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".

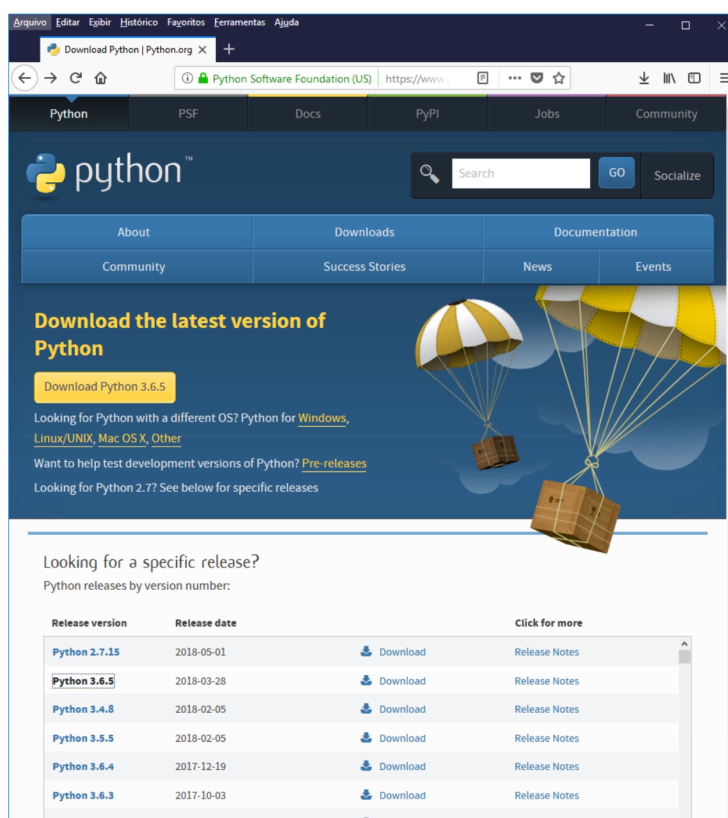

Figura 7.1: Baixando a versão mais recente de Python

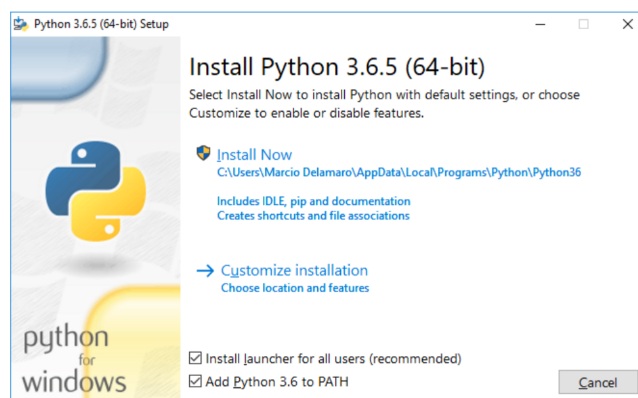

Figura 7.2: Instalador Python para Windows

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.

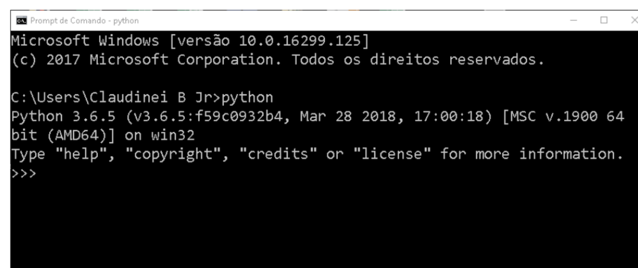

Figura 7.3: Executando Python no Prompt de comandos do Windows

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]

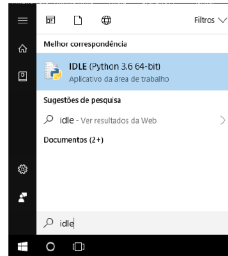

Figura 7.4: Executando o shell Python no windows

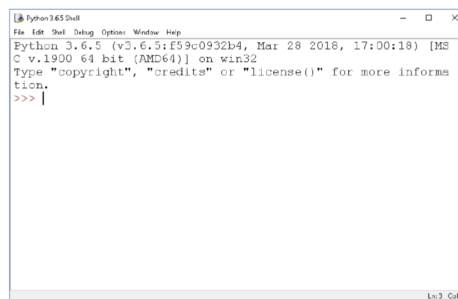

Figura 7.5: Janela do shell Python no Windows

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:

$$(A \mathbin{//} B) * B + A \,\%\, B = A$$

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. Definimos 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 incorparada 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 na prática a utilização deste recurso.

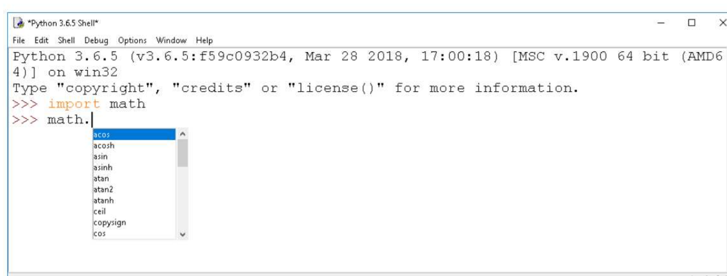

Figura 8.2: Exibição das funções do módulo `Math` no IDLE

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 na prática a utilização deste recurso.

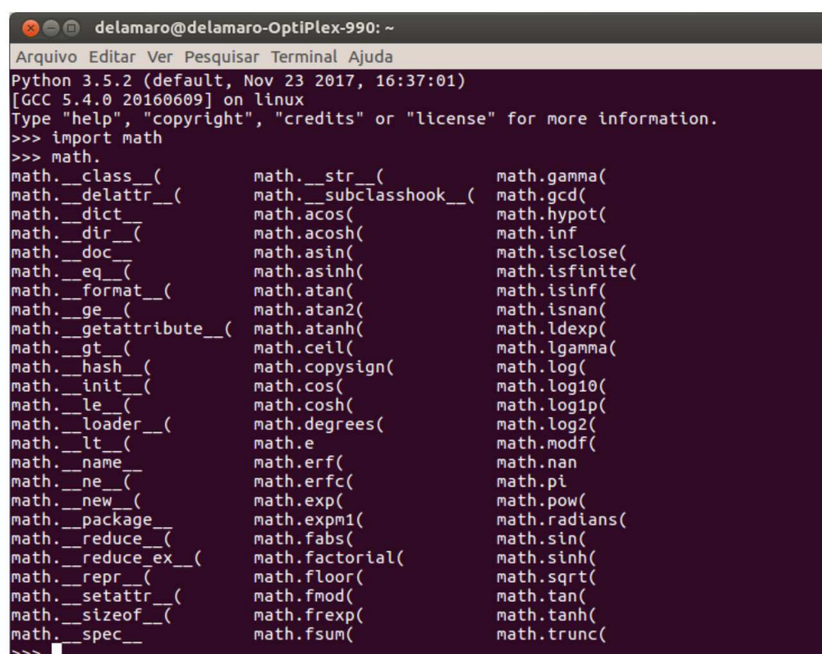

Figura 8.3: Exibição das funções do módulo `Math` no interpretador Python

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 $f(x) = x^3 - x^2 - 13x + 8$.

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.

3. Compute uma raiz da função $f(x) = sen^2(x) - cos(x)$.

```
>>> 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 $f(x) = x^3 - x^2 - 13x + 8$