>
```

Verificamos o valor da variável `delta`, para termos certeza que não é negativa. Como obtivemos o valor 1285425, podemos continuar e computar os valores de x_1 e x_2, que chamaremos de `x1` e `x2`, respectivamente.

```
>>> x1 = (-b + math.sqrt(delta)) / (2 * a)
>>> x2 = (-b - math.sqrt(delta)) / (2 * a)
>>> x1
1.6570874169509975
>>> x2
-2.0239705338341145
>>>
```

A solução apresenta duas raízes distintas, $X_1 = 1,6570874169509975$ e $x_2 = -2,0239705338341145$.

8.6.1 Exercícios

1. Repita o algoritmo de Bhaskara no interpretador Python, para outras equações como:

   a) $20x^2 + 214x + 572,45 = 0$

   b) $211x^2 - 14x + 103 = 0$

   c) $211x^2 - 103 = 0$

   d) $211x^2 - 14x = 0$

   e) $211x^2 = 0$

8.7 Aplicação: o método da bisseção

Podemos utilizar o interpretador Python também para computar as raízes de uma função contínua, usando o método da bisseção, embora seja um tanto trabalhoso aplicar passo a passo todas as iterações. Veremos mais adiante como isso pode ser automatizado.

Vamos utilizar o mesmo exemplo do Capítulo 2.1, da primeira parte deste livro. A função que queremos achar as raízes é: $f(x) = x^3 - x^2 - 13x + 8$. Para facilitar um pouquinho o nosso trabalho, vamos ver um recurso de Python chamado de "expressão lambda". Com esse recurso podemos armazenar em uma variável uma expressão qualquer e depois reutilizá-la sem ter que digitar novamente a expressão toda.

```
>>> f = lambda x: x ** 3 - x ** 2 - 13 * x + 8
>>> f(-4.0)
-20.0
>>> f(-3.5)
-1.625
>>>
```

A primeira linha está dizendo que criamos uma expressão que tem um parâmetro `x`, como os parâmetros das funções matemáticas que vimos. Essa expressão é armazenada na variável `f` que pode, então ser usada exatamente como usamos as funções. Ao utilizamos, por exemplo, `f(-3.5)`, o interpretar irá substituir o valor de `x` na expressão pelo valor -3.5 e computar o resultado de $3,5^3 - 3,5^2 - 13 \times 3,5 + 8$.

Podemos ter expressões lambda com mais do que um parâmetro, e elas podem ser aplicadas a qualquer tipo de valor, desde que as operações definidas na expressão sejam válidas para esses valores. No exemplo abaixo, definimos uma nova função que simplesmente soma dois valores. Se esses valores forem números, o resultado é a soma deles. Se forem strings, o resultado é a sua concatenação. Já a função `f` que definimos anteriormente não pode ser aplicada a ums string pois nela existem operações que não são aplicáveis a strings.

```
>>> s = = lambda x,y: x + y
>>> s(3,5.6)
8.6
>>> s('abc','def')
'abcdef'
>>> f('abc')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 1, in <lambda>
TypeError: unsupported operand type(s) for ** or pow(): 'str'
and 'int'
>>>
```

Uma vez definida a função que vamos utilizar, iniciamos o processo de busca pela raiz. Na primeira iteração, inicializamos as variáveis `a` e `b` com o intervalo de interesse, que é $(-4, -3)$. Em seguida, calculamos o ponto médio `c`, o tamanho do intervalo restante e por fim verificamos o sinal de `f(c)`.

```
>>> a = -4
>>> b = -3
>>> c = (a+b)/2
>>> (b-a)/2
0.5
>>> f(a) * f(c)
32.5
>>>
```

O quarto comando do interpretador acima serve para verificarmos o tama-

nho intervalo, depois de acharmos o ponto médio. Se ele for menor do que a tolerância desejada, podemos aceitar `c` como a resposta que procuramos. Caso contrário, continuamos o processo. No último comando estamos verificando como continuar. Se o resultado da multiplicação for zero, isso indica que `f(c) = 0` e portanto `c` é raiz exata da função. Como esse valor foi positivo, isso indica que `f(a)` e `f(c)` têm o mesmo sinal e portanto devemos continuar a busca no outro subintervalo, (c, b). Para isso, basta atribuirmos à variável `a` o valor da variável `c` e repetirmos o processo.

```
>>> a = c
>>> c = (a+b)/2
>>> (b-a)/2
0.25
>>> f(a) * f(c)
-8.708984375
>>>
```

O erro ainda não chegou no valor que desejamos (0.001) e dessa vez, o próximo intervalo a ser utilizado será (a, c), pois a multiplicação de `f(a)` e `f(c)` foi negativa. Então, para o próximo passo, atribuímos à variável `b` o valor de `c`.

```
>>> b = c
>>> c = (a+b)/2
>>> (b-a)/2
0.125
>>> f(a) * f(c)
-3.316650390625
>>>
```

A Figura 8.7 mostra os demais passos, até chegarmos ao valor aproximado da raiz da função. O valor, como esperado, é aquele que calculamos na primeira parte do livro: -3.4462890625. Notamos que no final da figura, quando o tamanho do intervalo de busca é menor do que 0.001 podemos parar de procurar e o resultado desejado é o valor da variável `c`.

Como já mencionamos, é um trabalho exaustivo aplicar o método de forma manual, como fizemos aqui. Embora não exija muito esforço intelectual, o trabalho manual realizado é demorado, cansativo e, ainda por cima, sujeito a erros. Por isso, precisamos aprender a escrever programas que automatizem a execução dos algoritmos que estamos estudando. Ao fazer isso temos um esforço intelectual que é o de criar um algoritmo para resolver o problema ou, em muitos casos, entender um algoritmo que já existe e, depois, traduzir para uma linguagem de programação aqueles passos que o algoritmo estabelece. Feito isso, economizamos um grande esforço manual e muito tempo.

Para que possamos automatizar essas tarefas, a linguagem de programação Python oferece vários recursos que permitem, por exemplo, escolher qual comando executar em uma dada situação (no exemplo, devo atribuir o valor de `c` para a variável `a` ou para `b`?). Ou comandos que permitem repetir várias vezes algumas ações, também como fizemos no exemplo, até que o valor do erro fosse menor que um determinado valor.

Nos próximos capítulos vamos tentar ensinar um pouco mais sobre esses comandos e mostra como usá-los nos problemas que já vimos até agora e em alguns outros.

8.8 Exercícios

1. Use o interpretador Python para computar as outras duas raízes da função $f(x) = x^3 - x^2 - 13x + 8$, com erro inferior a 0.001.

2. Use o interpretador Python para computar as raízes da função $f(x) = x^3 - x^2 - 13x + 8$, usando o método de Newton-Raphson, com erro inferior a 10^{-13}.

3. Compute uma raiz da função $f(x) = sen^2(x) - cos(x)$, com erro inferior a 10^{-13}.

4. Adicione na execução dos algoritmos anteriores uma variável `i` que deve registrar quantas iterações foram executadas. Ou seja, na primeira iteração essa variável deve receber o valor 1 e, a cada iteração, deve ser incrementada de uma unidade. Desse modo, no final da execução, podemos consultar o valor dessa variável para sabermos quantas iterações foram necessárias.

```
>>> b = c
>>> c = (a+b)/2
>>> (b-a)/2
0.0625
>>> f(a) * f(c)
-0.409820556640625
>>> b = c
>>> c = (a+b)/2
>>> (b-a)/2
0.03125
>>> f(a) * f(c)
1.0973014831542969
>>> a = c
>>> c = (a+b)/2
>>> (b-a)/2
0.015625
>>> f(a) * f(c)
0.1409673219313845
>>> a = c
>>> c = (a+b)/2
>>> (b-a)/2
0.0078125
>>> f(a) * f(c)
-0.0046784776259301
>>> b = c
>>> c = (a+b)/2
>>> (b-a)/2
0.00390625
>>> f(a) * f(c)
0.019414838510556365
>>> a = c
>>> c = (a+b)/2
>>> (b-a)/2
0.001953125
>>> f(a) * f(c)
0.0032784581515694633
>>> a = c
>>> c = (a+b)/2
>>> (b-a)/2
0.0009765625
>>> c
-3.4462890625
>>>
```

Figura 8.4: Passos finais para computar uma das raízes de $f(x) = x^3 - x^2 - 13x + 8$

Capítulo 9

Programando de verdade

Até agora, o processo que usamos para resolver nossos problemas foi bastante manual. Apenas utilizamos o interpretador Python para fazer umas contas e guardar alguns valores em variáveis. O que queremos é uma forma de fornecer os comandos para o interpretador e deixar por conta dele a execução.

Isso é muito fácil de fazer. Nós precisamos colocar a sequência de comandos que deve ser executada pelo interpretador em um arquivo texto e depois instruir o interpretador a executar aquele arquivo, ou mais precisamente, os comandos que estão listados naquele arquivo.

Por exemplo, suponha que colocamos os comandos do algoritmos de Bhaskara em um arquivo com nome *bhaskara.py*. Costumamos usar essa extensão para os programas Python, para, quando acharmos um arquivo com essa extensão, sabermos do que se trata. Para executar os comandos do arquivo devemos, no *prompt* do Windows ou no console do Linux, invocar o seguinte comando:

```
>> python bhaskara.py
```

A partir daí, o interpretador Python é carregado e passa a executar os comandos que estão no arquivo. Finalizada a execução dos comandos do arquivo, o interpretador também termina sua execução e o controle volta para o console do usuário. Para quem usa algum dos ambientes de programação mencionados no Apêndice A, não é necessário executar um console para invocar o interpretador e executar o programa. Dentro do próprio ambiente de programação existe uma opção para executar o programa.

É importante notar que, assim como acontece quando interagimos diretamente com o interpretador Python, ao executar o programa do arquivo é possível que exista algum erro nos comandos ali armazenados. Nesse caso, o interpretador pára a execução do programa, avisa ao programador qual o erro encontrado e qual a linha do programa onde ele ocorreu. Por exemplo, se esquecermos de incluir o comando `import math` no nosso programa, teremos o seguinte resultado:

```
>> python bhaskara.py
Traceback (most recent call last):
  File "bhaskara.py", line 8, in <module>
    x1 = (-b + math.sqrt(delta)) / (2 * a)
NameError: name 'math' is not defined

>>
```

9.1 Comando de saída

Vamos então escrever no arquivo *bhaskara.py* os comandos que usamos para computar as raízes da nossa função $f(x) = 308x^2 + 113x - 1033$. Programa 9.1.

Programa 9.1 Versão inicial do método de Bhaskara

```
1  import math
2
3  a = 308
4  b = 113
5  c = -1033
6
7  delta = b ** 2 - 4 * a * c
8
9  x1 = (-b + math.sqrt(delta)) / (2 * a)
10 x2 = (-b - math.sqrt(delta)) / (2 * a)
11
12 x1
13 x2
```

Porém, ao executarmos esse programa, conforme definimos anteriormente, nada acontece, ou seja, os valores de `x1` e `x2` não são apresentados, como gostaríamos que acontecesse. Isso ocorre porque, ao executarmos o programa armazenado em um arquivo precisamos incluir comandos que explicitamente indiquem que queremos apresentar algum valor "na saída" do programa, ou seja, no console. Para fazermos isso vamos usar a função *print*.

Essa função recebe zero, um ou mais parâmetros e como resultado, mostra o valor de cada um dos parâmetros, todos seguidos em uma única linha. Se for chamada sem nenhum parâmetro, a função simplesmente escreve uma "linha vazia", ou seja, não escreve nada mas o próximo *print* do programa ira ocupar uma nova linha. Então, se trocarmos os dois últimos comandos do Programa 9.1 por:

```
1 print(x1)
2 print(x2)
```

teremos como resultado os valores armazenados nas variáveis `x1` e `x2`.

```
>> python bhaskara.py
1.6570874169509975
-2.0239705338341145
>>
```

Podemos melhorar um pouco essa saída, fornecida por nosso programa. Já que a função *print* permite que passemos vários valores seguidos, podemos deixar a saída mais bonita ou mais explicativa com os comandos:

```
1 print("O valor da 1a raiz é ", x1)
2 print("O valor da 2a raiz é ", x2)
```

Esses dois comandos produzem como resultado do programa a seguinte saída:

```
>> python bhaskara.py
O valor da 1a raiz é 1.6570874169509975
O valor da 2a raiz é -2.0239705338341145
>>
```

Uma outra forma de obter o mesmo resultado é concatenando vários objetos para formar um único string dentro do *print*. Nesse caso, para fazer a concatenação precisamos converter todos os objetos para string. O programa, então, ficaria como mostrado a seguir.

Programa 9.2 Segunda versão do método de Bhaskara

```
1  import math
2
3  a = 308
4  b = 113
5  c = -1033
6
7  delta = b ** 2 - 4 * a * c
8
9  x1 = (-b + math.sqrt(delta)) / (2 * a)
10 x2 = (-b - math.sqrt(delta)) / (2 * a)
11
12 print("O valor da 1a raiz é " + str(x1))
13 print("O valor da 2a raiz é " + str(x2))
```

9.1.1 Exercícios

1. Modifique o arquivo *bhaskara.py* para que ele resolva as outras equações que vimos no capítulo anterior:

   a) $20x^2 + 214x + 572,45 = 0$

   b) $211x^2 - 14x + 103 = 0$

   c) $211x^2 - 103 = 0$

   d) $211x^2 - 14x = 0$

   e) $211x^2 = 0$

9.2 Comando de entrada

Se você resolveu os exercícios da seção anterior, deve ter notado um grande inconveniente no nosso programa. Acontece que para cada equação que queremos resolver precisamos alterar os primeiros comandos do arquivo *bhaskara.py*, nos quais os coeficientes são atribuídos às variáveis `a`, `b` e `c`. Seria mais interessante se tivéssemos um único programa para resolver qualquer equação do segundo grau, sem que precisássemos alterá-lo, para cada equação distinta.

A função *input()* é utilizada para fazer com que o usuário interaja com o seu programa, fornecendo a ele um valor. Esse valor pode, então, ser utilizado nos cálculos dentro do programa. Crie um arquivo com os seguintes comandos e verifique o seu resultado.

Programa 9.3 Uso da função *input*

```
1 s = input()
2 print(s)
```

O que acontece ao executar a primeira linha, é que o programa pára sua execução e fica esperando você digitar um texto qualquer, até terminar com um `<enter>`. O valor que você digitar é atribuído à variável `s` e depois exibida de volta para você, por meio da função *print*.

```
>> python teste_input.py
Vou digitar um texto
Vou digitar um texto
>>
```

A primeira ocorrência de "Vou digitar um texto", acima, corresponde ao texto que vai aparecendo à medida que você digita, até pressionar a tecla `<enter>`. A segunda ocorrência corresponde à execução do *print* com o valor da variável `s`.

Para facilitar um pouco, é possível associar à função *input*, uma mensagem que aparece para o usuário, para que ele saiba que o programa está esperando a

digitação de um valor. No exemplo abaixo, é mostrada uma mensagem "Digite seu nome" e então o programa espera a digitação de um texto, que é atribuído à variável `s`.

Programa 9.4 Uso da função *input*

```
1 s = input("Digite seu nome: ")
2 print(s)
```

Veja a execução:

```
>> python teste_input.py
Digite seu nome: José da Silva
José da Silva
>>
```

Podemos substituir agora os primeiros comandos do programa *bhaskara.py* pelo uso da função *input*. Só precisamos tomar cuidado pois os valores atribuídos às variáveis `a`, `b` e `c` devem ser do tipo `float` para que elas possam ser usadas em cálculos matemáticos mas a função *input* retorna um string. Então, precisamos converter os valores digitados para um número, antes de atribuí-los às variáveis.

Programa 9.5 Programa *bhaskara.py* com a função *input*

```
1  import math
2
3  s = input('Digite o valor de a: ')
4  a = float(s)
5  s = input('Digite o valor de b: ')
6  b = float(s)
7  s = input('Digite o valor de c: ')
8  c = float(s)
9
10
11 delta = b ** 2 - 4 * a * c
12
13 x1 = (-b + math.sqrt(delta)) / (2 * a)
14 x2 = (-b - math.sqrt(delta)) / (2 * a)
15
16 print("O valor da 1a raiz é ", x1)
17 print("O valor da 2a raiz é ", x2)
```

Executando esse programa, podemos digitar quaisque valores para os coeficientes da equação e, assim, não precisamos alterar o programa a cada execução. Se quisermos, por exemplo resolver a equação $20x^2 + 214x + 572,45 = 0$.

```
>> python bhaskara.py
Digite o valor de a: 20
Digite o valor de b: 214
Digite o valor de c: 572.45
O valor da 1a raiz é -5.35
O valor da 2a raiz é -5.35
>>
```

9.2.1 Exercícios

1. Use o programa desta seção para resolver as outras equações quadráticas que vimos anteriormente. O que acontece se a equação que você escolheu não for quadrática ($a = 0$) ou não possuir solução real ($\Delta < 0$)?

2. O que acontece se você digitar, ao ser solicitado o valor de uma das variáveis, alguma coisa que não seja um número, por exemplo: "abcd"?

9.3 Indentação e comentários

Você deve ter notado que os comandos no nosso programa (arquivo *.py*) possui alguma linha em branco, que servem para agrupar alguns comandos e separar outros. Por exemplo, o `import` está separado dos comandos de leitura das variáveis, que estão agrupados e por sua vez separados do cálculo do Δ e assim por diante. Nesse sentido, não existe restrição quanto ao uso de linhas em branco. Por outro lado, todas as linhas precisam começar sem nenhuma indentação, ou seja, na primeira coluna.

Veremos mais adiante que o Python usa a indentação das linhas para identificar a estrutura do programa. Além disso, mais adiante, quando for usar indentação, não misture espaços em branco com "tabs". Isso confunde o interpretador, que vai reclamar e não vai executar o seu programa.

Um outro aspecto importante na organização dos nossos programas é o uso de comentários. Um comentário dentro do seu programa não é executado pelo interpretador. Ele serve somente para que você documente o seu programa e quando você ou outra pessoa precisar ler e entender o que ele faz, a tarefa seja facilitada. Uma linha de comentário em um programa Python inicia com o símbolo "#" e a linha inteira depois dele é ignorada.

Não existe qualquer requisito de indentação em relação aos comentários, que podem, também, aparecer no final de uma linha, depois dos comandos. Como exemplo, veja em seguida como podemos adicionar comentários no nosso programa de resolução de equações quadráticas.

Programa 9.6 Comentários no programa *bhaskara.py*

```
1  # importa as funções matemáticas
2  import math
3
4  # inicializa as variáveis, lendo do teclado o valor
5  # dos coeficientes
6  s = input('Digite o valor de a: ')
7  a = float(s)
8  s = input('Digite o valor de b: ')
9  b = float(s)
10 s = input('Digite o valor de c: ')
11 c = float(s)
12
13 # aplica o método de Bhaskara
14 delta = b ** 2 - 4 * a * c # calcula o delta
15
16 # computa o valor das raízes
17 x1 = (-b + math.sqrt(delta)) / (2 * a)
18 x2 = (-b - math.sqrt(delta)) / (2 * a)
19
20 # exibe o resultado
21 print("O valor da 1a raiz é ", x1)
22 print("O valor da 2a raiz é ", x2)
```

Mais um detalhe importante em relação à formatação do nosso programa: se tivermos uma linha muito grande, ou se por qualquer outro motivo quisermos quebrar um comando em mais do que uma linha podemos usar o caractere \ no final da linha para indicar que o comando continua na linha seguinte. Na linha subsequente, não há restrição quanto à indentação pois essa linha é apenas uma continuação da linha anterior. Por exemplo, vamos quebrar as duas atribuições de `x1` e `x2`, em dois pontos diferentes do comando.

Programa 9.7 Quebrando uma linha do programa

```
1 # computa o valor das raízes
2 x1 = (-b + math.sqrt(delta)) / \
3      (2 * a)
4 x2 = (-b - math.sqrt(delta)) \
5      / (2 * a)
```

9.4 Exercícios

1. Faça um programa que leia duas notas de um aluno, calcule a média simples do aluno, e depois mostre como saída a média calculada.

2. Faça um programa que leia três notas de um aluno, onde a primeira e segunda nota possuem peso um e a terceira nota possui peso dois. Calcule a média ponderada destas notas e depois mostre na saída o resultado

3. Faça um programa que leia uma temperatura fornecida em graus Celsius e converta para graus Fahrenheit, exibindo o resultado na saída.

4. Faça um programa que leia o valor da hora de trabalho (em reais) e número de horas trabalhadas no mês, e exiba na tela o valor a ser pago ao funcionário, adicionando 10% sobre o valor calculado.

5. O valor do seno de x pode ser calculado pela série de Taylor, dada por:

$$sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!}...$$

   Faça um programa que leia o valor de x e compute o valor do seno usando os 5 primeiros termos desta série. Mostre o valor calculado e o valor fornecido pela função padrão do Python, $sin(x)$.

6. Escreva um programa para computar uma raiz da função $f(x) = x^3 - x^2 - 13x + 8$ usando 10 iterações do método de Newton-Raphson. Use um comando de entrada para ler o chute inicial.

7. Você conseguiria escrever um programa para implementar o método da bisseção, com 10 iterações? Qual seria o maior problema para fazer isso?

9.5 Formatação da saída

O comando `print` é relativamente limitado. Ele simplesmente mostra na saída os valores que são passados como parâmetros. Muitas vezes queremos dar uma forma mais agradável de visualização às saídas produzidas por nossos programas. É o caso, por exemplo, do programa *bhaskara.py* que poderia produzir uma saída com apenas 4 casas decimais. Ou então, uma saída que represente valores monetários, poderia, também ajustar a saída para esse padrão (R$10.00 por exemplo).

A linguagem Python fornece uma função `format` que serve para formatar um string, da maneira que desejarmos. Esse string pode, então, ser usado em um comando `print`. Para entender, vamos tomar como exemplo o programa para o método de Bhaskara, no qual queremos apresentar a como saída as duas raízes calculadas. Com o `format` podemos usar o seguinte comando:

```
1 print("A 1a raiz é {} e a 2a é {}".format(x1,x2))
```

O string usado na formatação é o que está antes do ponto (em vermelho), na chamada da função `format` e os parâmetros são os valores de `x1` e `x2`. Note que no string de formatação existem dois "campos" representados por `{}`. Esses campos vão ser substituídos pelos valores dos parâmetros passados para a função `format`, ou seja, os valores de `x1` e `x2`. Então, o valor do string que vai ser mostrado pelo comando `print` é:

```
A 1a raiz é 1.6570874169509975 e a 2a é -2.0239705338341145
```

Mas esse resultado poderíamos ter obtido apenas com algumas concatenações de strings. Porém, o string de formatação permite que adicionemos, dentro dos campos substituíveis, algumas instruções de como os parâmetros vão ser formatados. No exemplo a seguir, formatamos as saídas com quatro casas decimais. O ":" é obrigatório e indica o ínicio da formatação. A letra `f` indica que estamos formatando um valor `float` e o ".4" indica exatamente que queremos quatro casas decimais.

```
1 "A 1a raiz é {:.4f} e a 2a é {:.4f}".format(x1,x2)
```

O string produzido é mostrada a seguir. Note que na formatação os números não são apenas truncados na quarta casa decimal mas sim arredondados.

```
A 1a raiz é 1.6571 e a 2a é -2.0240
```

Podemos, também, estabelecer qual o tamanho total que cada campo vai ocupar no string final. Por exemplo, vamos estabelecer que queremos que cada valor ocupe nove espaços na saída, sendo quatro deles casas decimais. Além disso, o valor de `x1` vai ser ajustado à esquerda desses nove caracteres, por isso usamos o sinal "<". O valor de `x2` vai ser ajustado à direita, por isso usamos o sinal ">". Se quisermos centralizar o valor nesse espaço que determinamos, podemos usar "^".

```
1 "A 1a raiz é {:<9.4f} e a 2a é {:>9.4f}".format(x1,x2)
2 "A 1a raiz é {:^9.4f} e a 2a é {:^9.4f}".format(x1,x2)
```

As saídas produzidas nesses dois casos são:

```
A 1a raiz é 1.6571    e a 2a é   -2.0240
A 1a raiz é  1.6571   e a 2a é  -2.0240
```

Podemos usar as mesmas regras para formatar números inteiros, usando a letra "`d`" em vez do '`f`". Além disso, para inteiros não é permitido que se determine o número de casas decimais. O exemplo a seguir mostra vários números que são formatados todos com quatro espaços. Devemos reparar que quando um número não cabe no espaço reservado, como o último do exemplo que segue, não há nenhum truncamento no número. Ele simplesmente extrapola o espaço que deveria ocupar.

```
1 '{:>4d}{:>4d}{:>4d}{:>4d}'.format(12, 102, 1002, 10002)
```

No quadro a seguir vemos cada um dos números formatados e, na linha seguinte, uma indicação de onde termina cada um dos campos.

```
 12 102100210002
  |   |   |   |
```

A função `format` é bem mais complexa do que explicamos aqui. O que vimos aqui são algumas funcionalidades básicas mas que vão nos ajudar no restante deste livro. Para ter uma visão completa dessa funçao, o leitor deve consultar a documentação de Python em `https://docs.python.org/3/library/string.html#formatstrings`.

Uma outra forma muito comum de formatar a saída de um programa é utilizando a sequência "\n" dentro de um string em um comando `print`. Essa sequência faz com que haja uma quebra de linha, ou seja, a parte que vem depois dela no string é apresentada em uma nova linha na saída. Por exemplo,

```
1 print('x1 = {:9.4f}\nx2={:9.4f}'.format(x1,x2))
```

vai produzir a saída

```
x1 =    1.6571
x2 =   -2.0240
```

9.5.1 Exercícios

Repita os exercícios da seção anterior, formatando de maneira adequada as saídas produzidas pelos seus programas.

Capítulo 10

Comandos de seleção

Antes de apresentarmos mais alguns comandos, precisamos saber que existe um tipo de dado em Python chamado `bool` (de booleano). Diferente de números que podem assumir uma infinidade de valores, os objetos booleanos possuem apenas dois valores possíveis: verdadeiro (`True`) ou falso (`False`).

Esses valores são utilizados quando precisamos verificar se uma determina condição é satisfeita ou não. Por exemplo, para verificarmos se uma equação é do segundo grau, precisamos verificar se a variável `a` do nosso programa é igual a zero, ou não. Se for, então não devemos usar o método de Bhaskara e nosso programa deve avisar isso ao usuário.

Valores `bool` podem ser guardados em variáveis e podem ser utilizadas em expressões. Para criarmos expressões booleanas, existem operadores que servem para comparar operandos. Por exemplo:

```
>>> a = 10.01
>>> a == 10
False
>>> a > 10
True
>>> a != 10
True
>>> 'abc' < 'def'
True
>>>
```

A Tabela 10.1 mostra um resumo dos operadores que produzem resultados booleanos, também conhecidos por operadores relacionais. Note que eles podem ser utilizados para comparar valores de vários tipos diferentes como `int`, `float` ou string. Mas os tipos dos operandos têm que ser compatíveis. Não é possível comparar, por exemplo, um número e um string.

Existem também operadores lógicos que servem para combinar expressões booleanas. Por exemplo, se tivermos uma variável numérica `x` e gostaríamos

Operador	Significado	Exemplo	Resultado
==	Igual	10 == 10.1	False
		"abc"== "abc"	True
!=	Diferente	10 != 10.1	True
		"abc"!= "abc"	False
<	Menor	10.1 < 10	False
		"abc"< "def"	True
>	Maior	10.1 > 10	True
		"abc"> "def"	False
<=	Menor ou Igual	10.1 <= 10	False
		"abc"<= "abc"	True
>=	Maior ou Igual	10.1 >= 10	True
		"abc">= "def"	False

Tabela 10.1: Operados relacionais

de saber se ela está no intervalo $[0, 1]$. Para fazer tal verificação, precisamos, na verdade fazer duas comparações. A variável deve, ao mesmo tempo, ser menor ou igual a 1 e maior ou igual a 0. Note que na primeira expressão ambas as subexpressões são verdadeiras e portanto a expressão como um todo é verdadeira. Na segunda, a subexpressão da esquerda é verdadeira mas a da direita é falsa e portanto o resultado é falso.

```
>>> x = 0.5
>>> x >= 0.0 and x <= 1.0
True
>>> x = 2.0
>>> x >= 0.0 and x <= 1.0
False
>>>
```

Se, ao contrário do exemplo anterior, quiséssemos saber se o valor de `x` está fora do intervalo dado, precisaríamos verificar se ele é maior que 1 ou menor que 0. Ou seja, a expressão abaixo é verdadeira se pelo menos uma das subexpressões é verdadeira.

```
>>> x = 0.5
>>> x < 0.0 or x > 1.0
False
>>> x = 2.0
>>> x < 0.0 or x > 1.0
True
>>>
```

Nesse exemplo é impossível que as duas subexpressões sejam verdadeiras mas em outros casos isso é possível, o que faria com que o resultado da expressão toda seja verdadeiro pois pelo menos uma subexpressão deve ser verdadeira.

```
>>> x = 0.5
>>> y = 1.5
>>> x < 0.0 or y > 1.0
True
>>> x == 0.0 or x != y
True
>>>
```

Existe, ainda, o operador unário "not" cujo resultado é a negação do valor ao qual ele é aplicado. Por exemplo, se quisermos saber se o valor de `x`, no exemplo anterior, não está no intervalo $[0, 1]$ podemos utilizar a expressão que segue:

```
>>> x = 0.5
>>> not (x >= 0.0 and x <= 1.0)
False
>>> x = 2.0
>>> not (x >= 0.0 and x <= 1.0)
True
>>>
```

A Tabela 10.2 mostra o que chamamos "tabela verdade" para cada um dos operadores lógicos. Ou seja, para cada combinação possível dos operandos, ela mostra qual seria o resultado da expressão.

Tabela 10.2: Tabela verdade dos operadores lógicos

Operação	a	b	Resultado
a and b	False	False	False
	False	True	False
	True	False	False
	True	True	True
a or b	False	False	False
	False	True	True
	True	False	True
	True	True	True
not a	False	-	True
	True	-	False

10.1 O comando `if/else`

Agora que conhecemos os valores e expressões booleanos, podemos introduzir o comando de seleção `if`. Ele serve para modificar o fluxo de execução do nosso programa. Por exemplo, se o valor de Δ computado no método de Bhaskara for negativo, não queremos computar o valor das raízes, pois obteríamos um erro ao tentar calcular a raiz quadrada de um número negativo. O que queremos é mostrar apenas uma mensagem, dizendo que a equação não tem solução real.

O comando `if` pode ser entendido da seguinte maneira:

Programa 10.1 Como funciona o comando `if/else`

```
1 if <expressão booleana> :
2     comando executado se expressão for verdadeira
3     comando executado se expressão for verdadeira
4     comando executado se expressão for verdadeira
5 else :
6     comando executado se expressão for falsa
7     comando executado se expressão for falsa
```

Ele inicia com a palavra **if** que vem seguida de uma expressão cujo resultado deve ser do tipo `bool` seguida de ":". Se o resultado for verdadeiro, então os comandos que estiverem nas linhas seguintes são executados. Mas notem que para indicarmos quais são esses comandos, devemos colocá-los todos em um nível a mais de indentação. No exemplo acima, são três comandos executados, um após o outro, se a condição do `if` for `True`.

Em seguida, vem o comando `else`, que assim como o `if` deve ter um ":" no final da linha. O comando `else` deve estar no mesmo nível de indentação do `if` e abaixo dele vêm os comandos que serão executados se a expressão booleana der resultado `False`. Resumidamente podemos compreender o comando `if` como "se" e o comando `else` como "senão".

Então, podemos melhorar o nosso programa *bhaskara.py*, adicionando a verificação se a variável `delta` é negativa. Note que depois do comando `if` colocamos

um *print* que está no mesmo nível de indentação do `if` e dos comandos anteriores. Ou seja, não está "dentro" do comando `if` e será executado, qualquer que seja o valor da expressão booleana computada no `if`.

Programa 10.2 Uso do `if` no programa *bhaskara.py*

```
1  import math
2
3  a = float(input('Digite o valor de a: '))
4  b = float(input('Digite o valor de a: '))
5  c = float(input('Digite o valor de c: '))
6
7  delta = b ** 2 - 4 * a * c
8
9  # Verfica se delta é negativo
10 if delta < 0:
11     print('Essa equação não possui raízes reais')
12 else:
13     x1 = (-b + math.sqrt(delta)) / (2 * a)
14     x2 = (-b - math.sqrt(delta)) / (2 * a)
15
16     print('O valor da 1a raiz é {:.4f}'.format(x1))
17     print('O valor da 2a raiz é {:.4f}'.format(x2))
18
19
20 print('Final do programa')
```

O comando `else` deve sempre estar associado a um comando `if`. Ele não existe sozinho. Por outro lado, ele não é obrigatório, ou seja, podemos ter comandos `if` sem `else`.

Programa 10.3 Comando `if` sem `else`

```
1 if <expressão booleana> :
2     comando executado se expressão for verdadeira
3     comando executado se expressão for verdadeira
4     comando executado se expressão for verdadeira
```

Nesse caso, se a expressão booleana for verdadeira, os mesmos comandos que seguem o `if` são executados. mas se for falsa, nada é executado, ou melhor, o programa prossegue com o próximo comando depois do `if`, no mesmo nível de indentação. Um exemplo prático disso também pode ser mostrado no algoritmo de Bhaskara. Antes de executar os cálculos, nós vamos verificar se a equação realmente é quadrática, ou seja, se o valor da variável `a` é diferente de zero. Se ela não for quadrática, nosso programa deve apenas emitir um aviso e deve terminar, sem fazer o resto.

Programa 10.4 Uso do `if` sem `else` no programa *bhaskara.py*

```
1  import math
2  import sys
3
4  a = float(input('Digite o valor de a: '))
5  b = float(input('Digite o valor de a: '))
6  c = float(input('Digite o valor de c: '))
7
8  #verifica se é equação quadrática
9  if a == 0:
10     print('Essa equação não é quadrática.')
11     print('Terminando a execução')
12     sys.exit()
13
14 delta = b ** 2 - 4 * a * c
15
16 # Verfica se delta é negativo
17 if delta < 0:
18     print('Essa equação não possui raízes reais')
19 else:
20     x1 = (-b + math.sqrt(delta)) / (2 * a)
21     x2 = (-b - math.sqrt(delta)) / (2 * a)
22
23     print('O valor da 1a raiz é {:.4f}'.format(x1))
24     print('O valor da 2a raiz é {:.4f}'.format(x2))
25
26
27 print('Final do programa')
```

Veja que, após ler do teclado os valores dos coeficientes, nosso programa verifica se a variável `a` é igual a zero. Se for, ele emite uma mensagem e chama a função `sys.exit()` que termina a execução do programa imediatamente. Mas se não for, não existe uma alternativa no comando `if`. O programa simplesmente continua a sua execução no próximo comando, que é o cálculo do Δ.

10.1.1 Exercícios

1. Execute os programas 10.2 e 10.4 desta seção com valores diferentes e analise os resultados produzidos. Use pelo menos algum caso em que:

   a) a equação não tem solução real;

   b) a equação não é quadrática

10.2 Aplicação: método da bisseção

Embora de forma bastante precária, podemos agora implementar o método da bisseção. Por hora, temos que definir a função que desejamos usar com uma

expressão lambda, no início do programa. Ou seja, se quisermos mudar a função, temos que alterar o programa.

Vamos, também, inicializar os valores das variáveis que determinam o intervalo inicial e o erro que consideramos aceitável. Assim, o nosso programa inicia-se com os seguintes comandos:

Programa 10.5 Início do método da bisseção

```
1 import sys
2
3 f = lambda x: x ** 3 - x ** 2 - 13 * x + 8
4 a = -4.0
5 b = -3.0
6 erro = 0.001
```

Isso feito podemos começar as iterações para ir diminuindo o intervalo de busca da raiz. Ou seja, computamos o ponto médio entre `a` e `b`, verificamos o tamanho do intervalo, para decidir se podemos ou não parar e, se não podemos, alteramos o valor das variáveis para refletir o novo intervalo. A primeira iteração, então fica:

Programa 10.6 Uma iteração do método da bisseção

```
1 #iteração 1
2 c = (a+b)/2
3 if abs(( b - a ) / 2) < erro:
4     print('Achou raiz ', c, ' com erro ', (b-a)/2)
5     sys.exit();
6 if f(a) * f(c) < 0:
7     b = c
8 else:
9     a = c
```

Note que depois de executar esse trecho do programa, teremos, novamente que computar o valor de `c`, verificar o tamanho do intervalo etc. Isso significa que podemos começar a segunda iteração, que é exatamente igual à primeira. Ou seja, o resto do nosso programa é uma repetição, várias vezes, do código acima. Se quisermos que o nosso programa tenha dez iterações, repetimos dez vezes o mesmo trecho.

Ao executar o programa, os comandos vão sendo executados sequencialmente e quando o intervalo for menor do que o erro, a condição do primeiro `if` é executada e o programa termina. Isso pode acontecer na primeira, na segunda, ou qualquer uma das iterações, dependendo da função, do intervalo, e do valor escolhido para o erro. Por exemplo, usando os valores acima, o programa termina na décima iteração. Se a tolerância for de 0.01, o programa terminaria na sétima iteração.

Mas também é possível que após a décima iteração, o tamanho do intervalo continue maior do que a tolerância que definimos. Isso acontece, por exemplo, se definirmos a variável `erro` com um valor 0,0001. Para esses casos, precisamos colocar, no final do nosso programa, um comando para avisar ao usuário qual foi a solução encontrada e qual o erro obtido. Adicionamos, então, o seguinte comando:

Programa 10.7 Finalizando o método da bisseção

```
1 print('Valor calculado ', c, ' com erro ', (b-a)/2)
```

10.3 O comando `if/elif/else`

O comando `if` admite uma outra forma, que permite que várias expressões booleanas sejam testadas em sequência. Quando uma delas for verdadeira, os comandos correspondentes são executados e todas as outras expressões booleanas serão ignoradas. Caso nenhuma das expressões for verdadeira, podemos ter, opcionalmente o comando `else`.

Programa 10.8 Como funciona o comando `if`/`elif`/`else`

```
1 if <expressão 1> :
2     comando executado se expressão 1 for verdadeira
3 elif <expressão 2> :
4     comando executado se expressão 2 for verdadeira
5 elif <expressão 3> :
6     comando executado se expressão 3 for verdadeira
7 else :
8     comando executado se todas expressões forem falsas
```

Podemos ter quantas expressões quisermos. A primeira verdadeira determina os comandos que serão executados e as demais, abaixo dela, não serão verificadas e seus comandos não serão executados. O comando `else` pode aparecer ou não.

Se você tentou implementar o programa da bisseção, no final da seção anterior, poderia ter usado esse tipo de comando `if`. Depois de computar o valor médio entre `a` e `b`, é preciso escolher qual subintervalo utilizar na próxima iteração. Para isso, verificamos o valor de $f(a) * f(c)$. Se esse valor for negativo, iremos utilizar o intervalo entre `a` e `c`. Se for positivo, o intervalo entre `c` e `b`. E se for zero, isso significa que $f(c)$ é exatamento zero e portanto `c` tem o valor da raiz da função. Traduzindo isso para a linguagem Python, teríamos algo como:

Programa 10.9 Como funciona o comando `if/elif/else`

```
1 c = (a+b)/2
2 if f(a) * f(c) < 0:
3     b = c
4 elif f(a) * f(c) > 0:
5     a = c
6 else:
7     print('O valor exato da raiz é ', c)
8     sys.exit()
```

10.3.1 Exercícios

1. Reimplemente o algoritmo da bisseção com, no máximo, 10 iterações. Use o comando `if/elif/else` para verificar qual é o próximo subintervalo que o programa deve usar para procurar a raiz, conforme explicado nesta seção.

2. Reimplemente o algoritmo de Newton-Raphson com, no máximo, 10 iterações. A cada iteração, use o comando `if` para verificar se o erro está abaixo de 10^{-7}. Se estiver, termine imediatamente o programa, exibindo qual é a raiz e qual é o erro.

10.4 Comandos aninhados

O comando `if`, assim como outros comandos que veremos adiante, possui "dentro" dele outros comandos. Nos exemplos que vimos há pouco, utilizamos comandos simples como atribuições e o `print` mas podemos utilizar quaisquer comandos, inclusive o próprio `if`.

Tomando novamente como exemplo o método de Bhaskara (Programa 10.4) podemos reescrever a sua implementação da seguinte forma:

Programa 10.10 `ifs` aninhados no programa *bhaskara.py*

```
1  import math
2  import sys
3
4  a = float(input('Digite o valor de a: '))
5  b = float(input('Digite o valor de b: '))
6  c = float(input('Digite o valor de c: '))
7
8  #verifica se é equação quadrática
9  if a == 0:
10     print('Essa equação não é quadrática.')
11 else:
12     delta = b ** 2 - 4 * a * c
13
14     # Verfica se delta é negativo
15     if delta < 0:
16         print('Essa equação não possui raízes reais')
17     else:
18         x1 = (-b + math.sqrt(delta)) / (2 * a)
19         x2 = (-b - math.sqrt(delta)) / (2 * a)
20
21         print('A 1a raiz é {:.4f}'.format(x1))
22         print('A 2a raiz é {:.4f}'.format(x2))
23
24 print('Final do programa')
```

Neste programa, vemos que a condição do primeiro `if` (linha 9) é executada e, se for verdadeira, a execução segue para os dois `print` (linhas 10 e 11). Se a condição for falsa, a execução vai direto para os comandos que estão dentro do `else` da linha 12. Só que dentro desse `else` temos o cálculo do valor de `delta` e depois um outro comando `if/else`. Nesse caso, a condição da linha 16 é verificada e se for verdadeira executa-se o `print` da linha 17. Se for falsa, são os comandos das linhas 19 a 23 que serão executados.

Note que a indentação é que determina qual comando está aninhado a outro comando. O comando `if/else` que começa na linha 16 está dentro do `else` da linha 12. Por isso, está um nível de indentação à direita dele. Os comandos que estão dentro desse segundo `if`, ou seja, linhas 17 e 19 a 23, devem então estar um nível mais à direita ainda. É dessa forma que o interpretador Python identifica a estrutura de comandos do nosso programa.

10.4.1 Exercícios

1. Crie um programa que permita que o usuário informe um número qualquer e então o programa deve exibir a raiz quadrada deste número. Caso o número informado pelo usuário seja um número negativo, utilize o módulo deste número para calcular a raiz quadrada. Exemplos:

   (a) Caso o número informado pelo usuário seja 16, o resultado deve ser 4.

   (b) Caso o número informado pelo usuário seja -36, o resultado deve ser 6.

2. Crie um programa que permita que o usuário informe duas notas e com base nelas, calcule a média, mostre sua nota final e com base nesta nota final, exiba se ele foi aprovado, se ficou de recuperação ou se foi reprovado. Para ser aprovado o aluno tem que obter uma nota maior ou igual a 7. Para ser reprovado o aluno tem que obter uma nota menor que 5. Exemplos:

   (a) Se a primeira nota informada for 8 e a segunda for 7,5, o resultado deve ser "Nota Final: 7.75 | Aprovado".

   (b) Se a primeira nota informada for 7,5 e a segunda for 5,50, o resultado deve ser "Nota Final: 6.50 | Recuperação".

   (c) Se a primeira nota informada for 2,5 e a segunda for 6,5, o resultado deve ser "Nota Final: 4.50 | Reprovado".

3. Escreva um programa que gere aleatoriamente um número entre 0 e 100. Depois, o programa deve dar até 10 chances para o usuário adivinhar qual é o numero secreto. A cada palpite, o programa diz ao usuário se seu palpite é maior, menor, ou se ele acertou o valor. Se o usuário acertar o valor, o programa termina. Para gerar um número aleatório use a função *random.randint*.

4. Escreva um programa que lê 3 valores que representam os lados de um triângulo. O programa deve dizer se eles correspondem a um triângulo equilátero, isósceles ou escaleno ou, ainda se não correspondem a um triângulo.

5. Faça um programa que leia o sexo e a altura de uma pessoa e calcule o seu peso ideal, utilizando as seguintes fórmulas:

   para homem: (72.7 x h) – 58

   para mulher: (62.1 x h) – 44.7

6. Escreva um programa que lê o valor do salário de um trabalhador e calcula o valor do imposto de renda a ser recolhido na fonte, de acordo com a tabela da Receita Federal.

7. Escreva um programa que verifica se um determinado ano é ou não bissexto.

Capítulo 11

Comandos de repetição

Neste capítulo vamos ver como executar comandos de forma repetida e controlada. No programa da bisseção, com dez iterações, tivemos que repetir as mesmas instruções, dez vezes. Para cada iteração, precisamos dividir o intervalo corrente, computar o erro e escolher o próximo intervalo. Nada muda, os comandos executados são sempre os mesmos.

Programa 11.1 Programa de bisseção, sem comandos de repetição

```
1  #iteração 1
2  c = (a+b)/2
3  if abs(( b - a ) / 2) < erro:
4      print('Achou raiz ', c, ' com erro ', (b-a)/2)
5      sys.exit();
6  if f(a) * f(c) < 0:
7      b = c
8  else:
9      a = c
10 ...
11 #iteração 10
12 c = (a+b)/2
13 if abs(( b - a ) / 2) < erro:
14     print('Achou raiz ', c, ' com erro ', (b-a)/2)
15     sys.exit();
16 if f(a) * f(c) < 0:
17     b = c
18 else:
19     a = c
```

Antes de mais nada, é extremamente deselegante escrevermos um programa desta maneira. Além disso, o programa é muito inflexível, ou seja, ele tenta a solução em dez iterações e pronto. Se quisermos mudar isso, temos que editar nosso programa e adicionar ou remover várias linhas de código. Por isso, nas próximas seções, vamos conhecer alguns comandos que permitem executar uma sequência de instruções repetidamente.

11.1 O comando `while`

No comando `while`, assim como no comando `if`, existe uma expressão booleana que é calculada e, se o resultado for `True`, então são executados os comandos que estão "dentro" do `while`. Se a expressão for falsa, então os comandos dentro do `while` são simplesmente ignorados e executa-se o próximo comando. A diferença, em relação ao `if` é que, terminada a execução dos comandos dentro do `while`, a execução volta para o início do comando, ou seja, a expressão booleana é avaliada novamente, e tudo se repete.

Por exemplo, no trecho de programa abaixo, a variável `i`, inicialmente recebe o valor 1. Em seguida, inicia-se a execução do `while`, calculando a expressão `i < 10`, cujo resultado é `True`. Por isso, são executados os dois comandos dentro do `while`. Note que o segundo comando faz com que `i` passe a valer 2. Depois disso, como não temos mais comandos dentro do `while`, a execução volta para o seu início, calculando a expressão booleana que, novamente é verdadeira e, novamente os comandos dentro do `while` são executados.

Programa 11.2 Exemplo do comando `while`

```
1 i = 1
2
3 while i < 10:
4     print('O valor de i é: ', i)
5     i = i + 1
6
7 print('O valor final de i é: ', i);
```

A cada vez que a execução do programa entra no comando `while`, uma mensagem é exibida e o valor da variável `i` aumenta. A execução só passa para o próximo comando, que segue o `while`, quando a expressão booleana for false, ou seja, quando `i` atingir o valor 10. A saída gerada por esse programa é a seguinte:

```
O valor de i é: 1
O valor de i é: 2
O valor de i é: 3
O valor de i é: 4
O valor de i é: 5
O valor de i é: 6
O valor de i é: 7
O valor de i é: 8
O valor de i é: 9
O valor final de i é: 10
```

Podemos visualizar a execução desse trecho de programa por meio do diagrama da Figura 11.1.

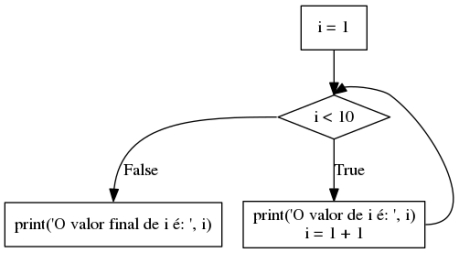

Figura 11.1: Fluxograma do comando `while`

Vamos, então, usar o comando `while` para implementar o algoritmo da bisseção. Mais precisamente, usamos esse comando para controlar o número de iterações do algoritmo. Se ele atingir um valor máximo, por exemplo 10, terminamos a execução e mostramos o resultado até aquele ponto. Inicialmente, fazemos as inicializações das variáveis que vamos utilizar no programa, ou seja, qual é a função, o intervalo, a tolerância, e o número máximo de iterações.

Programa 11.3 Inicialização de variáveis para método da bisseção

```
1 import sys
2
3 f = lambda x: x ** 3 - x ** 2 - 13 * x + 8
4 a = -4
5 b = -3
6 erro = 0.001
7 iteracoes = 10
```

Em seguida, precisamos calcular o valor médio entre `a` e `b` e verificar se o tamanho do intervalo é menor do que o erro. Se for, achamos já a solução e podemos terminar a execução. Se não for, calculamos um novo intervalo. Mas isso precisamos fazer várias vezes. No nosso caso, no máximo dez vezes, pois usamos `iteracoes = 10`. Temos, então, os seguintes comandos, depois da inicialização:

Programa 11.4 Implementação da bisseção usando o comando `while`

```
1  i = 1
2  while i <= iteracoes:
3      c = (a+b)/2
4      if abs(( b - a ) / 2) < erro:
5          print('Achou raiz ', c, ' com erro ', (b-a)/2)
6          sys.exit()
7      if f(a) * f(c) < 0:
8          b = c
9      else:
10         a = c
11     i = i + 1
12
13 print('Valor calculado ', c, ' com erro ', (b-a)/2)
```

A variável `i` é usada para controlar o número de iterações que foram executadas. Seu valor inicial é um pois quando o comando `while` é executado, o valor de `i` deve indicar qual é a iteração que está sendo iniciada. O comando `while` e os comandos que estão dentro dele serão executados enquanto o valor de `i` for menor ou igual ao número máximo de iterações, no caso, 10.

Note que caso o tamanho do intervalo seja menor do que o erro que determinamos no início, o programa termina de imediato, sem que seja preciso executar todas as iterações. Nesse caso, nosso programa mostra o valor calculado da variável `c` e termina a execução. Se o tamanho do intervalo não atingir um valor inferior ao erro nas dez iterações, o comando `while` termina e então o valor computado até aquela iteração é exibido, bem como o erro até aquele momento.

Para mudar o número de iterações desejadas não precisamos mudar substancialmente nosso programa. Basta mudar o comando que inicializa a variável `iteracoes`. Ou podemos fazer melhor do que isso. Podemos perguntar ao usuário quantas iterações ele deseja, qual o erro que ele quer usar e qual é o intervalo inicial.

Programa 11.5 Lendo o valor das variáveis para o método da bisseção

```
1  import sys
2
3  f = lambda x: x ** 3 - x ** 2 - 13 * x + 8
4  a = int(input('Forneça o valor inicial de a: '))
5  b = int(input('Forneça o valor inicial de b: '))
6  erro = float((input('Qual o valor da tolerância? '))
7  iteracoes = int(input('Número máximo de iterações: '))
```

Existem várias maneiras de implementar um mesmo algoritmo. Na listagem que segue, vemos uma forma alternativa de implementar o método da bisseção. A ideia aqui é que o programa termine quando atingimos o número máximo

de iterações ou quando o erro que temos for menor do que a tolerância que determinamos no início.

Mas como estamos utilizando o comando `while`, podemos dizer que o programa deve continuar enquanto não atingimos o número máximo de iterações e o erro for maior que a tolerância. A inicialização das variáveis continua a mesma mostrada anteriormente. O controle das iterações passa a ser:

Programa 11.6 Implementação da bisseção usando o comando `while` com uma condição composta

```
1  i = 1
2  c = (a+b)/2
3
4  while i <= iteracoes and abs(( b - a ) / 2) >= erro :
5      if f(a) * f(c) < 0:
6          b = c
7      else:
8          a = c
9      i = i + 1
10     c = (a+b)/2
11
12 print('Valor calculado ', c, ' com erro ', (b-a)/2)
```

Note que precisamos computar o valor de `c` antes do comando `while` pois é possível que a execução não entre laço nenhuma vez. Mesmo assim, o ponto médio entre `a` e `b` é a solução.

11.1.1 Exercícios

1. Escreva um programa para calcular o valor total de uma compra de supermercado. O valor dos produtos deve ser informado pelo usuário. O programa deve mostrar o valor final após o usuário informar um número negativo (o número negativo não deve fazer parte do cálculo do resultado final).

2. Incremente, usando o comando `while`, o Exercício 2 da Seção 10.4.1 e permita que o usuário informe o número de alunos que deverão ter suas notas calculadas. O resultado e a nota de cada aluno deverá ser exibido imediatamente após o usuário informar a nota. Exemplo: Caso o usuário informe que será calculada a nota de 3 alunos:

   (a) 1ª nota do 1º Aluno: 7. 2ª nota do 1º aluno: 8. O programa deve exibir "1o. Aluno: Nota Final: 7.75 | Aprovado".

   (b) 1ª nota do 2º Aluno: 7,5. 2ª nota do 2º aluno: 5,5. O programa deve exibir "2o. Aluno: Nota Final: 6,5 | Recuperação".

   (c) 1ª nota do 3º Aluno: 2,5. 2ª nota do 3º aluno: 4,5. O sistema deve exibir "3o. Aluno: Nota Final: 4.5 | Reprovado"

3. Incremente o exercício anterior, que até então calculava a média aritmética só de 2 notas e permita que o usuário informe o número de notas que serão utilizadas para calcular a nota final, ou seja, caso o usuário informe 1, deverá ser digitada 1 nota para cada aluno, caso o usuário informe 5, deverão ser digitadas 5 notas para cada aluno. Exemplo: Caso o usuário informe que serão calculadas 3 notas para cada aluno e 5 alunos:

   (a) 1ª nota do 1º Aluno: 7. 2ª nota do 1º aluno: 8. 3ª nota do 1º aluno: 6,85. O programa deve exibir "1o. Aluno: Nota Final: 7.28 | Aprovado".

   (b) 1ª nota do 2º Aluno: 7,5. 2ª nota do 2º aluno: 5,5. 3ª nota do 2º aluno: 7,5. O programa deve exibir "2o. Aluno: Nota Final: 6,83 | Recuperação".

   (c) 1ª nota do 3º Aluno: 2,5. 2ª nota do 3º aluno: 4,06. 3ª nota do 2º aluno: 5,2. O sistema deve exibir "3o. Aluno: Nota Final: 3.92 | Reprovado".

   (d) 1ª nota do 4º Aluno: 10. 2ª nota do 2º aluno: 8,9. 3ª nota do 2º aluno: 9,5. O programa deve exibir "4o. Aluno: Nota Final: 9.46 | Aprovado".

   (e) 1ª nota do 5º Aluno: 5. 2ª nota do 3º aluno: 7,25. 3ª nota do 2º aluno: 6,15. O sistema deve exibir "5o. Aluno: Nota Final: 6.13 | Recuperação".

11.2 O comando `for`

Outro comando de repetição é o `for`. Ele requer uma variável de controle e um objeto composto por diversas "partes". A cada iteração do comando `for`, uma dessas partes é atribuída à variável de controle. Esse comando é muito utilizado com listas, que veremos em um capítulo mais adiante mas também pode ser usado com outros objetos.

Por exemplo, como vimos anteriormente, um string é composto por vários caracteres, que podem ser acessados por um índice. Podemos utilizar o `for` para pegar esses caracteres, um de cada vez.

Programa 11.7 Uso do comando `for` para pegar os caracteres de um string

```
1 s = 'Python'
2 for c in s:
3     print(c)
```

A execução desse trecho de programa faz, inicialmente, que a letra 'P' seja atribuída à variável `c` e o comando `print` executado com esse valor. Retorna-se ao `for` e o segundo caractere é atribuído à variável `c` e o `print` é executado. E assim por diante, até o último caractere do string armazenado na variável `s`. Dizemos que a variável `c` é usada para "percorrer" os elementos do string.

A saída produzida por esse trecho de programa é o seguinte:

```
P
y
t
h
o
n
```

Dentro do comando `for` podemos ter vários comandos, e quaisquer comandos, como, por exemplo, um comando `if`. No programa abaixo, mostramos na saída somente as letras que forem minúsculas.

Programa 11.8 Uso do comando `for` para pegar os caracteres minúsculos de um string

```
1 s = 'Python'
2 for c in s:
3     if c.islower():
4         print(c)
```

A saída produzida por esse trecho de programa é o seguinte:

```
y
t
h
o
n
```

Outro objeto usado com o comando `for` é do tipo `range`. Ele pode ser obtido com a chamada da função `range`. Se passarmos como parâmetro um número inteiro n, ela produz um objeto que ao ser percorrido pelo `for` vai produzir números inteiros de zero até $n - 1$. Por exemplo, os valores 0, 1, 2, 3 e 4 são atribuídos, em sequência à variável `j` no programa a seguir.

Programa 11.9 Uso do comando `for` com a função `range`

```
1 for j in range(5):
2     print(j)
```

A saída produzida por esse trecho de programa é o seguinte:

```
0
1
2
3
4
```

Se usarmos a função `range` com dois parâmetros, por exemplo `range(3,10)`, queremos atribuir à variável de controle do `for` os valores 3, 4, 5, 6, 7, 8 e 9, ou seja, o primeiro parâmetro indica o valor inicial e o último parâmetro indica o valor final.

Programa 11.10 Uso do comando `for` com a função `range` com 2 parâmetros

```
1 for j in range(3, 10):
2     print(j)
```

A saída produzida por esse trecho de programa é o seguinte:

```
3
4
5
6
7
8
9
```

E se usarmos três parâmetros, o último indica qual é o salto entre uma atribuição e outra. Por exemplo, com `range(-5, 10, 3)`, vamos utilizar os valores -5, -2, 1, 4, 7 e 10.

Programa 11.11 Uso do comando `for` com a função `range` com 3 parâmetros

```
1 for j in range(-5, 10, 3):
2     print(j)
```

A saída produzida por esse trecho de programa é o seguinte:

```
-5
-2
1
4
7
```

11.2.1 Exercícios

- Implemente o método da bisseção usando o comando `for`
- Refaça os exercícios da Seção 11.1.1 utilizando o comando `for`.

11.3 Aplicação: integração numérica

Com o que aprendemos neste capítulo podemos criar um programa que implementa os métodos de integração numérica que vimos na primeira parte do livro. Vamos começar com o método dos trapézio. Nesse método, vamos dividir o intervalo em subintervalos e, então, computar a área de cada um deles, como já discutimos anteriormente.

Programa 11.12 Método dos trapézios usando o comando `while`

```
1  import math
2  f = lambda x:  x ** 3  + 8
3
4  n = 40 # intervalos
5  a = 0 #início do intervalo
6  b = 1 # fim do intervalo
7  h = (b - a) / n # tamanho de cada intervalo
8
9  s = 0.0 # variável para somar a integral
10 while a < b :
11     s += (f(a) + f(a+h)) / 2 * h
12     a += h
13
14 print('O valor da integral é {:.7f}'.format(s))
```

Assim como mostramos antes, a inicialização das variáveis `a`, `b` e `n` pode ser feita perguntando-se ao usuário quais valores deseja utilizar. Aqui mantivemos fixos, o intervalo $(0, 1)$ e dividimo-lo em 40 subintervalos. A função que estamos usando é $f(x) = x^3 + 8$.

Uma vez inicializados os parâmetros básicos do método, calculamos o tamanho de cada intervalo, guardando seu valor na variável `h`. A variável `s` guarda

o valor da integral. A cada iteração do método, soma-se a essa variável a área do subintervalo.

O comando `while` vai ser executado enquanto o valor de `a` for menor do que o valor de `b`. A cada execução do `while`, vamos adicionando o tamanho do intervalo à variável `a`. Assim, essa variável tem sempre o valor de início do próximo intervalo do qual vamos calcular a área. No primeiro comando dentro do `while`, calculamos a área do intervalo $(a, a + h)$ e somamos esse valor na variável `s` [1].

Ao final da execução do `while`, a variável `s` é exibida, indicando o valor da integral calculada.

Com o comando `for` nós utilizamos uma variável `i` que "conta" qual é o intervalo que vamos calcular a área. Para saber onde é o início do intervalo, soma-se $i * h$ ao valor inicial do intervalo todo, ou seja, a variável `a`. Esse valor é calculado a cada execução do `for` e atribuído à variável `x0`. Dessa forma não é necessário, ou melhor, seria incorreto ir incrementando o valor da variável `a`, como fizemos anteriormente.

Programa 11.13 Método dos trapézios usando o comando `for`

```
1  import math
2  f = lambda x:  x ** 3  + 8
3
4  n = 40 # intervalos
5  a = 0 #início do intervalo
6  b = 1 # fim do intervalo
7  h = (b - a) / n # tamanho de cada intervalo
8
9  s = 0.0 # variável para somar a integral
10 for i in range(n) :
11     x0 = a + i * h
12     s += (f(x0) + f(x0+h)) / 2 * h
13
14 print('O valor da integral é {:.7f}'.format(s))
```

Embora funcione bem para a função $f(x) = x^3 + 8$, esse nosso programa, em particular o que usa o comando `while` tem um problema. Se o executarmos com a função $f(x) = \sqrt{1 - x^2}$ veremos que um erro ocorre.

Isso acontece por um problema de precisão. Um número `float`, quando representado digitalmente, pode apresentar um erro que, embora muito pequeno, algumas vezes nos atrapalha. Nesse caso, o valor de `a` vai sendo incrementado e calculamos a área do trapézio definido entre esse valor e o valor de `a + h`. No último intervalo, o valor de `a + h` coincide com `b`. Mas, por causa desses erros minúsculos, é possível que o valor de `a + h` ultrapasse o valor de `b`, em uma quantidade muito, muito pequena (algo próximo de $10E - 16$). Isso não seria

[1] Quando usamos a notação "`x += <expressão>`" estamos dizendo que a expressão à direita é calculada e somada ao valor que já existia na variável `x`. É o mesmo que escrever `x = x + <expressão>`. Existem operadores de atribuição como esse também para os demais operadores aritméticos como `-=`, `*=` e `/=`

problema, em geral, pois faria com que a área do último trapézio aumentasse de um valor muito pequeno, e nosso erro seria, também, muito pequeno.

Porém, em alguns casos como o da função mencionada, acontece um erro pois a função não é definida além do intervalo de integração. Ou seja, $f(x)$ é indefinido para $x > 1$ e ao tentarmos computar, no nosso programa, o valor de `f(a + h)` obtemos um erro pois para o interpretador Python não se pode computar a raiz quadrada de um número negativo.

Para resolver esse problema, podemos tentar lidar com o erro de precisão dos `float`s ou podemos alterar um pouco nosso programa, que é o que faremos. Antes de computar a área do trapézio, verificamos se `a + h` ainda está dentro do intervalo. Se não estiver, usamos `b` como limite do intervalo e não `a + h`. Fica assim, então nosso programa:

Programa 11.14 Método dos trapézios usando o comando `while`, corrigido

```
1  import math
2  f = lambda x:  math.sqrt(1 - x**2)
3
4  n = 40 # intervalos
5  a = 0 #início do intervalo
6  b = 1 # fim do intervalo
7  h = (b - a) / n # tamanho de cada intervalo
8
9  s = 0.0 # variável para somar a integral
10 while a < b :
11     if ( a + h ) > b :
12         s += (f(a) + f(b)) / 2 * (b-a)
13     else:
14         s += (f(a) + f(a+h)) / 2 * h
15     a += h
16
17 print('O valor da integral é {:.7f}'.format(s))
```

O segundo método que vimos para computar a integral é pelo método de Simpson. Nesse caso precisamos apenas dividir o intervalo desejado em subintervalos e adicionar o valor da função em cada um dos pontos que limitam os subintervalos, incluindo `a` e `b`. O único detalhe é que cada um desses valores é multiplicado por uma constante. Para `a` e `b` essa constante é um. Para os demais, é dois para os pontos pares (o segundo, quarto etc) e quatro para os ímpares (o primeiro, terceiro etc). Ao final, a soma é multiplicada pelo tamanho do intervalo, dividido por três.

Como precisamos, em cada iteração, saber qual é o intervalo que estamos trabalhando, usaremos o comando `for` para implementar este programa.

Programa 11.15 Método de Simpson usando o comando `for`

```
1  import math
2  f = lambda x: math.sqrt(1 - x**2)
3
4  n = 40 # intervalos
5  a = 0 #início do intervalo
6  b = 1 # fim do intervalo
7  h = (b - a) / n # tamanho de cada intervalo
8
9  s = 0.0 # variável para somar a integral
10 for i  in range(1,n):
11     if i % 2 != 0: # verifica se i é impar
12         c = 4
13     else:
14         c = 2
15     x0 = a + i * h
16     s += c * f(x0)
17
18 s += f(a) + f(b)
19 s *= h / 3
20
21 print('O valor da integral é {:.7f}'.format(s))
```

11.4 Comandos `break` e `continue`

Existem dois comandos que podemos usar "dentro" de um comando de repetição e que ajudam a controlar sua execução. O primeiro deles é o `break`. Se esse comando é executado dentro de um `while` ou `for`, sua execução faz com que seja abortada a repetição e o próximo comando a ser executado é o que vem depois do `while` ou do `for`.

Por exemplo, vamos supor que temos dois números inteiros, armazenados nas variáveis `a` e `b` e `a` $<$ `b`. Queremos achar o menor valor entre esses dois que seja um divisor de `b`. Para isso, vamos incrementando o valor de `a` até acharmos um divisor ou até que seu valor chegue em `b`. Para isso, podemos usar um comando `break`.

Programa 11.16 Exemplo de uso do comando `break`

```
1 a = int(input('Digite o valor de a: '))
2 b = int(input('Digite o valor de b: '))
3 while a < b:
4     if b % a == 0: # verifica se a divide b
5         break
6     a += 1
7 print('O valor do divisor é: ', a)
```

Nesse exemplo, quando um divisor é encontrado, a condição do `if` é verdadeira e o comando `break` é executado. Isso faz a execução do programa "saltar" diretamente para o fim do comando `while`, ou seja, para o `print`. Assim, o laço de repetição é abortado no momento correto.

Podemos reescrever o programa 11.4 usando um `break` quando chegarmos a um erro menor do que a tolerância que estabelecemos. Nesse caso, a execução do `while` é interrompida e o resultado é mostrado no `print` que está no final, depois do `while`.

Programa 11.17 Implementação da bisseção usando os comandos `while` e `break`

```
1  i = 1
2  while i <= iteracoes:
3      c = (a+b)/2
4      if abs(( b - a ) / 2) < erro:
5          break
6      if f(a) * f(c) < 0:
7          b = c
8      else:
9          a = c
10     i = i + 1
11
12 print('Valor calculado ', c, ' com erro ', (b-a)/2)
```

O comando `continue` faz com que a a execução do laço seja interrompida mas não abandonada. A execução volta para o início do comando de repetição, ou seja, a condição vai ser testada novamente e, se for verdadeira, uma nova iteração do comando acontece. Se for falsa, o comando de repetição termina normalmente. Isso significa que em uma execução de um comando de repetição o `continue`, ao contrário do `break`, pode ser executado várias vezes.

Voltando ao exemplo do Programa 11.18, vamos supor que queremos achar o menor divisor mas ele não pode ser múltiplo de 11. Então, cada vez que um múltiplo de 11 aparecer nosso programa vai fazer a execução voltar ao comando de repetição. Note, também, que aqui vamos usar o comando `for` porque ele incrementa automaticamente o valor da variável de controle do laço. Se usássemos o comando `while` teríamos que incrementar essa variável antes de executar o `continue`.

Programa 11.18 Exemplo de uso do comando `continue`

```
1 a = int(input('Digite o valor de a: '))
2 b = int(input('Digite o valor de b: '))
3
4 for k in range(a,b+1):
5     if k % 11 == 0:
6         continue
7     if b % k == 0:
8         break
9 print('O valor do divisor é: ', k)
```

11.4.1 Exercícios

1. Implemente o Programa 11.18 usando o comando `while`.

2. Escreva um programa que gere aleatoriamente um número entre 0 e 100. Depois, o programa deve dar até 10 chances para o usuário adivinhar qual é o numero secreto. A cada palpite, o programa diz ao usuário se seu palpite é maior, menor, ou se ele acertou o valor. Se o usuário acertar o valor, o programa termina. Para gerar um número aleatório use a função *random.randint*.

3. Implemente o método de Simpson utilizando o comando de repetição `while`.

4. Escreva um programa que recebe como entrada um string que representa um número em algarismos romanos e apresenta como saída o valor decimal desse número. Seu programa não precisa verificar se o número fornecido é valido ou não. Apenas assuma que é.

5. Implemente o método de Newton Raphson para o cálculo das raízes de equações.

6. Uma outra forma de computar as raízes de uma função $f(x)$ no intervalo (a, b) é o seguinte:

   a) a partir de a, use uma variável t e vá incrementando essa variável com o valor $0,1$, até que o valor da função mude de sinal;

   b) quando isso acontecer, você terá um intervalo de tamanho $0,1$ no qual a raiz está localizada;

   c) repita a busca nesse intervalo, usando um incremento menor, $0,01$;

   d) repita esse processo, sempre diminuindo o incremento, até que seu intervalo seja menor que a tolerância desejada;

   e) quando isso acontecer, a solução pode ser dada pelo valor inicial do intervalo.

   Implemente esse algoritmo e verifique se ele realmente funciona. Qual é a desvantagem desse método em relação aos outros que estudamos?

7. Implemente a função de seno, que usa a série de Taylor até que o termo calculado seja menor do que 0,0001.

8. Para calcular o valor da raiz quadrada de um número x, podemos proceder da seguinte forma:

   a) escolhemos um chute inicial x_0, de preferência, próximo da raiz de x;

   b) em cada iteração, calculamos x_i como sendo a média entre x_{i-1} e x/x_{i-1};

   c) repetimos o passo (b) e terminamos o algoritmo quando a diferença entre x_i e x_{i-1} for inferior ao erro que desejamos;

   Escreva um programa que leia o valor de x e o chute inicial, compute o valor da raiz de x com erro inferior a 0.00001 e mostre o resultado com 5 casas decimais. Mostre também quantas iterações foram necessária para obter o resultado.

9. Escreva programas que mostrem as seguintes árvores, com qualquer número de linhas (cada programa mostra uma árvore diferente). Ou seja, o usuário escolhe quantas linhas quer. Se escolher 5, os resultados apresentados são os seguintes:

```
a)
     *
    **
   ***
  ****
 *****

b)
    *
   ***
  *****
 *******
*********
```

10. Escreva um programa que leia um número inteiro N. Depois ele deve ler N números inteiro e dizer quantos são ímpares.

11. Escreva um programa que leia um número inteiro e verifique se ele é primo.

12. Escreva um programa que leia uma sequência de números de ponto flutuante, um de cada vez, até que seja digitado o valor zero. Seu programa deve identificar e mostrar qual é o maior e qual é o menor de todos.

13. A série de Fibonacci é definida da seguinte maneira: o primeiro elemento da série é 1, o segundo também. Os demais são a soma dos dois elementos anteriores. Temos então: 1, 1, 2, 3, 5, 8, 13... Escreva um programa que lê um número inteiro N e mostra os N primeiros elementos da série de Fibonacci.

14. Escreva um programa que lê o primeiro valor, o último e a razão de uma PG e que exibe: todos os elementos da PG nesse intervalo e a soma dos elementos nesse intervalo.

Capítulo 12

Listas e similares

Uma lista de compras, uma lista de tarefas a realizar, uma lista de presença. Todos nós estamos habituados a esses objetos. Em Python, uma lista lista é mais ou menos isso: uma sequência de elementos, que podem ser acessados individualmente. E sobre a qual podemos realizar algumas operações como adicionar ou remover elementos, procurar um elemento ou modificar um elemento.

No caso de Python, os elementos são acessados pela posição que ocupam dentro da lista. O esquema é o mesmo que utilizamos para acessar os caracteres de um string. Vejamos alguns exemplos, inicialmente de como podemos criar listas em Python.

```
>>> q1 = []
>>> q2 = list()
>>> q3 = [1,2,3]
>>> q4 = ['Cerveja', 'Carne', 'Carvão']
>>> q5 = list(q4)
```

Os dois primeiros comandos criam listas vazias, ou seja, que não contêm nenhum elemento. O terceiro cria uma lista com três elementos, cada um deles um número inteiro. O quarto comando cria, também, uma lista com três elementos, mas cada um deles é um string. O último comando cria uma outra lista, que é igual à lista armazenada em `q4`, ou seja, com os mesmos strings.

Apesar de termos usado todos os elementos do mesmo tipo, isso não é necessário. Podemos misturar em uma lista valores de tipos diferentes. Por exemplo, a seguir construímos uma lista que tem números inteiros, `float`s e um string. Note, também, que essa lista contém elementos repetidos, ou seja, o número 3,14 aparece duas vezes dentro da lista.

```
>>> q = [2.3, 3, 3.14, 'Carvão', 7, 3.14]
>>> q
[2.3, 3, 3.14, 'Carvão', 7, 3.14]
```

Como dissemos, podemos acessar os elementos da lista usando um índice, que indica a sua posição. Assim como os caracteres de um string, as posições iniciam em zero e são incrementadas, de um em um. Ou seja, o primeiro elemento tem índice zero, o segundo tem índice um e assim por diante.

```
>>> q = [2.3, 3, 3.14, 'Carvão', 7, 3.14]
>>> q[0]
2.3
>>> q[3]
'Carvão'
>>> q[-1]
3.14
```

Assim como nos strings, um índice negativo representa a posição à partir do final da lista. Porém, um índice que não existe como seis ou sete negativo gera um erro, indicado pelo interpretador.

```
>>> q = [2.3, 3, 3.14, 'Carvão', 7, 3.14]
>>> q[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Podemos criar novas listas, também, a partir de pedaços de uma lista existente. Usamos a mesma notação dos strings, com o ":". Lembramos que a notação `[a:b]` inclui todos os elementos que estejam entre `a` e `b-1`. Ou seja, o valor na posição `b`, não entra na sublista. É importante notar que essa notação cria uma nova lista com os mesmos elementos da lista original.

```
>>> q = [2.3, 3, 3.14, 'Carvão', 7, 3.14]
>>> q[1:4]
[3, 3.14, 'Carvão']
>>> q[-4:4]
[3.14, 'Carvão']
```

Qualquer um dos valores na notação pode ser omitido. Se o primeiro for omitido, significa "a partir do início da lista". Se o segundo for omitido, significa "até o fim da lista". A seguir, alguns exemplos.

```
>>> q = [2.3, 3, 3.14, 'Carvão', 7, 3.14]
>>> q[:]
[2.3, 3, 3.14, 'Carvão', 7, 3.14]
>>> q[:4]
[2.3, 3, 3.14, 'Carvão']
>>> q[-4:]
[3.14, 'Carvão', 7, 3.14]
```

Ao contrário do que acontece no acesso a elementos individuais, na notação de sublista não ocorre um erro se usarmos um índice que não existe na lista. Nos dois exemplos abaixo, a lista criada é delimitada pelo início e pelo fim da lista original, no caso em que o índice usado está "abaixo do início" ou "acima do final", respectivamente.

```
>>> q = [2.3, 3, 3.14, 'Carvão', 7, 3.14]
>>> q[:17]
[2.3, 3, 3.14, 'Carvão', 7, 3.14]
>>> q[-30:17]
[2.3, 3, 3.14, 'Carvão', 7, 3.14]
```

Podemos, ainda, adicionar um terceiro valor nessa notação de lista. Esse terceiro valor indica de quantos em quantos elementos desejamos pegar da lista. No exemplo a seguir, usamos `[1:5:2]` para indicar que desejamos os elementos que estão entre a posição um até a posição cinco (essa não é incluída), de dois em dois, ou seja, as posições 1 e 3. No segundo exemplo no quadro a seguir criamos uma lista com elementos inteiros de 100 a 119 e em seguida criamos uma nova lista com os elementos entre as posições 3 e 18, de 3 em três.

```
>>> q = [2.3, 3, 3.14, 'Carvão', 7, 3.14]
>>> q1 = q[1:5:2]
>>> q1
[3, 'Carvão']
>>> t = list(range(100,120))
>>> t[3:18:3]
[103, 106, 109, 112, 115]
```

Os componentes da lista podem ser modificados, depois que a lista foi criada.

Por isso em Python diz-se que é uma estrutura mutável. Para trocar o valor armazenado em uma posição da lista basta fazer uma atribuição àquela posição. Mas não podemos atribuir um valor a uma posição que não existe ainda. Em particular, não podemos incluir um novo elemento na lista atribuindo um valor para a próxima posição "livre".

```
>>> q = [2.3, 3, 3.14, 'Carvão', 7, 3.14]
>>> q[2] = 'Carne'
>>> q
[2.3, 3, 'Carne', 'Carvão', 7, 3.14]
>>> q[-1] = 0
>>> q
[2.3, 3, 'Carne', 'Carvão', 7, 0]
>>> q[6] = 'Gelo'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

12.1 Inclusão e exclusão

Para incluir novos elementos na lista precisamos usar alguma função. A função `append` inclui um elemento no final da lista, que passa ter um elemento a mais. A função `insert` recebe um parâmetro a mais, que indica em que posição o novo elemento deve ser inserido. O elemento que estava naquela posição continua na lista, mas passa a ocupar a posição seguinte, o mesmo acontecendo com os elementos que estavam nas posições sucessivas. Ou seja, os elementos são deslocados, uma posição para frente. Note que a forma correta de chamar essas funções é usando a notação de ponto, que vimos anteriormente. Repare também que usar na função `insert` uma posição que não existe na lista não provoca um erro. O elemento novo é inserido no final ou no início da lista.

```
>>> q = [2.3, 3, 3.14, 'Carvão', 7, 3.14]
>>> q.append('Gelo')
>>> q
[2.3, 3, 3.14, 'Carvão', 7, 3.14, 'Gelo']
>>> q.insert(2,33)
>>> q
[2.3, 3, 33, 3.14, 'Carvão', 7, 3.14, 'Gelo']
>>> q.insert(10,'Linguiça')
>>> q
[2.3, 3, 33, 3.14, 'Carvão', 7, 3.14, 'Gelo', 'Linguiça']
>>> q.insert(-10, 8)
>>> q
[8, 2.3, 3, 33, 3.14, 'Carvão', 7, 3.14, 'Gelo', 'Linguiça']
```

Da mesma forma, para remover elementos da lista usamos funções. A função `pop` não requer nenhum parâmetro e remove o último elemento da lista. Além de remover, a chamada à função também retorna um valor, que é o valor que foi removido.

```
>>> q = [2.3, 3, 3.14, 'Carvão', 7, 3.14]
>>> x = 2 * q.pop()
>>> q
[2.3, 3, 3.14, 'Carvão', 7]
>>> x
6.28
```

Podemos, também, passar um parâmetro para a função `pop` que indica qual é a posição que queremos remover da lista. Note no quadro a seguir que vamos remover a primeira ocorrência de 3.14 da lista, e não a última como fizemos anteriormente. Se a posição indicada for inválida, o interpretador aponta um erro no comando.

```
>>> q = [2.3, 3, 3.14, 'Carvão', 7, 3.14]
>>> x = 2 * q.pop(2)
>>> q
[2.3, 3, 'Carvão', 7, 3.14]
>>> x
6.28
```

Em vez de especificar qual é a posição que queremos eliminar, podemos indicar qual é o valor a ser excluído. Caso o valor apareça mais do que uma vez na lista, apenas a primeira ocorrência será eliminada. E se o valor indicado não

estiver na lista, o interpretador aponta um erro na execução do comando.

```
>>> q = [2.3, 3, 3.14, 'Carvão', 7, 3.14]
>>> q.remove(3.14)
>>> q
[2.3, 3, 'Carvão', 7, 3.14]
>>> q.remove('Gelo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

Ainda para remover elementos da lista podemos utilizar o comando `del`. A vantagem é que podemos usar a mesma notação que usamos para acessar os elementos da lista, ou seja, a lista seguida do índice que desejamos remover.

```
>>> q = [2.3, 3, 3.14, 'Carvão', 7, 3.14]
>>> del q[2]
>>> q
[2.3, 3, 'Carvão', 7, 3.14]
>>> del q[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

Com o comando `del` podemos ainda remover vários elementos usando a notação de sublistas que vimos anteriormente. Por exemplo, no quadro a seguir vamos criar uma lista com os números inteiros de 100 até 119. Depois, vamos remover todos os elementos entre as posições 3 e 18, de 3 em 3. Ou seja, 103, 106, 109, 112 e 115.

```
>>> t = list(range(100, 120))
>>> del t[3:18:3]
>>> t
[100, 101, 102, 104, 105, 107, 108, 110, 111, 113, 114, 116,
117, 118, 119]
```

12.2 Outras operações

Nesta seção apresentamos algumas outras operações que são muito utilizadas com as listas. Ainda não mencionamos, mas podemos obter o número de elementos de uma lista com a função, que já conhecemos, `len`. Para verificar se um valor faz parte de uma lista utilizamos o operador `in`. Vejamos alguns exemplos:

```
>>> q = [2.3, 3, 3.14, 'Carvão', 7, 3.14]
>>> len(q)
6
>>> len(q[1:5:2])
2
>>> 'Carvão' in q
True
>>> 'Gelo' in q
False
```

A aplicação do operador `in` produz um valor booleano. Em um programa, se quisermos inserir um elemento na lista, apenas se ele não estiver presente, ou seja, não queremos elementos repetidos na lista, podemos ter o seguinte trecho de código:

Programa 12.1 Verificando se elemento existe em uma lista, antes de inserir

```
1 if not 'Carvão' in q:
2     q.append('Carvão')
```

Podemos comparar listas utilizando os operadores relacionais que vimos anteriormente. Duas listas são iguais quando elas são do mesmo tamanho e todos os seus elementos são iguais. Para comparar se uma lista é maior ou menor que outra, o interpretador compara o primeiro elemento de cada uma delas e obtém o resultado dessa comparação. Se os primeiros elementos das duas listas forem iguais, são comparados os segundos de cada uma delas, e assim por diante, até que se encontre uma diferença ou que uma das listas termine.

```
>>> q = [2.3, 3, 3.14, 'Carvão', 7, 3.14]
>>> q == [2.3, 3, 3.14, 'Carvão', 7, 3.14]
True
>>> q < [3, 3.14, 'Carvão', 7, 3.14]
True
>>> q > [2, 3, 3.14, 'Carvão', 7, 3.14]
True
>>> q < [2.3, 3, 3.14, 'Carvão', 7, 3.14, 'Gelo']
True
```

Duas lista podem ser somadas. Essa operação resulta em uma nova lista que é a concatenação das duas listas originais.

```
>>> q = [2.3, 3, 3.14, 'Carvão', 7, 3.14]
>>> q + q
[2.3, 3, 3.14, 'Carvão', 7, 3.14, 2.3, 3, 3.14, 'Carvão', 7,
3.14]
>>> q + ['Gelo']
[2.3, 3, 3.14, 'Carvão', 7, 3.14, 'Gelo']
>>> q[1:5:2] + q
[3, 'Carvão', 2.3, 3, 3.14, 'Carvão', 7, 3.14]
>>> [] + q + []
[2.3, 3, 3.14, 'Carvão', 7, 3.14]
```

E podemos até multiplicar uma lista por um número. O número indica quantas vezes a lista original deve ser repetida, para formar uma nova lista.

```
>>> q = [2.3, 3, 3.14, 'Carvão', 7, 3.14]
>>> 2 * q
[2.3, 3, 3.14, 'Carvão', 7, 3.14, 2.3, 3, 3.14, 'Carvão', 7,
3.14]
>>> ['Gelo'] * 3
['Gelo', 'Gelo', 'Gelo']
>>> q[1:5:2] * 4
[3, 'Carvão', 3, 'Carvão', 3, 'Carvão', 3, 'Carvão']
>>> 1024 * []
[]
```

12.2.1 Exercícios

Como a manipulação de listas e de strings é similar em vários aspectos, aproveitamos essa seção para praticar o uso desses dois tipos de dados.

1. Escreva um programa que coloque em uma lista os números inteiros de 1 a 100, em ordem crescente e depois mostre essa lista.

2. Escreva um programa que coloque em uma lista os números inteiros de 1 a 100, em ordem decrescente e depois mostre essa lista.

3. Faça um programa que leia dois strings. Após isso, o programa deve concatenar as informações lidas e mostrar o resultado para o usuário. Exemplo: Se o primeiro string digitado for "Bom dia, " e o segundo "moçada !", então o resultado deverá ficar: "Bom dia, moçada !".

4. Faça um programa que leia N elementos inteiros e coloque numa lista e depois leia um valor de código. Se o código for 1, o programa deve mostrar a lista na ordem direta, se o código for 2, deve mostrar a lista na ordem inversa.

5. Fazer um programa para ler um string e um caractere qualquer. Após isso, deve calcular o número de ocorrências desse caractere no string. Exemplo: Seja a string "USP - São Carlos" e o caractere "s", então o número de ocorrências é 3.

6. Escreva um programa que leia N elementos inteiros e coloque em uma lista. Após isso, seu programa deve percorrer a lista e mostrar somente os números pares.

7. Escreva um programa que leia N elementos inteiros e coloque em uma lista. Após isso, seu programa deve verificar se a sequência na lista representa uma sequência de Fibonacci. Seu programa pode considerar que qualquer sequência em que os elementos são a soma dos dois anteriores é uma sequência de Fibonacci. Por exemplo: 7 8 15 23 38.

8. Fazer um programa para ler um string e dois caracteres. Após isso, deve trocar todas as ocorrências do primeiro caractere pelo segundo. Mostre o string final na saída. Exemplo: Seja a entrada "ambiental" e os caracteres "a" e "b", então o string ficará "bmbientbl".

9. Escreva um programa que computa o valor de um polinômio em um determinado ponto. Essa programa deve receber como entradas:

   - um número inteiro que indica o grau do polinômio;
   - um lista de `float` com os coeficientes do polinômio. A posição k da lista corresponde ao coeficiente de x^k;
   - um `float` que indica o ponto no qual o polinômio deve ser calculado.

   Por exemplo, o polinômio $3x^5 - 12x^3 + 1.08x^2 - 3.9x + 8$ é representado pela lista: [8.0, -3.9, 1.08, -12.0, 0.0, 3.0].

10. Escreva um programa capaz de ler uma frase e contar o número de palavras dessa frase. Considere que as palavras estão separadas por espaços ou vírgulas.

11. Fazer um programa para ler uma frase e verificar se ela é palíndroma, isto é se ela é igual lida da esquerda para a direita e vice-versa. Exemplos: "Ana" é palíndroma, "arara" é palíndroma, "USP" não é palíndroma, "Anotaram a data da maratona" é palíndroma. (obs: os espaços em branco na frase lida devem ser desconsiderados em seu algoritmo).

12. Elabore um programa que receba uma linha de texto e conte as vogais apresentando o respectivo histograma na seguinte forma:

    Exemplo:
    Linha de texto passada: "A próxima quarta-feira é feriado."

    ```
    a : ****** (6)
    e : *** (3)
    i : *** (3)
    o : ** (2)
    u : * (1)
    ```

13. Escreva um programa que leia um número inteiro n e depois exiba as n primeiras linhas do triângulo de Pascal. Veja um exemplo abaixo. Não é difícil descobrir como essa matriz é formada. Sugestão: armazene sempre a linha corrente em uma lista.

    ```
    1
    1  1
    1  2  1
    1  3  3  1
    1  4  6  4  1
    1  5  10 10 5  1
    1  6  15 20 15 6  1
    ```

14. Elabore um programa que receba uma linha de texto e conte quantas vezes cada letra do alfabeto aparece nesse texto. Utilize uma lista para armazenar essa informação e a função `ord` para saber em que posição da lista está o contador de cada letra. Para saber se um caractere é ou não um aletra, você pode usar a função `isalpha`.

15. Implemente um programa de "criptografia" (codificação de dados visando a privacidade de acesso as informações). Dado um string seu programa deve codificar o string por meio de um processo de substituição de caracteres. Cada caractere do string tem uma representação numérica à qual você pode soma um determinado valor, alterando, assim, o conteúdo da mensagem. Para recuperar o string original, devemos tomar a mensagem criptografada e subtrair o mesmo valor. Implemente, também, o programa que recupera a mesnsagem original. Como dica, olhe as funções `ord` e `chr` da linguagem Python.

16. Escreva um programa que leia um inteiro n e depois leia duas listas, cada uma com n elementos. Em seguida, seu programa deve gerar uma terceira lista com os elementos das listas originais, intercalados.

12.3 Aplicação: estatística descritiva

Vamos usar, nesta seção, algumas operações com listas para implementar algumas das técnicas de estatística descritiva vistas na primeira parte deste livro. Inicialmente, precisamos coletar os dados que vamos usar. Para isso, por enquanto, temos que solicitar ao usuário que digite os dados, que vamos armazenar em uma lista.

Iniciamos perguntando ao usuário do nosso programa quantos são os dados que pretende fornecer. Depois, precisamos solicitar que ele digite aquela quantidade de valores e cada valor digitado adicionamos em uma lista.

Programa 12.2 Lendo os dados a serem tratados

```
1  n = int(input('Quantos valores serão digitados? '))
2  x = []
3  i = 1
4  while i <= n:
5      msg = 'Digite o valor ({}/{}): '.format(i,n)
6      r = float(input(msg))
7      x.append(r)
8      i += 1
9
10 print(x)
```

A variável `i` é utilizada para contar quantos números foram lidos. Ela é incrementada até atingir o valor de `n`. A cada iteração do comando `while`, um novo valor é lido e adicionado à lista com a função `append`. No final, mostramos a lista que foi lida, para que o usuário confira os valores. Executando esse programa com 3 valores, temos o seguinte resultado:

```
Quantos valores serão digitados? 3
Digite o valor (1/3): 5.3
Digite o valor (2/3): 3
Digite o valor (3/3): 8.1
[5.3, 3.0, 8.1]
```

Com o que aprendemos até agora, podemos melhorar um pouquinho o nosso programa. Podemos deixá-lo um pouco mais "amigável", ou seja, mais fácil para que vai usá-lo. Primeiro, em vez de perguntar quantos valores serão digitados, nosso programa deve ler e armazenar na lista todos os valores digitados, até que um valor negativo seja digitado. O segundo aperfeiçoamento é que supomos que os dados a serem digitados estão, por exemplo, no intervalo entre 0 e 100. Nesse caso, se o usuário digitar algum valor maior do que 100, vamos avisá-lo dessa restrição e o valor não é armazenado.

No programa que segue, a variável `r` é que controla a execução do comando `while`. Ele termina sua execução se o valor digitado e atribuído a `r` for negativo.

Ainda assim, mantivemos a variável `i` para que o usuário saiba quantos valores já foram digitados. Além disso, se o valor digitado for maior do que 100, uma aviso é dado ao usuário e o valor é ignorado. Se o valor for válido, ou seja, entre 0 e 100, ele é colocado na lista.

Programa 12.3 Lendo os dados e fazendo sua consistência

```
1  x = []
2  r = 0
3  i = 1
4  while r >= 0:
5      msg = 'Digite o valor ({}): '.format(i)
6      r = float(input(msg))
7      if r < 0:
8          print('Entrada de dados terminou')
9      elif r > 100:
10         print('Valor deve estar entre 0 e 100')
11     else:
12         x.append(r)
13         i += 1
14
15 print(x)
```

Executando esse programa, com os mesmos dados anteriores, temos a saída a seguir. Note que ao digitar o valor 3, o usuário enganou-se e digitou 300. Nosso programa alertou sobre o erro e ignorou o dado digitado errado.

```
Digite o valor (1): 5.3
Digite o valor (2): 300
Valor deve estar entre 0 e 100
Digite o valor (2): 3
Digite o valor (3): 8.1
Digite o valor (4): -1
Entrada de dados terminou
[5.3, 3.0, 8.1]
```

Uma vez armazenados os dados na nossa lista, podemos calcular as nossas estatísticas. Vamos iniciar com a média. Para isso precisamos somar todos os valores que estão na lista. Isso pode ser feito facilmente, já que uma lista é um tipo de objeto que podemos percorrer usando o comando `for`. No trecho de programa abaixo, a cada iteração do `for`, um valor que está na lista é atribuído à variável `r`. Então, na primeira execução, o valor que está em `x[0]` é colocado em `r`, na segunda o valor de `x[1]` e assim por diante, até o final da lista. A cada iteração o valor é somado ao valor da variável `soma`. E no final, a média é calculada e atribuída à variável `media`.

Programa 12.4 Calculando a média dos dados

```
1 soma = 0
2 for r in x:
3     soma += r
4
5 media = soma / len(x)
6 print('Valor da soma: {:.4f}'.format(soma))
7 print('Valor da média: {:.4f}'.format(media))
```

Mais uma vez, para computar a variância e o desvio padrão precisamos percorrer a lista e, para cada elemento, computar o quadrado da diferença em relação à média. Mais uma vez usamos o comando `for` para isso.

Programa 12.5 Calculando a variância e o desvio padrão

```
1 variancia = 0.0
2 for r in x:
3     variancia += (r - media) ** 2
4
5 variancia /= len(x) -1
6 dp = math.sqrt(variancia)
7 print('Valor da variância: {:.4f}'.format(variancia))
8 print('Valor do desvio padrão: {:.4f}'.format(dp))
```

Para o cálculo da mediana, precisamos saber qual é o elemento central da lista. Para isso, fica bem mais fácil se tivermos a lista ordenada de forma crescente. Ou seja, rearranjamos os valores, colocando os menores no início e os maiores no final da lista. Existem vários algoritmos conhecidos para fazer isso e poderíamos implementar qualquer um deles. Mas, a biblioteca do Python já fornece funções para isso.

```
>>>p = [5,-3,-1,8,13]
>>>sorted(p)
[-3, -1, 5, 8, 13]
>>>p
[5,-3,-1,8,13]
>>>p.sort()
>>>p
[-3, -1, 5, 8, 13]
>>>
```

Nesse exemplo, vemos duas formas de ordenar a lista. A primeira, usa a função `sorted` que cria uma cópia da lista passada como parâmetro, e não

modifica a ordenação dessa lista. Podemos, por exemplo, atribuir esse valor a uma segunda variável, fazendo `q = sorted(p)`. A segunda forma, usa a função `sort` que não retorna nenhum valor. Ela modifica a ordenação da lista original, no caso, armazenada em `p`. Então, se quisermos uma nova lista ordenada, preservando a lista original, usamos `sorted`. Se quisermos modificar a lista original, usamos `sort`.

Essas funções podem ser usadas em listas de qualquer tipo, incluindo listas com tipos diferentes de dados, desde que os elementos possam todos ser comparados entre si. Podemos, então, ordenar uma lista que só contem strings ou uma que contenha valores `int` e `float` mas não podemos comparar, por exemplo, uma que tenha valores `int` e strings.

```
>>>p = [5,-3,-1.8,8,13]
>>>sorted(p)
[-3, -1.8, 5, 8, 13]
>>>q = ['a', 'h', 'c']
>>>sorted(q)
['a', 'c', 'h']
>>>sorted(p+q)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: int() < str()
>>>
```

Voltando à estatística, uma vez que tenhamos a lista de valores ordenada, basta usar o seu elemento central como mediana. Se o número de elementos for par, usamos a média dos dois elementos centrais. Só precismos tomar cuidado de usar a divisão inteira, ao calcular o índice do elemento central.

Programa 12.6 Calculando a mediana

```
1 x_ordenado = sorted(x)
2 if len(x_ordenado) % 2 == 1:
3     mediana = x_ordenado[len(x_ordenado)//2]
4 else:
5     mediana = (x_ordenado[len(x_ordenado)//2] +
6                x_ordenado[len(x_ordenado)//2-1] ) / 2
7
8 print('Valor da mediana: {:.4f}'.format(mediana))
```

Para computar a moda, precisamos saber quantas vezes cada um dos elementos do conjunto de dados apareceu. Para isso, criamos o que chamamos de "vetor de frequências". Usando uma lista, vamos usar a variável `x_ordenado` para fazer essa contagem. A ideia é a seguinte: vamos criar uma lista `freq` que armazena na posição i, o número de vezes que aparece o elemento de `x_ordenado[i]`. Isso pode ser feito da seguinte maneira:

Programa 12.7 Calculando o vetor de frequências

```
1 freq = []
2 freq.append(1)
3 for i in range(1,len(x_ordenado)):
4     if x_ordenado[i] == x_ordenado[i-1]:
5         freq.append(freq[i-1]+1)
6     else:
7         freq.append(1)
8
9 print('O vetor de freqências: {}'.format(freq))
```

Como a lista `x_ordenado` está ordenada, todos os valores repetidos estão adjacentes. Então, na linha 2 do programa acima, sabemos que o primeiro valor de `x_ordenado` aparece uma única vez. No laço do comando `for`, visitamos a próxima posição dessa lista e se ela for igual ao valor anterior, inserimos em `freq` o valor anterior dessa lista incrementado de uma unidade. Caso contrário, ou seja, apareceu um novo valor na lista `x_ordenado`, então o valor inserido em `freq` é um. Na Figura 12.1, vemos um exemplo de como fica o vetor de frequência para um conjunto de dados.

x_ordenado:	0	2	3.3	3.3	6	7.5	8	8	8	8	11	13.5
freq:	1	1	1	2	1	1	1	2	3	4	1	1

Figura 12.1: Exemplo de um vetor de frequências

Depois disso, vamos achar na lista `freq` qual é o maior valor e a posição correspondente em `x_ordenado` é o valor da moda. No exemplo da Figura 12.1, o maior valor aparece na posição 9 do vetor de frequência, que corresponde ao valor 8, que esta em `x_ordenado[9]`. Só precisamos ter cuidado pois podemos ter mais do que um valor para a moda, ou seja, duas posições diferentes na lista `freq` que têm o valor máximo.

Programa 12.8 Calculando a moda

```
1 maximo = max(freq)
2 moda = []
3 for i in range(len(freq)):
4     if freq[i] == maximo:
5         moda.append(x_ordenado[i])
6
7 print('Valor da moda: {}'.format(moda))
```

No programa acima, a variável `maximo` recebe o valor do maior elemento da lista `freq`, dado pela função `max`, da biblioteca padrão Python. Em seguida,

percorremos cada elemento da lista `freq` e verificamos se ela contém esse valor máximo. Se contém, adicionamos o valor correspondente de `x_ordenado` em um anova lista `moda`. Ao final do laço, os elementos correspondentes à moda, estão armazenados nessa lista.

12.4 Tuplas

Tuplas são estruturas muito semelhantes a listas. A diferença é que, a tupla é "imutável", quer dizer que, uma vez criada, ela não pode ser modificada. Para criar uma tupla, pode-se usar os mesmos recursos que usamos para criar uma lista. Com algumas diferenças na forma de escrever os comandos.

```
>>>t = ('Marcio', 55, 1.73)
>>>r = ()
>>>q = tuple()
>>>p = tuple([1,2,3,4])
>>>
```

No primeiro comando, criamos uma tupla com um string, um número inteiro e um número de ponto flutuante. No segundo e no terceiro, mostramos como criar uma tupla vazia. No quarto comando, criamos uma tupla a partir de uma lista com quatro números inteiros.

Na verdade, podemos representar uma tupla apenas como uma sequência de valores separados por vírgula, ou seja, sem usar os parênteses. A seguir criamos a mesma tupla do exemplo anterior e uma outra com os números inteiros de 1 a 4.

```
>>>t = 'Marcio', 55, 1.73
>>>r = 1,2,3
>>>
```

O acesso aos elementos da tupla é igual ao dos elementos de uma lista. Além disso, podemos utilizar as mesmas funções que aplicamos às listas, desde que a tupla não seja modificada. Vejamos alguns exemplos.

```
>>>t = (5,4,3,2)
>>>t[2]
3
>>>t * 2
(5, 4, 3, 2, 5, 4, 3, 2)
>>>5.0 in t
True
>>>
```

Note que no terceiro comando, o uso da multiplicação não modifica o valor da tupla armazenada na variável `t`. O que ela faz é criar uma nova tupla que repete duas vezes os elementos de `t`. O último comando verifica se 5.0 é igual a algum valor da tupla em `t`. Para fins de comparação, o cinco `float` e `int` representam o mesmo valor, então, o resultado é verdadeiro, ou seja, 5.0 está presente na tupla.

Vamos ver como exemplo, um programa que recebe como entrada um número inteiro e produz como saída o dia da semana correspondentes àquele número. A solução abaixo utiliza uma tupla em que cada elemento é um string que representa o nome de um dia da semana. Depois de ler o número inteiro e verificar se ele está no intervalo correto, basta-nos mostra na saída o valor que está na lista, indexado pelo inteiro lido. Na verdade, temos que diminuir uma unidade desse valor pois os elementos da lista vão de zero a seis.

Programa 12.9 Exemplo de uso de uma tupla

```
1 dia_semana = ('Domingo','Segunda','Terça','Quarta', \
2                'Quinta','Sexta','Sábado')
3
4 d = int(input('Valor numérico do dia da semana: '))
5
6 if  d <= 7 and d >= 1:
7     print(dia_semana[d-1])
8 else:
9     print('Valor invãlido')
```

Note que faz sentido usarmos uma tupla, uma vez que os valores dessa sequência não mudam, por definição. Mesmo que ela seja usada um outros pontos do programa (supondo que o programa continua, fazendo outras coisas), os dias da semana devem permanecer o mesmo, não há motivo para alterar essa variável

12.4.1 Exercícios

1. Como podemos criar uma tupla com um único elemento?

2. Temos uma tupla que contém a mesma sequência de elementos de uma lista. Ao compararmos a igualdade entre a lista e a tupla, qual é o resultado?

12.5 Matrizes

Uma matriz, como todos conhecemos, pode ser utilizada em muitas aplicações como, por exemplo, na resolução de sistemas de equações, como vimo no Capítulo 5 da primeira parte desse livro.

Para representar uma matriz em Python, podemos, também, utilizar listas. Basta pensarmos em uma lista na qual cada elemento é uma lista. Por exemplo, se quisermos definir a matriz

$$M = \begin{bmatrix} 3 & 6 & -1 & 0 & 25 \\ -2 & 3 & 1 & 1 & 6 \\ 1 & -4 & 2 & 2 & 2 \\ -2 & -2 & 0 & 2 & 0 \end{bmatrix}$$

podemos estabelecer que teremos uma lista na qual cada posição corresponde a uma linha completa da matriz. Em código Python, teríamos algo como o quadro abaixo

```
>>> M = []
>>> M.append([3,6,-1,0,25])
>>> M.append([-2,3,1,1,6])
>>> M.append([1,-4,2,2,2])
>>> M.append([-2,-2,0,2,0])
>>>
```

Ou, de forma mais compacta:

```
>>> M = [[3,6,-1,0,25], [-2,3,1,1,6], [1,-4,2,2,2], [-2,-2,
0,2,0]]
>>>
```

Dessa forma, podemos fazer referência à matriz toda, às linhas da matriz e aos elementos individuais da matriz. Por exemplo:

```
>>> M
[[3, 6, -1, 0, 25], [-2, 3, 1, 1, 6], [1, -4, 2, 2, 2],
[-2, -2, 0, 2, 0]]
>>> M[1]
[-2, 3, 1, 1, 6]
>>> M[1][4]
6
>>>
```

Embora não seja muito comum, usando essa representação é possível termos linhas de tamanhos diferentes. Por exemplo, podemos atribuir a `M[1]` uma lista com apenas dois valores, e obtemos o resultado a seguir. Deixamos, porém, de ter uma matriz.

```
>>> M[1] = [-2, 3]
[[3, 6, -1, 0, 25], [-2, 3], [1, -4, 2, 2, 2],
[-2, -2, 0, 2, 0]]
>>>
```

Uma forma alternativa de representar uma matriz seria ter em cada posição da lista `M`, uma coluna da matriz. Então, para representar a matriz usada até aqui como exemplo teríamos:

```
>>> M = []
>>> M.append([3,-1,1,-2])
>>> M.append([6,3,-4,-2])
>>> M.append([-1,1,2,0])
>>> M.append([0,1,2,2])
>>> M.append([25,6,2,0])
>>>
```

Ou, de forma compacta,

```
>>> M = [ [3,-1,1,-2], [6,3,-4,-2], [-1,1,2,0], [0,1,2,2],
[25,6,2,0] ]
>>>
```

Mas, nesse caso, temos o inconveniente que nosso programa deveria acessar os elementos da matriz, trocando os valores de linhas e colunas. Por, exemplo para

acessar o elemento que está na primeira linha, quinta coluna (linha zero coluna quatro, lembrando que as listas são numeradas a partir do zero) deveriamos utilizar a notação abaixo, o que é nada intuitiva, considerando o modo usual de tratarmos as matrizes.

```
>>> M[4,0]]
25
>>>
```

Assim, a não ser que haja algum motivo específico, recomanda-se sempre o uso da primeira abordagem. A Figura 12.2 ilustra a diferença entre elas.

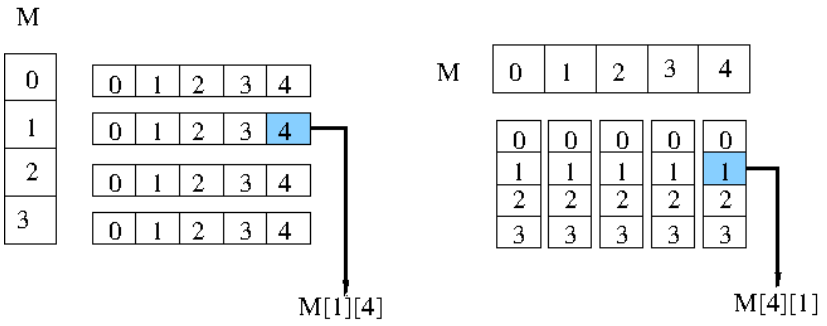

Figura 12.2: Duas formas de representar uma matriz

12.5.1 Exercícios

1. Faça e teste um programa que:

   - leia uma matriz quadrada;
   - some os elementos que estão acima da diagonal principal (inclusive estes); e
   - somar os elementos que estão abaixo da diagonal principal (inclusive estes).

2. Escreva um programa que leia todos os elementos de uma matriz 4×4. A restrição é que os números digitados que forem par devem ser armazenados somente em linhas pares e os números ímpares, somente em linhas ímpares. Quando não houver mais espaço para armazenar um número par ou ímpar, seu programa deve dar uma mensagem e continuar a ler os próximos números, até completar a matriz.

3. Escreva um programa que leia uma lista com 20 elementos. Depois, armazene esses elementos em matrizes de 2, 4 e 5 linhas e mostre as matrizes produzidas.

4. Escreva um programa para jogar o jogo da velha. O programa deve controlar o andamento do jogo com uma matriz 3X3. A cada lance o jogador deve informar qual a posição do tabuleiro que deve ser preenchida. A cada lance o seu programa deve mostrar o tabuleiro no formato abaixo

```
  X  |  O  |
-----+-----+-----
     |  O  |
-----+-----+-----
     |  X  |
```

5. Melhore o programa do exercício anterior de forma que, se houver um ganhador, seu programa interrompa o jogo e avise que o jogo terminou.

6. Escreva um programa que descubra e imprima o maior elemento da linha que contém o menor elemento em uma matriz.

7. Escreva um programa que leia e multiplique duas matrizes. Se as matrizes não puderem ser multiplicadas, seu programa deve dar uma mensagem de erro.

8. Escreva um programa que leia um número inteiro n e depois armazene em uma matriz e exiba as n primeiras linhas do triângulo de Pascal. Veja um exemplo abaixo. Não é difícil descobrir como essa matriz é formada.

```
1
1  1
1  2   1
1  3   3   1
1  4   6   4   1
1  5  10  10   5   1
1  6  15  20  15   6   1
```

12.6 Aplicação: método de Gauss

Como vimos, o método de Gaus usa matrizes para representar um sistema de equações lineares. O algoritmo é dividido em duas partes. A primeira que transforma a matriz de coeficientes em uma matriz triangular. E a segunda, que faz a substituição dos valores calculados para cada variável.

Vamos tomar uma matriz de coeficientes de ordem n. A matriz estendida que representa o sistema de equações tem, então, n linhas e $n + 1$ colunas. Na primeira parte, o programa a seguir utiliza a variável `i` para identificar o número da linha e da coluna do elemento que vai ser usado como pivô. Note que precisamos fazer isso para as linhas/colunas variando de 0 até $n - 2$ pois a última linha não precisa ser tratada pois abaixo dela não há mais elementos a serem zerados.

Para cada elemento na diagonal principal, precisamos fazer duas coisas: 1) escolher um elemento que seja o pivô pois elementos com valor zero não podem ser usados; 2) zerar os elementos que estão na mesma coluna, nas linhas abaixo do pivô.

Programa 12.10 Estrutura da parte inicial do método de Gauss. Transforma a matriz de coeficientes em uma matriz triangular superior

```
1 # M é a matriz estendida
2 M = [...]
3
4 for i in range(len(M)-1):
5     # procura um pivô que possa ser usado
6     ....
7
8     # zera os elementos abaixo do pivô
9     ....
```

Procurar o pivô, significa procurar na coluna `i`, uma linha que tenha um valor não nulo. Essa procura deve começar na linha `i`. Quando achamos a linha que tem essa característica, trocamos essa linha com a linha `i`. Isso garante que na posição `M[i][i]` teremos um pivô não nulo. Se o pivô naquela coluna não existe, ou seja, todos os elementos são zero, então não temos uma solução. O trecho de programa fica como mostrado a seguir.

Programa 12.11 Achando um pivô que possa ser usado

```
1  # M é a matriz estendida
2  M = [...]
3  for i in range(len(M)-1):
4      # procura um pivô que possa ser usado
5      for j in range(i,len(M)):
6          if M[j][i] != 0:
7              break # achou elemento não nulo
8      if j == len(M):
9          # não achou pivô
10         print('Pivô não encontrado na coluna', i)
11         sys.exit()
12     else:
13         # achou pivô na linha j. troca linhas i e j
14         M[j],M[i] = M[i],M[j]
15
16     # zera os elementos abaixo do pivô
17     .....
```

As linhas 5 a 7 incrementam o valor da variável `j` até que seja encontrado um valor diferente de zero na posição `M[j][i]` ou até que `j` atinja valor igual ao número de linhas. Na linha 8 essa condição é verificada. Se `j == len(M)` significa que não conseguimos encontrar o pivô e, então, o programa termina sem uma solução. Caso contrário, fazemos a troca das linhas `i` e `j`.

Aqui utilizamos o conceito de "atribuição de tuplas". Na linha 15 temos do lado esquerdo da atribuição uma tupla de variáveis e do lado direito uma tupla –

com o mesmo número de elementos – de valores. Primeiro, cada um dos valores do lado direito é calculado e então é atribuído para a variável correspondente do lado esquerdo. Assim, é possível trocar os valores de `M[i]` e `M[j]` de forma elegante.

Ao final desse trecho, precisamos zerar os elementos que estão na coluna `i`, em todas as linhas abaixo da linha `i`. Para isso, computamos o valor da variável `d`, que vai ser multiplicado por cada elemento da linha `i` e somado no elemento correspondente da linha `j`. Isso é feito nas linhas 10 e 11 do programa a seguir. Terminado esse trecho de programa, teremos um amatriz triangular superior, como desejamos.

Programa 12.12 Zerando os elementos abaixo do pivô

```
1  # M é a matriz estendida
2  M = [...]
3  for i in range(len(M)-1):
4      # procura um pivô que possa ser usado
5      ....
6      # zera os elementos abaixo do pivô
7      for j in range(i+1,len(M)):
8          d = - (M[j][i] / M[i][i])
9          #repete para cada elemento da linha
10         for k in range(i, len(M[0])):
11             M[j][k] = M[i][k] * d + M[j][k]
```

Por fim, tendo a matriz triangular, precisamos calcular o valor de cada incógnita do sistema de equações. Para isso, vamos usar uma lista que tem o mesmo tamanho da matriz de coeficientes, inicialmente com todos os seus valores iguais a zero (linha 9). Em seguida, o laço da linha 10 atribui à variável `i` o numero de cada uma das linhas que vamos trabalhar, de baixo para cima, ou seja, começando a última linha, até a linha zero.

A cada iteração do laço, uma nova posição da lista `v` é preenchida, começando, também pelas últimas posições. A variável `l` contém a linha que vai ser tratada em uma iteração. A variável `s` tem o valor que está no lado direito dessa mesma equação. Para calcular o valor de `v[i]`, multiplicamos os coeficientes da linha pelos valores das incógnitas já calculadas e subtraímos do valor de `s`. Isso é feito nas linhas 13 e 14. Em seguida, dividimos esse valor pelo coeficiente da incógnita que queremos calcular e isso nos dá o valor desejado (linha 15).

Programa 12.13 Substituindo os valores das incógnitas

```
1  # M é a matriz estendida
2  M = [...]
3  for i in range(len(M)-1):
4      # procura um pivô que possa ser usado
5      ....
6      # zera os elementos abaixo do pivô
7      ....
8  # faz as substituições para calcular as incógnitas
9  v = len(M) * [0]
10 for i in range(len(M)-1, -1, -1):
11     l = M[i]
12     s = l[len(v)]
13     for j in range(i+1, len(v)):
14         s -= l[j] * v[j]
15     v[i] = s / l[i]
16
17 print(v)
```

Para executar o nosso programa com a matriz do início da Seção 12.5 podemos ler cada um dos valores por meio do comando `input` ou podemos inicializar a variável `M` conforme o comando a seguir.

Programa 12.14 Inicialização da matriz

```
1  # M é a matriz estendida
2  M = [[3, 6, -1, 0, 25], [-2, 3, 1, 1, 6], [1, -4, 2, \
3        2, 2], [-2, -2, 0, 2, 0]]
```

Feito isso, executando o programa obtemos os resultados mostrados a seguir.

```
>>> python gauss.py
[3.0000000000000013, 2.4999999999999996, -1.0000000000000004,
5.5]
>>>
```

Exercícios

- Suponha que existe na biblioteca de Python uma função `determinante(m)` que computa e retorna o determinante de uma matriz `m`. Implemente um programa que use essa função para implementar o método de Cramer. Nesse caso talvez seja mais fácil utilizar a representação alternativa de

matriz vista no final da Seção 12.5 pois não precisamos acessar elementos individuais da matriz, apenas "pedaços" dela.

Capítulo 13

Definindo funções

Na Seção 8.5 introduzimos as funções que fazem parte da biblioteca Python. Essas funções facilitam muito nossa vida e economizam muito nosso trabalho. Imagine se, a cada vez que precisássemos computar o seno de um ângulo, tivéssemos que utilizar a série de Taylor, como vimos em um exercício anterior. De qualquer forma, o valor do seno que utilizamos quando invocamos a função `math.sin(x)` é calculado por um trecho de programa que alguém implementou.

A Figura 13.1 mostra o que acontece quando chamamos uma função e ela nos retorna um valor, no caso o seno do número passado como parâmetro. Na Figura 13.1(A), nosso programa é executado até o ponto em que a função é chamada. A execução do nosso programa é interrompida momentaneamente e o interpretador passa a executar o código da função `sin`, que está na biblioteca Python (Figura (B)). Esse programa vai utilizar o valor que passamos como parâmetro para computar o seno, utilizando algum método como a série de Taylor, ou qualquer outro método que, na verdade, não nos interessa. Terminada a função, a execução volta para o nosso programa, devolvendo também o resultado computado pela função (Figura (C)). Nosso programa pode, então, prosseguir a execução .

13.1 Definindo nossas próprias funções

Nós também podemos definir funções dentro dos nossos programas Python. Vamos usar como exemplo o nosso programa que calcula as estatísticas sobre um conjunto de dados. Se olharmos o programa, vemos que temos um código longo, e, talvez, difícil de se compreender. Isso acontece porque temos muitas coisas sendo feitas juntas, uma após a outra.

Vamos, então, tentar dividir as coisas. Vamos supor que temos diferentes funções para:

- ler os dados e armazená-los em uma lista;
- computar a média;
- computar a variância;
- computar mediana; e

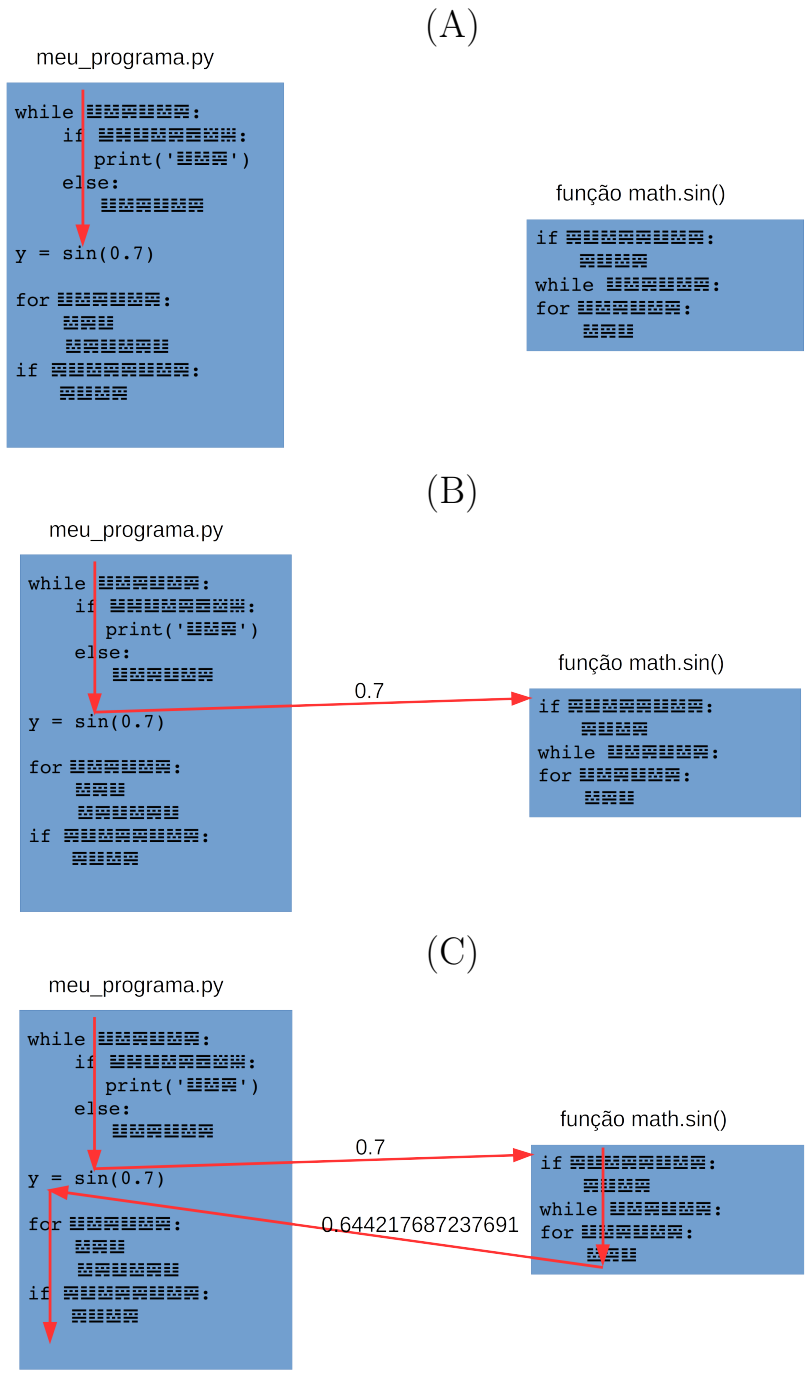

Figura 13.1: Mecanismo de chamada de uma função

- computar a moda.

Se tivermos essas funções disponíveis, podemos reescrever nosso programa

como mostrado a seguir. Percebe-se que o programa ficou muito mais claro e mais elegante.

Programa 13.1 Programa que calcula as estatísticas, usando funções

```
1  import math
2
3  x = le_dados()
4  print('Valor de média: {:.4f}'.format(media(x)))
5  print('Valor de variância: {:.4f}'.format(\
6                                          variancia(x)))
7  print('Valor do desvio padrão: {:.4f}'.format(\
8                                 math.sqrt(variancia(x))))
9  print('Valor de mediana: {:.4f}'.format(mediana(x)))
10 print('Valor de moda: {}'.format(moda(x)))
```

Precisamos, então, implementar aquele código que estava no nosso programa original na forma de funções. Vamos começar pela função que lê os dados de entrada. Para fazê-lo, utilizamos o seguinte comando:

```
1  def le_dados():
```

Com esse comando dizemos que os comandos a seguir fazem parte da definição da função que batizamos de `le_dados`. Os nomes de função devem seguir as mesmas restrições que vimos para nomes de variáveis. Os parênteses que seguem o nome indicam que essa função não utiliza nenhum parâmetro, ou seja, não precisa que nosso programa forneça nenhum dado para que ela faça o que deve fazer.

Vejamos, então, os comandos que executaremos dentro da função `le_dados`. Repare a indentação dos comandos. Todos eles devem estar pelo menos uma indentação além do comando `def`.

Programa 13.2 Função para ler um conjunto de dados e armazená-lo em uma lista

```
1  def le_dados():
2      dados = []
3      r = 0
4      i = 1
5      while r >= 0:
6          msg = 'Digite o valor {}:'.format(i)
7          r = float(input(msg))
8          if r < 0:
9              print('Entrada de dados terminou')
10         elif r > 100:
11             print('Valor deve estar entre 0 e 100')
12         else:
13             dados.append(r)
14             i += 1
15     return dados
```

O significado de cada um dos comandos não é mais novidade. Vamos, então, nos concentrar em alguns pontos que são importantes. O primeiro é que "o que acontece na função, fica na função". Ou seja, todas as variáveis utilizadas na função não fazem sentido fora dela. Por exemplo, se tentarmos utilizar as variáveis `dados` ou `r` ou `i` lá no nosso Programa 13.1 obteremos uma mensagem de erro do interpretador. Por isso são consideradas "locais".

Mas se não podemos usar os dados processados na função, para que ela serve? Existe uma forma de transferir dados que foram obtidos ou calculados dentro da função para o código que a chamou. Isso é feito com o comando `return` na linha 15 da nossa função. Esse comando está dizendo que o resultado da chamada da função é o conteúdo da variável `dados`, que é uma lista. Então, ao executarmos o comando da linha 3 do Programa 13.1, estamos atribuindo à variável `x` esse resultado. Da mesma forma que podemos fazer `y = sin(0.7)`, podemos fazer `x = le_dados()`.

A diferença entre chamar a função do seno e a `le_dados()`, é que para o seno precisamos dizer qual é o ângulo sobre o qual queremos calcular o seno. Significa que precisamos passar um parâmetro para essa função, o que não acontece com a função `le_dados()`, que não necessita de parâmetros. Já a função `media`, que chamamos na linha 4 do Programa 13.1, necessita de um parâmetro, que é a lista de valores sobre a qual desejamos calcular a média. Então, na definição da função `media` precisamos indicar que ela aceita um parâmetro.

Programa 13.3 Função para computar a média de valores armazenados em uma lista

```
1 def media(dados):
2     soma = 0.0
3     for r in dados:
4         soma += r
5     return soma / len(dados)
```

Isso é feito colocando-se entre os parênteses, após o nome da função, uma lista de parâmetros que devem ser usados para chamar-se a função. Se tivermos mais do que um parâmetro, os seus nomes aparecem separados por vírgula. O nome dos parâmetros são tratados como variáveis locais da função.

A função para o cálculo da variância é mostrada a seguir. Não temos muitas novidades a destacar, a não ser o fato que essa função chama a função `media`. Isso é perfeitamente válido, ou seja, ter chamada de uma função dentro de outra função.

Programa 13.4 Função para computar a variância de valores armazenados em uma lista

```
1 def variancia(dados):
2     m = media(dados)
3     var = 0.0
4     for r in dados:
5         var += (r - m) ** 2
6     return var / (len(dados) -1)
```

Podemos melhorar um pouco essa função, deixando o nosso código mais elegante. Nas linhas 5 e 7 do Programa 13.1, chamamos duas vezes a função `variancia`. Na segunda, extraímos o raiz quadrada do resultado para computar o desvio padrão. Usando uma tupla, podemos fazer com que a função `variancia` retorne mais do que um resultado. No caso, a variância e o desvio padrão. A nova versão da função é mostrada a seguir.

Programa 13.5 Função para computar a variância e o desvio padrão de valores armazenados em uma lista

```
1 def variancia(dados):
2     m = media(dados)
3     var = 0.0
4     for r in dados:
5         var += (r - m) ** 2
6     var /= (len(dados) -1)
7     dp = math.sqrt(var)
8     return  (var,dp)
```

E a parte "principal" do nosso programa é mostrada a seguir. Note que temos uma tupla, com duas variáveis que irá receber o retorno da função `variancia`: a variável `v` recebe o primeiro elemento da tupla retornada, que é a variância, e a variável `dp` recebe o segundo, que é o desvio padrão.

Programa 13.6 Chamada de função que retorna uma tupla

```
1 x = le_dados()
2 print('Valor de média: {:.4f}'.format(media(x)))
3 (v,dp) = variancia(x)
4 print('Valor de variância: {:.4f}'.format(v))
5 print('Valor do desvio padrão: {:.4f}'.format(dp))
6 print('Valor de mediana: {:.4f}'.format(mediana(x)))
7 print('Valor de moda: {}'.format(moda(x)))
```

No Programa 13.6, a última linha mostra uma lista ou uma tupla, já que podemos ter mais do que um valor como moda. Por isso, não é possível utilizar o mesmo comando `format` usado nos comandos anteriores. Para melhorar um pouco isso, podemos criar uma função `print_moda`, que mostra os resultados da moda formatados da maneira que desejamos. Note, no programa a seguir, que essa função não retorna nenhum valor. Ela simplesmente faz o processamento necessário e sua execução termina com um comando `return`, sem nenhum valor associado a ele.

Programa 13.7 Função sem retorno de valor

```
1 def mostra_moda(moda):
2     if len(moda) == 1:
3         s = 'O valor da moda é: '
4     else:
5         s = 'Os valores da moda são: '
6     for x in moda:
7         s += '{:.4f} '.format(x)
8     print(s)
9     return
```

13.1.1 Exercícios

1. Complete o programa das estatísticas, implementando as demais funções.

2. Escreva uma função que recebe três parâmetros que representam os lados de um triângulo. O programa deve retornar: 0 se os valores não correspondem a um triângulo; 1 se correspondem a um triângulo equilátero; 2 se correspondem a um triângulo isósceles; 3 se correspondem a um triângulo escaleno.

3. Faça um programa com duas funções (uma para homens e uma para mulheres) que computam o peso ideal, baseado na altura de uma pessoa:
   - para homem: (72.7 x h) – 58
   - para mulher: (62.1 x h) – 44.7
4. Escreva uma função que recebe como parâmetro o valor do salário de um trabalhador e calcula o valor do imposto de renda a ser recolhido na fonte.
5. Escreva uma função que verifica se um determinado ano é ou não bissexto. A função deve retornar o valor False se o ano não for bissexto e True se ele for bissexto.
6. Implemente as funções de seno, cosseno e tangente, usando a série de Taylor, vista anteriormente.
7. Escreva uma função que transforma qualquer ângulo dado em radianos em um ângulo em graus, no intervalo [0 – 360).
8. Utilize funções para simplificar o programa que implementa o método de Gauss (Programas 12.10 a 12.13) para resolução de sistemas de equações.

13.2 Recursão

Quando falamos de uma linguagem de programação, o termo "recursão" refere-se à possibilidade de uma função fazer chamadas a ela mesma. Em termos mais gerais, isso significa que para resolver um determinado problema usamos a solução de um caso mais simples do mesmo problema.

Um exemplo típico é o cálculo do fatorial. Podemos definir $n!$ como:

- a multiplicação de todos os inteiros de 1 até n; ou
- $n \times (n-1)!$. E $0! = 1$

A primeira definição é iterativa. A segunda é recursiva pois para definir o fatorial de um número n, usamos a definição do fatorial de $n-1$. Temos, também, que definir um caso base, que é o caso mais simples, e que serve para dar fim à recursão. Nesse caso, precisamos definir qual o valor de 1!. Dessa forma, se quisermos saber qual o valor de 4!, teremos:

$$4! = 4 \times 3! = 4 \times 3 \times 2! = 4 \times 3 \times 2 \times 1! = 4 \times 3 \times 2 \times 1 \times 0! = 4 \times 3 \times 2 \times 1 \times 1 = 24$$

Se formos traduzir essa ideia em uma função Python, teremos o trecho de programa a seguir. Na execução dessa função, o interpretador vai criando várias chamadas da função `fatorial`, cada uma com um valor menor do parâmetro `n`, até que esse chegue a 0. Quando isso acontece, é possível calcular o valor para todas as chamadas, com valores maiores. Quer, dizer, tendo computado `fatorial(0)`, que retorna o resultado 1, é possível computar `fatorial(1)` que também retorna o valor 1, e assim por diante.

Programa 13.8 Função fatorial recursiva

```
1 def fatorial(n):
2     if n == 0:
3         return 1;
4     return n * fatorial(n-1)
```

Um segundo exemplo de função que pode ser definida recursivamente é o Máximo Divisor Comum entre dois números inteiros. Sabe-se que para computar essa função, valem as seguintes propriedades:

- $MDC(x, x) = x$;
- $MDC(x, y) = MDC(x - y, y)$, se $x > y$;
- $MDC(x, y) = MDC(y, x)$.

A primeira propriedade espelha o caso base, ou seja, é onde devemos chegar para terminar a recursão. A segunda, mostra como calculamos o valor do MDC, baseado em um caso mais simples. E o terceiro nos ajuda no caso em que $x < y$. O programa a seguir mostra como implementar essa função.

Programa 13.9 Função MDC recursiva

```
1  def mdc(x,y):
2      if x == y:
3          return x
4      if x > y:
5          return mdc(x-y, y)
6      return mdc(y,x)
7
8  x = int(input('Entre com o 1o. valor: '))
9  y = int(input('Entre com o 2o. valor: '))
10
11 print('O valor do MDC é: {}'.format(mdc(x,y)))
```

Muitas vezes, a definição de uma função recursiva acaba sendo bem mais simples e mais elegante do que a versão iterativa. Porém, devemos ser cuidadosos na sua utilização. Cada chamada de função em Python vai gastar uma certa quantidade de memória. Se tivermos muita chamadas de funções simultâneas, todas "abertas" ao mesmo tempo, é possível que tenhamos um erro apontado pelo interpretador. Por exemplo, se executarmos o programa do MDC com os valores 1234567 e 890, obtemos o resultado mostrado a seguir.

```
>  python mdc.py
Entre com o 1o. valor: 1234567
Entre com o 2o. valor: 890
Traceback (most recent call last):
  File "mdc.py", line 13, in <module>
    print('O valor do MDC é: '.format(mdc(x,y)))
  File "mdc.py", line 6, in mdc
    return mdc(x-y, y)
  [Previous line repeated 994 more times]
  File "mdc.py", line 3, in mdc
    if x == y:
RecursionError: maximum recursion depth exceeded in
comparison
```

13.2.1 Exercícios

Resolva os problemas abaixo, sempre utilizando funções recursivas.

1. Crie uma função recursiva que receba um número inteiro N e calcule a soma dos números de 1 até N.

2. Escreva uma função recursiva para somar os elementos de uma lista de números. Ou seja, a função recebe uma lista como parâmetro e retorna um número, que é a soma dos elementos da lista.

3. Escreva uma função recursiva para encontrar o maior valor de uma lista de números.

4. Implemente uma função `inverte` que recebe uma lista como parâmetro e inverte a ordem dos seus elementos.

5. Para calcular o resto da divisão inteira entre dois números (função MOD), pode-se usar a seguinte definição:

   - $MOD(x, x) = 0$;
   - $MOD(x, y) = x$, se $x < y$;
   - $MOD(x, y) = MOD(x - y, y)$, se $x > y$.

   Escreva uma função recursiva para calcular a função MOD.

6. Escreva uma função recursiva que determine quantas vezes um dígito K ocorre em um número natural N. Por exemplo, o dígito 2 ocorre 3 vezes em 762021192.

7. Escreva a função recursiva que une duas listas, sem elementos repetidos, ordenadas de forma crescente, considerando que as duas listas não têm elementos em comum. Ou seja, o resultado da função deve ser uma lista ordenada de forma crescente que contém todos os elementos das listas originais.

8. Escreva a função recursiva que une duas listas, sem elementos repetidos, ordenadas de forma crescente, considerando que as duas listas podem ter elementos em comum e que a lista resultante pode ter elementos repetidos. Ou seja, o resultado da função deve ser uma lista ordenada de forma crescente que contém todos os elementos das listas originais.

9. Escreva a função recursiva que une duas listas, sem elementos repetidos, ordenadas de forma crescente, considerando que as duas listas podem ter elementos em comum e que a lista resultante não pode ter elementos repetidos. Ou seja, o resultado da função deve ser uma lista ordenada de forma crescente que contém todos os elementos das listas originais, sem repetições.

13.3 Aplicação: o método de Cramer

Como vimos anteriormente, uma das formas para resolver um sistema de equações, que não seja muito grande, é o método de Cramer. Para aplicar esse método, é essencial computar o determinante de algumas matrizes. Por isso, vamos definir uma função, recursiva, que calcula o determinante pelo método de Laplace, como mostrado na primeira parte deste livro.

No Programa 13.10 vemos essa função, que recebe um único parâmetro, a matriz da qual se deseja o determinante. Se a matriz tem ordem um, ou seja, é composta de um único elemento, então o valor desse elemento é o determinante. Essa condição e verificada nas linhas 2 e 3 da função.

Caso contrário, iremos calcular e somar os cofatores dos elementos da primeira coluna, ou no caso da nossa lista, a coluna de número 0. Isso é feito nas linhas 6 a 14. Note que estamos trabalhando em cima da coluna 0 ou da linha 0, dependendo de como estamos representando a nossa matriz, conforme apresentado na Seção 12.5. Como veremos adiante, nesse programa, escolhemos a representação menos convencional, que é a de uma coluna por posição da nossa lista.

Programa 13.10 Função para calcular o determinante

```
1  def determinante(M):
2      if len(M) == 1:
3          return M[0][0]
4
5      det = 0.0
6      for i in range(len(M)):
7          #faz uma copia da matriz
8          #sem coluna 0 e sem linha i
9          M2 = copia_matriz(M,0,i)
10         if i % 2 == 0:
11             sig = 1
12         else:
13             sig = -1
14         d = determinante(M2)
15         det += M[0][i] * sig * d
16
17     return det
```

A função `copia_matriz` retorna uma matriz `M2` igual à matriz `M`, mas sem a coluna 0 e sem a linha `i`, ou seja, com a ordem uma unidade inferior. Sobre essa nova matriz chamamos a mesma função `determinante`. Note que assim vamos diminuindo a ordem das matriz que necessitamos computar o determinante até chegar à ordem um. Como mencionado anteriormente, o problema é que para matrizes grandes, o número de matrizes torna-se absurdamente grande, fazendo com que torne-se impraticável aplicar este método. Na função, a variável `sig` indica qual o sinal do cofator que devemos usar na soma dos cofatores (1 ou -1). Como usamos sempre a coluna 0, esse valor depende apenas do valor de `i`.

A função `copia_matriz` é apresentada a seguir. Ela é relativamente simples. Inicialmente, faz uma cópia exatamente igual do parâmetro `M`. Em seguida, percorre todas as posições da nova lista `M2`, que também são listas, eliminando os elementos na posição `j` dessas listas. Ou seja, elimina a linha `j` da matriz. Em seguida elimina o elemento na posição `i` da matriz `M2`, eliminando, portanto, toda a linha `i`. Essa função requer que nosso programa contenha um `import copy` para o uso da função `deepcopy`.

Programa 13.11 Função que elimina uma linha e uma coluna da matriz

```
1  def copia_matriz(M, i, j):
2      M2 = copy.deepcopy(M)
3      for l in M2:
4          del l[j]
5      del M2[i]
6      return M2
```

Finalmente, falta implementar o método de Cramer, propriamente dito.

Para isso, inicialmente computamos o determinante da matriz de coeficientes e armazenamos na variável `detM`, no Programa 13.12. Em seguida, substituímos cada coluna dessa mesma matriz, pela última coluna da matriz estendida, que representa os valores à direita da igualdade. Para cada uma dessas substituições, computamos o determinante da matriz obtida e armazenada na variável `A2` (linha 11). Finalmente, dividimos esses determinantes pelo determinante da matriz de coeficientes, obtendo os valores desejados das incógnitas.

Programa 13.12 Programa que implementa o método de Cramer

```
1  #cada posição da lista representa uma COLUNA da matriz
2  A = [[6,5,-2,3], [0,-2,3,1], [0,3,2,1], [2,3,-11,0],
3  [6,5,4,3]]
4  detM = determinante(A[:-1])
5  if detM == 0:
6      print('Sistema sem solução.')
7      sys.exit()
8  b = A[-1]
9  r = []
10 for i in range(len(A)-1):
11     A2 = A[:-1]
12     A2[i] = b
13     det = determinante(A2)
14     r.append(det/detM)
15 print(r)
```

Um ponto importante a ressaltar é que na representação escolhida, cada elemento das listas `A` e `A2` representam uma coluna da matriz. Essa representação não é a mais habitual, como já mencionamos, mas facilita a substituição de cada uma das colunas da matriz de coeficientes, na linha 11 do Programa 13.12.

Capítulo 14

Arquivos

Estamos muito acostumados a lidar com arquivos. Seja um documento de texto, uma planilha, uma figura ou um programa Python, tudo fica armazenado em um arquivo no nosso disco rígido ou outra mídia. Nossos programa também podem interagir com os arquivos, lendo dados a partir deles ou escrevendo dados neles.

Em particular, um tipo de arquivo que vamos utilizar é o "arquivo texto" ou "arquivo de texto". Como o nome sugere, nele armazenamos apenas texto, ou seja, uma sequência de caracteres. Podemos pensar no arquivo como um string, que pode ser muito grande, e que está armazenado no disco do nosso computador. Em geral, usamos para esse tipo de arquivo a extensão ".txt". Mas outras categorias de arquivos também são simplesmente texto como, por exemplo, os arquivos ".py" que temos usado.

Para criar ou editar um arquivo texto podemos usar vários programas como o Bloco de Notas do Windows ou mesmo o Geany ou outro ambiente de programação. Nas próximas seções vamos ver algumas das operações que podemos fazer sobre os arquivos texto. Daqui pra frente vamos nos referir a eles apenas como "arquivos".

14.1 Abrindo e fechando

Para identificar um arquivo armazenado no nosso computador precisamos saber seu nome e precisamos saber onde (qual pasta) ele está. Assim, todo arquivo tem um 'nome absoluto' que indica as esses duas coisas. Por exemplo, usando o nosso Bloco de Notas do Windows (Figura 14.1), vamos criar um arquivo "Poema.txt" e vamos armazená-lo na pasta "C:\Users\Eng\Documents". O arquivo contém parte do poema "O Engenheiro" de João Cabral de Melo Neto.

Para acessar esse arquivo dentro de nosso programa Python usamos a função `open` da biblioteca padrão. Para essa função devemos passar o nome do arquivo que queremos acessar e ela devolve um objeto que vai identificar, dentro do programa, esse arquivo. Por exemplo:

Programa 14.1 Abrindo um arquivo

```
1 f = open('C:\Users\Eng\Documents\Poema.txt')
```

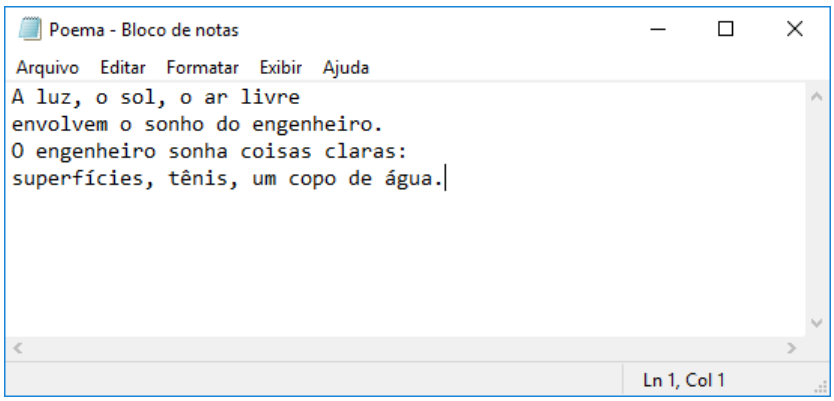

Figura 14.1: Arquivo texto editado no Bloco de Notas

A partir desse ponto, qualquer uso da variável `f` irá referenciar o arquivo *Poema.txt*. Costumamos dizer que o arquivo está "aberto" para podemos trabalhar com ele.

Podemos, também, identificar o nome do arquivo em relação à localização do nosso programa. Por exemplo, se o arquivo estiver na mesma pasta que o nosso programa, podemos simplificar e escrever:

Programa 14.2 Abrindo um arquivo, usando seu nome relativo ao programa

```
1 f = open('Poema.txt')
```

Quando terminarmos de usar o arquivo, precisamos fazer a operação contrária: devemos fechar o arquivo. Isso significa que deixamos de ter acesso àquele arquivo por meio da variável que usamos com a função `open`. Essa operação é importante pois ela avisa o sistema operacional que nosso programa deixará de acessar o arquivo. Ela evita, também, que nosso arquivo fique desatualizado ou seja corrompido. A seguir, o esquema geral de como utilizamos um arquivo.

Programa 14.3 Abrindo e fechando um arquivo

```
1 f = open('Poema.txt') #abre o arquivo
2
3 # aqui nosso programa acessa o arquivo
4
5 f.close() #fecha o arquivo
```

É importante notar que depois de fechado, não podemos mais usar a variável `f`. Se tentarmos fazer isso, o interpretador Python irá indicar um erro.

14.2 Lendo dados

Vamos admitir que o nosso arquivo *Poema.txt* foi criado e reside na mesma pasta do nosso arquivo. Agora, queremos ler o seu conteúdo, ou seja, usar aquilo que está no arquivo, dentro do nosso programa. Da mesma forma, existem funções específicas para se fazer isso.

A primeira forma de enxergarmos o arquivo é como um grande string. Assim, podemos simplesmente "ler" esse string do arquivo para dentro do nosso programa e, então, manipulá-lo como quisermos. Isso é feito pela função `read`.

Programa 14.4 Lendo dados de um arquivo

```
1 f = open('Poema.txt') #abre o arquivo
2 s = f.read()
3 print(s)
4 f.close() #fecha o arquivo
```

Ao ler o arquivo na linha 3, transferimos todo o seu conteúdo para a variável `s`. O comando `print` a seguir, vai apresentar esse conteúdo, como mostrado a seguir. Uma coisa a ser destacada é que nesse string existem alguns caracteres de final de linha, o tal de "\n" que vimos anteriormente. Por isso, a saída exibida aparece em quatro linhas.

```
> python poema.py
A luz, o sol, o ar livre
envolvem o sonho do engenheiro.
O engenheiro sonha coisas claras:
superfícies, tênis, um copo de água.
```

Uma outra forma de enxergar o arquivo texto é como uma sequência de linhas. Assim, podemos, também, ler o arquivo separando as suas linhas em uma lista usando a função `readlines`. Nesse caso, o retorno da chamada dessa função é uma lista na qual cada elemento é um string que corresponde a uma linha do arquivo.

Programa 14.5 Lendo linhas de um arquivo

```
1 f = open('Poema.txt') #abre o arquivo
2 linha = f.readlines()
3 print(linhas)
4 f.close() #fecha o arquivo
```

A saída desse programa é mostrada a seguir. A lista `linhas` tem 4 elementos, cada um é um string que contém uma linha do arquivo.

```
> python poema2.py
['A luz, o sol, o ar livre\n', 'envolvem o sonho do
engenheiro.\n', 'O engenheiro sonha coisas claras:\n',
'superfícies, tênis, um copo de água.\n']
```

Uma função, que não está diretamente ligada com arquivos, mas que é muito útil quando lemos dados deles é a `split`, que é aplicada um um objeto do tipo string. Ela quebra o string, colocando cada "palavra" em uma posição de uma lista. Por exemplo:

Programa 14.6 Lendo e quebrando as palavras de um arquivo

```
1 f = open('Poema.txt') #abre o arquivo
2 s = f.read()
3 palavras = s.split()
4 print(palavras)
5 f.close() #fecha o arquivo
```

Esse programa vai criar uma lista, que é mostrada na saída abaixo. Veja que cada espaço em branco do string é usado para separar as palavras. A saída é mostra a seguir.

```
> python poema3.py
['A', 'luz,', 'o', 'sol,', 'o', 'ar', 'livre', 'envolvem',
'o', 'sonho', 'do', 'engenheiro.', 'O', 'engenheiro',
'sonha', 'coisas', 'claras:', 'superfícies,', 'tênis,', 'um',
'copo', 'de', 'água.']
```

Essa função é útil, por exemplo, se tivermos um arquivo que contém os dados de consumo de cerveja de uma turma de estudantes de engenharia. O arquivo poderia ser representado da seguinte maneira:

```
 1.5  2  2  4
0  2.5  3.5
2  6  3.5
```

Então, para ler esses dados com o o intuito de calcular, por exemplo, as estatísticas que vimos anteriormente, bastaria implementar a seguinte função. Na linha 3 o arquivo é lido e o string retornado é quebrado e colocado em uma lista. Em seguida, cria-se uma nova lista, na qual os elementos, transformados para

`float`, são inseridos. Isso é necessários pois queremos uma lista com números e não com strings.

Programa 14.7 Lendo números em uma lista

```
1 def le_dados(arquivo):
2     f = open(arquivo)
3     s = f.read().split()
4     f.close()
5     r = []
6     for c in s:
7         r.append(float(c))
8     return r
```

14.3 Lendo com o comando `for`

Podemos ter arquivos muito grandes, com milhares de linhas. Ao usar a função `read` todo o conteúdo do arquivo é transferido para o nosso programa, ou seja, para a memória do computador. Isso pode ser um problema pois podemos não ter tanta memória disponível. Por isso, podemos adotar a estratégia de ler e processar apenas uma linha por vez.

No programa abaixo, a função `readline` é usada para ler uma linha do arquivo. Para saber até quando devemos ler, ou seja, para descobrir quando o arquivo lido terminou, usamos o tamanho do string devolvido pela função `readline`. Quando esse string for vazio, ou seja, tiver tamanho zero, então, terminamos de ler o arquivo.

Programa 14.8 Lendo linhas com `readline`

```
1 f = open('numeros.txt')
2 s = f.readline()
3 while len(s) > 0:
4     # usa string em s
5     s = f.readline()
6 f.close()
```

Uma forma mais simples de fazer isso é utilizando o comando `for`. Assim como fazemos com listas, podemos usar esse comando para percorrer cada elemento – no caso, cada linha – do arquivo. No programa a seguir fazemos isso.

Programa 14.9 Lendo linhas com o comando `for`

```
1 f = open('numeros.txt')
2 for s in f:
3     # usa string em s
4 f.close()
```

14.4 Escrevendo dados em um arquivo

Nosso programa pode, também, usar arquivos para armazenar dados que foram gerados. Para isso, algumas diferenças existem em relação à leitura de arquivos. A primeira, é que precisamos, no nosso programa, avisar que não vamos ler dados do arquivo, mas sim, vamos colocar dados nele. Isso é feito ao abrirmos o arquivo.

Programa 14.10 Abrindo o arquivo para escrever

```
1 f = open('Poema2.txt', 'w')
2 f.write('O lápis, o esquadro, o papel;\n')
3 f.write('o desenho, o projeto, o número:\n')
4 f.write('o engenheiro pensa o mundo justo,\n')
5 f.write('mundo que nenhum véu encobre.\n')
6 f.close()
```

No programa acima, a variável `f` refere-se a um arquivo no qual podemos gravar dados. O segundo parâmetro do comando `open`, ou seja, o string "w" indica justamente que o arquivo vai ser usado para escrever dados, não para ler. Isso permite que, nas linhas seguintes possamos usar o comando `write` para escrever as quatros linhas seguintes do poema de João Cabral de Melo Neto.

De certa forma, o comando `write` se assemelha ao comando `print`, que já conhecemos muito bem. Algumas diferenças, porém, precisam ser apontadas:

- o comando aceita um único parâmetro;
- esse parâmetro tem que ser, necessariamente, um string;
- o comando `write` não insere automaticamente um caractere de final de linha como é o comportamento padrão do `print`. Por isso, nos comandos do nosso programa temos um "\n" no final de cada string que escrevemos no arquivo;
- o string, no caso do `write` vai aparecer no arquivo correspondente e não na saída do nosso programa, como no caso do `print`.

O Programa 14.10 poderia utilizar apenas um comando `write` em vez de quatro. Bastaria termos um único string com as quebras de linha posicionadas corretamente e o resultado seria o mesmo. Mostramos esse programa a seguir.

Programa 14.11 Usando um único `write`

```
1 f = open('Poema2.txt', 'w')
2 f.write('O lápis, o esquadro, o papel;\no desenho, \
3 o projeto, o número:\no engenheiro pensa o mundo \
4 justo,\nmundo que nenhum véu encobre.\n')
5 f.close()
```

No caso de querermos escrever outros tipos de dados que não sejam strings precisamos transforma-los em strings antes. No exemplo a seguir, vamos tomar uma lista de números do tipo `float` que representam, por exemplo, o consumo de cerveja dos alunos de uma turma de engenharia e vamos escrevê-los em uma arquivo. Para complicar um pouco, vamos sempre escrever 3 valores em cada linha do arquivo e cada valor formatado com duas casas decimais.

Programa 14.12 Escrevendo números de uma lista

```
1 def grava_dados(arquivo,lista):
2     f = open(arquivo, 'w')
3     i = 0
4     for c in lista:
5         f.write('{:7.2f}'.format(c))
6         i += 1
7         if i % 3 == 0:
8             f.write('\n')
9     f.close()
```

Para uma lista como $[1.5, 2, 2, 4, 0, 2.5, 3.5, 2, 6, 3.5]$ teríamos a saída, gravada no arquivo cujo nome é passado por parâmetro, mostrada a seguir.

```
 1.50   2.00   2.00
 4.00   0.00   2.50
 3.50   2.00   6.00
 3.50
```

14.5 Modos de abertura

O segundo parâmetro usado na função `open` indica como queremos ou para que queremos abrir o arqui. Por isso é chamado de "mode de abertura".

Existem várias formas de abrir um arquivo. A seguir descrevemos aquelas mais usuais e que mais nos interessam.

Modo "r": indica que o arquivo vai ser usado para ler dados. Nesse caso, o arquvo precisa existir no sistema de arquivos. Caso ele não exista, o interpretador vai abortar a execução do programa indicando um erro de execução, como vemos a seguir.

```
>>>f = open('arquivo_nao_existe.txt', 'r')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory:
'arquivo_nao_existe.txt'
```

Esse modo de abertura é o modo padrão. Ou seja, se passarmos apenas um parâmetro para a função `open` assume-se que esse será o modo de abertura do arquivo.

Modo "w": indica que queremos ler dados do arquivo. Nesse caso, se arquivo não existe não é um problema pois o arquivo é criado, sem nenhum dado nele. Se o arquivo existe, ele é descartado e um novo arquivo, com o mesmo nome é criado, sem nenhum dado nele. Isso quer dizer que se executarmos duas vezes o Programa 14.10 vamos, na primeira vez, criar o arquivo e nele escrever o poema. Na segunda vez, o arquivo é deletado, em seguida criado e por fim escrevemos o poema nele. Ou seja, no final, teremos o arquivo contendo as quatro linhas do poema, exatamente como se tivessemos executado o programa uma única vez.

Ainda assim, podemos ter um erro ao executarmos a função `open`, se tentarmos criar o arquivo em uma pasta que não existe ou em uma pasta que não temos autorização para modificar. No exemplo abaixo, tentamos criar o arquivo na pasta "inexiste", que não existe.

```
>>>f = open('inexiste/Poema.txt', 'w')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory:
'inexiste/Poema.txt'
```

Modo "a": é o modo para adicionar coisas no final do arquivo. Se o arquivo não existe, ele é criado, desde que seja em uma pasta válida. Se o arquivo já existe, o seu conteúdo não é descartado e tudo que nosso programa escreve no arquivo vai para o final do conteúdo que já existe. Então, se executarmos duas vezes o Programa 14.10 mas com o modo de abertura "a" em vez do "w", teríamos o seguinte conteúdo no arquivo *Poema2.txt*:

```
O lápis, o esquadro, o papel;
o desenho, o projeto, o número:
o engenheiro pensa o mundo justo,
mundo que nenhum véu encobre.
O lápis, o esquadro, o papel;
o desenho, o projeto, o número:
o engenheiro pensa o mundo justo,
mundo que nenhum véu encobre.
```

Na primeira execução, são gravadas as quatro primeiras linhas do arquivo. Na segunda execução, nosso programa iria abrir o arquivo e começar a escrever no seu final, ou seja, a segunda cópia do poema, que vemos acima.

14.5.1 Exercícios

1. Faça um programa que crie um arquivo texto, com o nome "dados.txt", e escreva neste arquivo em disco uma contagem que vá de 1 até 100, com um número em cada linha. Abra este arquivo em um editor de textos, como por exemplo o Notepad ou o Geany do Windows.

2. Faça um programa que leia um arquivo texto do disco, lendo linha a linha, e exibindo cada uma das linhas numeradas na tela. A ideia é podermos pegar um arquivo texto qualquer (pode ser inclusive o arquivo do programa fonte – arquivo ".py") e mostrar na tela com as linhas numeradas. As primeiras linhas do arquivo iriam ser exibidas na tela da seguinte forma:

```
001:  def h_peso_ideal(altura):
002:      peso = altura * 72.7 - 58.0
003:   return peso
004:
005:  def m_peso_ideal(altura):
006:      peso = altura * 62.1 - 44.7
007:      return peso
```

3. Complemente o programa anterior para que ele grave o resultado em um outro arquivo.

4. Suponha arquivo "mega-sena.txt" tem os números sorteados em todos os concursos da mega sena, um concurso em cada linha do arquivo. Escreva um programa que leia esse arquivo e descubra qual a dezena que mais apareceu e qual a que menos apareceu.

5. Escreva um programa que lê um arquivo texto e exibe: o número de linhas, o número total de caracteres, o número de espaços em branco e o número de caracteres não brancos encontrados. O nome do arquivo de entrada deve ser digitado pelo pelo usuário. Sugestão: use a função isspace para verificar se um caractere é um espaço.

6. O seu professor gostaria de avaliar o desempenho de uma turma, em relação às notas de uma avaliação. Para isso, criou o arquivo "notas.txt", com o seguinte formato, representando o número de matrícula dos alunos e as notas obtidas:

   ```
   9654819 5.2
   7788950 5.2
   9560923 9.7
   6095479 0.6
   8006694 5.5
   9569242 8.7
   9889433 9.9
   ```

   Escreva um programa que mostre o seguinte relatório:

   ```
   ------------------------------
   Matrícula      Nota   Difer.
   9654819        5.20    -1.20
   7788950        5.20    -1.20
   9560923        9.70     3.30
   6095479        0.60    -5.80
   8006694        5.50    -0.90
   9569242        8.70     2.30
   9889433        9.90     3.50
   ------------------------------
   Média: 6.40
   Acima da média: 3
   Abaixo da média: 4
   ```

   A terceira coluna representa a diferença entre a nota do aluno e a média. Notas maiores ou iguais à médias contam como"Acima da média". O relatório deve ser exibido na tela do computador e também gravado em um arquivo "relatório.txt" Use funções para: 1) calcular a média; 2) calcular a terceira coluna do relatório; 3) mostrar os relatórios.

7. Um linguista muito famoso está fazendo a análise dos nomes dos alunos da sua universidade e para isso, criou um arquivo "nomes.txt" com o seguite formato:

   ```
   Alecio Navarro
   Ályson Buarque
   Amanda de Mello
   Ana Flávia Correia
   Ana Julia Picard
   Ana Luiza Severiano
   Bruna Nordsveri
   ```

   ou seja, um nome por linha. Agora ele quer saber a frequência com que cada nome aparece. Para isso, você deve criar um programa que gera o seguinte relatório:

```
---------------------------------
Nome              Ocorr.  % total
Alecio                 1    14.29%
Ályson                 1    14.29%
Amanda                 1    14.29%
Ana                    3    42.86%
Bruna                  1    14.29%
---------------------------------
Média ocorrencias: 1.40
Média percentagens: 20.00
```

A terceira coluna representa a porcentagem de vezes que o nome apareceu, no total de alunos analisados. O relatório deve ser exibido na tela do computador e também gravado em um arquivo "relatório.txt". Use funções para: 1) contar quantas vezes cada nome aparece; 2) mostrar os relatórios; 3) outras que você achar necessário.

14.6 Manipulação externa de arquivos

A biblioteca Python possui um módulo chamado `os` que permite que comandos do sistema operacional sejam executados de dentro dos nossos programas, utilizando-se funções pré-definidas. A descrição desse módulo pode ser encontrada em `https://docs.python.org/3/library/os.html`.

Um conjunto particularmente interessante é das funções que manipulam pastas e arquivos. A sua descrição pode ser encontrada em `https://docs.python.org/3/library/os.html#os-file-dir`. Vamos a seguir descrever algumas dessas funções.

Quando executamos um programa Python usando o interpretador, essa execução possui uma pasta corrente. Quando executamos um programa de dentro de um ambiente de programação, esse ambiente acerta as coisas de modo que a pasta corrente da execução seja a pasta onde nosso programa está. Então, executando o seguinte programa

Programa 14.13 Exibindo a pasta corrente

```
1 import os
2
3 print(os.getcwd())
```

obtemos a saída mostrada a seguir, supondo que o arquivo do nosso programa está na pasta `/home/user/PythoParaEng/`. Note que a responsável por retornar um string que contém a pasta corrente é a função `os.getcwd`.

```
/home/user/PythonParaEng
```

De modo diverso, se fizermos a chamada do programa a partir do console do sistema operacional, o valor retornado pela função `os.getcwd`, ou seja, a pasta corrente do programa é a pasta corrente do console que fez a chamada. Por exemplo:

```
> pwd
/home/user
> python PythonParaEng/testa_getcwd.py
/home/user
```

Note que nesse exemplo, a pasta corrente do console é `/home/user` e executando-se o programa *testa_ getcwd.py* que está dentro da pasta *PythonParaEng*, obtemos como resposta não a pasta onde está o arquivo mas sim a pasta corrente.

A função `os.chdir` serve para mudar o diretório corrente. Ela recebe como parâmetro um string que representa a pasta que deve passar a ser a pasta corrente. Esse string pode ser especificado em relação à pasta corrente do programa – se não inicia com uma "/" – ou como um caminho absoluto, caso contrário. No exemplo a seguir, vemos os dois caso. Na linha 3, altera-se a pasta corrente para *PythonParaEng/dados*, que está dentro da pasta corrente. No segundo caso, linha 4, a pasta corrente passa a ser */home/user/PythonParaEng*, qualquer que seja a pasta corrente. Nas linhas 5 e 6, mostramos que os os nomes especiais de pasta ".." e "." também podem ser usados com essa função. O primeiro faz com que a pasta corrente seja aquela "acima" da pasta corrente o segundo simplesmente não altera a pasta corrente.

Programa 14.14 Alterando a pasta corrente

```
1 import os
2
3 os.chdir('PythonParaEng/dados')
4 os.chdir('/home/delamaro')
5 os.chdir('..')
6 os.chdir('.')
```

No caso em que a mudança de pasta não possa ser feita – por exemplo por não existir nova pasta ou por não ter o programa acesso a ela – um erro será apontado pelo interpretador Python.

Uma função muito útil é `os.listdir`. Ela retorna uma lista que contém todas as entradas – arquivos e subpastas – presentes em uma determinada pasta. Essa pasta deve ser passada como parâmetro ou, caso não seja, a função devolve o resultado relativo à pasta corrente do programa. No exemplo a seguir, nosso programa obtém a lista de entradas da pasta corrente e, em seguida, mostra o nome de cada uma delas, indicando se é um arquivo ou uma subpasta. Para isso, usamos as funções `os.path.isdir` e `os.path.isfile`.

Programa 14.15 Identificando as entradas de uma pasta

```
1 import os
2 import os.path
3
4 entradas = os.listdir()
5 for nome in entradas:
6     if os.path.isdir(nome):
7         print('{} é uma pasta'.format(nome))
8     elif os.path.isfile(nome):
9         print('{} é um arquivo'.format(nome))
```

Outras funções para manipulação de arquivos e pastas estão sumarizadas na Tabela 14.1.

Nome da função	**Significado**
`os.mkdir(pasta)`	Cria uma nova pasta.
`os.remove(arquivo`	Deleta um arquivo
`os.rename(arq1,arq2)`	Traca o nome do arquivo `arq1` para `arq2`
`os.rmdir(pasta)`	Deleta uma pasta, que tem que estar vazia
`os.unlink(arquivo)`	O mesmo que `os.remove(arquivo)`

Tabela 14.1: Algumas funções para manipulação externa de arquivos e pastas

14.6.1 Exercícios

Capítulo 15

Exceções

Quando programamos, às vezes, nem tudo ocorre como esperamos. Por exemplo, no programa a seguir, o que acontece se o usuário digitar algum string que não corresponde a um número inteiro?

Programa 15.1 Entrada de um número inteiro

```
1 n = int(input('Quantos valores serão digitados? '))
2 ...
```

Como já vimos anteriormente, o interpretador Python vai terminar a execução do programa já na linha 1 – mais exatamente na execução da função `int` – e vai mostrar qual foi o erro que ocorreu. No caso, supondo que o usuário digitou "13o" em vez de "130', a mensagem dada é:

```
ValueError: invalid literal for int() with base 10: '13o'
```

Vários tipos de erros podem acontecer, como também já vimos anteriomente. Esses erros, no contexto de Python são chamados de "Exceções". Cada exceção tem um tipo específico. Para saber o tipo de uma exceção, basta olhar o nome que aparece no início da mensagem de erro. No caso acima, temos uma exceção chamada `ValueError`.

Para ter uma noção das exceções que são definidas no ambiente Python, o leitor pode consulta a documentação da linguagem em `https://docs.python.org/3/library/exceptions.html`.

15.1 Tratamento de exceções

Se soubermos os tipos de exceções que podem ocorrer nos comandos dos nossos programas, podemos tentar contorná-las, ou melhor dizendo, tratá-las.

Por exemplo, no Programa 15.1, sabemos que se o usuário digitar alguma coisa que não corresponde a um inteiro, teremos uma exceção do tipo `ValueError`. Então, em vez de deixar o interpretador Python indicar um erro, podemos tomar

a seguinte medida, dentro do nosso programa: pedimos para o usuário digitar novamente o número, até que ele acerte. Para implementar isso podemos ter a seguinte função.

Programa 15.2 Tratando uma exceção

```
1  def le_int(msg):
2      while True:
3          try:
4              n = int(input(msg))
5              return n
6          except ValueError:
7              print('Valor digitado não é válido')
8
9  n = le_int('Quantos valores serão digitados? ')
10 print(n)
11 ...
```

Nesse programa, temos um laço que só termin quando o comando `return` é executado. No comando `try` vamos fazer a leitura usando o `input` e vamos transformar string retornado em um `int`. Mas se isso não funcionar, ou seja, se um `ValueError` acontecer, a execução do programa continua no comando `except`. Ou seja, em vez de deixar o interpretador exibir o erro e terminar a função, nosso programa vai exibir uma mensagem (linha 7) e a execução continua: uma nova iteração do `while` vai fazer com que seja solicitada uma nova digitação. Até que não ocorra uma execeção. Um exemplo é mostrado a seguir.

```
>  python le_inteiro.py
Quantos valores serão digitados? 3o
Valor digitado não é válido
Quantos valores serão digitados? abc
Valor digitado não é válido
Quantos valores serão digitados? 30
30
>
```

Duas coisas são importantes de se notar no Programa 15.2, em relação ao tratamento de exceções:

- a primeira, é que os comandos dentro do comando`try` vão sendo executados sequencialmente, como sempre, e a ocorrência de uma exceção termina a execução desse comando. Então, no nosso exemplo, se houver um erro no comando da linha 4, o comando da linha 5 não é executado;
- a segunda é que os comandos dentro do `except` só são executados se o erro que ocorre dentro do `try` for do tipo especificado, ou seja um `ValueError`.

Se outro tipo de error ocorrer, nosso programa não está pronto para tratá-lo e, então, o interpretador Python é quem vai interromper a execução e mostrar a mensagem de erro que já conhecemos.

Também é possível ter, no mesmo comando `try`, tratamento para diferentes tipos de exceções. No exemplo a seguir, o usuário é solicitado a digitar um número inteiro, na linha 2. Caso ocorra um erro ele é tratado a partir da linha 6. Caso contrário, a função `open` é executada. Ela também pode gerar uma exceção, que seria tratada na linha 4. Depois disso, o programa continua na linha 8.

Programa 15.3 Tratando mais do que uma exceção

```
1 try:
2     n = int(input('Digite um número inteiro: '))
3     f = open('nao_existe.txt')
4 except FileNotFoundError:
5     print('Arquivo não existe')
6 except ValueError:
7     print('Valor inválido')
8 ...
```

Podemos ter, também, um comando `except` que trate todos os possíveis erros que possam acontecer dentro do `try`. Isso é feito não especificando, no `except` qual seria a exceção a ser tratada. Como mostrado a seguir.

Programa 15.4 Tratando uma exceção genérica

```
1 try:
2     n = int(input('Digite um número inteiro: '))
3     f = open('nao_existe.txt')
4 except :
5     print('Algum erro ocorreu')
6 ...
```

Um ponto importante que ainda não foi mencionado é que uma exceção, quando ocorre e não é tratada, vai fazer com que a função corrente seja terminada e a exceção seja "transmitida" para o ponto que fez chamada da função, que tem, também a oportunidade de fazer o tratamento. Apenas no caso em que ninguém faz esse tratamento, então o interpretador irá mostrar aquela conhecida mensagem de erro.

No programa a seguir, apenas para ilustração, temos uma função que lê um string e o transforma para `int`. Como vimos, isso pode gerar uma exceção e pode terminar a execução da função. Mas, na parte principal do nosso programa, podemos tratar essa exceção, por exemplo, chamar a função novamente.

Programa 15.5 Exceção sendo propagada para o ponto de chamada

```
1  def le_int(msg):
2      n = int(input(msg))
3      return n
4
5  while True:
6      try:
7          k = le_int('Digite um número inteiro: ')
8          break
9      except ValueError:
10         print('Valor inválido')
```

Então, no Programa 15.5 continuamos chamando a função `le_int` que não faz tratamento de possíveis exceções. Quem faz o tratamento é quem chama a função `le_int`, ou seja, a parte principal do nosso programa.

15.2 Gerando uma exceção

Não são apenas as funções da biblioteca Python que podem gerar exceções. Podemos programar nossas funções para que elas possam criar exceções caso alguma situação anormal seja detectada.

Vamos supor que queremos uma função que leia um número inteiro a partir do teclado e, além disso, seja feita uma validação como, por exemplo, verificar se esse número está num determinado intervalo. Podemos organizar nosso programa da seguinte maneira:

Programa 15.6 Gerando uma exceção

```
1  def le_int(msg):
2      n = int(input(msg))
3      if n < 0 or n > 100:
4          raise ValueError()
5      return int
6
7  while True:
8      try:
9          k = le_int('Digite um número inteiro: ')
10         break
11     except ValueError:
12         print('Valor inválido')
```

O comando `raise` é usado para "lançar" a exceção. Ou seja vai terminar a execução da função e vai fazer com que uma nova exceção seja disponibilizada e, nesse caso, tratada na linha 11 do nosso programa.

É possível associar uma mensagem a uma exceção. Ela deve indicar porque aquela exceção ocorreu. Por exemplo, se o número digitado estiver fora do intervalo válido, podemos indicar esse fato. Essa mensagem será mostrada pelo interpretador Python, caso a exceção não seja tratada. Isso é mostrado no programa a seguir e na saída do programa, mostrada na sequência.

Programa 15.7 Gerando uma exceção com mensagem

```
1 def le_int(msg):
2     n = int(input(msg))
3     if n < 0 or n > 100:
4         raise ValueError('Valor fora do intervalo: '
5                                 + str(n))
6     return int
7
8 le_int('Digite um número inteiro.')
```

```
> python le_inteiro2.py
Traceback (most recent call last):
  File "except4.py", line 13, in <module>
    le_int('Digite um inteiro: ')
  File "except4.py", line 4, in le_int
    raise ValueError('Valor fora do intervalo: ' + str(n))
ValueError: Valor fora do intervalo: 101
>
```

A última linha da saída reflete a mensagem que foi utilizada para criar a exceção. Ela mostra, inclusive, o valor que foi digitado pelo usuário e que está fora do intervalo.

15.2.1 Exercícios

1. Escreva um programa que cria uma lista com 10 elementos de ponto flutuante, todos iguais a zero. Em seguida, seu programa deve pedir ao usuário que digite vários valores de ponto flutuante e indique em que posição ele deve ser colocado na lista. Use o tratamento de exceções para os casos em que:

   a) o usuário digitar alguma coisa inválida; ou

   b) a posição indicada não existir; ou

   c) a posição indicada já estiver ocupada.

   O programa termina quando todas as posições da lista estiverem ocupadas.

15.3 O comando `finally`

15.3.1 Exercícios

Capítulo 16

Conjuntos e dicionários

Capítulo 17

Pacotes para engenheiros

Apêndice A

Complementos de Python

Nesse apêndice vamos mostrar como instalar e utilizar alguns softwares que ajudam a trabalhar com o ambiente Python. São ambientes de programação, um instalador de software e um criador de ambientes virtuais.

A.1 Ambientes de programação

Uma das muitas vantagens da linguagem de programação Python é que não é a sua facilidade para escrever programas. É possível escrever *scripts* Python utilizando um editor de texto convencional. Contudo, também é possível utilizar ambientes de desenvolvimento integrados (IDEs), que possuem uma série de funcionalidades, como editor de código-fonte, ferramentas para automatizar a execução do código, ferramenta de *debug*, ferramenta de auto-complete de código, dentre uma série de outras características.

A.1.1 IDLE

O IDLE é o Ambiente de Desenvolvimento e Aprendizagem Integrado do Python. É uma IDE básica e possui os seguintes recursos:

- desenvolvido em Python 100% puro, usando o *toolkit* `GUI tkinter`;
- multi-plataforma, funcionando basicamente da mesma maneira nos sistemas operacionais *Windows*, *Unix* e *Mac OS*;
- interpretador interativo com colorização código, saída e mensagens de erro;
- editor de texto com vários recursos, como: desfazer, identação inteligente, dicas de chamada, preenchimento automático de codigo;

Instalando o IDLE

Para o sistema operacional *Windows*, ao instalar o kit de desenvolvimento Python, o IDLE será automaticamente instalado.

Para sistemas operacionais *Linux* e *macOS*, que já possuem nativamente uma versão do Python instalada, para instalar o IDLE é necessário realizar a execução do comando[1] abaixo:

```
> sudo apt-get install idle-python3.7
```

Visão geral

Ao iniciar o IDLE, Figura A.1, a janela principal do IDLE será visualizada. Tal janela corresponde ao interpretador (ou "*shell*") do IDE. O Interpretador permite escrever comandos Python e, assim que for executado, o Python irá exibirá o seu resultado. Sua principal vantagem se dá pela resposta imediata aos comandos executados.

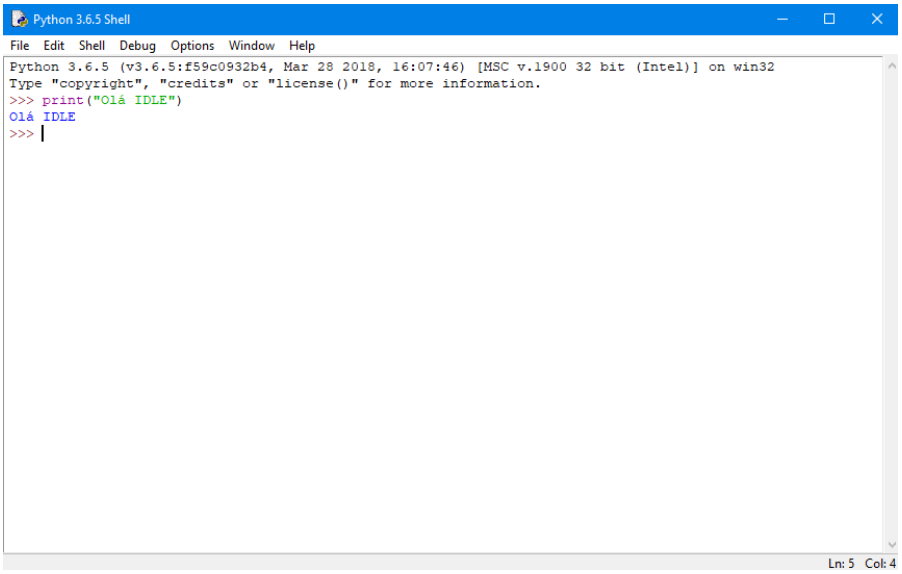

Figura A.1: Visão do interpretador shell do IDLE.

Observe que, se fecharmos o interpretador do IDLE e o iniciarmos novamente, perderemos todos os comandos que foram previamente executados.

A solução para esse problema é utilizar o editor de código do IDLE, de tal forma que possamos criar um arquivo com os comandos desejados, possibilitando salvar esse arquivo como um documento Python. Dessa forma, futuramente, podemos abrir esse arquivo, edita-lo e executa-lo novamente.

Para abrir o editor de código do IDLE vá até o menu `"File->New File"`, e então, uma nova janela será aberta. Conforme apresentado na Figura A.2.

[1]Observe que é necessário especificar a versão do Python correspondente ao IDLE. Caso queria executar o IDLE com a versão Python 2.7, a versão a ser instalada deve ser `idle-python2.7`.

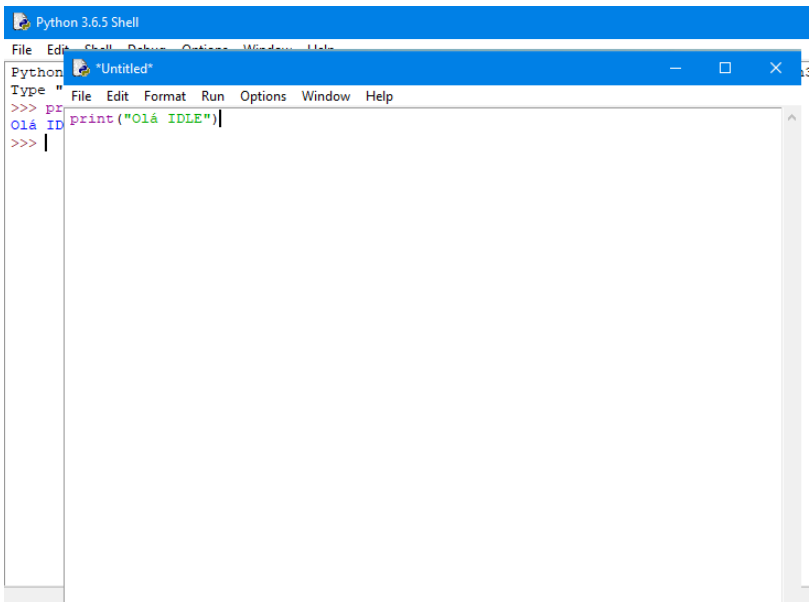

Figura A.2: Visão do editor de código do IDLE.

No editor de código do IDLE é possível escrever o código em Python e, posteriormente, salva-lo e executa-lo. É possível executar o arquivo criado por meio do `menu` do editor de código `"Run->Run Module"` ou, simplesmente, apertando a tecla `F5`.

Observe que a saída produzida pela execução do código será mostrada no interpretador do IDLE.

Para uma utilização eficiente do IDLE, o programador alterna constantemente entre a janela do interpretador e a janela do editor de código. Essa tarefa nem sempre é muito produtiva. Contudo, o IDLE foi desenvolvido com o intuito de ser um pequeno laboratório para experimentar funcionalidade da linguagem Python e apenas para o desenvolvimento de pequenos programas.

A.1.2 Geany

Geany é um editor de texto que usa o kit de ferramentas GTK + e possui os recursos básicos de uma IDE. Ele é gratuito, de código aberto[2] e foi desenvolvido para fornecer ao usuário uma IDE rápida e de pequeno porte, que possui uma pequena lista dependências de outros pacotes. Dessa forma a IDE tem como objetivo ser o mais independente possível de um ambiente, podendo ser, facilmente, executada em diversos sistemas operacionais.

O Geany suporta as principais linguagens de programação do mercado (`C, Java, PHP, HTML, Perl, Pascal` e Python) e oferece recursos, como realce de sintaxe, auto-complete de código, auto-complete de instruções utilizadas com frequência (if, for, while), preenchimento automático de *tags* XML e HTML, dicas, além da possibilidade de estender as suas funcionalidade por meio de *plugins*.

[2] `https://github.com/geany/geany`

Instalando o Geany

Para instalar o Geany em distribuições Linux, BSD e similares, a maneira mais recomendada é a utilização de pacotes. Dessa forma, o usuário se beneficiará de futuras atualizações automáticas do Geany fornecidas pelo gerenciador de pacotes da distribuição. Pacotes estão disponíveis para a maioria das distribuições Linux, incluindo Debian, Fedora, Ubuntu, entre outros.

```
> sudo apt-get install geany
```

Para instalação em sistemas operacionais como Windows e MacOS, pacotes binários pré-compilados podem ser encontrados no site oficial[3] e realizar os procedimentos convencional de instalação de um software, conforme o sistema operacional desejado.

Visão geral

Ao abrir a IDE, o usuário é apresentado a uma interface, conforme apresentado na Figura A.3. Todas as mensagens da IDE aparecerão na caixa na parte inferior da janela. A partir da interface principal, é possível criar um arquivo, conforme destacado em azul na Figura A.3 e especificar a linguagem de programação desejada. A IDE cuida das configurações básicas para o arquivo.

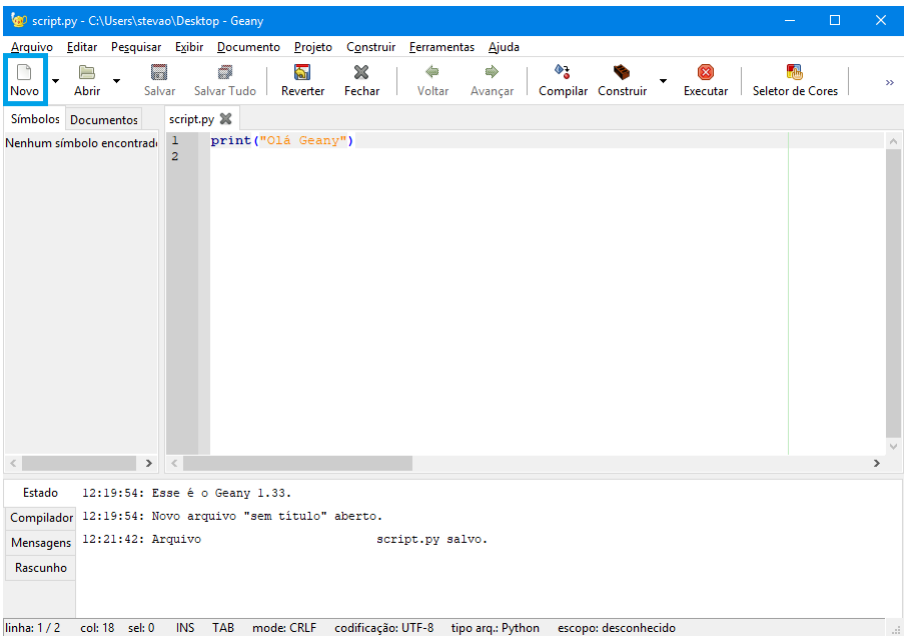

Figura A.3: Visão do editor de código do Geany.

[3] https://www.geany.org/Download/Releases

Para executar código criado é muito simples. Basta clicar em um botão para executar, conforme destacado em vermelho na Figura A.4. A saída produzida pela execução do código-fonte será exibida no *shell* do sistema operacional. Após a execução do programa, basta apertar a tecla `Enter`, para sair do *shell* e retornar à IDE.

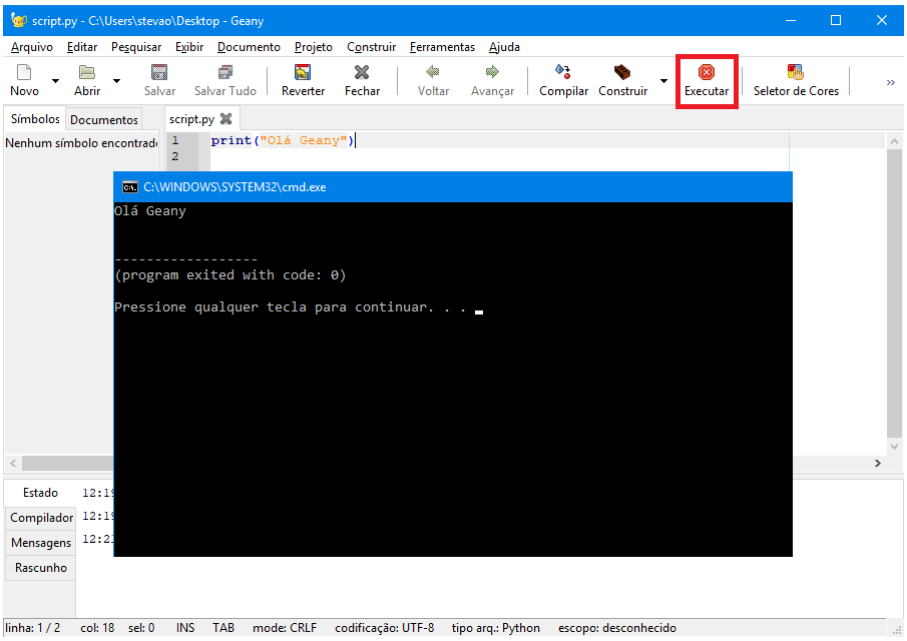

Figura A.4: Visão do editor de código do Geany em execução.

O Geany é uma alternativa de IDE para as mais diversas plataformas e tem como principal vantagem a sua facilidade de uso e configuração.

A.1.3 Spyder 3

O Spyder é um poderoso ambiente científico escrito em Python, para Python e projetado para cientistas, engenheiros e analistas de dados. Ele oferece uma combinação de recursos avançados de edição, análise, depuração e criação de perfil de uma ferramenta de desenvolvimento abrangente com exploração de dados, execução interativa, inspeção detalhada e recursos de visualização.

Além de muitos recursos internos, as capacidades dessa IDE podem ser estendidas por meio do seu sistema de *plugins* e API.

Instalando o Spyder 3

É possível instalar o Sypder[4] em diversos sistemas operacionais. A maneira mais prática e recomendada é a instalação por meio da utilização do gerenciador de pacotes do Python. Utilizando o gerenciador de pacotes `pip` (ver Seção A.2) a instalação do Spyder é realizada por meio da execução do comando especificado abaixo:

[4] `https://www.spyder-ide.org/`

```
> sudo pip3 install spyder
```

Caso seja utilizada uma distribuição Linux Debian ou distribuições baseadas no mesmo, também é possível realizar o processo de instalação por meio do comando:

```
> sudo apt-get install spyder3
```

No Windows, O processo de instalação pode ser realizado instalando algum dos pacotes de distribuição científica do Python como, por exemplo, Anaconda, WinPython e Python (x, y). A instalação por meio de uma das distribuições científicas do Python é realizada a partir das recomendações disponibilizadas na página oficial da respectiva distribuição científica. Contudo, a maneira recomendada é instalar o Spyder por meio do gerenciador de pacotes `pip`, conforme anteriormente citado.

No macOS, além da instalação por meio do gerenciador de pacotes, a instalação do Spyder pode ser realizada de outras duas formas: (i) instalando a distribuição científica Anaconda; ou (ii) por meio da instalação do arquivo DMG. A instalação da distribuição científica Anaconda pode ser realizada por meio das recomendações disponibilizadas na página oficial da distribuição[5]. Por fim, a instalação do arquivo de instalação DMG é realizada por meio do download do arquivo de instalação[6].

Visão geral

Após a instalação correta do ambiente de programação Spyder, o usuário terá a disposição todos os recursos necessários para uso da IDE. Caso o usuário tenha optado por instalar o Sypder por meio da instalação de alguma distribuição científica do Python como, por exemplo, o Anaconda, o usuário terá a disposição uma tela similar a Figura A.5.

Assim como ilustrado na Figura A.5, o usuário pode abrir a IDE Sypder e ter acesso a instalação de diversas bibliotecas científicas a partir do Anaconda. Abrindo a IDE Sypder, o usuário terá a disposição uma tela similar a Figura A.6.

A interface da IDE Spyder é simples e oferece um ambiente de programação em Python com diversos recursos para o usuário. Os dois principais recursos disponibilizados pela Spyder é o Editor (em vermelho na Figura A.6), o Console (em amarelo na Figura A.6) e a Barra de Ferramentas (em laranja na Figura A.6). O editor de texto oferece diversas funcionalidades como, por exemplo, edição multilinguagem, verificação automática de sintaxe, análise de código em tempo real, dicas de chamadas, etc. O console possibilita que o usuário possa entrar, interagir e visualizar os dados por meio de um interpretador Python.

[5] `https://www.anaconda.com/download/`
[6] `https://github.com/spyder-ide/spyder/releases`

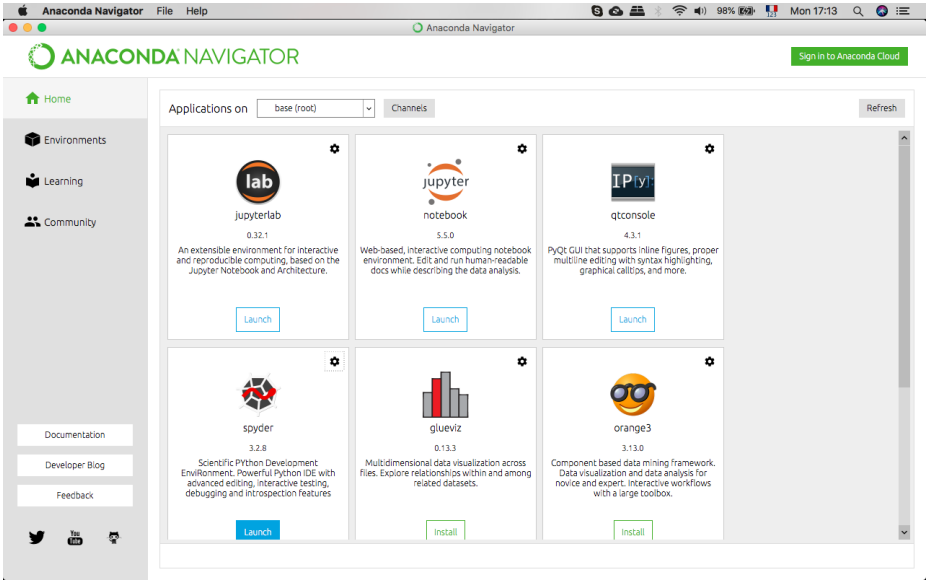

Figura A.5: Visão da distribuição científica Anaconda.

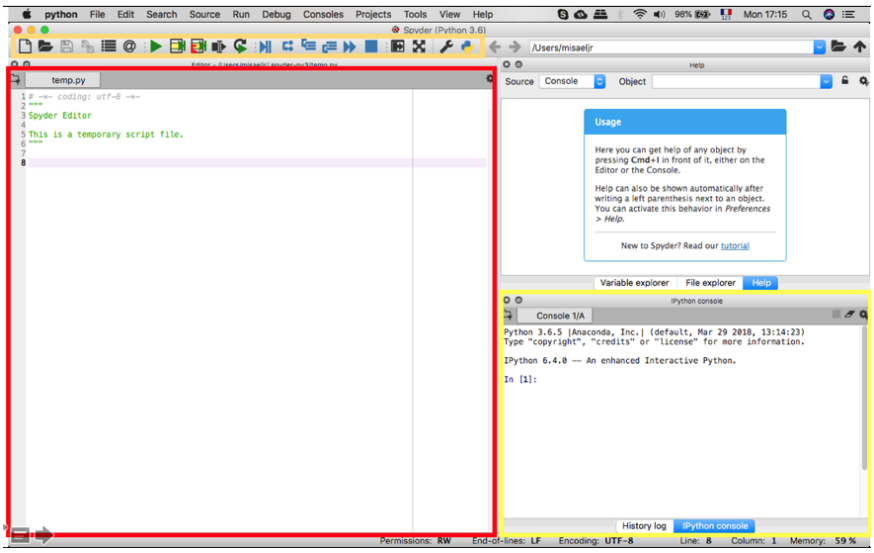

Figura A.6: Visão do editor de código do Spyder.

Por fim, a barra de ferramentas disponibiliza opções como, por exemplo: criar um novo arquivo, abrir um novo arquivo, salvar o arquivo e depurar o programa.

A Figura A.7 apresenta um exemplo da execução de um programa no Spyder. O programa lê os dados a partir de um arquivo e os armazena em dois vetores que correspondem aos eixos do gráfico PxR (Precisão x Revocação). Os dados correspondem aos valores de precisão e revocação a partir da consulta de um sistema `CBIR` (*Content-Based Image Retrieval*). Os valores de precisão e revocação são calculados a partir do resultado de cada consulta no sistema

e correspondem a qualidade do sistema em retornar as imagens esperadas pelo usuário.

Note que no exemplo foram utilizadas bibliotecas de científicas de dados como, por exemplo, o `Numpy` e o `Matplotlib`. Tais bibliotecas já se encontram nativamente inclusas na distribuição Anaconda (e em grande parte das distribuições de Python focadas em análise de Dados e programação científica). O usuário pode executar o programa pressionando `F5` ou no ícone de execução na barra de ferramentas (em vermelho na Figura A.7). O resultado da execução é apresentado no console.

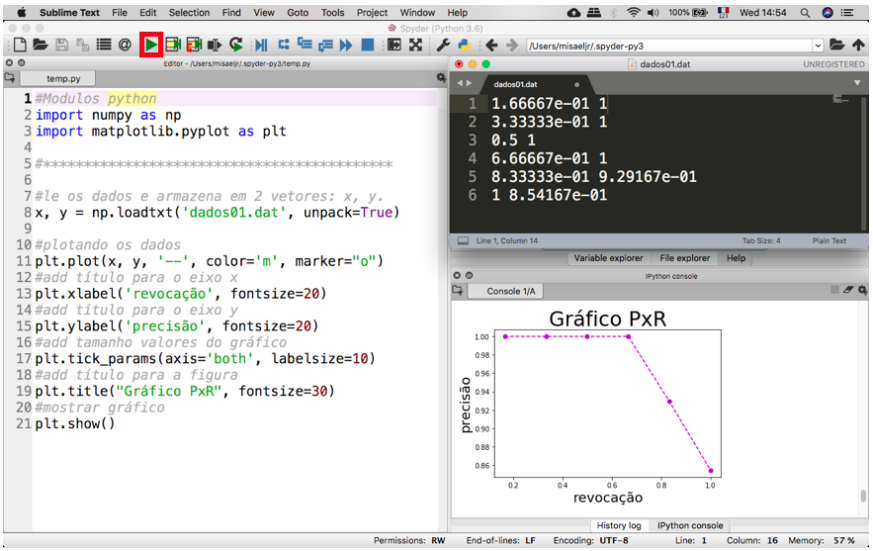

Figura A.7: Execução de um programa no Spyder.

O Spyder deve ser usado como um ambiente de trabalho para computação científica em Python, por isso, suas principais características disponibilizam recursos para esse fim. O Spyder possui os principais recursos para o desenvolvimento de aplicações convencionais em Python, mas, a menos que o foco do projeto a ser desenvolvido envolva o uso de pacotes para computação científica, é possível encontrar outras IDEs que melhor se adequam para o desenvolvimento do projeto.

Mais informações sobre o uso da IDE Sypder podem ser consultadas por meio da documentação disponibilizada na página oficial da IDE[7].

A.2 pip: Instalando pacotes

Pacotes são mecanismos de distribuição que contém arquivos com conjuntos de definições de comandos, sub-rotina e funções. Sua principal finalidade é facilitar a construção de programa. Em linhas gerais, um pacote pode ser definido como um conjunto de métodos claramente definidos para desempenhar uma determinada atividade.

[7] `https://pythonhosted.org/spyder/overview.html`

Um pacote facilita o processo de construção de programas ao abstrair a implementação de determinadas rotinas, expondo apenas objetos, ações e comportamento que o desenvolvedor precisa para construir a solução para um determinado problema.

Suponha que para um dado problema seja necessário realizar o calculo das funções `seno`, `cosseno` e `tangente`. O pacote `Math` incorpora uma série de subrotinas e funções que facilitam o desenvolvimento de softwares que necessitam de conceitos matemáticos e pode facilmente solucionar esse problema:

```
> import math
> math.sin(90)
0.8939966636005579
> math.cos(90)
-0.4480736161291701
> math.tan(90)
-1.995200412208242
```

Contudo, existe uma serie de pacotes que são desenvolvidos pela comunidade Python, que não compõem a biblioteca padrão da linguagem de programação. Nessa seção veremos como podemos instalar tais pacotes, a fim de facilitar a construção de um programa.

O *pip* é um sistema de gerenciamento de pacotes que visa simplificar a instalação e a gestão de pacotes de software escritos na linguagem de programação Python como, por exemplo, os disponibilizados no *Python Package Index (PyPI)*. Dessa forma, o *pip* transfere um módulo ou programa especificado de um repositório central para a máquina do usuário.

Caso a versão do Python instalada na máquina do usuário seja o Python 2.7.9 (ou superior) ou o Python 3.4 (ou superior), o *pip* vem instalado com o Python por padrão. Por outro lado, para versões mais antigas do Python é necessário executar algumas etapas de instalação especificadas nas subseção a seguir.

A.2.1 Instalando o *pip*

Para instalar o *pip* no *Windows* é necessário o download do arquivo **get-pip.py**[8]. Após o download do arquivo, é necessário abrir o terminal, navegar até a localização do arquivo e executá-lo a partir do seguinte comando:

```
> python3 get-pip.py
```

Caso o comando acima não funcione, a instalação do *pip* pode ser realizada manualmente. Para tal, é necessário realizar o download do arquivo **setup.py**[9],

[8] `https://bootstrap.pypa.io/get-pip.py`
[9] `https://github.com/pypa/pip`

descompacta-lo, navegar até o arquivo, via *prompt* de comando, e executar o seguinte comando:

```
> python3 setup.py install
```

Para instalar o *pip* em distribuições *Linux* como, por exemplo, *Debian*, *Ubuntu* e derivados, é necessário, a princípio, atualizar o gerenciador de pacotes por meio do seguinte comando:

```
> sudo apt-get update
```

Após atualizar o gerenciador de pacotes, o comando especificado a seguir deve ser executado para instalação do *pip*:

```
> sudo apt-get install python3-pip
```

Para verificar se o *pip* foi instalado corretamente, o seguinte comando deve ser executado:

```
> pip3 --version
pip 10.0.1 from /Library/python36-32/lib/site-packages/
pip (python 3.6)
```

O retorno do comando deve ser similar ao exibido acima, embora o número da versão possa variar.

A.2.2 Usando o pip

Com o *pip* instalado, diversas operações podem ser executadas. Instalação de pacotes, desinstalação de pacotes, controle de versão e busca de pacotes são algumas das operações que podem ser executadas a partir do *pip*.

Uma das melhores fontes de informações sobre o *pip* é a própria documentação disponibilizada pela ferramenta. Para ter acesso, basta digitar o comando abaixo no *prompt* de comando:

```
> pip3 help
Commands:
  install          Install packages.
  uninstall        Uninstall packages.
  freeze           Output installed packages...
```

Então serão listados todos os comandos e opções que podem ser utilizados, como, por exemplo: install, uninstall, freze, etc. Também é possível utilizar o comando help para receber maiores informações a respeito de um comando especifico. Para isso, basta especificar o comando a ser inspecionado após a diretiva help, conforme exemplificado abaixo:

```
> pip3 help install

Usage:
  pip install [options] <requirement specifier> ...
  pip install [options] -r <requirements file>  ...
```

Com isso serão exibidas todas as opções de configurações disponíveis para o comando especificado.

A seguir, algumas das operações disponibilizadas pelo *pip* são apresentadas.

- **install:** comando utilizado para instalar um pacote especificado pelo usuário. O processo de instalação pode ser realizado a partir dos comandos especificados a seguir.

Para instalar a versão mais atual do repositório, no caso, o nome do pacote é "numpy":

```
> pip3 install numpy
Successfully installed numpy-1.14.5
```

Para instalar uma versão específica de um pacote:

```
> pip3 install numpy==1.14.0
Successfully installed numpy-1.14.0
```

- **upgrade:** comando utilizado para atualizar a versão de um pacote, caso você já o tenha instalado, mas queira instalar a versão mais recente.

```
> pip3 install --upgrade numpy

Successfully installed numpy-1.14.5
```

- **search:** comando utilizado para buscar um determinado pacote que deseja-se instalar. O comando *search* é bastante utilizado quando não se tem certeza sobre o nome completo de um pacote. O comando retorna uma lista de pacotes similares a partir do que foi especificado a partir no comando.

Para buscar um determinado pacote a partir das iniciais, por exemplo, "numpy":

```
> pip3 search numpy
numpy (1.14.5)           - NumPy: array processing for
numbers, strings, records, and objects.
numpy-sugar (1.2.5)      - Missing NumPy functionalities
numpy-turtle (0.1)       - Turtle graphics with NumPy
p4a-numpy (1.13.3.post2) - recipe for python-for-android
```

- **show:** comando utilizado para obter detalhes a respeito de um determinado pacote que já encontra-se instalado no ambiente.

```
> pip3 show numpy
Name: numpy
Version: 1.14.3
Summary: NumPy: array processing for numbers, strings...
Home-page: http://www.numpy.org
Author: Travis E. Oliphant et al.
```

- **list:** comando utilizado para obter uma lista de todos os programas instalados no ambiente em execução.

```
> pip3 list
Package           Version
----------------- -------
numpy             1.14.5
setuptools        39.0.1
```

Para retornar uma lista apenas dos programas instalados que estão desatualizados:

```
> pip3 list -o
Package    Version Latest Type
---------- ------- ------ -----
setuptools 39.0.1  40.0.0 wheel
```

Para retornar uma lista apenas dos programas que não estão desatualizados:

```
> pip3 list -u
Package           Version
----------------- -------
numpy             1.14.5
```

- **freeze:** exibe a uma lista com os pacotes instalados no ambiente. Os pacotes são listados em uma ordem classificada não sensível a maiúsculas e minúsculas.

```
> pip3 freeze
numpy-1.14.5
```

A lista de pacotes exibida pelo comando *freeze* pode ser facilmente exportada para um arquivo texto. Isso possibilita, posteriormente, reinstalar todos os pacotes listados no arquivo. O comando abaixo criará um arquivo *requirements.txt*, no qual ficarão listados todos os pacotes instalados no ambiente.

```
> pip3 freeze > requiriments.txt
```

– **uninstall:** comando utilizado para desinstalar um determinado pacote.

```
> pip3 uninstall numpy
Successfully uninstalled numpy-1.14.5
```

Posteriormente, é possível reinstalar todos os pacotes a partir do arquivo *requiriments.txt* anteriormente gerado.

```
> pip3 install -r requirements.txt
Successfully installed numpy-1.14.5
```

Desenvolvedores Python mais experientes estabeleceram o *pip* como um padrão anos atrás. Formalizando isso, o *pip* também tem uma recomendação oficial (a PEP 453[10] diz: "oficialmente recomenda-se o uso do *pip* como o instalador padrão para pacotes Python").

Por esses motivo, o *pip* é o padrão mais seguro atualmente disponível para a instalação de pacotes. Além disso, é possível elencar as seguintes vantagens:

- é multiplataforma;
- possui interface simples por linha de comando; e
- facilitam o processo de gerenciamento, instalação, atualização e remoção de pacotes.

Apesar de todas as vantagens, o *pip* não incluiu um mecanismo capaz de isolar pacotes uns dos outros. Para solucionar esse problema, foram propostos `virtualenvs`, que serão abordados na próxima seção.

A.3 Criando ambientes virtuais

O `virtualenv` é uma ferramenta para criar ambientes virtuais Python isolados. Ela deve ser executada a partir do console do Linux ou Windows.

Os problemas básicos a serem resolvidos são o de dependências, versões e, indiretamente, problemas de permissões. Imagine que um dado projeto precisa da versão 1 do pacote `LibFoo`, contudo outro projeto requer a versão 2. Como é possível executar ambos os projetos?

Se a instalação das dependências é realizada no ambiente global do Python, ou seja, no seu sistema operacional, é provável acabar em uma situação na qual, involuntariamente, serão atualizados pacotes específicos para um dado projeto que não deveriam ser atualizados. Além disso, existem situações em que

[10]PEP significa Python Enhancement Proposal. Um PEP é um documento que fornece informações para a comunidade Python ou descreve novos recurso para futuras versões do Python. Os PEP são mantido pelos desenvolvedores oficiais da linguagem da linguagem Python.

não é possível instalar pacotes no diretório global de pacotes do Python. Por exemplo, em um cenários em que um servidor de hospedagem é compartilhado por múltiplos usuários.

Em todos esses casos o `virtualenv` pode ajudar. Ele cria um ambiente que tem seus próprios diretórios de instalação, que não compartilham bibliotecas com outros ambientes virtuais. O `virtualenv` é um mecanismo para separar diferentes ambientes Python para repositórios distintos.

Um cenário ideal de execução é um cenário em que cada projeto possua um ambiente isolado, no qual existam apenas os pacotes e as dependências com as versões necessárias para a execução do projeto. Isso é o que o `virtualenv` possibilita fazer.

O `virtualenv` é instalado por meio do gerenciador de pacotes Python *("pip")* com a execução do comando abaixo:

```
> pip3 install virtualenv
```

Após a instalação do `virtualenv` é possível criar ambientes virtuais Python independentes. Para a criação de um novo ambiente é necessário a execução do seguinte comando, supondo que você queira batizar o seu projeto de "projeto1_ambiente":

```
> virtualenv projeto1_ambiente
```

Ao executar o comando, observa-se que será criado um diretório denominado "projeto1_ambiente"e serão instalados uma versão do *setuptools* e do gerenciador de pacotes *(pip)* para o ambiente recém criado. Para ativar o ambiente recém criado é necessário a execução do comando abaixo:

Windows

```
> projeto1_ambiente\Scripts\activate.bat
```

Linux

```
> source projeto1_ambiente/bin/activate
```

Após a execução do comando o novo ambiente Python entrará em execução. É possível observar no quadro abaixo que aparecerá uma identificação (projeto1_ambiente) no *prompt* de comando, especificando o ambiente Python que

está em execução.

```
(projeto1_ambiente)>
```

Também é possível verificar qual a instancia do *Python* em execução por meio do comando:

Windows

```
(projeto1_ambiente)> where python
C:\Users\User\projeto1_ambiente\Scripts\python.exe
```

Linux

```
(projeto1_ambiente)> which python
/home/User/projeto1_ambiente/bin/python
```

Observe que após o comando, o caminho apontado será referente ao caminho do ambiente virtual recém criado. O mesmo pode ser verificado para o gerenciador de pacotes do Python (*pip*) e o resultado a ser apresentado também será referente ao ambiente recém criado.

Ao executar o comando abaixo:

```
(projeto1_ambiente)> pip3 list
Package    Version
---------- -------
pip        10.0.1
setuptools 39.2.0
wheel      0.31.1
```

É possível observar que apenas o próprio pacote *pip* e os pacotes *setuptools* e *wheel* encontram-se instalados e que as bibliotecas instaladas anteriormente não são listadas para esse ambiente.

Nesse estágio é possível trabalhar com o ambiente local, instalando os pacotes necessários para a execução de um dado projeto. Posteriormente, é possível exportar a lista de pacotes instalados no ambiente local por meio do comando:

```
(projeto1_ambiente)> pip3 freeze --local > requirements.txt
```

O argumento *–local* especifica que apenas os pacotes instalados no ambiente em execução devem ser exportados. Após a execução do comando, um arquivo denominado de *requiriments.txt* será criado no diretório atual. O arquivo *requiriments.txt* contém uma lista com a descrição dos pacotes instalados no presente repositório.

Considera-se uma boa pratica manter um arquivo *requiriments.txt* para cada projeto desenvolvido. Dessa forma, é possível manter o estado dos pacotes instalados a fim de facilmente replicar a instalação dos pacotes em outra máquina, ou outro ambiente virtual.

Para restaurar o estado de um conjunto de pacotes descritos em um arquivo *requiriments.txt*, dado que o *prompt* de comando está no mesmo diretório que o arquivo, é necessário realizar a execução do comando abaixo:

```
(projeto2_ambiente)> pip3 install -r requirements.txt
```

Para sair de um ambiente *Python* local e retornar para o *prompt* global é necessário realizar a execução do comando abaixo:

```
(projeto1_ambiente)> deactivate
```

Após a execução do comando, observe que a identificação (projeto1_ambiente) no *prompt* de comando ira desaparecer, indicando que o ambiente de execução passa a ser o ambiente global. De forma similar, ao executar os comandos:

Windows

```
> where python
C:\Users\User\AppData\Local\Programs\Python\Python36\pyth
on.exe
```

Linux

```
> which python
/usr/bin/python
```

O resultado produzido deverá indicar o local de instalação do Python e não mais o ambiente local como anteriormente apresentado. De forma similar, se o comando "`pip list`" for executado, serão exibidos apenas os pacotes instalados no ambiente global.

Para remover um ambiente local é necessário apenas deletar o diretório (projeto1_ambiente) que foi anteriormente criado.

Ambientes virtuais foram propostos para possibilitar o gerenciamento adequado de dependências para projetos em Python, contudo os mesmos foram projetados para serem independentes de projetos, de tal forma que os mesmos possam ser deletados e alterados sem um risco de danos ao projeto que está sendo desenvolvido.

Dessa forma, orienta-se que seja criado um diretório, independente, específico para o gerenciamento de ambientes locais e que tal diretório não possua qualquer correlação com o diretório em que se encontra o projeto a ser desenvolvido.