1

- Criando e executando Programas PYTHON.
- “O lindo coração” do Python

Rápida Introdução à programação procedural

Este capítulo irá lhe fornecer informação suficiente para iniciar sua escrita de programas Python. Nós recomendamos, de forma veemente, que você instale o Python caso ainda não o tenha feito, a fim de que possa experenciar algo concreto com tudo o que aprender aqui. (A introdução explica como obter e instalar Python com as principais plataformas – ver página 4).

Esta primeira seção do capítulo vai lhe mostrar como criar e executar programas Python. Você pode usar o seu editor de texto simples para escrever seu código Python, mas o ambiente de programação IDLE, discutido nesta seção, não lhe fornece apenas um editor para códigos, mas também algumas funcionalidades adicionais, incluindo facilidades para experimentação com código Python e para a depuração de programas Python.

A segunda seção apresenta oito partes-chave de Python, que, por si mesmas, são o suficiente para escrever programas Python muito úteis. Estas partes serão todas abordadas de maneira mais extensa em capítulos posteriores, e, conforme o livro for progredindo, eles serão suplementados por todo o restante do Python; portanto, ao final do livro, você já terá aprendido toda a linguagem e estará apto a utilizar tudo o que isto pode oferecer para seus programas.

A seção final deste capítulo introduz dois pequenos programas, os quais usam o subset das características do Python, introduzidos na segunda seção. Desta forma, você poderá provar imediatamente, o gosto da programação Python.

Criando e executando Programas Python

Codificadores de caracteres ☞ 85

O código Python pode ser escrito com qualquer editor de texto simples, que possa carregar e salvar o texto usando ambos os codificadores de caracteres: o ASCII ou o UTF-8. Por padrão, os arquivos Python assumem usar o codificador de caracteres UTF-8, um superset de ASCII que pode representar muito bem qualquer caractere, em qualquer linguagem. Arquivos Python geralmente possuem uma

extensão `.py`, ainda que, em alguns sistemas semelhantes ao Unix (e.g., linux e Mac OS X), algumas aplicações Python não possuam extensão, e os programas Python GUI (Graphical User Interface) geralmente vêm com uma extensão `.pyw`, particularmente no Windows e no Mac OS X. Neste livro, nós sempre usamos uma extensão `.py` para programas de terminais do Python e módulos Python, e `.pyw` para programas GUI. Todos os exemplos apresentados neste livro executam sem alterações em todas as plataformas em que Python 3 está disponível.

Apenas para ter certeza de que tudo está funcionando corretamente e para criar o primeiro exemplo clássico, crie um arquivo chamado `hello.py` em um editor de texto simples (o bloco de notas do Windows já serve muito bem – nós logo iremos usar um editor melhor) com os seguintes itens:

```
#!/usr/bin/env  python3

print ("hello", "World!")
```

A primeira linha é um comentário. Em Python, os comentários começam com um # e continuam até o final da mesma linha. (Nós iremos explicar o comentário mais crítico adiante).

A segunda linha está em branco Python ignora linhas em branco, mas elas são normalmente muito usadas pelas pessoas para quebrar blocos de código muito grandes: uma maneira de torná-los mais fáceis de se ler. A terceira linha é um código Python. Aqui, a função `print ( )` foi evocada com dois argumentos, cada um do tipo `str` (string; i.e., uma sequência de caracteres).

Cada declaração encontrada em um arquivo `.py` é executada em turnos, começando com o primeiro e, progressivamente, linha por linha. Isto é diferente de algumas outras linguagem, por exemplo, C++ ou Java, as quais possuem uma função particular ou um método de nome especial de onde elas iniciam. O fluxo de controle pode divergir, como veremos assim que discutirmos as estruturas de controle do Python, na próxima seção.

Nós iremos supor que os usuários do Windows mantenham seus códigos python no diretório `C:\py3eg` e que os usuários Unix, Linux e Mac OS X mantenham seus códigos no diretório `$HOME/py3eg`. Salve, então, `hello.py` dentro do diretório e feche o editor de texto.

Agora que já temos um programa, podemos executá-lo. Os programas Python são executados pelo interpretador do Python e, normalmente, isto é feito dentro de um terminal do Windows. No Windows, este terminal é chamado de "`Prompt de Comando`" ou "`DOS Prompt`" ou "`MS-DOS Prompt`", ou algo assim, e, normalmente, está disponível em `Iniciar => Todos os programas => Acessórios`. No Mac OS X, o terminal é fornecido através do programa `Terminal.app` (localizado por default em `Aplicativos/Utilitários`), que também pode ser encontrado utilizando o "Finder", e, em outros Unixes, nós podemos usar um xterm ou um terminal que é fornecido pelo ambiente windowing, por exemplo, konsole ou gnome-terminal.

Inicie um Prompt de Comando e, no Windows, entre com os seguintes comandos (os quais pressupõem que o Python já esteja instalado no local padrão) – a saída do prompt de comando pode ser vista em **negrito**; e, o que você digitou, em `Courier New`:

```
C:\>cd  c:\py3g
C:\py3eg\>C:\Python30\python.exe  hello.py
```

Desde que o comando cd (mudar diretório) possua uma lista de diretórios absoluta, não importa de qual diretório você inicie.

Usuários Unix devem entrar com (supondo que o Python 3 esteja no PATH):*

```
$  cd $HOME/py3eg
$  python3  hello.py
```

Em ambos os casos, a saída será a mesma:

```
Hello World!
```

Note que, a não ser declarado ao contrário, o comportamento do Python no Mac OS X é o mesmo do que em qualquer sistema Unix. De fato, a qualquer momento que nos referimos a "Unix", isto pode ser subentendido como Linux, BSD, Mac OS X, e a maioria dos sistemas Unixes e afins.

Print ( ) ☞ 171

Ainda que o programa possua apenas uma declaração executável, uma vez que esta seja executada, é possível inferir alguma informação sobre a função `print( )`. Por um lado, `print ( )` é um item construído dentro da linguagem Python– não precisamos "importar" ou "incluir" o mesmo de uma biblioteca para fazer uso dele. Ademais, ele separa cada item que será impresso com um único espaço, e imprime uma nova linha após o último item ser impresso. Estes são comportamentos padrão que podem ser modificados, como veremos a seguir. Por outro lado, algo digno de nota a respeito de `print ( )` é que ele pode suportar quantos argumentos nos convir dar a ele.

Digitar essas linhas de comando para invocar nosso programa Python poderia rapidamente se transformar em algo extremamente tedioso. Felizmente, tanto no Windows como no Unix, podemos usar uma abordagem mais conveniente. Supondo que estamos no diretório do Windows `py3eg`, nós podemos simplesmente digitar:

```
C:\py3eg\>hello.py
```

O Windows usa associações de registro de arquivos para chamar automaticamente o interpretador Python, quando um nome-de-arquivo com extensão `.py` é iniciado dentro de um terminal.

Se a saída no Windows aparecer como:

```
('hello','World!')
```

---

* O Prompt do Unix pode ser diferente do que o $ mostrado aqui; caso isso aconteça, saiba que não é algo relevante.

Isto significa que o Python 2 está no sistema e está sendo invocado no lugar do Python 3. Uma solução é modificar a associação do arquivo `.py` do Python 2 para o Python 3.

Outra solução (menos conveniente, mais segura) é colocar o interpretador do Python 3 no caminho (supondo que ele esteja instalado no local padrão) e executar explicitamente todas as vezes:

```
C:\py3eg\>path=c:\python30;%path%
C:\py3eg\>python hello.py
```

Isto pode se tornar mais conveniente para criar um arquivo `py3.bat` com a linha única `path=c:\python30;%path%` e salvar este arquivo no diretório `C:\Windows`. Depois, toda vez que você iniciar um terminal para executar programas do Python3, comece executando `py3.bat`. Ou,de forma alternada, você pode executar `py3.bat` automaticamente. Para fazer isso, modifique as propriedades do terminal (encontre o terminal no menu Iniciar, depois clique com botão direito do mouse para aparecer na tela de Propiedades o texto " /u /k c:\windows\py3.bat" (note os espaços antes, entre e depois das opções "/u" e "/k" e certifique-se de ter adicionado este no final, após "cmd.exe").

No Unix, devemos primeiro fazer do arquivo um executável e, somente depois, poderemos executá-lo:

```
$ chmod +x hello.py
$ ./hello.py
```

Nós precisamos executar o comando `chmod` apenas uma vez, é claro; depois disso, podemos digitar simplesmente `./hello.py` e o programa irá rodar.

No Unix, quando um programa é invocado no terminal, os primeiros dois bytes do arquivo são lidos.* Se estes bytes são os caracteres ASCII #!, o shell assume que o arquivo deve ser executado por um interpretador e que a linha do primeiro arquivo especifica qual interpretador usar. Esta linha é chamada de linha `shebang` (execução shell) e, se presente, deverá ser a primeira linha do arquivo.

A linha `shebang` é comumente escrita em uma destas duas formas:

```
#!/usr/bin/python3
```

Ou

```
#!/usr/bin/env python 3
```

Se a escrita usar a primeira forma, o interpretador especificado é usado. Esta forma pode ser necessária para os programas Python que serão executados por um servidor da web, ainda que o caminho especificado possa ser diferente do que foi mostrado. Se a escrita usar a segunda forma, o primeiro interpretador `Python3` encontrado no ambiente corrente do shell é usado. A segunda forma é mais versátil porque permite a possibilidade de o interpretador Python3 não ficar locali-

* A interação entre o usuário e o terminal é gerenciado por um programa "shell". A distinção entre o terminal e o shell não nos diz respeito aqui, então iremos utilizar os termos intercambiávelmente.

zado em `/usr/bin` (e.g., ele poderá ficar em `/usr/local/bin` ou instalado em `$HOME`). A linha shebang não é necessária (mas é inofensiva) no Windows; todos os exemplos neste livro possuem uma linha shebang da segunda forma, ainda que não a mostremos.

Perceba que para os sistemas Unix supomos que o nome do executável do Python3 (ou um link simbólico) no `PATH` é `Python3`. Se este não é seu caso, você vai precisar mudar a linha shebang nos exemplos para usar o nome correto (ou corrigir o nome e caminho caso você esteja usando a primeira forma) ou criar um link simbólico do executável Python3 para o nome `Python3` em algum lugar no `PATH`.

Muitos editores de textos simples, mas poderosos, tais como o Vim e o Emacs, vêm com um suporte embutido para editar programas Python. Este suporte tipicamente envolve destaque colorido de sintaxe e uma correta identação ou desindentação das linhas. Uma alternativa é usar o ambiente de programação IDLE do Python. No Windows e no Mac OS X, o IDLE é instalado de modo padrão; nos Unixes, ele é normalmente fornecido como um pacote separado, como descrito na Introdução.

Obtendo e instalando Python ☜ 4

Como aparece na tela que a Figura 1.1. mostra, o IDLE possui um estilo que nos transporta à época do Motif no Unix e do Windows 95. Isto acontece porque ele usa a biblioteca baseado em Tk Tkinter GUI (vista no capítulo 13), ao invés de usar as mais poderosas e modernas bibliotecas GUI, como PyGtk, PyQt ou wxPython. As razões para o uso da Tkinter são uma mistura de história, licença liberal, e o fato de que a Tkinter é muito menor que as outras bibliotecas GUI.

Por outro lado, o IDLE vem como padrão com o Python e é muito simples de aprender e usar.

IDLE fornece três facilidades-chave: a habilidade para entrar com expressões e código Python e ver os resultados diretamente no Shell Python; um editor de código que fornece destaque colorido de sintaxe e suporte de indentação; e um

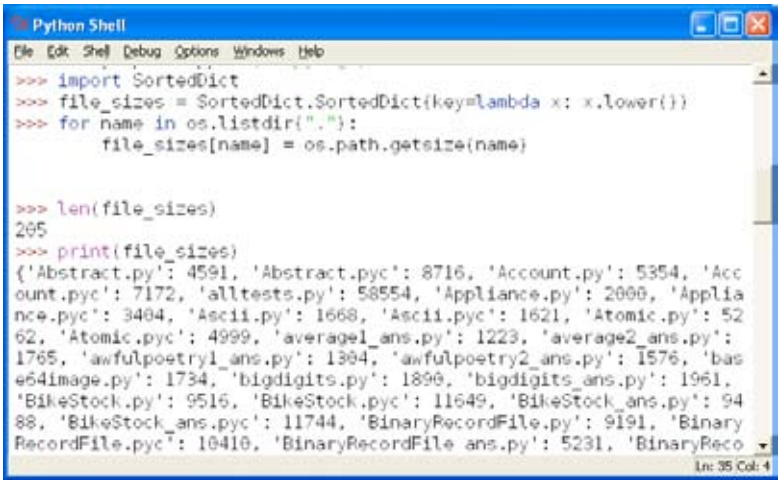

Figura 1.1. *Shell Python do IDLE.*

depurador que pode ser usado para avançar através do código a fim de ajudar a identificar e acabar com os erros de programação (bugs). O Shell Python é especialmente útil para testar algoritmos simples, fragmentos de código e expressões regulares, além de ser usado também como uma calculadora poderosa e muito flexível.

Muitos outros ambientes de desenvolvimento Python estão disponíveis, mas nós recomendamos que você use o IDLE, pelo menos no começo. Uma alternativa é criar seus programas no editor de texto simples da sua escolha e depurá-los usando invocações para `print()`.

É possível invocar o interpretador do Python sem um programa Python específico. Se isto for feito, o interpretador inicia de modo interativo. Neste modo, é possível entrar com declarações Python e ver os resultados exatamente da mesma forma como se usássemos a janela IDLE do Python Shell, e com os mesmo prompts>>>>. Mas o IDLE é muito mais fácil de usar, portanto, nós recomendamos usar o IDLE para experimentação de fragmentos de código. Os pequenos e interativos exemplos que mostramos até agora pressupõem serem executados em um interpretador interativo do Python ou no IDLE Python Shell.

Agora, nós já sabemos como criar e executar programas Python, mas, objetivamente, não vamos chegar muito longe conhecendo apenas uma única função. Na próxima seção, iremos aumentar consideravelmente nosso conhecimento Python. Isso nos tornará aptos a criar pequenos, mas muito úteis, programas Python, algo que iremos fazer na última seção deste capítulo.

"Lindo coração" do Python

Nesta seção, nós iremos aprender sobre as oito partes da linguagem Python, e, na próxima seção, iremos mostrar como estas partes podem ser usadas para escrever um punhado de pequenos, mas realistas, programas. Há muito mais a dizer sobre todas as coisas que iremos ver nesta seção, logo, conforme você for lendo, e sentir que esta faltando algo no Python ou que as coisas algumas vezes são feitas de uma forma muito enfadonha, espie mais adiante usando como guia as referências adiante, ou usando a lista de tabelas ou índice remissivo. Agindo dessa forma, certamente você irá encontrar as características do Python que procura e mais outras concisas formas de expressões do que estamos mostrando por enquanto – e muitas outras coisas.

Parte #1: Tipos de dados

Algo fundamental, que qualquer linguagem de programação deve estar apto a fazer, é representar os itens de dados. Python fornece vários tipos de dados embutidos, mas nós iremos nos preocupar com apenas dois deles por enquanto. Python representa inteiros (números inteiros positivos e negativos) usando o tipo `int`, e apresenta strings (sequências de caracteres de UniCode) usanto o tipo `str`. Aqui estão alguns exemplos de inteiros e strings, literalmente:

```
-973
21062458333711437339583605536734086463779019080109822250862195507 2
0
"Infinitely Demanding"
'Simon Critchley'
'positively abg €ã'
' '
```

Incidentalmente, o segundo número apresentado é 2^{217} – o tamanho dos inteiros do Python é limitado apenas pela memória do computador, não por um número fixado de bytes. Já as string, podem ser delimitadas através de "quotas" duplas ou simples, enquanto o mesmo tipo é usado em ambos os finais, e, já que Python usa o UniCode, as string não são limitadas para os caracteres ASCII, assim como mostra a penúltima string. Uma string vazia é simplesmente uma com nada entre os delimitadores.

O Python usa colchetes ([ ]) para acessar os itens de uma sequência com uma string. Por exemplo, se nós estamos em um Shell Python (tanto no interpretador interativo quanto no IDLE) poderemos entrar com o seguinte – a saída do Shell Python é exibida em **negrito** e o que você digitou é exibido em `Courier New`:

```
>>> "Tempos Difíceis" [5]
'T'
>>>"girafa"[0]
"g"
```

Tradicionalmente, Shell Python usa >>> como seus prompts, ainda que isto possa ser modificado. A sintaxe colchete pode ser usada com itens de dados de qualquer tipo de dados que seja uma sequência, assim como string e listas. Esta consistência de sintaxe é uma das razões para que a linguagem Python ser tão bonita. Perceba que todas as posições de índice do Python começa com 0.

No Python, tanto `str,` e os tipos básicos numéricos, como `int` são imutáveis – isto é, uma vez atribuidos, seus valores não podem ser modificados. De primeira instância, isto pode parecer uma limitação muito estranha, a sintaxe do Python não é um problema de prática. A única razão para mencionarmos este fato é que, ainda que possamos usar colchetes para resgatar o caractere dado na posição do índice em uma string, nós não poderemos usá-lo para atribuir um novo caractere. (Perceba que, no Python um caractere é simplesmente uma string de extensão 1.)

Para converter um item de dados de um tipo para outro, podemos usar a sintaxe `datetype (item)`. Por exemplo:

```
>>> int ("45")
45
>>> str (912)
'912'
```

A conversão `int( )` é tolerante à espaços em branco, logo, `int("45")` funcionaria bem também. A conversão `str( )` pode ser aplicada em quase nenhum itens de dados. Nós podemos facilmente fazer nosso próprio suporte de dados

customizados da conversão `str( )`, e também com o `int( )` ou qualquer outra conversão, se elas fizerem sentido, como veremos a seguir no Capítulo 6. Se uma conversão falha, uma exceção é lançada – nós iremos introduzir a manipulação de exceção brevemente na Parte #5, e cobrir totalmente as exceções no Capítulo 4.

Strings e inteiros serão totalmente cobertos no capítulo 2, junto com outros tipos de dados embutidos e alguns tipos de dados da biblioteca padrão do Python. Este capítulo também vai cobrir operações que podem ser aplicadas a sequências imutáveis, assim como strings.

Parte #2: Referência de Objetos

Uma vez que tenhamos alguns tipos de dados, a próxima coisa que precisamos são variáveis, nas guias armazenamos os tipos de dados. Python não possui variáveis como estas, mas, ao invés disso possui *referencia de objetos*. Quando eles se transformam em objetos imutáveis, como `ints` e `strs`, não existe uma diferença discernível entre a variável e uma referencia de objeto. Quanto a objetos mutáveis, existe uma diferença, isto raramente importa colocados em prática. Nós iremos usar os termos *variável* e *referencia de objetos* intercambiavelmente. Vamos ver alguns pequenos exemplos e, depois, discutir alguns detalhes.

```
x= “azul”
y= “verde”
z= x
```

A sintaxe é simplesmente `referencia de objeto = valor`. Não há necessidade para uma pré-declaração ou para especificar o tipo do valor. Quando o Python executa a primeira declaração, ele cria um objeto `str` com o texto “azul” e cria uma referência de objeto chamado `x` que se refere ao objeto `str`. Para todos os objetivos práticos nós podemos dizer que “variável `x` foi atribuida a string "azul". A segunda declaração é similar. A terceira declaração cria uma nova referencia de objeto chamado `z` e o ajusta para se referenciar com o mesmo objeto que a referencia de objeto `x` se refere (neste caso, o `str` contem o texto “azul”).

O operador = não é o mesmo operador de atribuição variável de algumas outras linguagens. O operador = ata um referencia de objeto a um objeto da memória. Se a referencia de objeto já existe, ele é simplesmente reatado para se referenciar ao objeto à direita do operador =; se a referencia de objeto não existe, ele é criado pelo operador =.

Vamos continuar com o exemplo `x, y, z` e fazer alguns reatamentos – como notado anteriormente, os comentários começam com um # e continuam até o fim da linha:

```
print (x,  y,  z)  # print: azul  verde   azul
z = y
print (z,  y,  z)  # print: azul  verde   azul
x = z
print (x,  y,  z)  # print: azul  verde   azul
```

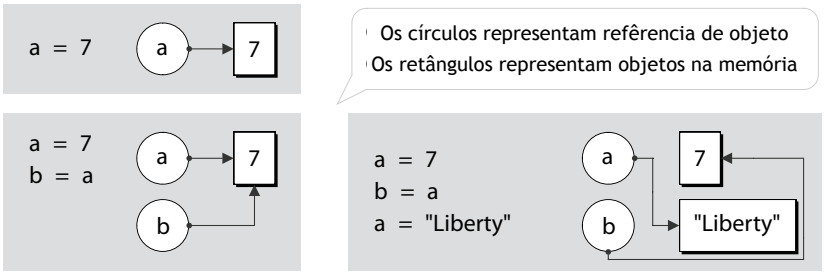

Figura 1.2 *Referência de Objetos e objetos.*

Após a quarta declaração `(x = z)`, todos as três referência de objetos são referidos a mesma `str`. Já que não existe mais referência de objetos para a string "azul", Python está livre para joga-los fora.

Figura 1.2 mostra a relação entre objetos e referência de objetos de forma esquematizada:

Os nomes usados para referência de objetos (chamados *identificadores*) possuem poucas restrições. Em particular, eles não podem ser iguais a nenhuma palavra-chave do Python, e devem iniciar com uma letra ou um traço inferior (*underscore*), e serem seguidas por zero ou alguma letra diferente de espaço sublinhado ou dígito. Não existe limite para o tamanho, e as letras e dígitos são definidos pelo UniCode, isto é, elas são incluídas, mas não limitadas, às letras e dígitos do ASCII ("a", "b", ..."z", "A", "B",..."Z", "0", "1", ..., "9"). Os identificadores Python diferenciam entre letras maiúsculas e minúsculas, então, por exemplo, `LIMITE`, `Limite` e `limite` são três diferentes identificadores. Ainda mais detalhes e alguns exemplos um tanto quanto exóticos são dados no Capitulo 2.

Python utiliza dynamic typing (tipagem dinâmica), o que significa que uma referência de objeto pode ser reatado para se referir a um objeto diferente (o qual pode ser um tipo diferente de dados) a qualquer momento. Linguagens que utilizam uma tipagem forte (como C++ e Java) permitem apenas operações que são definidas pelos tipos de dados envolvidos para serem executados. Python também aplica esta restrição, mas não pode ser classificada como tendo uma tipagem forte em seu caso, porque as operações válida podem mudar – por exemplo, se um referência de objeto é reatado a um objeto de um tipo diferente de dados. Por exemplo:

```
rota= 866
print (rota, type (rota)) # imprimir: 866 <class 'int'>
rota = "Norte"
print(rota, type (rota)) #imprimir: Norte <class 'str'>
```

Aqui, nós criamos um nova referência de objeto chamado `rota` e o ajustamos para se referir a um novo `int` de valor 866. Até este ponto, nós poderíamos usar / com `rota`, já que a divisão é uma operação válida para inteiros. Depois, nós reusamos a referência de objeto `rota` para que ele se refira a uma nova referência de objeto `str` de valor `"Norte"`, e o objeto `int` é programado para a garbage collection, já que agora nenhum referência de objeto se refere a ele. Até este pon-

to, usar / com `rota` poderia causar um `typeError` para ser lançada, já que / não é mais uma operação válida para uma string.

A função `type ( )` retorna os tipos de dados (também conhecido como a "classe") do item de dados que é fornecido- esta função pode ser muito utilizada para testar e depurar, mas normalmente não vai aparecer no código de produção, já que existe uma alternativa bem melhor, como veremos no Capítulo 6.

Se nós estamos experimentando o código Python dentro de um interpretador interativo ou em um Shell Python, bem como o único fornecido pelo IDLE, simplesmente digitar o nome de uma referência de objeto já é o bastante para fazer com que Python imprima seu valor. Por exemplo:

```
>>> x = "azul"
>>> y = "verde"
>>> z = x
>>> x
'azul'
>>> x, y, z
('azul', 'verde', 'azul')
```

Isto é muito mais conveniente do que invocar a função `print ( )` a todo momento, mas funciona apenas quando usamos o Python interativamente – qualquer programa e módulos que escrevemos devem usar `print ( )` ou funções similares para produzir saídas. Perceba que Python exibiu a última saída em parênteses separados por vírgulas – isto significa uma `tuple`, isto é, uma sequência ordenada e imutável de objetos. Nós iremos ver a respeito de `tuple` na parte seguinte.

Parte #3: Tipos de dados colecionáveis

Normalmente, é conveniente reter coleções inteiras de itens de dados. Python fornece muitos tipos de dados colecionáveis que podem reter itens, incluindo `arrys` associativas a `sets`. Mas, aqui, nós iremos introduzir apenas dois deles: tuplas e lista. As `tuple` e `list` do Python podem ser usadas para reter qualquer número de itens de dados de qualquer tipo de dados. Tuplas são *imutáveis*, portanto, uma vez que foram criadas, não é permitido modificá-las depois. Listas são *mutáveis*, logo, podemos facilmente inserir e remover itens da maneira que nos for aprazível.

Tuplas são criadas usando vírgulas (,) como nos exemplos mostrados a seguir:

```
>>> "Denmark", "Norway", "Sweden"
('Denmark', 'Norway', 'Sweden')
>>> "um",
('um',)
```

Quando Python imprimi uma tupla, ele a cerca em parênteses. Muitos programadores emulam isso e sempre cercam as tuplas literais escrevendo-as, entre parênteses. Se nós temos um item tupla e queremos usar parênteses, é preciso

ainda usar a vírgula – por exemplo, (1,). Uma tupla vazia é criada usando parênteses vazios, ( ). A vírgula é também usada para separar argumentos chamados de funções, portanto, se queremos passar uma tupla literal como um argumento, devemos cerca-la entre parênteses para fugir de confusões.

Shallow e Deep Copy ☞ 34

Aqui estão alguns exemplos de listas:

```
[1, 4, 9, 16, 25, 36, 49]
[‘alfa’, ‘bravo’, ‘charlie’, ‘delta’, ‘eco’]
[‘zebra’, 49, -879, ‘aardvark’, 200]
[]
```

Listas podem ser criadas a partir de colchetes ([ ]), assim como fizemos aqui; mais tarde, nós veremos outras maneiras de criar listas. A quarta lista mostra uma lista vazia.

Por baixo da cobertura, listas e tuplas não armazenam itens de dados, mas sim referência de objetos. Quando listas e tuplas são criadas (e quando itens são inseridos no caso das listas), elas fazem cópias das referenciais de objetos que lhes formam dados. No caso de itens literais como inteiros ou strings, um objeto do tipo de dados apropriado é criado na memória e adequadamente inicializado, e, depois, uma referência de objeto referindo-se ao objeto criado e é posto na lista ou na tupla.

Shallow e Deep Copy ☞ 136

Como tudo mais em Python, tipos de dados colecionáveis são objetos, então, nós podemos encaixar tipos de dados colecionáveis dentro de outros tipos de dados colecionáveis, por exemplo, para criar listas de listas, sem formalidades. Em algumas situações, o fato que listas, tuplas e grande parte de outros tipos de dados colecionáveis do Python retêm referência de objetos ao invés de objetos fazem um diferencial – isto é abordado no Capítulo 3 (que começa na página 99).

Em programação procedural, nós podemos invocar funções e, normalmente, passar itens de dados como argumentos. Por exemplo, nós já vimos a função `print ( )`. Outra função frequentemente usada pelo Python é `len( )`, a qual toma um único item de dados como seu argumento e retorna o “tamanho” do item como um `int`. Aqui há algumas chamadas para `len( )` – nós não temos usado negrito para as saídas do interpretador, pois supomos que você possa perceber o que você digita e o que é representado pelo interpretador, como agora:

```
>>> len( (“um”,))
1
>>>len([3,  5,  1,  2,  “pausar”  , 5])
6
>>> len(“automaticamente”)
15
```

Tuplas, listas e strings são “sized”, isto é, elas são tipos de dados que possuem uma noção de tamanho e qualquer item de dados de qualquer tipo de dados pode ser, de forma significativa, passada para a função `len ( )`. (Uma exceção é lançada se um tipo de dados sem tamanho é transferido para `len ( )`.)

Todos os itens de dados do Python são *objetos* (também chamados de *instâncias*) de um tipo de dados particular (também chamado de *classe*). Nós iremos usar os

termos *tipos de dados* e *classe* alternativamente. Uma diferença chave entre um objeto e os itens simples de dados que outras linguagens fornecem (e.g., C++ ou tipos numéricos embutidos do Java) é que um objeto pode ter `métodos`. Essencialmente, um método é simplesmente uma função que é invocada para um objeto particular. Por exemplo, o tipo lista possui um método `append ( )`, portanto, podemos anexar qualquer objeto a uma lista como esta:

```
>>> x = ["zebra", 49, -879, "aardvark", 200]
>>> x.append ("mais")
>>>x
['zebra', 49, -879, 'aardvark, 200, 'mais']
```

O objeto `x` sabe que é uma lista (todos os objetos Python sabem qual o seu próprio tipo de dados), logo, nós não precisamos especificar o tipo de dados explicitamente. Na implementação do método `append( )`, o primeiro argumento irá ser o objeto `x` por si mesmo – isto é feito automaticamente pelo Python como parte de seu suporte sintático para métodos.

O método `append( )` se transforma, isto é, muda a lista original. Isto é possível porque listas são mutáveis. Isto também é potencialmente mais eficiente do que criar uma nova lista com os itens originais e o item extra e, depois, reatar a referencial do objeto à nova lista, particularmente para lista muito longas.

Em uma linguagem procedural, a mesma coisa pode ser obtida usando o `append ( )` das listas como este (que é perfeitamente válido para a sintaxe Python):

```
>>> list.append (x,   "extra")
>>>x
['zebra', 49, -879, 'aardvark, 200, 'mais', 'extra']
```

Aqui, nós especificamos os tipos de dados e o método do tipo de dados, e lhe demos, como o primeiro argumento, o item de dados do tipo de dados que queríamos para invocar o método, seguido por qualquer argumento adicional. (Diante da hereditariedade existe uma semântica sutil que difere entre as duas sintaxes; a primeira forma é uma que é mais comumente usada em prática. Hereditariedade é abordada no Capítulo 6).

Se você não está familiarizado com programação via orientada à objetos isto pode parecer um pouco estranho de início. Por agora, apenas aceite que o Python possui funções convencionais, como estes: `funçãoNome(argumentos);` e métodos que são chamados, por exemplo: `objetoNome.metodoNome(argumentos)`. (Programação via objeto orientado será abordada no Capítulo 6).

O operador ponto ("atributo de acesso") é usado para acessar um atributo do objeto. Um atributo pode ser qualquer tipo de objeto, ainda que, por enquanto, nós tenhamos mostrado apenas atributos de métodos. Já que um atributo pode ser um objeto que possui atributos, os quais, em troca podem, ter atributos, e assim por diante, nós podemos usar quantos operadores ponto forem necessários para acessar o atributo particular que quisermos.

O tipo `list` possui muitos outros métodos, incluindo `insert( )`, que é usado para inserir um item numa posição dada no índice, e `remove( )`, que remove

um item numa posição dada no índice. Como já foi notado anteriormente, os índices do Python são sempre baseados em 0.

Nós vimos anteriormente que podemos obter caracteres das strings usando operadores em colchetes, e notamos que, no momento, este operador poderá ser usado com qualquer sequência. Listas são sequências, portanto, nós podemos fazer coisas como esta:

```
>>>x
['zebra', 49, -879, 'aardvark, 200, 'mais', 'extra']
>>> x[0]
'zebra'
>>> x[4]
200
```

Tuplas também são sequências, logo, se x tem sido uma tupla, nós podemos recuperar itens usando colchetes, exatamente do mesmo modo como fizemos na lista x. Mas, já que listas são mutáveis (diferente de string e tuplas, que são imutáveis) , podemos também usar o operador em colchetes para agrupar elementos na lista. Por exemplo:

```
>>>> x[1] = "quarenta e nove"
>>> x
['zebra', 'quarenta e nove', -879, 'aardvark', 200, 'mais', 'extra']
```

Se nós dermos uma posição de índice que esteja fora de alcance, uma exceção será lançada – nós iremos fazer uma introdução muito breve sobre manipulação de exceções na parte #5, e abordar completamente exceções no Capítulo 4.

Nós usamos o termo *sequência* uma porção de vezes até agora, contando com uma compreensão informal de seu significado, e iremos continuar a fazer isso adiante. De qualquer maneira, Python define precisamente quais características uma sequência deve suportar, e assim por diante, para várias outras categorias que um tipo de dados possa pertencer, como veremos no Capítulo 8.

Listas, tuplas e outros tipos de dados colecionáveis do Python serão abordados no Capítulo 3.

Parte #4: Operações Lógicas

Uma das características fundamentais de qualquer linguagem de programação são suas operações lógicas. Python fornece quatro grupos de operações lógicas, e nós iremos rever os aspectos fundamentais de todos eles aqui.

O operador de identidade

Já que todas as variáveis do Python são realmente referência de objetos, algumas vezes, faz sentido de perguntar dois ou mais referência de objetos são referidos para os mesmo objetos. O operador `is` trata-se de um operador binário que retorna `true` se sua referência de objeto esquerda está se referindo ao mesmo referencia de objeto da direita.

Aqui estão alguns exemplos:

```
>>> a= ["Retenção", 3, None]
>>> b = ["Retenção", 3, None]
>>> a e b
False
>>>b = a
>>>a e b
True
```

Perceba que, usualmente, não faz sentido usar o `is` para comparar `ints, strs` e a maioria dos tipos de dados, já que a grande maioria invariavelmente quer comparar seus valores. De fato, usar `is` para comparar itens de dados pode levar a resultados não muito intuitivos, como vimos no exemplo anterior, em que, ainda que `a` e `b` foram atribuídos inicialmente a mesma lista de valores, as listas, por si mesmas, são retidas como objetos `list` separados e, portanto, `is` retorna como `False` da primeira vez que o utilizamos.

Um dos benefícios oferecidos pelas comparações de identidade é que elas são muito rápidas. Isto ocorre porque os objetos referidos não precisam ser examinados por si próprios. O operador `is` precisa comparar apenas a memória de endereços dos objetos – o mesmo endereço significa o mesmo objeto.

O caso mais comum do uso do `is` ocorre quando se compara um item de dados com um objeto nulo, `None`, o qual é normalmente utilizado como um valor placemarking (marcador) que significa *"unknow"* ou *"nonexistent":*

```
>>> a = "Algo"
>>>b= None
>>> a is not none , b is none
(true, true)
```

Para inverter o teste de identidade, nós usamos o `not is`.

O objetivo do operador de identidade é verificar se, entre duas referências de objetos, há referência ao mesmo objeto, ou verificar se um objeto é `None`. Se quisermos comparar os valores dos objetos devemos usar um operador de comparação ao invés deste procedimento.

Operadores de comparação

O Python fornece o padrão para operadores de comparação binária, com as semânticas esperadas: < menor que, <= menor que ou igual a, == igual a, != não igual a, > = maior que ou igual a, e > maior que. Estes operadores comparam valores de objetos, isto é, os objetos que a referência do objeto utiliza em sua comparação para referir-se a ele. Aqui estão alguns exemplos digitados num Shell Python:

```
>>>b = 6
>>>a = = b
False
>>>a<b
True
```

```
>>>a<= b, a !=b,a =b a>b
(True, True, False, False)
```

Tudo está como esperávamos com os inteiros. Similarmente, as strings aparecem também para comparar adequadamente:

```
>>>a = "muitos caminhos"
>>>b = "muitos caminhos"
>>> a is b
False
>>>a == b
True
```

Ainda que `a` e `b` sejam objetos diferentes (possuem identidades diferentes), eles possuem os mesmo valores, então, eles comparam igualmente. Esteja atento, porém, porque o Python usa o UniCode para a representação de strings, comparando strings que contêm caracteres não ASCII, logo, isto pode se tornar um problema muito sério e complicado, ainda que não pareça de início – nós iremos discutir isso de forma mais completa no Capítulo 2.

Particularmente, uma característica muito boa dos operadores de comparação do Python é que eles podem ser conectáveis. Por exemplo:

Comparando Strings ☞ 63

```
>>>a = 9
>>>0 <= a <= 10
True
```

Este é um jeito bacana de testar um determinado item de dados que está no range ao invés de ter que fazer duas comparações separadas e ligadas o `and`, como a maioria das linguagens requerem. Também possui a adicional virtude de avaliar o item de dados apenas uma vez (desde que ele apareça apenas uma vez na expressão), algo que pode fazer uma grande diferença caso a computação de dados do valor do item seja muito cara, ou caso acessar o item de dados cause efeitos colaterais.

Graças ao "forte" aspecto da tipagem dinâmica do Python, comparações que não fazem sentido irão causar uma exceção que será lançada. Por exemplo:

```
>>> "três" < 4
Traceback (most recent call last):
...
TypeError :unordenable types : str( )< int( )
```

Quando uma exceção é lançada e não é tratada, Python imprime um traceback junto com a mensagem de erro da exceção. Para esclarecer melhor, nós omitimos a parte da saída da traceback, substituindo-a com uma elipse*. O mesmo `TypeError` poderia ocorrer se nós escrevêssemos "3"<4, já que o Python não vai tentar adivinhar nossas intenções- a abordagem correta é explicitar a conversão, por exemplo, int ("3")<4, ou usar os tipos comparáveis, que são ambos os inteiros ou ambas as strings.

Python nos possibilita uma fácil criação de tipos de dados personalizados que irão integrar-se de maneira perfeita; por exemplo, nós poderíamos criar nosso próprio tipo numérico personalizado o que seria capaz de participar de compa-

rações com o tipo `int` inteiro e com outros tipos numéricos personalizados, mas não com a string ou outros tipos não numéricos.

O operador de Associação

Fuzzy-Bool Alternativo ☞ 248

Para tipos de dados sequenciais ou colecionáveis, como strings, listas e tuplas, nós podemos testar a associação com o operador `in` e, para ausência, o operador `not in`. Por exemplo:

```
>>> p= (4, "sapo", 9, -33, 9, 2)
>>> 2 inp
True
>>> "cachorro" not in p
True
```

Para listas e tuplas, o operador `in` usa uma procura linear, que pode se tornar lenta para coleções muito grandes (dezenas de milhares de itens ou mais). Por outro lado, `in` é muito rápido quando usado com um dicionário ou um set. Esses dois tipos de dados colecionáveis são abordados no Capítulo 3. Aqui está como `in` pode ser usado com uma string:

```
>>frase = "A Paz não é permitida durante as festas"
>>> "v" in frase
True
>>> "anel" em frase
True
```

Convenientemente, no caso das strings, o membership operator pode ser usado para teste de substrings de qualquer tamanho. (Como percebido anteriormente, um caractere é apenas uma string de tamanho 1.)

Operadores Lógicos

Python fornece três operadores lógicos: `and`, `or` e `not`. Tanto `and` como `or` usam um curto-circuito lógico e retornam a operando que determinou o resultado – eles não retornam um boleano (a menos que eles realmente possuam operando boleanas). Vamos ver o que isto significa na prática:

```
>>> cinco = 5
>>> dois = 2
>>> zero = 0
>>> cindo and dois
2
>>> dois and cinco
5
>>> cinco and zero
0
```

*Um "traceback" (alguma vezes chamado de um backtrace) é uma lista que contém todas as invocações feitas do ponto de onde a exceção não tratada ocorreu de volta ao topo da pilha de invocações.

Se a expressão ocorre em um contexto boleano, o resultado é avaliado como um boleano; logo, a expressão anterior pode se tornar `True, True e False`, isto é, uma declaração `if`.

```
>>>nada= 0
>>>cinco or dois
5
>>>dois or cinco
2
>>>zero or cinco
5
>>>zero or nada
0
```

O operador `ou` é similar; aqui, o resultado num contexto boleano poderia ser `True, True` e `False`.

O operador unário `not` avalia se ele é um argumento em um contexto boleano, e, como é de se esperar, sempre acaba retornando um resultado boleano; então, para continuar o exemplo anterior, `not (zero or nada)` poderá produzir `True` e `not dois` poderá produzir `False`.

Parte #5: Controle do fluxo das declarações

Nós mencionamos anteriormente que cada declaração encontrada em um arquivo `.py` é executada em turno, começando com a primeira e, progressivamente, linha por linha. O fluxo (ou corrente) de controle pode ser desviado por uma função ou método invocado ou por um controle de estrutura, tal como um desvio condicional ou uma declaração *loop*. O controle também é desviado quando uma exceção é lançada.

Nesta subseção, nós iremos ver a declaração Python `if` e seus loops `while` e `for`; as considerações deferidas das funções ficam para a Parte #8, e os métodos ficam para o Capítulo 6. Nós iremos também, verificar as bases da manipulação de exceções; e iremos abordar o sujeito no Capítulo 4. Mas, primeiro, vamos iluminar melhor alguns itens de terminologia.

Uma expressão *Boleana* é qualquer coisa que possa ser avaliada para produzir um valor Boleano (`True ou False`). No Python, uma expressão é avaliada como `False,` caso seja a constante pré-definida `False`, o objeto especial `None`, uma sequência de coleção vazia (e.g., uma string, lista ou tupla vazia), ou um item de dado numérico de valor 0, qualquer outra coisa é considerada como `True`. Quando criamos nossos próprios tipos de dados personalizados (e.g., no Capítulo 6), podemos decidir por nós mesmos o que eles irão retornar num contexto Boleano.

Na linguagem Python, um bloco de códigos, isto é, uma sequência de uma ou mais declarações, é chamado de uma *suite*. Porque algumas das sintaxes Python requerem que uma suíte esteja presente, o Python providencia a palavra-chave `pass,` que é uma declaração que não faz nada e que pode ser usada quando uma suíte é requisitada (ou onde nós queiramos indicar, o que é considerado ser um caso particular), mas nenhum processamento é necessário.

A declaração if

A sintaxe geral para a declaração `if` do Python é esta:*

```
If  boleana_expressão1:
    Suite1
elif boleana_expressão2:
    Suite2
...
elif boleana_expressãoN:
    suiteN
else:
    else_suíte
```

Pode existir zero ou mais cláusulas `elif`, e, a cláusula final `else` é opcional. Se quisermos relatar para um caso particular, mas não quisermos fazer nada se isto ocorrer, podemos usar o `pass` como uma suíte de desvio.

A primeira coisa que chama a atenção de usuário de C++ ou Java é que não existem parênteses ou chaves. Outra coisa que se nota é dois pontos (delimitador): este é parte da sintaxe e é muito fácil de se esquecer a princípio. Dois pontos são usados com `else`, `elif` e, essencialmente, em qualquer outro lugar onde uma *suite* estiver adiante.

Diferente da maioria das outras linguagens de programação, Python utiliza uma indentação para dar significado às suas estruturas de blocos. Alguns programadores não gostam disso, especialmente antes de tentarem usar isso , e acabam levando esta questão para o lado pessoal. Mas, o código é necessário apenas alguns dias para se acostumar com isso, e, após algumas semanas ou meses, se torna muito melhor e menos desordenado para se ler do que um código que utiliza colchetes.

Já que suítes são indicadas com o uso de uma indentação, a pergunta que naturalmente nos incomoda é: "Que tipo de indentação?". A guia de boas práticas do Python recomenda quatro espaços por nível de indentação, e somente espaços (não podem ser tabs). A maioria dos editores de texto mais modernos pode ser configurada para gerenciar este problema automaticamente (o editor do IDLE faz isso, é claro, e muitos outros editores compatíveis com o Python também). Python irá funcionar bem com qualquer número de espaços, tabulações ou uma mistura dos dois, desde que usado uma identação consistente. Neste livro, nós iremos seguir a guia de boas práticas oficial do Python.

Aqui está um exemplo muito simples da declaração se:

```
if x:
    print ("x é não zero)
```

Neste caso, se a condição `(x)` é avaliada como `True` a suíte (a invocação da função `print( )`), a mesma será executada.

---

* Neste livro, elipses (...) são usadas para indicar linhas que não foram mostradas.

```
if linhas < 1000:
    print("pequeno")
elif linhas <10000:
    print ("médio")
else:
    print("grande")
```

Esta é uma declaração sutilmente mais elaborada que imprime uma palavra que descreve o valor das variáveis `lines`.

A declaração while

A declaração `while` é usada para executar uma suíte zero ou mais vezes: o número de vezes vai depender do estado do `while` da expressão boleana no loop. Aqui está a sintaxe:

```
while boleana_expressão
    suite
```

Na verdade, a sintaxe completa do loop de `while` é mais sofisticada do que isto, já que tanto `break` como `continue` são suportadas, e também uma cláusula opcional `else` que nós iremos discutir no Capítulo 4. A declaração `break` transfere o controle para a declaração seguinte, o mais interno do loop, no qual a declaração `break` aparece – isto é, ela quebra o loop. A declaração `continue` transfere o controle ao início do loop. Tanto `break` quanto `continue` são usados, normalmente, dentro da declaração `if` para mudar condicionalmente um comportamento do loop.

```
while True
    item = get_next_item( )
    if not item:
       break
    process_item(item)
```

Este loop `while` possui uma estrutura muito típica e vai funcionar na medida em que ainda existam itens para ele processar. (Tanto `get_next_item ( )` quanto `process_item( )` são supostas funções personalizadas, definidas num outro lugar). Neste exemplo, a suíte da declaração `while` contém uma declaração `if`, a qual, por si mesma, tem uma suíte – como deve ser – neste caso, consiste de uma única declaração `break`.

A declaração for...in

O loop do Python `for` usa novamente a palavra chave `in` (a qual, em outros contextos, é o membership operator) e possui a seguinte sintaxe:

```
for variável in reiterável
  Suíte
```

Assim como o loop `while`, o loop `for` suporta `break` e `continue`, e também pos-

sui uma cláusula opcional `else`. A `variável` é configurada para referir-se para cada objeto no `interable` em turnos. Um `interable` é qualquer tipo de dados que possa ser iterado e estar apto a ter strings incluídas (em que a iteração é caracterizada pelo caractere), listas, tuplas e outros tipos de dados colecionáveis do Python.

```
for pais in["Denmark" , "Finlad", "Norway", "Sweden"]:
  print (pais)
```

Aqui, nós levamos em consideração uma abordagem muito simples para imprimir uma lista de países. Na prática, isto é muito mais comum do que usar uma variável:

```
Paises = ["Denmark", "Finland", "Norway", "Sweden"]
for pais em paises
  print (pais)
```

De fato, uma lista completa (ou tupla) pode ser impressa usando a função `print ( )` diretamente, por exemplo, `print(paises)`, mas nós normalmente preferimos imprimir coleções usando um loop `for` (ou uma lista de compreensão, abordada mais adiante) para alcançar um controle completo sobre a formatação.

```
for letra in "ABCDEFGHIJKLMNOPQRSTUWXYZ":
  if letra in "AEIOU":
    print(letra, "é uma vogal")
  mais:
    print(letra, "é uma consoante")
```

Neste fragmento, o primeiro uso da palavra-chave `in` é parte de uma declaração de `for`, com a variável `letra` tomando os valores `"A"`, `"B"` e assim por diante, até `"Z"`, mudando a cada iteração do loop. Na segunda linha do fragmento, nós usamos `if` novamente, mas, desta vez, como operador teste de associação. Note também que este exemplo mostra suites aninhadas. A suíte do loop `for` é a declaração `if...else`, e tanto o desvio `if` como o `else` possuem suas próprias suites.

Lista de compreensões ☞ 110

Manipuladores de exceção básicos

Muitas das funções e métodos Python indicam erros ou outros eventos importantes lançando uma exceção. Uma exceção é um objeto como qualquer outro objeto Python, e, quando convertido em uma string (e.g., quando imprimido), a exceção produz uma mensagem de texto. Uma forma simples da sintaxe para manipuladores de exceção é esta:

```
Try:
  Testar_suíte
Except exceção as variável:
  Exceção_suíte1
...
Except exceçãoN as variávelN>
  Exceção_suíteN
```

Perceba que a parte `as variável` é opcional; nós podemos apenas notar que uma exceção particular foi lançada e não realmente importanta-nos

A sintaxe completa é mais sofisticada; por exemplo, cada cláusula `excepct` pode identificar múltiplas exceções e há uma cláusula opcional `else`. Tudo isto é abordado no Capítulo 4.

A lógica funciona da mesma maneira. Se as declarações de `try` da suíte do bloco de códigos forem todas executadas sem lançar uma exceção, os blocos `except` serão omitidos. Se uma exceção é lançada dentro do bloco `try`, o controle é imediatamente passado para a suíte correspondente à primeira exceção equivalente – isto significa que qualquer declaração na suíte que siga aquela que causou a exceção não será executada. Se isto ocorrer e se a parte `try variável` for determinada e, depois, colocada dentro da suite manipulador-de-exceção, `variável` irá se referir ao objeto de exceção.

Se uma exceção ocorre na manipulação do bloco `except`, ou se uma exceção que for lançada não bater com qualquer um dos blocos `except`, em primeiro lugar, Python vai procurar por um bloco `except` correspondente mais próximo do escopo seguinte. A procura por um manipulador de exceção adequado trabalha de escopo e até a chamada de pilha até que um identificador adequado seja encontrado e que a exceção seja tratada, mas, se não for encontrado nenhum manipulador adequado, o programa encerra com uma exceção não manipulada. No caso de uma exceção não manipulada, Python imprime uma traceback, assim como as mensagens de texto da exceção.

Aqui está um exemplo:

```
s= imput("entre um iteirador: ")
try:
  i= int (s)
  print ("inteiro valido introduzido: ", i)
Except ValueError as err:
  print (err)
```

Caso o usuário entre com "3.5", a saída será:

```
invalid literal for int() with base 10: '3.5'
```

Mas, se a entrada for feita com "13", a saída será:

```
Inteiro válido introduzido: 13
```

Muitos livros consideram que o manipulador de exceções seja um tópico muito avançado e adiam o assunto o quanto for possível. Mas, lançar e, especialmente, manipular exceções, é fundamental para que o Python funcione; fato que nos levou a fazer uso deste conceito desde o princípio. E, assim como nós pudemos ver, usar manipuladores de exceções pode fazer com que os códigos fiquem muito mais legíveis, separando os casos "excepcionais" dos processos que realmente importam.

Parte #6: Operadores Aritméticos

Python fornece um conjunto completo de operadores aritméticos, incluindo operadores binários para as quatro operações matemáticas básicas: = adição, - subtração, * multiplicação e / divisão. Além do mais, muitos tipos de dados Python podem ser usados com operadores de incremento associado como + = *. Os operadores+, - e * comportam-se como o esperado quando ambos operadores são inteiros:

```
>>> 5+6
11
>>> 3-7
-4
>>> 4*8
32
```

Perceba que o – pode ser usado tanto como um operador unário (negativo) quanto como um operador binário (subtração); algo comum nas outras linguagens de programação. Onde Python se difere dos demais é na parte da divisão:

```
>>> 12/3
4.0
>>> 3/2
1.5
```

O operador de divisão produz um valor de ponto flutuante, não um inteiro; muitas outras linguagens irão produzir um inteiro, truncando qualquer parte fracionada. Se nós precisarmos de um resultado inteiro, poderemos sempre converter usando `int ( )` (ou usando o operador de divisão de truncamento //, discutido mais adiante).

```
>>> a = 5
>>> a
5
>>> a += 8
>>> a
13
```

Num primeiro aspecto, as declarações anteriores não são muito surpreendentes, particularmente para as pessoas familiarizadas com linguagens como C.

Nessas linguagens, associação de incremento é um atalho para associar os resultados de uma operação – por exemplo, a+=8 é, essencialmente, o mesmo que a=a+8. De qualquer maneira, existem duas sutilezas importantes aqui, um específico do Python e um sobre operadores de incremento em qualquer outra linguagem.

O primeiro ponto para relembrar é que o tipo de dados `int` é imutável – isto é, uma vez designado, um valor `int` não pode ser modificado. Logo, o que realmente acontece atrás das cenas quando um operador de incremento associado é utilizado em um objeto imutável, é que a operação é feita e um objeto que contêm resultado é criado, e, depois, a referência-alvo do objeto é reatada para referir-se ao objeto resultado ao invés do objeto que foi referido anteriormente. Portanto, no caso anterior, quando a declaração a+=8 foi encontrada, Python computou a+8, armazenando o resultado em um novo objeto int e, depois, amarrando novamen-

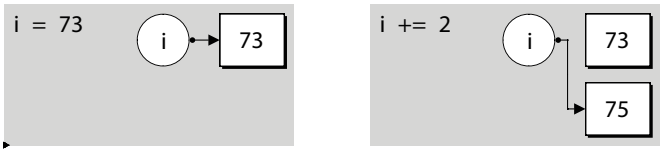

Figura 1.3 *Incremento Associável de um objeto imutável*

te, para que ele se refira a este novo int. (E, se o objeto original `a` foi referido de modo que não possua mais referência de objeto referidos a ele, este será programado para o garbage collection). Figura 1.3 ilustra este ponto.

Outra sutileza é que `a operador= b` não é igual ao `a= a operador b`. A versão incrementada olha o valor de `a` apenas uma vez, logo, isto pode ser potencialmente rápido. Ademais, se `a` é uma expressão complexa (e.g., uma lista de elementos com uma posição de índice calculado como sendo itens[offset+índice], usar a versão incrementada causará menos erros, já que o calculo precisa modificar o mantenedor e este precisa mudar para apenas uma ao invés de duas expressões.

Python sobrecarrega (i.e., reusos para um tipo de dados diferente) os operadores o(a)+ e += tanto em string quanto em listas; o primeiro significa concatenação e o outro significa anexação para string e extend (anexar outro lista) para listas:

```
>>>nome = "John"
>>>nome=+ " Doe"
'JohnDoe'
>>>nome+= "Doe"
>>>nome
'John Doe'
```

Assim como os inteiros, as strings são imutáveis, logo, quando += é utilizado, uma nova string é criada e a referênci do lado esquerdo da expressão é reatado a ela, exatamente como escrito anteriormente pelo `ints`. Listas suportam a mesma sintaxe, mas são diferentes por trás das cenas:

```
>>>semeante = ["gergelim", "girassol"]
>>>semente=+ ["abóbora"]
>>>semente
['gergelim', 'girassol', 'abóbora']
```

Já que listas são mutáveis, quando =+ é usado, o objeto lista original é modificado, então, nenhum reatamento de `seeds` é necessário. Figura 1.4 mostra como isto ocorre.

Já que a sintaxe Python, de maneira muito inteligente, esconde a distinção entre tipos de dados mutáveis e imutáveis, afinal de contas, por que são necessários os dois tipos? As razões para isso são mais pelo desempenho. Tipos imutáveis são potencialmente muito mais eficientes para implementar (já que eles nunca mudem) do que tipos mutáveis. Ademais, alguns tipos de dados colecionáveis, por exemplo, sets, podem trabalhar apenas com tipos imutáveis. Por outro lado, tipos mutáveis são bem mais convenientes de usar. Onde esta distinção realmente importa, nós discutiremos isso – por exemplo, no Capítulo 4, quando discutirmos argumentamos setados como default para funções customizada no Capítulo 3, quando discutirmos

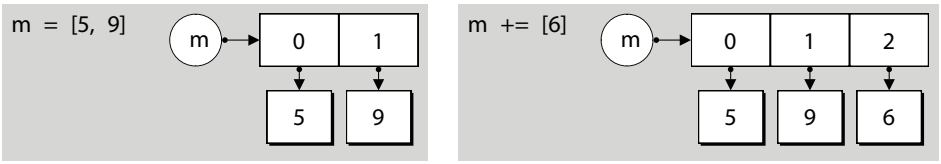

Figura 1.4 *Incremento Associado de um objeto mutável*

sobre listas, sets e alguns outros tipos de dados, e novamente no Capítulo 6,quando discuture. como criar tipos de dados personalizados.

A operador do lado direito para o operador += da lista deve ser um interable; caso não seja, uma exceção é lançada:

```
>>> seeds += 5
Traceback (most recent call last):
...
TypeError: 'int' object is not iterable
```

A forma correta para prolongar uma lista é usar um objeto interable, como outra lista:

```
>>>semente += [5]
>>>semente
['gergelim', 'girassol', 'abóbora', 5]
```

E, é claro, o objeto iterable utilizado para prolongar a lista pelo fato de ter, por si mesmo, mais de um item:

```
>>>semear += [9, 1, 5, "papoula"]
>>>semear
['geregelim', 'girassol', 'abóbora', 5, 9, 1, 5,  'papoula']
```

Anexando uma string simples – por exemplo, "durian" – ao invés de uma lista contendo uma string, ["durian"], leva a um resultado lógico, mas, quem sabe, um resultado surpreendente:

```
>>>semente = ['gergelim', 'girassol', 'abóbora']
>>>semente += "durian"
>>>semente
['gergelim', 'girassol', 'abóbora', 'd', 'u', 'r', 'i', 'a', 'n', ]
```

O operador += da lista estende a lista anexando a cada item cada iterable que é fornecido; e, uma vez que uma string é iterável, ela leva cada caractere na string a ser anexado individualmente. Se nós usarmos na lista o método `append( )`, o argumento será sempre adicionado como um único item.

Parte #7: Entrada/Saída

Para estarmos aptos a escrever programas genuinamente completos, nós devemos estar também aptos a produzir entradas – por exemplo, do usuário para o terminal, e dos arquivos para o terminal- e, da mesma forma, produzir saídas,

ambas para o terminal ou para os arquivos. Nos já fizemos uso da função embutida do Python `print ( )`, ainda que nós tenhamos adiado sua abordagem completa para o Capítulo 4. Nesta subseção, nós iremos nos concentrar no terminal I/O e usar redirecionamentos shell para ler e escrever arquivos.

Python fornece a função embutida `input( )` para aceitar entrada dos usuários. Esta função toma um argumento de string opcional (que é impresso no terminal); ele, então, espera o usuário digitar uma resposta e finalizar pressionando `Enter` (ou `Return`). Se o usuário não digitar texto algum, mas apenas apertar o `Enter`, a função `input( )` retorna uma string vazia; de qualquer maneira, ele retorna uma string contendo o que o usuário digitou, sem qualquer finalizador de linha.

Aqui está nosso primeiro e completo programa "útil"; ele se adianta em muito se o compararmos com os programas anteriores – a única coisa realmente nova é exibida na função `( )`:

```
print("Type integers, each followed by Enter; or just Enter to finish")

total = 0
count = 0

while True:
   line = input("integer: ")
   if line:
      try:
          number = int(line)
      except ValueError as err:
          print(err)
          continue
      total += number
      count += 1
   else:
          break
   if count:
      print("count =", count, "total =", total, "mean =", total /
      count)
```

O programa (no arquivo `sum1.py` nos exemplos do livro) possui apenas 17 linhas executáveis. Aqui está o que acontece em uma típica execução:

```
Type integers, each followed by Enter; or just Enter to finish
number: 12
number: 7
number: 1x
invalid literal for int() with base 10: '1x'
number: 15
number: 5
number:
count = 4 total = 39 mean = 9.75
```

Ainda que este programa seja muito curto, é amplamente robusto. Se o usuário entrar com uma string que não possa ser convertida em um inteiro o problema

é assumido por um manipulador de exceção que imprime uma mensagem adequada e envia o controle para o início do loop ("continuar o loop"). E a última declaração se assegura que o usuário não entre com qualquer número; o sumário não é colocado na saída e a divisão por zero é prevenida.

Exemplos do livro ☜ 3

A manipulação de arquivos é totalmente abordada no Capítulo 7; mas, por enquanto, nós podemos criar arquivos simples direcionando as saídas da função `print ( )` do terminal. Por exemplo:

```
C:\>teste.py>resultado.txt
```

causará a saída de uma função simples `print( )` chamada no programa fictício `teste.py`, que deve ser escrito no arquivo `resultados.txt`. Esta sintaxe funciona com o terminal do Windows e do Unix. Para o Windows, nós devemos escrever `C:\Python30\python.exe teste.py>resultados.txt`, caso o Python 2 estiveja configurada como padrão no computador, ou `python teste.py>resultados.txt`, se Python 3 vier primeiro no PATH (ainda que nós não queiramos falar sobre isso novamente), e para Unixes nós devemos fazer o programa executável (`chmod +x teste.py`) e, então, invocá-lo escrevendo `./teste`: a menos que aconteça de o diretório estar no PATH.

Ler dados pode ser alcançado redirecionando um arquivo dado como entrada numa forma analógica feito para redirecionar a saída. De qualquer maneira, se nós usarmos o redirecionamento com `sum1.py`, o programa poderá falhar. Isto ocorre porque a função `output( )` lança uma exceção quando recebe um caractere EOF (fim de arquivo). Aqui está uma versão muito mais robusta (`sum2.py`) que pode aceitar entradas do usuário, quando este digitar manualmente no teclado, ou via redirecionamento de arquivo:

```
print("Type integers, each followed by Enter; or ^D or ^Z to fi-
nish")
total = 0
count = 0

while True:
   try:
       line = input()
       if line:
         number = int(line)
         total += number
         count += 1
   except ValueError as err:
       print(err)
       continue
   except EOFError:
       break

if count:
   print("count =", count, "total =", total, "mean =", total /
   count)
```

Dada a linha de comando `sum2.py<dados\sum2.dad` (onde o arquivo `sum2.`

`dado` contêm uma lista de números, um por linha, e está nos exemplos do subdiretório `dados`), a saída para o terminal será:

```
Type integers, each followed by Enter; or ^D or ^Z to finish
count = 37 total = 1839 mean = 49.7027027027
```

Nós criamos muitas pequenas mudanças para fazer o programa mais adequado para usar tanto de forma interativa quanto usar o redirecionamento. Primeiro, nós temos que mudar a terminação de uma linha em branco para um caractere EOF (Ctrl+D no Unix, Ctrl+Z, Enter no Windows). Isto torna o programa mais robusto quando ele precisa manipular arquivos de entrada que contêm linhas em branco. Precisamos também deixar de imprimir um prompt para cada número, já que não faz sentido ter um para cada redirecionamento de saída. E, por fim, temos de usar um bloco único `try`, com dois manipuladores de exceção.

Perceba que, se um inteirador inválido for acionado (tanto via teclado ou devido a uma linha "ruim" dos dados em um redirecionamento de arquivos de entrada), a conversão `int( )` vai lançar uma exceção `ValueError` e o fluxo do controle será imediatamente enviado para o bloco relevante `except` – isto significa que tanto `total` quanto `count` não serão incrementados quando um dado inválido for encontrado, o que é exatamente o que esperamos que aconteça.

Nós poderíamos simplesmente ter usado dois manipuladores de exceção separados ao invés de todo este processo:

```
while True:
   try:
       line = input()
       if line:
   try:
       number = int(line)
       except ValueError as err:
         print(err)
   continue

   total += number
   count += 1
except EOFError:
   break
```

Mas é preferencialmente melhor agrupar as exceções juntas no final do que fazer o processo principal de organização, o quanto for possível.

Parte #8: Criando e Chamando Funções

É perfeitamente possível escrever programas usando os tipos de dados e controles de estrutura que já abordamos nas partes anteriores. De qualquer maneira, normalmente, nós iremos fazer, essencialmente, o mesmo processo repetidamente, mas com uma pequena diferença, assim como uma diferença de valor inicial.

Python fornece vários sentidos para suítes encapsuladas, como funções que podem ser parametrizadas através dos argumentos que lhe são passados. Aqui está a sintaxe geral para criar uma função:

```
def funçãoNome(argumentos):
Suite
```

Os `argumentos` são opcionais, e múltiplos argumentos devem estar separados por vírgulas. Toda função Python possui um valor de retorno, este valor é `None` por padrão, a menos que nós retornemos a função usando a sintaxe `return value`, em que será retornado o `valor` em cada caso. O valor retornado pode ser apenas um valor ou uma tupla de valores. O valor retornado pode ser ignorado por um invocador que, em cada caso, será simplesmente ignorado.

Note que `def` é uma declaração que trabalha de um jeito similar ao operador associação. Quando def é executado, um objeto de função é criado e uma referência objeto com o nome especificado é criado e configurado para referir-se ao objeto de função. Já que funções são objetos, elas podem ser armazenadas em um tipo de dados colecionável e passadas como argumentos para outras funções, assim como vimos nos capítulos anteriores.

Uma necessidade frequente quando escrevemos aplicações em terminais interativos é que temos de obter um inteiro do usuário. Aqui está a função que faz exatamente isto:

```
def get_int(msg):
 while True:
 try:
Python's "Beautiful Heart" 35
   i = int(input(msg))
   return i
except ValueError as err:
 print(err)
```

Esta função toma um argumento, `msg`. Dentro do loop `while`, o usuário é abordado com um prompt, a fim de dar entrada num inteiro. Se eles entrarem com algo inválido, uma exceção `ValueError` será lançada, a mensagem de erro será impressa e o loop se repetirá. Uma vez que um iteirador válido seja acionado, ele é retornado ao invocador. Aqui está como nós deveremos chamá-lo:

```
idade = obter_int("entrar sua idade:")
```

Neste exemplo, o único argumento unitário é mandatório, porque nós não podemos fornecer um valor padrão. De fato, Python suporta uma sintaxe muito sofisticada e flexível para parâmetros de funções, os quais suportam argumentos padrões. Toda esta sintaxe é abordada no Capítulo 4.

Ainda que criemos nossas próprias funções e isto seja muito satisfatório, em muitos casos não será necessário. Isto ocorre porque Python possui um monte de funções embutidas e uma grande gama de funções nos módulos da biblioteca padrão, então, o que queremos provável e já está pronto e disponível.

Um modulo Python é apenas um arquivo `.py` que contém código Python, assim como funções personalizadas e definições de classes (tipos de dados personalizados), e algumas variáveis. Para acessar a funcionalidade em um módulo, nós devemos importá-lo. Por exemplo:

```
Import sys
```

Para importar um módulo, nós usamos a declaração `import` seguida pelo nome de um arquivo `.py`, mas omitindo a extensão*. Uma vez que um módulo tenha sido importado, nós podemos acessar qualquer funcionalidade, classes ou variáveis que ele contenha. Por exemplo:

```
Print(sys.argv)
```

Este módulo `sys` fornece a variável `argv` – uma lista em que o primeiro item é o nome no qual o programa foi invocado e o segundo e também os itens subsequentes, os argumentos da linha de comando do programa. As duas linhas anteriores mencionadas constituem o programa completo `ecoargs.py`. Se o programa foi invocado com a linha de comando `ecoargs.py`, ele será impresso `['ecoargs.py', '-v']` no terminal. (No Unix, a primeira entrada poderá ser `'./ecoargs.py'`)

No geral, a sintaxe para usar a função de um módulo é `moduloNome.funçãoNome(argumentos)`. Isto faz uso do operador de ponto ("acessar atributos") que nós introduzimos na parte #3. A biblioteca padrão contém vários módulos e nós

iremos fazer uso de muitos deles ao longo do livro. O módulo padrão possui, quebra linha, nomes em minúsculo; logo, alguns programadores irão usar nomes com a primeira letra em maiúsculo (e.g., *MeuModulo*) para seus próprios módulos, a fim de manter a distinção entre um e outro.

Vamos ver um exemplo: o módulo `random` (no arquivo `random.py` da biblioteca padrão), que fornece funções muito utilizadas:

```
Import random
X= random.randint(1, 6)
Y= random.choice(["maçã", "banana", "cereja", "durião"])
```

Operador(.) de ponto operador ☜ 18

Depois de estas declarações terem sido executadas, `x` irá conter um inteiro, inclusive entre 1 e 6; e `y` irá conter uma das strings da lista passada para a função `random.choice( )`.

É conveniente colocar todas as declarações `import` no início dos arquivos `.py`, depois da linha shebang e depois da documentação do módulo. (Documentação de módulos será abordada no Capítulo 5.) Nós recomendamos importar os módulos da biblioteca principal primeiro, depois os módulos da biblioteca de terceiros e, finalmente, seus próprios módulos.

*O módulo `sys`, outro embutido dos módulos, e módulos implementados em C não possuem necessariamente e correspondentemente arquivos `.py` – mas eles são usados exatamente do mesmo jeito que estes outros também o são.

Exemplos

Na seção anterior, nós aprendemos o suficiente sobre Python para escrever programas reais. Nesta seção, nós iremos estudar programas completos que usam apenas o que já foi abordado sobre Python antes. Estes exemplos são tanto para mostrar o que é possível fazer com o que aprendemos quanto para ajudar a consolidar tudo que já foi aprendido até este ponto.

Linha shebang ☜ 10

Nos capítulos subsequentes, nós iremos melhorar a abordagem sobre linguagem Python e suas bibliotecas; dessa forma, estaremos aptos a escrever programas mais concisos e mais robustos do que estes ilustrados aqui – mas, primeiro, devemos dar fundamento para estes exemplos posteriores.

Bigdigits.py

O primeiro programa que nós iremos rever é extremamente curto, ainda que ele tenha alguns aspectos sutis, incluindo uma lista de listas. Aqui está o que acontece: dado um número na linha de comando, o programa libera o mesmo número dentro do terminal usando "grandes" dígitos.

Pelos sites afora, existem punhados de usuários partilhando uma linha de imprimir altamente rápida, que é usada para práticas comuns para cada trabalho impresso do usuário, precedido por uma página de capa que exibirá seus nomes de usuário e alguns outros detalhes de identificação impresssos, que utiliza este tipo de técnica.

Nós iremos rever o código em três partes: a importação, a criação das listas que atam os dados aos programas usados e o processo por si mesmo. Mas, primeiro, vamos ver este exemplo em funcionamento:

```
bigdigits.py 41072819
```

```
   *     *     ***    *****   ***    ***    *    ****
  **    **    *   *       *  *   *  *   *  **   *   *
 * *     *   *     *     *   *  *   *   *   *   *   *
*  *     *   *     *    *      *     ***    *    ****
******   *   *     *   *      *     *   *   *       *
   *     *    *   *   *      *     *   *   *       *
   *    ***    ***    *      *****   ***   ***      *
```

Nós não precisamos mostras o prompt do terminal (ou o ./ para usuários Unix); nós iremos esconcedê-lo daqui para frente.

```
Import sys
```

Já que nós devemos ler em um argumento da linha de comando (o número para a saída), nós precisamos acessar a lista `sys.argv`; logo, começaremos importando o módulo `sys`.

Nós representamos cada número como uma lista de strings. Por exemplo, aqui é zero:

```
Zero = ["  ***  ",
        " *   * ",
        "*     *",
        "*     *",
        "*     *",
        " *   * ",
        "  ***  "]
```

Um detalhe que se deve notar é que a lista `Zero` das strings é espalhada por linhas múltiplas. As declarações Python normalmente ocupam uma única linha, mas elas podem espalhar por múltiplas linhas se estiverem em: uma expressão entre parênteses, uma lista, set ou dicionário literal, uma chamada função de lista de argumentos ou uma declaração de multilinhas, em que cada caractere final de linha exclui a última que escapou, precedendo-a com uma barra lateral (`\`). Em todos estes casos, qualquer número de linhas pode ser espalhada a indêntação não importa para a segunda linha ou para as linhas subsequentes.

Cada lista representa um número que possui sete strings, todas com uma largura uniforme, ainda que esta largura difere de número para número. A lista para outros números segue o mesmo padrão que é usado para zero, ainda que elas estejam "deitadas" para usar menos espaço ao ivés da forma anterior mais clara:

```
One = [" * ", "** ", " * ", " * ", " * ", " * ", "***"]
Two = [" *** ", "*   *", "*  * ", "  *  ", " *   ", "*    ", "*****"]
# ...
Nine = [" ****", "*   *", "*   *", " ****", "    *", "    *", "    *"]
```

Tipos de Set ☞ 112

Tipos de dicionários ☞ 118

A última parte dos dados que precisamos é uma lista de todas as listas de dígitos:

```
Digits = [Zero, One, Two, Three, Four, Five, Six, Seven, Eight, Nine]
```

Nós poderíamos ter criado as listas de `Digits` diretamente, e evitar a criação de variáveis extras. Por exemplo:

```
Digits = [
    ["  ***  ", " *   * ", "*     *", "*     *", "*     *",
     " *   * ", "  ***  "], # Zero
    [" * ", "** ", " * ", " * ", " * ", " * ", "***"], # One
    # ...
    [" ****", "*   *", "*   *", " ****", "    *", "    *",
     "    *"] # Nine
    ]
```

Nós preferimos usar uma variável separada para cada número, tanto pela facilidade de entender melhor quanto porque o código fica mais elegante quando se usa as variáveis.

Nós iremos mencionar o resto do código em uma tentativa para que você possa tentar entender como funciona antes de ler as explicações a seguir.

```
try:
   digits = sys.argv[1]
   row = 0
   while row < 7:
       line = ""
       column = 0
       while column < len(digits):
         number = int(digits[column])
         digit = Digits[number]
         line += digit[row] + " "
         column += 1
   print(line)
   row += 1
except IndexError:
   print("usage: bigdigits.py <number>")
except ValueError as err:
   print(err, "in", digits)
```

O código completo está empacotado em um manipulador de exceção que pode capturar os dois erros que podem ocorrer. Nós começamos restaurando o argumento da linha de comando do programa. A lista `sys.argv` é baseada em 0, como todas as listas Python; o item que está na posição 0 do índice é o nome do programa invocado, logo, quando o programa está rodando, esta lista sempre começa com pelo menos um item. Se nenhum argumento foi dado, nós iremos tentar acessar o segundo item em uma lista de um-item, e isto irá causar uma exceção `IndexError`, que será lançada. Se isto ocorrer, o fluxo do controle é imediatamente enviado para o bloco de manipulação de exceções correspondente e, aqui, nós simplesmente imprimimos o uso do programa. Após o final do bloco try a execução continua; mas não terá mais códigos; logo, o programa simplesmente termina.

Se nenhum `IndexError` ocorrer, a string de `digits` guarda o argumento da linha de comando, que nós esperamos que seja uma sequência de caracteres de dígitos. (Relembrando a parte #2, na qual já explicamos que os identificadores são sensitivos a letras maiúsculas e minúsculas, `digits` e `Digits` são diferentes). Cada grande dígito é representado por sete string e, para dar a saída do número corretamente, nós devemos enviar para a saída o topo da fileira de cada dígito, depois, na próxima fileira, e assim por diante, até que todas as sete fileiras tenham sido enviadas. Nós usamos um loop `while` para iterar cada fileira. Ao invés disso, nós poderíamos também ter feito simples e facilmente o seguinte: `for rowin (0,1,2,3,4,5,6)`: e, depois, nós iremos obter uma maneira muito melhor para atuar com a função `range( )`.

Nós usamos a string da `line` para guardar a fileira de string com todos os dígitos envolvidos. Depois, utilizamos o loop por colunas, isto é, em cada caractere, suces-

sivamente, no argumento da linha de comando. Nós restauramos cada caractere com `digits[column]` e convertemos o dígito em um inteiro chamado `number`. Se a conversão falhar, uma exceção de `ValueError` será lançada e o fluxo do controle será imediatamente transferido para o manipulador de exceções correspondente. Neste caso, nós imprimimos uma mensagem de erro e o controlador a retoma depois de o bloco `try`. Como notado anteriormente, desde que não haja mais códigos até este ponto, o programa irá simplesmente terminar.

Range ☞131

Se a conversão obtiver sucesso, nós usaremos `number` como um índice remissivo dentro da lista `Digits`, a qual nós iremos extrair da lista `digit` das strings. Depois, nós adicionamos a string `row-th` desta lista para a linha que nós estamos construindo e, finalmente, anexamos dois espaços para dar alguma separação horizontal entre cada dígito.

Cada vez que o loop interno `while` finalizar, nós devemos imprimir a linha que estava sendo construída. A chave para entender este programa está onde nós anexamos cada string de fileira de dígitos para a linha de fileiras correntes. Tente executar o programa para sentir como isto funciona. Nós iremos retornar a este programa nos exercícios a fim de melhorar um pouco esta saída.

Generate_grid.py

Uma necessidade que ocorre frequentemente é a geração de dados de teste. Não existe um programa genérico para fazer isto, já que dados de teste são normalmente enormes. Python é geralmente usada para produzir dados de teste porque ela é muito fácil para escrever e modificar programas Python. Nesta subseção, nós iremos criar um programa que gera uma rede de inteiros aleatórios; os usuários podem especificar quantas fileiras e colunas eles desejam o range que abrange os interior. Nós iremos começar olhando (um exemplo) sendo executado:

```
generate_grid.py
rows: 4x
invalid literal for int() with base 10: '4x'
rows: 4
columns: 7


minimum (or Enter for 0): -100

maximum (or Enter for 1000):

   554 720 550 217 810 649 912

   -24 908 742 -65 -74 724 825

   711 968 824 505 741 55 723

   180 -60 794 173 487 4 -35
```

O programa trabalha interativamente e, no início, nós fizemos um erro de digitação quando entramos com o número de fileiras. O programa respondeu imprimindo uma mensagem de erro e depois pedindo-nos para entrar com o número

de fileiras novamente. Para o máximo, nós apenas pressionamos `Enter`, a fim de aceitar a configuração padrão.

Nós iremos rever o código em quatro partes: a importação, a definição de uma função `get_int( )` (uma versão mais sofisticada do que aquela exibida na Parte #8), a interação do usuário para obter os valores de uso e o processo por si mesmo.

```
Importar aleatório
```

Nós precisamos do módulo aleatório para nos dar acesso à função `random.randint( )`.

```
def get_int(msg, minimum, default):
   while True:
       try:
line = input(msg)
         if not line and default is not None:
         return default
         i = int(line)
         if i < minimum:
         print("must be >=", minimum)
         else:
return i
   except ValueError as err:
       print(err)
```

aleatório. aleaint ( ) ☞36

Esta função necessita de três argumentos: uma string de mensagem, um valor mínimo e um valor padrão. Se o usuário apenas pressionar o Enter, haverá duas possibilidades. Se `default` é `None`, isto é, nenhum valor padrão foi dado, o fluxo do controlador irá cair através da linha `int( )`. Ali, a conversão irá falhar (já que "não seja convertido em um espaço") e uma exceção `ValueError` será lançada. Mas, se `default` não for `None`, então ele será retornado. De outra maneira, a função irá empreender a conversão do texto que o usuário entrou em um iteirador, e, se a conversão é bem-sucedida, ela irá, então, checar se o inteiro possui um valor pelo menos igual ao mínimo que foi especificado.

Portanto, a função irá sempre retornar, tanto `default` (se o usuário pressionou Enter) como em um inteiro válido, que será bem maior do que, ou igual, o `minimum` especificado.

```
Fileiras = obter_int("fileiras: ", 1, Nada)
Colunas = obter_int("colunas:", 1, Nada)
Mínimo = obter_int("mínimo (ou Enter para 0): ", -1000000. 0)

Padrão = 1000
Se padrão <mínimo
     Padrão = 2 * mínimo
Máximo = obter_int("máximo (ou Enter para " + str(padrão) +"): ",
            Mínimo, padrão)
```

Nossa função `get_int` transforma em algo fácil obter o número de fileiras e colunas e o valor aleatório mínimo que o usuário deseja. Para fileiras e colunas, nós damos um valor padrão de `None`, isto é, nenhum padrão; logo, o usuário deve en-

trar com um iteirador. Neste caso, para mínimo, nós fornecemos um valor padrão de 0 e, para o máximo, nós fornecemos um valor padrão de 1000, ou duplamente o mínimo, caso o mínimo seja maior ou igual a 1000.

Como pôde ser notado nos exemplos anteriores, a função do argumento para chamar as listas pode passar por cima de qualquer número de linhas, e a indentação é irrelevante para suas linhas secundárias ou subsequentes.

Uma vez que nós já saibamos quantas fileiras e colunas o usuário exige e o valor mínimo e máximo dos números aleatórios que eles querem, já é possível estarmos prontos para fazer o processamento.

```
rows = get_int("rows: ", 1, None)
columns = get_int("columns: ", 1, None)
minimum = get_int("minimum (or Enter for 0): ", -1000000, 0)
default = 1000

if default < minimum:
   default = 2 * minimum
maximum = get_int("maximum (or Enter for " + str(default) + "): ",
minimum, default)
```

Para gerar a grade, nós usamos três loops `while`: o externo, que trabalha por colunas, o do centro, que também funciona por colunas, e o do interior, que funciona por caracteres. No centro do loop, nós obtemos um número aleatório no range especificado e, então, convertido em uma string. O loop `while` interno é usado para encher a string com os espaços impressos entre as linhas; logo, cada número é representado por uma string de 10 caracteres no total. Nós usamos a string `line` para acumular os números em cada fileira e imprimir a linha após cada número de colunas ter sido adicionado. Isto completa nosso segundo exemplo.

Python fornece funcionalidades de formatação de string muito sofisticadas, assim como um excelente suporte para loops `for...in`, então, versões mais realistas do `bigdigits.py` e do `generate_grid.py` poderiam usar loops `for...in`, e `generate_grid.py` poderia ser usado na capacitação de formatações de string do Python ao invés de grosseiros "enchedores" de espaços. Mas nós estivemos limitando-nos às oito partes do Python, introduzidas neste capítulo, e elas são suficientes o bastante para escrever programas completos e muito úteis. Em cada capítulo subsequente, nós iremos aprender novas características Python, de modo a progredir pelos programas que veremos através do livro e, por fim, estaremos aptos a escrever de modo sofisticado.

Sumário

Neste capítulo, nós aprendemos como editar e executar programas Python e revisamos alguns pequenos, mas completos, programas. Mas a maioria das páginas destes capítulos foram dedicadas às oito partes do "lindo coração" do Python – suficiente para escrever programas reais.

Nós começamos com dois dos mais básicos tipos de dados do Python, `int` e `str`.

Inteiros literais são escritos do mesmo jeito que muitas outras linguagens de programação o fazem. Strings literais são escritas usando aspas únicas ou duplas; não importa se, qua tipo desde que, os mesmo tipos de aspas são usadas. Nós podemos fazer conversão entre string e inteiros, por exemplo, `int` ("250") e `str`(125). Se a conversão de um inteiros falhar, uma exceção `ValueError` será lançada; enquanto que quase tudo o mais pode ser convertido em uma string.

Strings são sequências, logo, aquelas funções e operações que podem ser usadas com sequências podem ser usadas com string. Por exemplo, nós podemos acessar um caractere particular usando o item operador acessor ([ ]), concatenando string usando + e anexando uma string à outra usando +=. Já que string são imutáveis por trás das cenas, a anexação cria uma nova string que é concatenada às string dadas e reata o objeto da string do lado esquerdo referente à string resultante. Nós podemos, também, iterar, em uma string, caractere por caractere, usando um loop `for...in`. E também é possível usar a função embutida `len( )` para informar quantos caracteres contêm uma string.

Para objetos imutáveis, como strings, inteiros e tuplas, nós podemos escrever nossos códigos, ainda que uma referência de objeto seja o próprio objeto a ser referenciado. É possível fazer isto com objeto mutáveis, ainda que qualquer mudança feita em um objeto mutável afete todas as ocorrências do objeto (e.g., todos as referências de objetos para o objeto); nós iremos abordar este assunto no Capítulo 3.

Python fornece muitos tipos de dados colecionavéis embutidos e possui alguns outros em sua biblioteca padrão. Nós aprendemos sobre os tipos `lista` e `tupla` e, de forma particular, como criar tuplas e listas de literais, por exemplo, `even=[2,4,6,8]`. Listas, como tudo o mais no Python, são objetos, portanto, nós podemos chamar métodos para elas – por exemplo, `even.append(10)` irá adicionar um item extra para a lista. Já que string, listas e tuplas são sequências, é possível, portanto, iterá-las, item por item, usando um loop `for...in` e descobrir em quantos itens elas têm usado a função `len( )`. Nós podemos, também, recuperar um item particular numa lista ou tupla usando o operador de acesso a itens ([ ]), concatenando duas listas ou tuplas, usando + e anexando uma na outra usando +=. Se nós quisermos anexar um único item para uma lista, nós devemos também usar `list`, `append( )` ou usar += com o item criado dentro de uma lista de um só item- por exemplo, `even +=[12]`. E, já que listas são mutáveis, é possível usar [ ] para modificar itens individuais, por exemplo, `even[1]=16`.

Os rápidos operadores de identidade `is` e `not is` podem ser usados para checar se existe, entre duas referência de objetos, referência à mesma coisa – isto é particularmente muito utilizado quando se está checando em oposição ao objeto singular `None`. Todos os operadores de comparação usuais estão disponíveis (<, <=, = =, !=, >=, >), mas eles podem ser usados apenas com tipos de dados compatíveis e, então, apenas as operações são suportadas. Os tipos de dados que vimos até agora – `int, str, list` e `tuple` – são todos capazes de suportar o grupo de operadores de comparação. Ao comparar tipos incompatíveis, por exemplo, comparar um `int` com um `str` ou uma `list`, iremos produzir, de maneira completa e sensível, uma exceção `TypeError`.

Python suporta os operadores lógicos padrões `and`, `or` e `not`. Tanto `e` como `ou` são operadores de curto-circuito que retornam a operação que determinou seus resultados – e este pode não ser um boleano (ainda que este resultado possa ser convertido num boleano); já o operador `not` sempre retorna como `False` ou `True`.

Nós podemos testar associações de tipos de sequências, incluindo listas, strings e tuplas, usando os operadores `in` e `not in`. Testes de associações usam uma procura linear lenta nas listas e tuplas e um algoritmo híbrido, potencialmente mais rápido para strings, mas o resultado é raramente abordado, exceto para string, listas e tuplas muito longas. No Capítulo 3, nós iremos aprender sobre os vetores de associação do Python e como configurar tipos de dados colecionáveis, ambos do teste de associação que foi executado de maneira muito rápida. Também é possível encontrar um objeto de tipo variável (i.e, o tipo do objeto que a referância do objeto se refere) usando `type ( )` – mas esta função é normalmente utilizada apenas para depurar e testar.

Python fornece muitas estruturas de controle, incluindo a ramificação condicional com `if...elif...else`, loopins condicionais com `while`, loop sobre sequências com `for...in`, e manipulador de exceção com blocos `try...except`. Tanto `while` quanto o loop `for...in` podem ser finalizados prematuramente usando a declaração `break` e ambos podem transferir o controle para o início usando `continue`.

Os operadores aritméticos comuns são suportados, incluindo +, -, * e /, ainda que Python seja incomum com /; esta sempre produz um resultado em ponto flutuante, mesmo que nas duas operações sejam inteiros. (A divisão de truncamento que muitas outras linguagens utilizam está também disponível no Python como //.) Python também fornece operadores de incremento associado como += e *; estes criam novos objetos e os reatam por debaixo do processo se seus operadores do lado esquerdo forem imutáveis. Os operadores aritméticos são sobrecarregados pelos tipos `str` e `lista`, como pudemos notar anteriormente.

O terminal I/O pode ser executado usando `output( )` e `print( );` e usar um redirecionamento de arquivos no terminal torna possível usarmos estas mesmas funções embutidas para ler e escrever arquivos.

Em adição à rica funcionalidade dos embutidos do Python, sua extensa biblioteca padrão é também disponível com módulos acessíveis; desde que eles tenham sido importados, através da declaração `import`. Um módulo comumente importado é o `sys`, que retém a lista `sys.argv` da linha de comando dos argumentos. E, quanto Python não possuir a função que precisamos, podemos facilmente criar uma que poderá fazer o que é necessário utilizando a declaração `def`.

Fazendo uso das funcionalidades descritas neste capitulo será possível escrever programas Python pequenos, porém muito úteis. No capítulo seguinte, nós iremos aprender mais sobre os tipos de dados do Python, indo mais a fundo nos `ints` e `strs`, e também aprenderemos mais sobre tuplas e listas e alguma coisa sobre outros tipos de dados colecionáveis do Python. Depois, no Capítulo 4, nós iremos abordar as estruturas de controle do Python em muito mais detalhes e iremos aprender como criar nossas próprias funções, a fim de que possamos

empacotar funcionalidades para evitar a duplicação dos códigos e promover, também, o reuso de códigos.

Exercícios

O objetivo dos exercícios propostos aqui e por todas as partes do livro é encorajar você a fazer uma experiência real com o Python e lhe fornecer a experiência suficiente para absorver o material exposto em cada um dos capítulos. Os exemplos e exercícios abrangem tanto a parte de processamento numérico quanto de texto; eles apelam para uma audiência tão ampla quanto possível, sem contar que são bastante pequenos; dessa forma, damos ênfase nos atos pensar e aprender, ao invés de apenas copiar códigos. Todos os exercícios possuem um resultado, fornecidos com os exemplos do livro.

1. Uma variação muito bacana do programa `bigdigits.py` é que, ao invés de imprimir `*`s, o dígito relevante é que é impresso no lugar. Por exemplo:

Exemplos do livro ☜ 3

```
bigdigits_ans.py 719428306
77777   1    9999     4     222    888    333     000     666
    7  11   9   9    44    2   2  8   8  3   3   0   0   6
   7    1   9   9   4 4    2  2   8   8      3  0     0  6
  7     1    9999  4  4      2     888     33   0     0  6666
 7      1       9  444444   2     8   8      3  0     0  6   6
7       1       9     4    2      8   8  3   3   0   0   6   6
7      111      9     4    22222   888    333     000     666
```

   Dois caminhos podem ser tomados. O mais fácil é simplesmente mudar o `*`s nas listas. Mas isto não é muito versátil e não é a solução mais adequada para se escolher. Ao invés disso, mude o processamento do código, de maneira que, ao invés de adicionar cada string de fileiras dos dígitos na linha em que você está, você adiciona caractere por caractere e, mesmo que um `*` seja encontrado, você usa o dígito relevante.

   Isto pode ser feito copiando `bigdigits.py` e modificando cinco linhas. Isto vem a ser algo difícil, mas é um pouco sutil.

Uma solução é fornecida em `bigdigits_ans.py`.

2. IDLE pode ser usado como uma calculadora muito poderosa e flexível, mas, algumas vezes, é mais conveniente que se tenha uma calculadora para este uso específico. Crie um programa que forneça prompts ao usuário a fim de que ele possa entrar com um número num loop `while`; gradualmente, vá construindo uma lista com os números dados. Quando o usuário tiver terminado (simplesmente pressionando o Enter), imprima os números dados, o cálculo dos números, a soma dos números, o maior e o menor número dado e o peso dos números (sum/count). Aqui está uma execução simples:

```
average1_ans.py
enter a number or Enter to finish: 5
enter a number or Enter to finish: 4
enter a number or Enter to finish: 1
enter a number or Enter to finish: 8
enter a number or Enter to finish: 5
enter a number or Enter to finish: 2
enter a number or Enter to finish:
numbers: [5, 4, 1, 8, 5, 2]
count = 6 sum = 25 lowest = 1 highest = 8 mean = 4.16666666667
```

Isto irá ocupar cerca de quatro linhas para inicializar as variáveis necessárias (uma lista vazia é simplesmente [ ]) e menos de 15 linhas para o loop `while`, incluindo a manipulação de erros básica. É possível imprimir no final somente com algumas linhas, então todo o programa, incluindo as linhas em branco com o objetivo de uma clareza maior, pode ser feito com cerca de 25 linhas.

3. Em algumas situações, nós precisamos gerar textos de teste – por exemplo, para popular o design de um site da web antes que o conteúdo real esteja disponível, ou para fornecer testes de conteúdo quando está desenvolvendo um escritor de relatórios. Para este fim, escreva um programa que gere poesias, diga-se de passagem, horrorosas (do tipo que deixaria você envergonhado).

   Crie algumas listas de palavras, por exemplo, artigos (“o/a”, “um/uma”, etc.), substantivos ("ou", "a", "um", "uma", etc), verbos (“cantar”, “correr”, “pular”) e advérbios (“vagarosamente”, “silenciosamente”, “bem”, “mal”). Depois, faça um loop cinco vezes e, em cada iteração, utilize a função `randow.choice( )` para apanhar, dentre eles, um artigo, um substantivo, um verbo e um advérbio. Utilize `random.randint( )` para escolher entre duas estruturas de sentenças, ou apenas um artigo, substantivo ou verbo, e imprima as sentenças. Aqui está um exemplo executado:

   ```
   awfulpoetry1_ans.py
   her man heard politely
   his boy sang
   another woman hoped
   her girl sang slowly
   the cat heard loudly
   ```

Randow.randint( ) e randow.choice ☜ 36

   Você vai precisar importar o módulo `random` (aleatório). As listas podem ser executadas por volta de 4 a 10 linhas, dependendo de quantas palavras você colocou nelas e o loop, por si mesmo, já exige menos do que dez linha; portanto, com algumas linhas em branco, todo o programa pode ser feito em algo por volta de 20 linhas de código. Uma solução é fornecida em `awfulpoetry_ans.py`.

4. Para fazer programas de “poesias horríveis” mais versáteis, adicione alguns códigos a ele; dessa forma, se o usuário entrar com um número na linha de comando (entre 1 e 10, inclusive), o programa irá imprimir as várias linhas. Se nenhum argumento para a linha de comando for dado, o padrão é imprimir cinco linhas, assim como antes. Você vai precisar mudar o loop principal (e.g., para um loop `while`). Tenha em mente que os operadores de comparação do

Python podem ser conectados, portanto, não há necessidade de usar lógica e checar se o argumento está ao alcance. A funcionalidade adicional pode ser executada adicionando cerca de dez linhas de código. Uma solução está em e `awfulpoetry2_ans.py`.

5. Seria bacana se conseguíssemos calcular a média (valor central) e também peso os significados igual do programa de média do Exercício 2. No Python, uma lista pode facilmente ser classificada usando o método `list.sorted( )`, mas não iremos abordar este método por enquanto, portanto, ele não será utilizado aqui. Estenda o programa de média com um bloco de códigos que classifique a lista dos números – se este desempenho lhe for motivo de preocupação, resolva o problema utilizando a abordagem mais fácil que você possa pensar. Uma vez que a lista esteja classificada, a média será o valor central se a lista possuir um número ímpar de itens. Calcule a média e imprima junto com as outras informações.

   Este procedimento pode se tornar uma armadilha, principalmente para programadores inexperientes. Se você possuir alguma experiência Python, ainda assim vai lhe parecer algo desafiante, pelo menos se você limitou-se a usar apenas o que foi ensinado até agora. A classificação pode ser feita em cerca de doze linhas e o cálculo da média (em que você não poderá usar o operador de módulo, já que este ainda não foi abordado), em quatro linhas. A solução está em `average2_ans.py`.