FRANCISCO MARCELO DE BARROS MACIEL

Python & Django

DESENVOLVIMENTO WEB MODERNO E ÁGIL

ALTA BOOKS
EDITORA
Rio de Janeiro, 2020

SUMÁRIO

1 CONCEITOS BÁSICOS

ESTE CAPÍTULO SERVIRÁ PARA que aqueles que nunca tiveram contato com a linguagem Python, ou mesmo nunca programaram antes (se esse for o seu caso, parabéns pelo bom gosto, Python é uma ótima escolha para a primeira linguagem de programação), possam conhecer seus fundamentos. Se já estiver confortável com essa linguagem de programação, fique à vontade para pular. Mas, mesmo que esse seja o seu caso, não custa nada fazer uma leitura dinâmica do capítulo — nunca se sabe quando pode ter esquecido de estudar algum tópico enquanto aprendia a programar com uma linguagem diferente, não é verdade?

As principais características de Python são:

- **Alto nível** — é uma linguagem de alto nível, ou seja, sua sintaxe é, conceitualmente, mais próxima da linguagem natural que da linguagem de máquina.
- **Interpretada** — O código-fonte é rodado por outro programa, o interpretador. De fato, primeiro o código é convertido para um formato intermediário, denominado bytecode (tecnicamente, o código é ***compilado*** para esse formato). Os arquivos no formato bytecode têm extensão .pyc. Esse código intermediário é então convertido pelo interpretador, que é específico para cada sistema operacional, em chamadas nativas ao S.O. da máquina na qual o programa está sendo executado. Esse processo tem vantagens e "desvantagens". A principal vantagem é a **portabilidade de código** — código convertido para bytecode de uma determinada arquitetura deve executar em qualquer interpretador que implemente a mesma arquitetura. Coloquei a palavra desvantagens entre aspas porque, de fato, o maior problema dessa abordagem, quando ela surgiu, era o tempo que a máquina perdia no processo de tradução e execução, porém, atualmente os computadores são tão rápidos que esse tempo é, quase sempre, irrelevante (com exceção de sistemas para os quais o tempo de execução é crítico, mas esses costumam ser desenvolvidos com outras linguagens).

- **De script** — Pode ser utilizada para escrever pequenos programas (scripts), esse é um ponto muito forte de Python. É possível automatizar facilmente diversas tarefas com pouca codificação.
- **Multiparadigma** — Python não se enquadra em um único paradigma de linguagem de programação, sendo considerada, ao mesmo tempo, imperativa[1] e funcional. Dentro do paradigma imperativo pode ser classificada, simultaneamente, como procedural e orientada a objetos.
- **Tipagem forte** — Uma linguagem de programação é dita **fortemente tipada** se os tipos de dados são verificados em todas as operações, seja em tempo de compilação ou de execução. Ficou confuso? Veja um exemplo:

Em JavaScript, se declarar:

```
'2' + 3     // resultado: '23'
5 + True // resultado: 5True
```

Pois o interpretador não restringe os tipos permitidos, então você consegue somar uma string com um número ou um número com um booleano; em Python, tais somas resultariam em erros de compilação.

- **Tipagem dinâmica** — Uma linguagem possui **tipagem dinâmica** se ela infere o tipo de dados que uma variável armazena com base no valor recebido, sem a necessidade de o programador declará-lo explicitamente. Nas linguagens *estaticamente tipadas*, como é o caso de C e Pascal, por exemplo, o programador deve especificar o tipo da variável. Além disso, em uma linguagem dinamicamente tipada, a variável pode mudar o seu tipo de dado em tempo de execução. Não entenda isso como uma limitação, há tarefas para as quais uma determinada linguagem é mais indicada que outra.
- ***Case sensitive*** — Uma linguagem é ***case sensitive*** quando faz distinção entre maiúsculas e minúsculas. Assim, dois identificadores (nomes), *Something*1 e *something*1 são coisas diferentes para o interpretador Python.

AVISO

Se você nunca programou e não entendeu coisa alguma desta seção, não se preocupe. No Capítulo 2, quando falarei sobre variáveis e tipos de dados, isso ficará mais claro.

[1] Não há, ainda, consenso entre os teóricos da área sobre se a orientação a objetos é um novo paradigma de linguagem computacional ou apenas uma subdivisão do paradigma imperativo. Tal definição é, para os propósitos deste livro, desnecessária.

Histórico da linguagem Python

A linguagem Python se tornou bastante "famosa" de uns tempos para cá (ao menos entre programadores), o que faz parecer que se trata de uma linguagem recente, mas basta um pouco de pesquisa para verificar que ela surgiu há um bom tempo.

Python foi criada no final dos anos 1980, embora sua concepção tenha acontecido ainda mais no passado: suas raízes estão no time de desenvolvimento de outra linguagem, denominada ABC. O ano era 1982, e seu idealizador, Guido Van Rossum, trabalhava, na época, no CWI (*Centrum Wiskunde & Informatica* — Centro de Matemática e Ciência da Computação — onde também surgiu a linguagem *Algol 68*), em Amsterdã, na Holanda.

Em 1987, a linguagem ABC foi abortada; Guido foi transferido para o grupo de trabalho Amoeba, cujo intuito era desenvolver um novo sistema operacional. Lá ele percebeu a necessidade de uma linguagem que tomasse menos tempo que a C para apresentar resultados e que, ao mesmo tempo, não sofresse as limitações da linguagem de comandos shell script utilizada por eles.

Em 1989, começou, de fato, o desenvolvimento do Python; no final de 1990, a popularidade da nova linguagem tinha aumentado tanto no CWI que já era mais utilizada que a própria ABC.

A partir de 2000, houve um período de "reinado" de Java, que se tornou, talvez, a linguagem de programação mais utilizada em todo o mundo. Por outro lado, a "prolixidade" tradicionalmente associada a Java levou muitos desenvolvedores a buscarem outra linguagem, ou, pelo menos, linguagens que rodem sobre a *Java Virtual Machine* (lembro-me de ouvir, certa feita, em uma palestra: "Java está morto, mas a JVM está bem viva"). Muitos de nós (inclusive eu) programaram e continuam programando em Java até hoje! Não se trata de ter uma "linguagem favorita" e sim de utilizar "a ferramenta certa para cada tipo de trabalho".

Por conta das características positivas de Python, a década seguinte viu um crescimento cada vez maior do número de adeptos. Pessoalmente, programo em ambas as linguagens e gosto das duas.

Além de ser muito útil para automatizar tarefas, Python tem ganhado grande visibilidade nas áreas de desenvolvimento online e machine learning, pois sua sintaxe concisa permite fazer muito com menos código e suas regras de formatação rígida para o código-fonte favorecem a legibilidade dos programas. De acordo com um estudo publicado no final de 2018 pela empresa de análise de mercado Redmonk, direcionada aos desenvolvedores de software, Python está entre as cinco linguagens mais populares da atualidade.

Uma curiosidade é a origem do nome da linguagem: uma homenagem ao grupo de comédia britânico Monty Python Flying Circus. Como existe uma cobra com o mesmo nome (em português, Píton ou Pitão), vários produtos relacionados à linguagem costumam adotar uma cobra como símbolo.

Por que Python é tão popular atualmente

Discutir a popularidade de qualquer coisa é um assunto traiçoeiro, sempre se esbarra em preferências pessoais. Porém, pode-se dizer que, entre as diversas justificativas para a linguagem Python ser muito utilizada, destacam-se:

- **Eficiência** — Você pode realizar muita coisa em Python com poucas linhas de código. Às vezes, com uma biblioteca adequada, poupará horas ou até mesmo dias de trabalho. Essa característica faz da linguagem uma grande escolha para automação de tarefas.
- **Comunidade ativa** — A comunidade em torno da linguagem é bastante presente e cresce a cada dia. Se procurar no Python Package Index (PyPI), encontrará ferramentas para todas as suas necessidades, é uma verdadeira "superloja de ferramentas" para o desenvolvedor.
- **Simplicidade** —Python tem uma curva de aprendizado muito mais suave que outras linguagens tradicionais, como Java ou C++. É uma ótima escolha como primeira linguagem de programação.
- **Forte presença na academia** — Hoje, vários cursos de graduação na área de informática incluem programação em Python em seus currículos. Em particular, na área de pesquisa em "machine learning" (aprendizado de máquina), Python tem uma de suas mais difundidas utilizações.
- **Tendência** — Várias ferramentas para áreas "quentes" em computação foram desenvolvidas em Python. Tome como exemplo a área de **inteligência artificial**: vários guias de profissões e análises de mercado citam "cientista de dados" como a melhor profissão da América, e inteligência artificial como a área mais promissora para o futuro (e não se trata de um futuro distante, se você já usou um smartphone ou um videogame com câmera que detecta movimentos, saiba que ambos usam algoritmos que já estão publicamente disponíveis). Python está ganhando rapidamente a preferência nesse nicho. As bibliotecas NumPy, Pandas, Scikit-learn, Tensorflow, todas escritas em Python, só para citar alguns exemplos, são as preferidas por esses profissionais.

PEP: Python Enhancement Proposal

PEP, Python Enhancement Proposal ou "Proposta de Aperfeiçoamento do Python", de acordo com o arquivo oficial, é um documento cujo objetivo é fornecer à comunidade informações sobre a linguagem ou descrever novas características sobre a linguagem, seu processo ou ambiente. A mais conhecida é a PEP8, que descreve o "Guia de Estilo para o Código Python".

Se quer ser um programador Python, você **DEVE** ler a PEP8. Ao menos, se quer criar códigos verdadeiramente elegantes.

DICA

Regras de Formatação do Código Python

No site Python Brasil, Pedro Werneck fez a enorme gentileza de traduzir para português do Brasil **todo o conteúdo** da PEP8. O texto está disponível em https://wiki.python.org.br/GuiaDeEstilo. Não copiarei as normas aqui pois sei o quanto os leitores se irritam com "enchimento de linguiça" em livros técnicos (eu, pelo menos, não fico feliz).

Conhecendo o IDLE

OK. Chega de teoria. Vamos colocar a "mão na massa"! Se você não tem o Python instalado, siga as instruções do Apêndice I e depois retorne a este ponto.

Qualquer editor de texto voltado para programação pode ser usado para escrever código Python. Na seção "Algumas palavras sobre IDEs", ainda neste capítulo, discutirei o uso de IDEs (Integrated Development Enviroments). Se está começando e não quer perder tempo com muitos recursos, o Python vem com uma ferramenta muito simples: o IDLE (Integrated Development and Learning Enviroment). Para propósitos educacionais, o IDLE fornece o mínimo necessário caso você não tenha acesso ou não queira usar um IDE profissional.

Para acessar o IDLE no Windows:

Execute o atalho com o título IDLE (Python ***versão***, sendo ***versão*** o número da versão do Python instalada). Dependendo da versão do Windows que você utilizar, precisará digitar "python" na caixa de busca (search) para versões a partir do Windows 8 ou localizá-lo no menu Iniciar para as demais versões. Em ambos os casos, será mostrada a tela da figura:

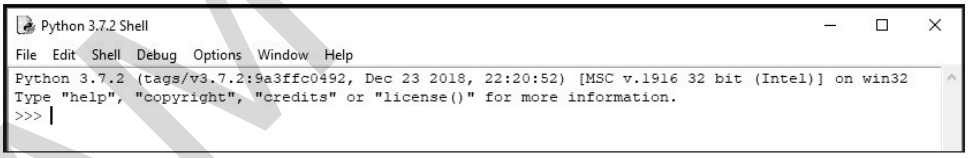

FIGURA 1.1: Shell do Python sendo executado no Windows

No Ubuntu Linux, abra um novo terminal e digite:

```
idle3
```

O resultado deverá ser o mesmo — o IDLE abrirá.

Você aprenderá alguns recursos daquele editor na seção "Um 'clássico' da programação: escrevendo 'Hello, World' em Python", na qual mostrarei o primeiro programa nessa linguagem.

DICA

Um recurso bastante útil do IDLE é o *autocomplete* — trata-se da capacidade de completar automaticamente o código, à medida que ele vai sendo digitado. Para utilizá-lo, simplesmente comece a digitar um comando qualquer do Python e tecle *tab*- o restante do comando será completado para você.
Se você quiser aproveitar código de linhas anteriores em um script que esteja digitando no IDLE (recurso muito útil quando se está escrevendo código repetitivo), use as combinações de teclas *alt* + *P* e *alt* + *N* para repetir, respectivamente, a linha anterior (**P**rior) e a próxima (**N**ext).

Python "interativo": o shell do Python

A janela inicial do IDLE, mostrada na Figura 1.1, permite rodar comandos do Python interativamente, via linha de comando, como no prompt de comando do Windows ou no terminal do Linux; diz-se, nesse caso, que você está interagindo com o shell do Python. Se quiser usar o shell fora do IDLE (por exemplo, para um teste rápido do resultado de alguma instrução):

1. No Windows:
   Abra um novo prompt de comando.
   Digite:
   `python`

2. No Linux:
   Abra um novo terminal.
   Digite:
   `python`

Pronto. Você está no shell do Python. Observe que o ***prompt*** mudará para um sinal de >>>.

Sempre que digitar comandos nesse local, eles serão executados no mesmo momento, assim que teclar **<enter>**.

Experimente digitar alguma operação matemática simples, como 2 + 2, e teclar <enter>. O resultado será mostrado na tela.

O shell é um grande auxiliar para os iniciantes, pois permite testar seus códigos sem a necessidade de escrever programas de fato. Mesmo programadores profissionais recorrem a ele, de tempos em tempos, para testar trechos rápidos de código antes de inseri-los em seus programas.

Para sair do shell, simplesmente digite: `exit()`

AVISO

Shell do Python e IDLE
Como você deve ter percebido, quando abre o IDLE, o título da janela mostra "Python **versão** shell". De fato, o IDLE usa o shell para executar suas tarefas.

Um "clássico" da programação: escrevendo "Hello, World" em Python

Como já é tradição em livros de programação, testarei os recursos do IDLE criando um programa que escreve "Hello, World!" ("Alô, Mundo!") na tela. Para tal, abra o IDLE e clique no menu File — comando New File ou tecle **<ctrl> + N**. Será exibida uma tela de edição em branco. De fato, podemos dizer que o IDLE é composto de shell + editor. Na janela do editor, digite o seguinte código:

```
# O famoso "Hello, World"
print("Hello, World!")
```

LISTAGEM 1.1: Hello, World! (hello.py)

A primeira linha é apenas um comentário. Qualquer coisa em um código Python após o símbolo # é considerada como tal e ignorada pelo interpretador.

A linha seguinte realiza uma chamada à função[2] `print()`, que, grosso modo, escreve na tela tudo o que for colocado entre seus parênteses – os chamados ***argumentos*** ou ***parâmetros*** da função.

DICA

Para alguns iniciantes, pode parecer estranho um trecho de programa que não será executado nunca. Explico melhor: a finalidade de um comentário no código é melhorar a **documentação** dos softwares que você desenvolverá. Use com sabedoria: um comentário bem escrito servirá para ajudá-lo a relembrar o que queria dizer ao escrever determinado trecho de código e, anos depois, quando precisar revisá-lo, será uma preciosa ajuda. Por outro lado, não comente apenas por comentar, seu comentário deve ter um **propósito**. Do contrário, servirá apenas para poluir o programa. Um bom uso, por exemplo, é explicar o que certo trecho do programa faz, **quando não for óbvio para quem lê**.

Um uso muito comum dos comentários, mesmo entre programadores profissionais, é para fazer o programa ignorar códigos "obsoletos", mas que você deseja que fiquem como um "lembrete" na listagem. Considero essa uma **má prática de programação**. Um software de controle de versões pode evitar esse vício.

[2] Uma ***função*** em Python, grosso modo, é um trecho de programa que é executado separadamente e pode ou não retornar um valor para o código que o chamou. Esse assunto será detalhado no Capítulo 9, "Funções personalizadas".

Na excelente obra *Código Limpo*,[3] publicada pela Alta Books, Robert C. Martin ensina:

"Uma das motivações mais comuns para criar comentários é um código ruim.(...) Códigos claros e expressivos com poucos comentários são de longe superiores a um amontoado e complexo com muitos comentários."

Observe que ele não proíbe seu uso — apenas recomenda que seja usado ***somente* quando realmente fizer sentido**.

É possível notar que, à medida que você digita, o IDLE destaca a sintaxe de seu código com cores diferentes, usando uma cor para o comando `print()` e outra para a string[4] Hello, World!

Clique em File — Save ou digite **<ctrl> + S**.

Surgirá uma caixa de diálogo pedindo um nome e local para salvar o arquivo. Selecione uma pasta do seu agrado e salve-o com o nome de *hello*. Será adicionada, automaticamente, a extensão .py ao nome do arquivo.

Para conferir se o arquivo foi realmente salvo, feche o editor e, novamente na janela do IDLE, escolha File — Open ou **<ctrl> + O**.

Seu código será carregado.

Para executar o arquivo, **com o código já carregado no editor**, clique no menu Run — comando Run Module ou tecle **<F5>**. O arquivo será executado, mostrando uma resposta semelhante à da figura:

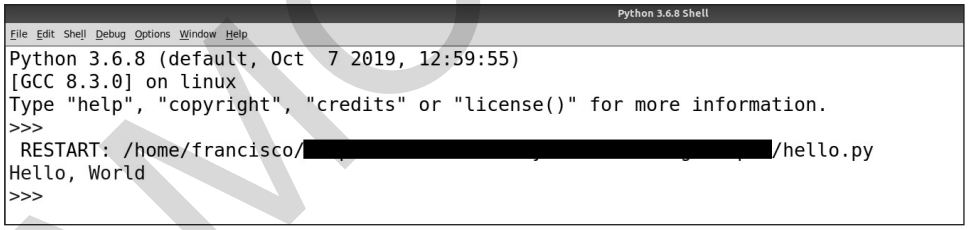

FIGURA 1.2: Resultado da execução do Hello, world

Você também pode executar seu código fora do IDLE, acessando por linha de comando a pasta onde o arquivo foi salvo e digitando:

```
python <nome do arquivo>
```

No exemplo atual:

```
python hello.py
```

[3] Não se trata de propaganda na obra, esse livro realmente vale a compra. Acredito que todo desenvolvedor deveria lê-lo, independentemente da linguagem que utiliza.

[4] Cadeia de caracteres — se essa é a primeira vez que vê esse termo, não se preocupe, ele será explicado novamente quando eu falar acerca de tipos de dados, no capítulo seguinte.

DICA

Sempre que se aprende uma nova linguagem de programação, há uma certa "barreira cognitiva" há ser transposta — novos conceitos, técnicas e ferramentas precisam ser absorvidos e cada pessoa tem o seu ritmo para esse processo. Forneci no site da editora Alta Books o código-fonte de todos os exemplos do livro, separados por capítulo; porém, minha sugestão a você, leitor, se está começando a programar agora, é que **digite o programa** e use o código do site apenas para conferir, caso algo não funcione como esperado. Em minha experiência, tendemos a aprender mais **fazendo** que olhando algo que os outros fizeram. Trata-se apenas de uma sugestão e fica a seu critério segui-la.

Aqueles que já leram outros livros de programação, muitas vezes, ao se depararem com um programa "Hello, World", têm aquela sensação de *déja-vu* e até mesmo de aborrecimento ("Hello, World"?! De novo?! Fala sério!). Defendo o seu uso como um modo simples de começar uma nova linguagem, interagindo com as capacidades de escrita na tela e, ainda, aprendendo as regras mais básicas sobre estrutura de um programa na nova tecnologia.

Se você não leu o PEP8, recomendo fazê-lo. Vários padrões de formatação de código estão descritos lá. Uma convenção que não faz parte dele, mas é bastante enfatizada pela comunidade ao nomear arquivos, é a chamada snake_case ou lower_case_with_underscores: simplesmente dê nome aos seus arquivos usando letras **minúsculas** e separe cada palavra da outra com um símbolo de underscore (_).

Algumas palavras sobre IDEs

Em várias linguagens de programação, e Python não é exceção, se tornou comum o uso de IDEs (Integrated Development Enviroments — "Ambientes de Desenvolvimento Integrado"). Essas ferramentas, em geral, são recheadas de recursos que ajudam a desenvolver software em uma ou mais linguagens.

Em geral, seu uso traz aumento de produtividade, embora aquelas também possuam a sua própria curva de aprendizado.

Há vários "sabores" de IDEs disponíveis para Python: PyCharm, Eclipse (com o plugin "PyDev"), Spyder... e a lista continua a aumentar.

Outra opção é utilizar um bom editor de textos voltado para programação, em conjunto com as ferramentas de linha de comando. Nessa categoria, posso citar: Brackets (gratuito e de código aberto), Sublime Text, UltraEdit, entre outros.

Minha sugestão: se está começando, use um editor de textos free para se familiarizar com as técnicas sem a complexidade extra de uma IDE. À medida que for ganhando experiência, teste IDEs gratuitas ou não (as que são pagas costumam ter versões de avaliação) e escolha o que lhe dá mais produtividade. Se sua escolha recair sobre um produto pago, pondere em quanto o aumento de produtividade lhe permitirá melhorar seus prazos de entrega e, portanto, seus ganhos. Por outro lado, se sua escolha for por

uma ferramenta gratuita, tenha em mente que você deverá acompanhar, periodicamente, as atualizações e notícias sobre a mesma, para continuar com seu código existente funcionando.

Variáveis: uma definição informal

Em Python, uma variável é um nome que se atribui a uma posição da memória do computador para que se possa armazenar e manipular informações. Pense nelas como o conceito de variável que você aprendeu com seu professor/sua professora de matemática do ensino fundamental: um nome que representa um valor, a princípio, desconhecido. O tópico será explicado melhor no Capítulo 2, na seção "Declarando variáveis". Por ora, para compreender os exemplos deste capítulo, basta saber que, em Python, variáveis são criadas no momento em que você lhes atribui um valor; isso é feito por meio da sintaxe:

```
nome_da_variável = valor
```

Em que:

- `nome_da_variável` — Um nome que identificará unicamente a variável durante todo o seu ciclo de vida. Esse nome deve **OBRIGATORIAMENTE** começar por uma letra ou underscore (_) e pode ser seguido por zero, letras, underscores ou números, em maiúsculas ou minúsculas.
- `Valor` — O valor que a variável conterá inicialmente (a operação de atribuir um valor a uma variável é conhecida como ***inicialização***).
  O ***tipo de dados*** de uma variável pode ser numérico, string (que contém texto), booleano (que só aceita valores **True** ou **False**), lista (que, como o nome indica, contém uma lista de itens), tupla (parecida com uma lista — também armazena grupos de informações, porém, tem algumas particularidades) e dicionário (armazena pares do tipo-chave: valor). Não se preocupe em entender o que cada tipo faz nesse momento. Eles serão apresentados em detalhes no Capítulo 2, "Variáveis e Tipos de Dados". Por ora, entenda apenas que cada variável armazena um valor e ele pode ser acessado pelo seu nome. Por convenção, variáveis também são nomeadas usando o padrão ***snake_case***.

AVISO

Também não são permitidos nomes de variáveis que sejam palavras reservadas da linguagem Python. Tais palavras são:
and, as, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while, with, yield

No código da seção seguinte, serão vistos alguns exemplos de inicialização de variáveis.

Debugando código Python

A tarefa de "caçar" erros em programação é denominada *debugging* (ou, como alguns autores adotam, "depuração"). Em geral, *debugar* um programa exige uma ferramenta que permita examinar o valor de estruturas de dados, variáveis, objetos etc. Nessa tarefa, o uso de uma IDE é **fortemente recomendado**, embora não seja indispensável.

Na maioria dos IDEs, existem recursos para visualizar o conteúdo de estruturas de dados; rodar o código linha por linha, o que pode ser de grande ajuda para encontrar erros; e definir breakpoints, que são locais do código em que o IDE pausará a execução sempre que o fluxo do programa chegar àquele ponto, permitindo examinar os dados que estão na memória.

Se você não dispuser de uma ferramenta desse tipo, pode usar simplesmente comandos `print()` no meio do seu código para exibir as informações que deseja conferir.

Para ilustrar o processo, mostrarei como debugar um pequeno programa, usando o IDLE. Vou inserir um erro de sintaxe proposital no código para que possa ilustrar o processo de *debugging*.

Execute o IDLE, conforme visto na seção "Conhecendo o IDLE".

Clique em *File — New File* e, na janela que surgirá, digite o script seguinte:

```
#Python e Django: Desenvolvimento Web Moderno e ágil.
print('Usuário: Francisco Maciel')
print('Aniversário: 04 de outubro')
idade=42
serie="Game of Thrones"
livro="Odisseia"
print('Idade:', idade, ' anos')
print('Série favorita: ', serie)
print('Livro favorito: ', livro)
```

LISTAGEM 1.2: Teste de debugging (debug.py)

Seu script deverá ficar como na Figura 1.3.

FIGURA 1.3: Código antes do erro

3. Clique no menu *Run — Run Module* ou tecle **F5**. Será mostrada uma caixa de diálogo avisando que o código deverá ser salvo antes e pedindo a confirmação. Clique em OK, forneça um nome ao script e salve-o em uma pasta qualquer.

O código será executado e exibirá a janela da Figura 1.4.

```
Usuário: Francisco Maciel
Aniversário: 04 de outubro
Idade: 42  anos
Série favorita:  Game of Thrones
Livro favorito:  Odisseia
>>>
```

FIGURA 1.4: Resultado da impressão

Agora, altere a última linha do script, modificando a palavra "livro" para "livros" e execute-o novamente. Surgirá uma mensagem de erro semelhante a:

```
Traceback (most recent call last):
  File "/home/francisco/Alta Books/código/cap01/debug.py", line 9,
in <module>
    print('Livro favorito: ', livros)
NameError: name 'livros' is not defined
```

A mensagem identifica a linha do erro (linha 9) e o tipo de erro que ocorreu (*NameError: name 'livros' is not defined*). No caso, o erro indica que uma variável denominada "livros" não foi definida antes de ser usada (lembre-se: Python é **fortemente tipado**).

Esse é o tipo de erro mais fácil de identificar e corrigir — um **erro de sintaxe**. Erros dessa natureza são identificados pelo próprio interpretador.

Corrija o código, salve-o e execute-o novamente para ter certeza de que está tudo funcionando.

Um outro recurso muito útil, que qualquer debugger minimamente aceitável possui, é o de inspecionar trechos de código em tempo de execução para conferir os valores utilizados.

Vou escrever um outro programa, um pouco mais "intelectualizado". Utilizarei aqui dois operadores do Python aos quais você ainda não foi apresentado: * e **. Eles serão explicados, brevemente, nos comentários do próprio programa. Mais adiante, na seção "Operadores aritméticos", eles serão retomados mais detalhadamente. Feche o script atual e crie um novo (<ctrl> + N). Digite o código a seguir:

```
altura = 1.5
largura = 5.0
comprimento = 6.0
area_base = largura * comprimento# * é o operador de multiplicação
volume = altura * area_base # Volume em m3
print('O volume em m3 é ', volume)
```

LISTAGEM 1.3: Cálculo de volume (volume.py)

DICA

Para melhorar seu aprendizado

Sempre que aprendo uma nova linguagem de programação, gosto de evitar o ato de "copiar e colar" o código — mesmo que seja da página online do livro que está lendo. Recomendo digitar todo o conteúdo e usar o código fornecido com a obra para conferir o resultado. Assim, os princípios ficam "gravados em pedra". Essa é a forma que funciona melhor para mim. Experimente fazer o mesmo e use o método que achar mais apropriado para você.

Execute o programa. Você deverá obter o resultado da figura:

```
O volume em m3 é  45.0
>>>
```

FIGURA 1.5: Resultado do cálculo de volume

Agora, revelarei um tipo de erro mais traiçoeiro — na lógica do programa. Esse tipo de falha não é identificada pelo interpretador, que pensa que está tudo certo, afinal, o código foi escrito de forma que pode ser executado.

Imagine que digitou, por descuido (sim, isso acontece até com programadores profissionais!), dois asteriscos em vez de um na linha que calcula a área da base, deixando-a assim:

```
area_base = largura ** comprimento
```

Como existe um operador ** no Python, o interpretador não "reclama" de nada — o programa é válido. Porém, o resultado de x * y (produto de x por y) tem um significado completamente diferente de x ** y (x elevado a y), e será obtido um resultado (errado), cuja origem é mais fácil de identificar com um debugger.

Salve o código com o nome de volume_com_erro.py

Se você rodar o programa da forma como está agora, obterá um resultado estranho:

```
O volume em m3 é  23437.5
>>>
```

FIGURA 1.6: Resultado do volume com erro

Vou detalhar, então, como debugar um programa usando o IDLE.

A primeira coisa a fazer é criar um breakpoint (literalmente, "ponto de ruptura"): uma linha de código em que, uma vez que o fluxo de execução a atinja, o programa para e o IDE devolve o controle ao programador para que esse possa inspecionar valores e testar hipóteses sobre o código em execução e o ambiente em que o programa está rodando. Na prática, um breakpoint é um marcador que informa ao debugger que ele deve executar o código até aquele ponto e, então, devolver o controle ao programador. Você pode definir múltiplos breakpoints. De fato, é muito comum existirem vários durante o desenvolvimento de um sistema complexo.

Coloque um breakpoint no código assim: no menu Debug, clique no comando Debugger. Surgirá a caixa de diálogo *Debug Control* e o Python Shell passa a mostrar a mensagem [***DEBUG ON***].

FIGURA 1.7: Caixa de diálogo Debug Control

Agora, posicione o cursor na janela do editor do código, na linha em que você quer definir um breakpoint. No caso presente, a linha área base = largura ** comprimento.

Escolhi essa linha, pois, a partir dela, os cálculos começam a ser efetuados. Clique na linha com o botão direito do mouse e escolha Set Breakpoint no menu de contexto. Observe que a linha ficará destacada em uma cor diferente. No IDLE, breakpoints não são salvos ao final da sessão de depuração; cada vez que usar o debugger, precisará redefini-los.

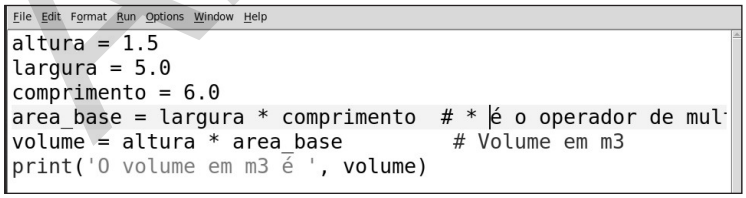

FIGURA 1.8: Definindo um breakpoint

Agora, execute novamente (F5) o programa.

O programa começará a rodar e vai parar na primeira linha de código **efetivamente executável** dele, ou seja, vai ignorar comentários que, porventura, existam no começo do seu código. Nesse momento, será exibida a janela Debug Control novamente.

Agora você pode:

- Executar o código normalmente, até que um breakpoint seja encontrado; para isso, basta clicar em Go. Esse recurso é usado quando se quer identificar a origem de um erro em um código por meio de várias paradas na execução, nos trechos em que algum dado é alterado. A cada interrupção, o debugger mostrará os valores efetivamente armazenados nas variáveis, além de outras informações úteis. Geralmente, quando se programa um sistema grande, com vários arquivos de código, costuma colocar diversos breakpoints nos lugares em que se "suspeita" que um erro possa ter começado.
- Executar o programa sequencialmente, linha por linha; para isso, clique no botão Step na caixa de diálogo Debug Control. Ao fazê-lo, apenas uma linha do programa será executada e o painel Locals do Debug Control mostrará que você está agora na linha 2 e o código em execução é: `largura = 5.0`.

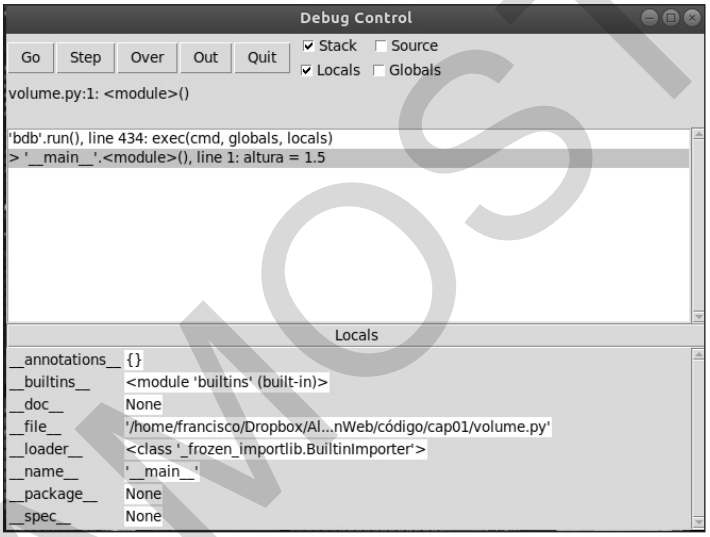

FIGURA 1.9: Programa atingiu um breakpoint

- "Pular uma linha" na execução, clicando em Over — geralmente, isso é feito ao atingir uma linha que se sabe ser irrelevante para a localização do erro. Se houver uma chamada de função no local, esta será executada por completo, sem exibir detalhes de sua execução ou variáveis internas e, em seguida, o controle retorna ao debugger. Por exemplo, eu poderia tê-lo feito em qualquer das três primeiras linhas do script, pois elas apenas inicializam variáveis e seus valores podem ser facilmente conferidos no painel Locals.
- Retornar ao programa chamador clicando em Out. Sempre que estiver em um código que foi chamado a partir de outro, por exemplo, em uma função,

clicar em Out executará o restante do código chamado e devolverá o controle ao programa chamador. Esses trechos de código chamados a partir de outros são também conhecidos como sub-rotinas.

- Encerrar a execução, clicando em Quit. Prossiga, clicando em Go. O programa vai rodar e parar no breakpoint que foi definido, na linha 4. Nesse momento, observe os valores das variáveis no painel Locals:

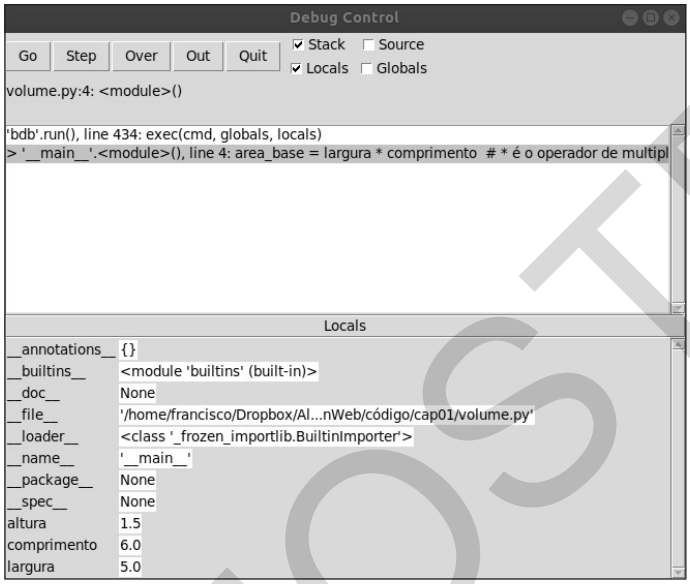

FIGURA 1.10: Inspecionando as variáveis com o Debugger

Todos os valores estão corretos, como esperado. Clique agora em Step. Nesse momento, aparecerá na lista uma nova variável, `area_base`, com valor 15625.0, que está claramente errado (o correto seria base x altura x comprimento = 1,5 x 5,0 x 6,0 = 45,0).

A linha causadora do erro foi encontrada. Uma observação atenta e é possível perceber o operador ** no lugar de *. Apagando o * extra e rodando o programa outra vez, vê-se que o erro não acontece mais.

Esse exemplo ilustra o processo de localizar e corrigir bugs em aplicações. Você pode usar outras ferramentas (recomendo um IDE para essa tarefa — eles costumam ter debuggers bem mais fáceis de usar), porém, o procedimento básico é o mesmo. No Capítulo 12, "Quando as Coisas Dão Errado", serão demonstradas outras técnicas para prevenir e detectar erros.

Escrevendo na saída padrão: a função `print()`

Nesta seção, examinarei mais de perto a função `print()`. Nos programas anteriores, eu a utilizei para escrever na tela. Como a tela é o dispositivo de saída padrão na maioria dos equipamentos, diz-se que essa função "escreve na saída padrão".

Abra o IDLE e digite:

```
print("Aprendendo Python")
```

A mensagem será exibida na tela, conforme esperado. Agora, experimente digitar:

```
print("Aprendendo Python", " a partir de um livro")
```

Será impresso na tela:

```
Aprendendo Python a partir de um livro
```

Esse exemplo serviu apenas para demonstrar que, se passar uma lista de textos[5] separados por vírgula para a função `print()`, eles serão mostrados em sequência na mesma linha. Isso pode ser útil quando você tiver um texto longo a imprimir na tela — quebre-o em mais de uma linha e seu programa ficará mais legível.

Agora, lembra de quando, na última seção, escrevi uma linha de código como a seguinte?

```
print('O volume em m3 é ', volume)
```

Quando você passar uma variável como parâmetro para a função `print()`, será mostrado o valor armazenado nela (a menos que a delimite com aspas — nesse caso, será mostrado o nome da variável).

Do mesmo modo, é possível imprimir números:

```
print('O valor aproximado de pi é ', 3.141592)
```

AVISO

Cuidado ao digitar o exemplo anterior! O separador de decimais do Python é um **ponto** e não uma **vírgula**, como escrevemos em português.

Se passar uma **expressão matemática**, aparecerá o resultado da conta:

```
print('2 + 2 = ', 2 + 2)
```

Será exibido:

```
2 + 2 = 4
```

Note que o primeiro argumento passado à função ("2 + 2 =") é um **texto**, pois está delimitado por aspas (tanto faz se você usar aspas simples ou duplas). Os textos, em lin-

[5] De fato, Python tem ótimos recursos para processamento de listas de dados — característica muito utilizada para implementar scripts de automação de tarefas.