Python Tutorial

Release 2.7.18

Guido van Rossum
and the Python development team

maio 07, 2020

Python Software Foundation
Email: docs@python.org

Sumário

Python é uma linguagem fácil de aprender e poderosa. Ela tem estruturas de dados de alto nível eficientes e uma abordagem simples mas efetiva de programação orientada a objetos. A elegância de sintaxe e a tipagem dinâmica do Python aliadas com sua natureza interpretativa, o fazem a linguagem ideal para programas e desenvolvimento de aplicações rápidas em diversas áreas e na maioria das plataformas.

O interpretador Python e a extensiva biblioteca padrão estão disponíveis gratuitamente em código ou na forma binária para toda as maiores plataformas no endereço eletrônico do Python, https://www.python.org/, e pode ser livremente distribuído. O mesmo endereço contém distribuições de diversos módulos, programas e ferramentas gratuitos produzidos por terceiros e documentação adicional.

O interpretador Python pode ser facilmente estendido com novas funções e tipos de dados implementados em C ou C++ (ou outras linguagens chamadas a partir de C). Python também é adequada como uma linguagem de extensão para aplicações personalizáveis.

Este tutorial introduz informalmente o leitor aos conceitos básicos e aos recursos da linguagem e do sistema Python. É mais fácil se você possuir um interpretador Python para uma experiência prática, mas os exemplos são autossuficientes e, portanto, o tutorial pode apenas ser lido off-line também.

Para uma descrição detalhada dos módulos e objetos padrões, veja library-index. Em reference-index você encontra uma definição mais formal da linguagem. Para escrever extensões em C ou C++ leia extending-index e c-api-index. Existe também uma série de livros que cobrem Python em profundidade.

Este tutorial não espera ser abrangente e cobrir todos os recursos ou mesmo os recursos mais usados. Ele busca introduzir diversos dos recursos mais notáveis do Python e lhe dará uma boa ideia do sabor e estilo da linguagem. Depois de lê-lo, você terá condições de ler e escrever programas e módulos Python e estará pronto para aprender mais sobre os diversos módulos descritos em library-index.

O *Glossário* também vale a pena ser estudado.

CAPÍTULO 1

Abrindo seu apetite

Se você trabalha muito com computadores, acabará encontrando alguma tarefa que gostaria de automatizar. Por exemplo, você pode querer fazer busca-e-troca em um grande número de arquivos de texto, ou renomear e reorganizar um monte de arquivos de fotos de uma maneira complicada. Talvez você gostaria de escrever um pequeno banco de dados personalizado, ou um aplicativo GUI especializado, ou um jogo simples.

Se você é um desenvolvedor de software profissional, pode ter que trabalhar com várias bibliotecas C/C++/Java, mas o tradicional ciclo escrever/compilar/testar/recompilar é muito lento. Talvez você esteja escrevendo um conjunto de testes para uma biblioteca e está achando tedioso codificar os testes. Ou talvez você tenha escrito um programa que poderia utilizar uma linguagem de extensão, e você não quer conceber e implementar toda uma nova linguagem para sua aplicação.

Python é a linguagem para você.

Você poderia escrever um script para o shell do Unix ou arquivos em lote do Windows para algumas dessas tarefas, mas scripts shell são bons para mover arquivos e alterar textos, mas não adequados para aplicações GUI ou jogos. Você poderia escrever um programa em C/C++/Java, mas pode tomar tempo de desenvolvimento para chegar até um primeiro rascunho. Python é mais simples, está disponível em Windows, Mac OS X, e sistemas operacionais Unix, e vai ajudá-lo a fazer o trabalho mais rapidamente.

Python é fácil de usar, sem deixar de ser uma linguagem de programação de verdade, oferecendo muito mais estruturação e suporte para programas extensos do que shell scripts ou arquivos de lote oferecem. Por outro lado, Python também oferece melhor verificação de erros do que C, e por ser uma linguagem de *muito alto nível*, ela possui tipos nativos de alto nível, tais como dicionários e vetores (arrays) flexíveis. Devido ao suporte nativo a uma variedade de tipos de dados, Python é aplicável a um domínio de problemas muito mais vasto do que Awk ou até mesmo Perl, ainda assim muitas tarefas são pelo menos tão fáceis em Python quanto nessas linguagens.

Python permite que você organize seu programa em módulos que podem ser reutilizados em outros programas escritos em Python. A linguagem provê uma vasta coleção de módulos que podem ser utilizados como base para sua aplicação — ou como exemplos para estudo e aprofundamento. Alguns desses módulos implementam manipulação de arquivos, chamadas do sistema, sockets, e até mesmo acesso a bibliotecas de construção de interfaces gráficas, como Tk.

Python é uma linguagem interpretada, por isso você pode economizar um tempo considerável durante o desenvolvimento, uma vez que não há necessidade de compilação e vinculação (*linking*). O interpretador pode ser usado interativamente, o que torna fácil experimentar diversas características da linguagem, escrever programas "descartáveis", ou testar funções em um desenvolvimento debaixo para cima (*bottom-up*). É também uma útil calculadora de mesa.

Python permite a escrita de programas compactos e legíveis. Programas escritos em Python são tipicamente mais curtos do que seus equivalentes em C, C++ ou Java, por diversas razões:

- os tipos de alto nível permitem que você expresse operações complexas em um único comando;
- a definição de bloco é feita por indentação ao invés de marcadores de início e fim de bloco;
- não há necessidade de declaração de variáveis ou parâmetros formais;

Python é *extensível*: se você sabe como programar em C, é fácil adicionar funções ou módulos diretamente no interpretador, seja para desempenhar operações críticas em máxima velocidade, ou para vincular programas Python a bibliotecas que só estejam disponíveis em formato binário (como uma biblioteca gráfica de terceiros). Uma vez que você tenha sido fisgado, você pode vincular o interpretador Python a uma aplicação escrita em C e utilizá-la como linguagem de comandos ou extensão para esta aplicação.

A propósito, a linguagem foi batizada a partir do famoso programa da BBC "Monty Python's Flying Circus" e não tem nada a ver com répteis. Fazer referências a citações do programa na documentação não é só permitido, como também é encorajado!

Agora que você está entusiasmado com Python, vai querer conhecê-la com mais detalhes. Partindo do princípio que a melhor maneira de aprender uma linguagem é usando-a, você está agora convidado a fazê-lo com este tutorial.

No próximo capítulo, a mecânica de utilização do interpretador é explicada. Essa informação, ainda que mundana, é essencial para a experimentação dos exemplos apresentados mais tarde.

O resto do tutorial introduz diversos aspectos do sistema e linguagem Python por intermédio de exemplos. Serão abordadas expressões simples, comandos, tipos, funções e módulos. Finalmente, serão explicados alguns conceitos avançados como exceções e classes definidas pelo usuário.

CAPÍTULO 2

Utilizando o interpretador Python

2.1 Chamando o interpretador

The Python interpreter is usually installed as `/usr/local/bin/python` on those machines where it is available; putting `/usr/local/bin` in your Unix shell's search path makes it possible to start it by typing the command

```
python
```

to the shell. Since the choice of the directory where the interpreter lives is an installation option, other places are possible; check with your local Python guru or system administrator. (E.g., `/usr/local/python` is a popular alternative location.)

On Windows machines, the Python installation is usually placed in `C:\Python27`, though you can change this when you're running the installer. To add this directory to your path, you can type the following command into the command prompt in a DOS box:

```
set path=%path%;C:\python27
```

Digitando um caractere de fim-de-arquivo (`Control-D` no Unix, `Control-Z` no Windows) diretamente no prompt força o interpretador a sair com status de saída zero. Se isso não funcionar, você pode sair do interpretador digitando o seguinte comando: `quit()`.

The interpreter's line-editing features usually aren't very sophisticated. On Unix, whoever installed the interpreter may have enabled support for the GNU readline library, which adds more elaborate interactive editing and history features. Perhaps the quickest check to see whether command line editing is supported is typing `Control-P` to the first Python prompt you get. If it beeps, you have command line editing; see Appendix *Edição de entrada interativa e substituição de histórico* for an introduction to the keys. If nothing appears to happen, or if `^P` is echoed, command line editing isn't available; you'll only be able to use backspace to remove characters from the current line.

O interpretador trabalha de forma semelhante a uma shell de Unix: quando chamado com a saída padrão conectada a um console de terminal, ele lê e executa comandos interativamente; quando chamado com um nome de arquivo como argumento, ou com redirecionamento da entrada padrão para ler um arquivo, o interpretador lê e executa o *script* contido no arquivo.

Uma segunda forma de rodar o interpretador é `python -c command [arg] ...`, que executa um ou mais comandos especificados na posição *comando*, analogamente à opção de shell `-c`. Considerando que comandos Python frequentemente têm espaços em branco (ou outros caracteres que são especiais para a shell) é aconselhável que o *comando* esteja dentro de aspas duplas.

Alguns módulos Python são também úteis como scripts. Estes podem ser chamados usando `python -m modulo [arg] ...`, que executa o arquivo fonte do *módulo* como se você tivesse digitado seu caminho completo na linha de comando.

Quando um arquivo de script é utilizado, às vezes é útil executá-lo e logo em seguida entrar em modo interativo. Isto pode ser feito acrescentando o argumento `-i` antes do nome do script.

All command-line options are described in using-on-general.

2.1.1 Passagem de argumentos

Quando são de conhecimento do interpretador, o nome do script e demais argumentos da linha de comando da shell são acessíveis ao próprio script através da variável `argv` do módulo `sys`. Pode-se acessar essa lista executando `import sys`. Essa lista tem sempre ao menos um elemento; quando nenhum script ou argumento for passado para o interpretador, `sys.argv[0]` será uma string vazia. Quando o nome do script for `'-'` (significando entrada padrão), o conteúdo de `sys.argv[0]` será `'-'`. Quando for utilizado `-c` *comando*, `sys.argv[0]` conterá `'-c'`. Quando for utilizado `-m` *módulo*, `sys.argv[0]` conterá o caminho completo do módulo localizado. Opções especificadas após `-c` *comando* ou `-m` *módulo* não serão consumidas pelo interpretador mas deixadas em `sys.argv` para serem tratadas pelo comando ou módulo.

2.1.2 Modo interativo

Quando os comandos são lidos a partir do console, diz-se que o interpretador está em modo interativo. Nesse modo ele solicita um próximo comando através do *prompt primário*, tipicamente três sinais de maior (`>>>`); para linhas de continuação do comando atual, o *prompt secundário* padrão é formado por três pontos (`...`). O interpretador exibe uma mensagem de boas vindas, informando seu número de versão e um aviso de copyright antes de exibir o primeiro prompt:

```
python
Python 2.7 (#1, Feb 28 2010, 00:02:06)
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Linhas de continuação são necessárias em construções multi-linha. Como exemplo, dê uma olhada nesse comando `if`:

```
>>> the_world_is_flat = 1
>>> if the_world_is_flat:
...     print "Be careful not to fall off!"
...
Be careful not to fall off!
```

Para mais informações sobre o modo interativo, veja *Modo interativo*.

2.2 O interpretador e seu ambiente

2.2.1 Edição de código-fonte

By default, Python source files are treated as encoded in ASCII. To declare an encoding other than the default one, a special comment line should be added as the *first* line of the file. The syntax is as follows:

```
# -*- coding: encoding -*-
```

onde *encoding* é uma das `codecs` válidas com suporte do Python.

Por exemplo, para declarar que a codificação Windows-1252 deve ser usada, a primeira linha do seu arquivo fonte deve ser:

```
# -*- coding: cp1252 -*-
```

Uma exceção para a regra da *primeira linha* é quando o código-fonte inicia com uma *linha com UNIX "shebang"*. Nesse caso, a declaração de codificação deve ser adicionada como a segunda linha do arquivo. Por exemplo:

```
#!/usr/bin/env python
# -*- coding: cp1252 -*-
```

CAPÍTULO 3

Uma introdução informal ao Python

Nos exemplos seguintes, pode-se distinguir entrada e saída pela presença ou ausência dos prompts (»> e ...): para repetir o exemplo, você deve digitar tudo após o prompt, quando o mesmo aparece; linhas que não começarem com um prompt são na verdade as saídas geradas pelo interpretador. Observe que quando aparece uma linha contendo apenas o prompt secundário você deve digitar uma linha em branco; é assim que se encerra um comando de múltiplas linhas.

Muitos dos exemplos neste manual, até mesmo aqueles digitados interativamente, incluem comentários. Comentários em Python são iniciados pelo caractere #, e se estendem até o final da linha física. Um comentário pode aparecer no início da linha, depois de um espaço em branco ou código, mas nunca dentro de uma string literal. O caractere # em uma string literal não passa de um caractere #. Uma vez que os comentários são usados apenas para explicar o código e não são interpretados pelo Python, eles podem ser omitidos ao digitar os exemplos.

Alguns exemplos:

```
# this is the first comment
spam = 1  # and this is the second comment
          # ... and now a third!
text = "# This is not a comment because it's inside quotes."
```

3.1 Usando Python como uma calculadora

Vamos experimentar alguns comandos simples em Python. Inicie o interpretador e aguarde o prompt primário, `>>>`. (Não deve demorar muito.)

3.1.1 Números

O interpretador funciona como uma calculadora bem simples: você pode digitar uma expressão e o resultado será apresentado. A sintaxe de expressões é a usual: operadores `+`, `-`, `*` e `/` funcionam da mesma forma que em outras linguagens tradicionais (por exemplo, Pascal ou C); parênteses (`()`) podem ser usados para agrupar expressões. Por exemplo:

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5.0*6) / 4
5.0
>>> 8 / 5.0
1.6
```

Os números inteiros (ex. `2`, `4`, `20`) são do tipo `int`, aqueles com parte fracionária (ex. `5.0`, `1.6`) são do tipo `float`. Veremos mais sobre tipos numéricos posteriormente neste tutorial.

The return type of a division (`/`) operation depends on its operands. If both operands are of type `int`, *floor division* is performed and an `int` is returned. If either operand is a `float`, classic division is performed and a `float` is returned. The `//` operator is also provided for doing floor division no matter what the operands are. The remainder can be calculated with the `%` operator:

```
>>> 17 / 3  # int / int -> int
5
>>> 17 / 3.0  # int / float -> float
5.666666666666667
>>> 17 // 3.0  # explicit floor division discards the fractional part
5.0
>>> 17 % 3  # the % operator returns the remainder of the division
2
>>> 5 * 3 + 2  # result * divisor + remainder
17
```

Com Python, podemos usar o operador `**` para calcular potências[1]:

```
>>> 5 ** 2  # 5 squared
25
>>> 2 ** 7  # 2 to the power of 7
128
```

O sinal de igual (`'='`) é usado para atribuir um valor a uma variável. Depois de uma atribuição, nenhum resultado é exibido antes do próximo prompt:

```
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

[1] Uma vez que `**` tem precedência mais alta que `-`, `-3**2` será interpretado como `-(3**2)` e assim resultará em `-9`. Para evitar isso e obter `9`, você pode usar `(-3)**2`.

Se uma variável não é "definida" (não tem um valor atribuído), tentar utilizá-la gerará um erro:

```
>>> n  # try to access an undefined variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

Há suporte completo para ponto flutuante (*float*); operadores com operandos de diferentes tipos convertem o inteiro para ponto flutuante:

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

No modo interativo, o valor da última expressão exibida é atribuída a variável _. Assim, ao utilizar Python como uma calculadora, fica mais fácil prosseguir com os cálculos, por exemplo:

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
```

Essa variável especial deve ser tratada como *somente para leitura* pelo usuário. Nunca lhe atribua explicitamente um valor — do contrário, estaria criando uma outra variável (homônima) independente, que mascararia a variável especial com seu comportamento mágico.

Além de `int` e `float`, o Python suporta outros tipos de números, tais como `Decimal` e `Fraction`. O Python também possui suporte nativo a números complexos, e usa os sufixos `j` ou `J` para indicar a parte imaginária (por exemplo, `3+5j`).

3.1.2 Strings

Além de números, Python também pode manipular strings (sequências de caracteres), que podem ser expressas de diversas formas. Elas podem ser delimitadas por aspas simples (`'...'`) ou duplas (`"..."`) e teremos o mesmo resultado[2]. `\` pode ser usada para escapar aspas:

```
>>> 'spam eggs'  # single quotes
'spam eggs'
>>> 'doesn\'t'  # use \' to escape the single quote...
"doesn't"
>>> "doesn't"  # ...or use double quotes instead
"doesn't"
>>> '"Yes," they said.'
'"Yes," they said.'
>>> "\"Yes,\" they said."
'"Yes," they said.'
>>> '"Isn\'t," they said.'
'"Isn\'t," they said.'
```

[2] Ao contrário de outras linguagens, caracteres especiais como `\n` têm o mesmo significado com as aspas simples (`'...'`) e duplas (`"..."`) . A única diferença entre as duas é que, dentro de aspas simples, você não precisa escapar o `"` (mas você deve escapar a `\'`) e vice-versa.

In the interactive interpreter, the output string is enclosed in quotes and special characters are escaped with backslashes. While this might sometimes look different from the input (the enclosing quotes could change), the two strings are equivalent. The string is enclosed in double quotes if the string contains a single quote and no double quotes, otherwise it is enclosed in single quotes. The `print` statement produces a more readable output, by omitting the enclosing quotes and by printing escaped and special characters:

```
>>> '"Isn\'t," they said.'
'"Isn\'t," they said.'
>>> print '"Isn\'t," they said.'
"Isn't," they said.
>>> s = 'First line.\nSecond line.'  # \n means newline
>>> s  # without print, \n is included in the output
'First line.\nSecond line.'
>>> print s  # with print, \n produces a new line
First line.
Second line.
```

Se não quiseres que os caracteres sejam precedidos por `\` para serem interpretados como caracteres especiais, poderás usar *strings raw* (N.d.T: "crua" ou sem processamento de caracteres de escape) adicionando um `r` antes da primeira aspa:

```
>>> print 'C:\some\name'  # here \n means newline!
C:\some
ame
>>> print r'C:\some\name'  # note the r before the quote
C:\some\name
```

As strings literais podem abranger várias linhas. Uma maneira é usar as aspas triplas: `"""..."""` ou `'''...'''`. O fim das linhas é incluído automaticamente na string, mas é possível evitar isso adicionando uma `\` no final. O seguinte exemplo:

```
print """\
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
"""
```

produz a seguinte saída (observe que a linha inicial não está incluída):

```
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
```

Strings podem ser concatenadas (coladas) com o operador `+`, e repetidas com `*`:

```
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'ununununium'
```

Duas ou mais *strings literais* (ou seja, entre aspas) ao lado da outra são automaticamente concatenados.

```
>>> 'Py' 'thon'
'Python'
```

Esse recurso é particularmente útil quando você quer quebrar strings longas:

```
>>> text = ('Put several strings within parentheses '
...         'to have them joined together.')
```

(continua na próxima página)

(continuação da página anterior)

```
>>> text
'Put several strings within parentheses to have them joined together.'
```

Isso só funciona com duas strings literais, não com variáveis ou expressões:

```
>>> prefix = 'Py'
>>> prefix 'thon'  # can't concatenate a variable and a string literal
  ...
SyntaxError: invalid syntax
>>> ('un' * 3) 'ium'
  ...
SyntaxError: invalid syntax
```

Se você quiser concatenar variáveis ou uma variável e uma literal, use +:

```
>>> prefix + 'thon'
'Python'
```

As strings podem ser *indexadas* (subscritas), com o primeiro caractere como índice 0. Não existe um tipo específico para caracteres; um caractere é simplesmente uma string cujo tamanho é 1:

```
>>> word = 'Python'
>>> word[0]  # character in position 0
'P'
>>> word[5]  # character in position 5
'n'
```

Índices também podem ser números negativos para iniciar a contagem pela direita:

```
>>> word[-1]  # last character
'n'
>>> word[-2]  # second-last character
'o'
>>> word[-6]
'P'
```

Note que dado que -0 é o mesmo que 0, índices negativos começam em -1.

In addition to indexing, *slicing* is also supported. While indexing is used to obtain individual characters, *slicing* allows you to obtain a substring:

```
>>> word[0:2]  # characters from position 0 (included) to 2 (excluded)
'Py'
>>> word[2:5]  # characters from position 2 (included) to 5 (excluded)
'tho'
```

Observe como o início sempre está incluído, e o fim sempre é excluído. Isso garante que `s[:i] + s[i:]` seja sempre igual a `s`:

```
>>> word[:2] + word[2:]
'Python'
>>> word[:4] + word[4:]
'Python'
```

Os índices do fatiamento possuem padrões úteis; um primeiro índice omitido padrão é zero, um segundo índice omitido é por padrão o tamanho da string sendo fatiada:

```
>>> word[:2]    # character from the beginning to position 2 (excluded)
'Py'
>>> word[4:]    # characters from position 4 (included) to the end
'on'
>>> word[-2:]  # characters from the second-last (included) to the end
'on'
```

Uma maneira de lembrar como fatias funcionam é pensar que os índices indicam posições *entre* caracteres, onde a borda esquerda do primeiro caractere é 0. Assim, a borda direita do último caractere de uma string de comprimento *n* tem índice *n*, por exemplo:

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

A primeira fileira de números indica a posição dos índices 0…6 na string; a segunda fileira indica a posição dos respectivos índices negativos. Uma fatia de *i* a *j* consiste em todos os caracteres entre as bordas *i* e *j*, respectivamente.

Para índices positivos, o comprimento da fatia é a diferença entre os índices, se ambos estão dentro dos limites da string. Por exemplo, o comprimento de `word[1:3]` é 2.

A tentativa de usar um índice que seja muito grande resultará em um erro:

```
>>> word[42]  # the word only has 6 characters
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

No entanto, os índices de fatiamento fora do alcance são tratados graciosamente (N.d.T: o termo original “gracefully” indica robustez no tratamento de erros) quando usados para fatiamento. Um índice maior que o comprimento é trocado pelo comprimento, um limite superior menor que o limite inferior produz uma string vazia:

```
>>> word[4:42]
'on'
>>> word[42:]
''
```

As strings do Python não podem ser alteradas — uma string é *imutável*. Portanto, atribuir a uma posição indexada na sequência resulta em um erro:

```
>>> word[0] = 'J'
  ...
TypeError: 'str' object does not support item assignment
>>> word[2:] = 'py'
  ...
TypeError: 'str' object does not support item assignment
```

Se você precisar de uma string diferente, deverá criar uma nova:

```
>>> 'J' + word[1:]
'Jython'
>>> word[:2] + 'py'
'Pypy'
```

A função embutida `len()` devolve o comprimento de uma string:

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

Ver também:

typesseq Strings, and the Unicode strings described in the next section, are examples of *sequence types*, and support the common operations supported by such types.

string-methods Both strings and Unicode strings support a large number of methods for basic transformations and searching.

formatstrings Informações sobre formatação de string com o método `str.format()`.

string-formatting The old formatting operations invoked when strings and Unicode strings are the left operand of the `%` operator are described in more detail here.

3.1.3 Unicode Strings

Starting with Python 2.0 a new data type for storing text data is available to the programmer: the Unicode object. It can be used to store and manipulate Unicode data (see http://www.unicode.org/) and integrates well with the existing string objects, providing auto-conversions where necessary.

Unicode has the advantage of providing one ordinal for every character in every script used in modern and ancient texts. Previously, there were only 256 possible ordinals for script characters. Texts were typically bound to a code page which mapped the ordinals to script characters. This lead to very much confusion especially with respect to internationalization (usually written as `i18n` — `'i'` + 18 characters + `'n'`) of software. Unicode solves these problems by defining one code page for all scripts.

Creating Unicode strings in Python is just as simple as creating normal strings:

```
>>> u'Hello World !'
u'Hello World !'
```

The small `'u'` in front of the quote indicates that a Unicode string is supposed to be created. If you want to include special characters in the string, you can do so by using the Python *Unicode-Escape* encoding. The following example shows how:

```
>>> u'Hello\u0020World !'
u'Hello World !'
```

The escape sequence `\u0020` indicates to insert the Unicode character with the ordinal value 0x0020 (the space character) at the given position.

Other characters are interpreted by using their respective ordinal values directly as Unicode ordinals. If you have literal strings in the standard Latin-1 encoding that is used in many Western countries, you will find it convenient that the lower 256 characters of Unicode are the same as the 256 characters of Latin-1.

For experts, there is also a raw mode just like the one for normal strings. You have to prefix the opening quote with 'ur' to have Python use the *Raw-Unicode-Escape* encoding. It will only apply the above `\uXXXX` conversion if there is an uneven number of backslashes in front of the small 'u'.

```
>>> ur'Hello\u0020World !'
u'Hello World !'
>>> ur'Hello\\u0020World !'
u'Hello\\\\u0020World !'
```

The raw mode is most useful when you have to enter lots of backslashes, as can be necessary in regular expressions.

Apart from these standard encodings, Python provides a whole set of other ways of creating Unicode strings on the basis of a known encoding.

The built-in function `unicode()` provides access to all registered Unicode codecs (COders and DECoders). Some of the more well known encodings which these codecs can convert are *Latin-1*, *ASCII*, *UTF-8*, and *UTF-16*. The latter two are variable-length encodings that store each Unicode character in one or more bytes. The default encoding is normally set to ASCII, which passes through characters in the range 0 to 127 and rejects any other characters with an error. When a Unicode string is printed, written to a file, or converted with `str()`, conversion takes place using this default encoding.

```
>>> u"abc"
u'abc'
>>> str(u"abc")
'abc'
>>> u"äöü"
u'\xe4\xf6\xfc'
>>> str(u"äöü")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-2: ordinal␣
↪not in range(128)
```

To convert a Unicode string into an 8-bit string using a specific encoding, Unicode objects provide an `encode()` method that takes one argument, the name of the encoding. Lowercase names for encodings are preferred.

```
>>> u"äöü".encode('utf-8')
'\xc3\xa4\xc3\xb6\xc3\xbc'
```

If you have data in a specific encoding and want to produce a corresponding Unicode string from it, you can use the `unicode()` function with the encoding name as the second argument.

```
>>> unicode('\xc3\xa4\xc3\xb6\xc3\xbc', 'utf-8')
u'\xe4\xf6\xfc'
```

3.1.4 Listas

Python inclui diversas estruturas de dados *compostas*, usadas para agrupar outros valores. A mais versátil é *list* (lista), que pode ser escrita como uma lista de valores (itens) separados por vírgula, entre colchetes. Os valores contidos na lista não precisam ser todos do mesmo tipo.

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

Like strings (and all other built-in *sequence* type), lists can be indexed and sliced:

```
>>> squares[0]  # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:]  # slicing returns a new list
[9, 16, 25]
```

All slice operations return a new list containing the requested elements. This means that the following slice returns a new (shallow) copy of the list:

```
>>> squares[:]
[1, 4, 9, 16, 25]
```

Lists also supports operations like concatenation:

```
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Diferentemente de strings, que são *imutáveis*, listas são *mutáveis*, ou seja, é possível alterar elementos individuais de uma lista:

```
>>> cubes = [1, 8, 27, 65, 125]  # something's wrong here
>>> 4 ** 3  # the cube of 4 is 64, not 65!
64
>>> cubes[3] = 64  # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
```

Você também pode adicionar novos itens no final da lista, usando o *método* `append()` (estudaremos mais a respeito dos métodos posteriormente):

```
>>> cubes.append(216)  # add the cube of 6
>>> cubes.append(7 ** 3)  # and the cube of 7
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

Atribuição a fatias também é possível, e isso pode até alterar o tamanho da lista ou remover todos os itens dela:

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # replace some values
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # now remove them
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # clear the list by replacing all the elements with an empty list
>>> letters[:] = []
>>> letters
[]
```

A função embutida `len()` também se aplica a listas:

```
>>> letters = ['a', 'b', 'c', 'd']
>>> len(letters)
4
```

É possível aninhar listas (criar listas contendo outras listas), por exemplo:

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
```

(continua na próxima página)

(continuação da página anterior)

```
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

3.2 Primeiros passos para a programação

Of course, we can use Python for more complicated tasks than adding two and two together. For instance, we can write an initial sub-sequence of the *Fibonacci* series as follows:

```
>>> # Fibonacci series:
... # the sum of two elements defines the next
... a, b = 0, 1
>>> while b < 10:
...     print b
...     a, b = b, a+b
...
1
1
2
3
5
8
```

Este exemplo introduz diversas características ainda não mencionadas.

- A primeira linha contém uma atribuição múltipla: as variáveis `a` e `b` recebem simultaneamente os novos valores 0 e 1. Na última linha há outro exemplo de atribuição múltipla demonstrando que expressões do lado direito são sempre avaliadas primeiro, antes da atribuição. As expressões do lado direito são avaliadas da esquerda para a direita.
- The `while` loop executes as long as the condition (here: `b < 10`) remains true. In Python, like in C, any non-zero integer value is true; zero is false. The condition may also be a string or list value, in fact any sequence; anything with a non-zero length is true, empty sequences are false. The test used in the example is a simple comparison. The standard comparison operators are written the same as in C: < (less than), > (greater than), == (equal to), <= (less than or equal to), >= (greater than or equal to) and != (not equal to).
- O *corpo* do laço é *indentado*: indentação em Python é a maneira de agrupar comandos em blocos. No console interativo padrão você terá que digitar tab ou espaços para indentar cada linha. Na prática você vai preparar scripts Python mais complicados em um editor de texto; a maioria dos editores de texto tem facilidades de indentação automática. Quando um comando composto é digitado interativamente, deve ser finalizado por uma linha em branco (já que o interpretador não tem como adivinhar qual é a última linha do comando). Observe que toda linha de um mesmo bloco de comandos deve ter a mesma indentação.
- The `print` statement writes the value of the expression(s) it is given. It differs from just writing the expression you want to write (as we did earlier in the calculator examples) in the way it handles multiple expressions and strings. Strings are printed without quotes, and a space is inserted between items, so you can format things nicely, like this:

  ```
  >>> i = 256*256
  >>> print 'The value of i is', i
  The value of i is 65536
  ```

  A trailing comma avoids the newline after the output:

```
>>> a, b = 0, 1
>>> while b < 1000:
...     print b,
...     a, b = b, a+b
...
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

Note that the interpreter inserts a newline before it prints the next prompt if the last line was not completed.

CAPÍTULO 4

Mais ferramentas de controle de fluxo

Além do comando `while` recém apresentado, Python tem as estruturas usuais de controle de fluxo conhecidas em outras linguagens, com algumas particulares.

4.1 `if` Statements

Provavelmente o mais conhecido comando de controle de fluxo é o `if`. Por exemplo:

```
>>> x = int(raw_input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print 'Negative changed to zero'
... elif x == 0:
...     print 'Zero'
... elif x == 1:
...     print 'Single'
... else:
...     print 'More'
...
More
```

There can be zero or more `elif` parts, and the `else` part is optional. The keyword 'elif' is short for 'else if', and is useful to avoid excessive indentation. An `if` ... `elif` ... `elif` ... sequence is a substitute for the `switch` or `case` statements found in other languages.

4.2 `for` Statements

The `for` statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python's `for` statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example (no pun intended):

```
>>> # Measure some strings:
... words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print w, len(w)
...
cat 3
window 6
defenestrate 12
```

If you need to modify the sequence you are iterating over while inside the loop (for example to duplicate selected items), it is recommended that you first make a copy. Iterating over a sequence does not implicitly make a copy. The slice notation makes this especially convenient:

```
>>> for w in words[:]:  # Loop over a slice copy of the entire list.
...     if len(w) > 6:
...         words.insert(0, w)
...
>>> words
['defenestrate', 'cat', 'window', 'defenestrate']
```

4.3 A função `range()`

If you do need to iterate over a sequence of numbers, the built-in function `range()` comes in handy. It generates lists containing arithmetic progressions:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The given end point is never part of the generated list; `range(10)` generates a list of 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the 'step'):

```
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

Para iterar sobre os índices de uma sequência, combine `range()` e `len()` da seguinte forma:

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print i, a[i]
...
0 Mary
```

(continua na próxima página)

(continuação da página anterior)

```
1 had
2 a
3 little
4 lamb
```

Na maioria dos casos, porém, é mais conveniente usar a função `enumerate()`, veja *Técnicas de iteração*.

4.4 `break` and `continue` Statements, and `else` Clauses on Loops

O comando `break`, como no C, sai imediatamente do laço de repetição mais interno, seja `for` ou `while`.

Loop statements may have an `else` clause; it is executed when the loop terminates through exhaustion of the list (with `for`) or when the condition becomes false (with `while`), but not when the loop is terminated by a `break` statement. This is exemplified by the following loop, which searches for prime numbers:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print n, 'equals', x, '*', n/x
...             break
...     else:
...         # loop fell through without finding a factor
...         print n, 'is a prime number'
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

(Sim, o código está correto. Olhe atentamente: a cláusula `else` pertence ao laço `for`, e **não** ao comando `if`.)

When used with a loop, the `else` clause has more in common with the `else` clause of a `try` statement than it does that of `if` statements: a `try` statement's `else` clause runs when no exception occurs, and a loop's `else` clause runs when no `break` occurs. For more on the `try` statement and exceptions, see *Tratamento de exceções*.

A instrução `continue`, também emprestada da linguagem C, continua com a próxima iteração do laço:

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print "Found an even number", num
...         continue
...     print "Found a number", num
Found an even number 2
Found a number 3
Found an even number 4
Found a number 5
Found an even number 6
Found a number 7
Found an even number 8
Found a number 9
```

4.5 pass Statements

O comando pass não faz nada. Ela pode ser usada quando a sintaxe exige um comando mas a semântica do programa não requer nenhuma ação. Por exemplo:

```
>>> while True:
...     pass  # Busy-wait for keyboard interrupt (Ctrl+C)
...
```

Isto é usado muitas vezes para se definir classes mínimas:

```
>>> class MyEmptyClass:
...     pass
...
```

Another place pass can be used is as a place-holder for a function or conditional body when you are working on new code, allowing you to keep thinking at a more abstract level. The pass is silently ignored:

```
>>> def initlog(*args):
...     pass   # Remember to implement this!
...
```

4.6 Definindo funções

Podemos criar uma função que escreve a série de Fibonacci até um limite arbitrário:

```
>>> def fib(n):    # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...         print a,
...         a, b = b, a+b
...
>>> # Now call the function we just defined:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

A palavra reservada def inicia a *definição* de uma função. Ela deve ser seguida do nome da função e da lista de parâmetros formais entre parênteses. Os comandos que formam o corpo da função começam na linha seguinte e devem ser indentados.

Opcionalmente, a primeira linha do corpo da função pode ser uma literal string, cujo propósito é documentar a função. Se presente, essa string chama-se *docstring*. (Há mais informação sobre docstrings na seção *Strings de documentação*.) Existem ferramentas que utilizam docstrings para produzir automaticamente documentação online ou para imprimir, ou ainda, permitir que o usuário navegue interativamente pelo código. É uma boa prática incluir sempre docstrings em suas funções, portanto, tente fazer disso um hábito.

The *execution* of a function introduces a new symbol table used for the local variables of the function. More precisely, all variable assignments in a function store the value in the local symbol table; whereas variable references first look in the local symbol table, then in the local symbol tables of enclosing functions, then in the global symbol table, and finally in the table of built-in names. Thus, global variables cannot be directly assigned a value within a function (unless named in a global statement), although they may be referenced.

Os parâmetros reais (argumentos) de uma chamada de função são introduzidos na tabela de símbolos local da função no momento da chamada; portanto, argumentos são passados *por valor* (onde o *valor* é sempre uma *referência* para objeto,

não o valor do objeto).[1] Quando uma função chama outra função, uma nova tabela de símbolos é criada para tal chamada.

Uma definição de função introduz o nome da função na tabela de símbolos atual. O valor associado ao nome da função tem um tipo que é reconhecido pelo interpretador como uma função definida pelo usuário. Esse valor pode ser atribuído a outros nomes que também podem ser usados como funções. Esse mecanismo serve para renomear funções:

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Coming from other languages, you might object that `fib` is not a function but a procedure since it doesn't return a value. In fact, even functions without a `return` statement do return a value, albeit a rather boring one. This value is called `None` (it's a built-in name). Writing the value `None` is normally suppressed by the interpreter if it would be the only value written. You can see it if you really want to using `print`:

```
>>> fib(0)
>>> print fib(0)
None
```

É fácil escrever uma função que devolve uma lista de números da série de Fibonacci, ao invés de exibi-los:

```
>>> def fib2(n):  # return Fibonacci series up to n
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)    # see below
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # call it
>>> f100                # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Este exemplo demonstra novos recursos de Python:

- The `return` statement returns with a value from a function. `return` without an expression argument returns `None`. Falling off the end of a function also returns `None`.
- A instrução `result.append(a)` chama um *método* do objeto lista `result`. Um método é uma função que 'pertence' a um objeto, e é chamada `obj.nomemetodo`, onde `obj` é um objeto qualquer (pode ser uma expressão), e `nomemetodo` é o nome de um método que foi definido pelo tipo do objeto. Tipos diferentes definem métodos diferentes. Métodos de diferentes tipos podem ter o mesmo nome sem ambiguidade. (É possível definir seus próprios tipos de objetos e métodos, utilizando *classes*, veja em *Classes*) O método `append()`, mostrado no exemplo é definido para objetos do tipo lista; adiciona um novo elemento ao final da lista. Neste exemplo, ele equivale a `result = result + [a]`, só que mais eficiente.

[1] Na verdade, *passagem por referência para objeto* seria uma descrição melhor, pois, se um objeto mutável for passado, quem chamou verá as alterações feitas por quem foi chamado (por exemplo, a inclusão de itens em uma lista).

4.7 Mais sobre definição de funções

É possível definir funções com um número variável de argumentos. Existem três formas, que podem ser combinadas.

4.7.1 Argumentos com valor padrão

A mais útil das três é especificar um valor padrão para um ou mais argumentos. Isso cria uma função que pode ser invocada com menos argumentos do que os que foram definidos. Por exemplo:

```
def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
    while True:
        ok = raw_input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise IOError('refusenik user')
        print complaint
```

Essa função pode ser chamada de várias formas:

- fornecendo apenas o argumento obrigatório: `ask_ok('Do you really want to quit?')`
- fornecendo um dos argumentos opcionais: `ask_ok('OK to overwrite the file?', 2)`
- ou fornecendo todos os argumentos: `ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')`

Este exemplo também introduz o operador `in`, que verifica se uma sequência contém ou não um determinado valor.

Os valores padrões são avaliados no momento da definição da função, e no escopo em que a função foi *definida*, portanto:

```
i = 5

def f(arg=i):
    print arg

i = 6
f()
```

irá exibir `5`.

Aviso importante: Valores padrões são avaliados apenas uma vez. Isso faz diferença quando o valor é um objeto mutável, como uma lista, dicionário, ou instâncias de classes. Por exemplo, a função a seguir acumula os argumentos passados, nas chamadas subsequentes:

```
def f(a, L=[]):
    L.append(a)
    return L

print f(1)
print f(2)
print f(3)
```

Isso exibirá:

```
[1]
[1, 2]
[1, 2, 3]
```

Se não quiser que o valor padrão seja compartilhado entre chamadas subsequentes, pode reescrever a função assim:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

4.7.2 Argumentos nomeados

Funções também podem ser chamadas usando *argumentos nomeados* da forma `chave=valor`. Por exemplo, a função a seguir:

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    print "-- This parrot wouldn't", action,
    print "if you put", voltage, "volts through it."
    print "-- Lovely plumage, the", type
    print "-- It's", state, "!"
```

aceita um argumento obrigatório (`voltage`) e três argumentos opcionais (`state`, `action`, e `type`). Esta função pode ser chamada de qualquer uma dessas formas:

```
parrot(1000)                                          # 1 positional argument
parrot(voltage=1000)                                  # 1 keyword argument
parrot(voltage=1000000, action='VOOOOOM')             # 2 keyword arguments
parrot(action='VOOOOOM', voltage=1000000)             # 2 keyword arguments
parrot('a million', 'bereft of life', 'jump')         # 3 positional arguments
parrot('a thousand', state='pushing up the daisies')  # 1 positional, 1 keyword
```

mas todas as formas a seguir seriam inválidas:

```
parrot()                     # required argument missing
parrot(voltage=5.0, 'dead')  # non-keyword argument after a keyword argument
parrot(110, voltage=220)     # duplicate value for the same argument
parrot(actor='John Cleese')  # unknown keyword argument
```

Em uma chamada de função, argumentos nomeados devem vir depois dos argumentos posicionais. Todos os argumentos nomeados passados devem corresponder com argumentos aceitos pela função (ex. `actor` não é um argumento nomeado válido para a função `parrot`), mas sua ordem é irrelevante. Isto também inclui argumentos obrigatórios (ex.: `parrot(voltage=1000)` funciona). Nenhum argumento pode receber mais de um valor. Eis um exemplo que não funciona devido a esta restrição:

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: function() got multiple values for keyword argument 'a'
```

When a final formal parameter of the form `**name` is present, it receives a dictionary (see typesmapping) containing all keyword arguments except for those corresponding to a formal parameter. This may be combined with a formal parameter

of the form `*name` (described in the next subsection) which receives a tuple containing the positional arguments beyond the formal parameter list. (`*name` must occur before `**name`.) For example, if we define a function like this:

```
def cheeseshop(kind, *arguments, **keywords):
    print "-- Do you have any", kind, "?"
    print "-- I'm sorry, we're all out of", kind
    for arg in arguments:
        print arg
    print "-" * 40
    keys = sorted(keywords.keys())
    for kw in keys:
        print kw, ":", keywords[kw]
```

Pode ser chamada assim:

```
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper='Michael Palin',
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```

e, claro, exibiria:

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
----------------------------------------
client : John Cleese
shopkeeper : Michael Palin
sketch : Cheese Shop Sketch
```

Note that the list of keyword argument names is created by sorting the result of the keywords dictionary's `keys()` method before printing its contents; if this is not done, the order in which the arguments are printed is undefined.

4.7.3 Listas de argumentos arbitrários

Finalmente, a opção menos usada é especificar que a função pode ser chamada com um número arbitrário de argumentos. Esses argumentos serão empacotados em uma tupla (ver *Tuplas e sequências*). Antes dos argumentos em número variável, zero ou mais argumentos normais podem estar presentes.

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

4.7.4 Desempacotando listas de argumentos

A situação inversa ocorre quando os argumentos já estão numa lista ou tupla mas ela precisa ser explodida para invocarmos uma função que requer argumentos posicionais separados. Por exemplo, a função `range()` espera argumentos separados, *start* e *stop*. Se os valores já estiverem juntos em uma lista ou tupla, escreva a chamada de função com o operador `*` para desempacotá-los da sequência:

```
>>> range(3, 6)             # normal call with separate arguments
[3, 4, 5]
>>> args = [3, 6]
```

(continua na próxima página)

(continuação da página anterior)

```
>>> range(*args)            # call with arguments unpacked from a list
[3, 4, 5]
```

Da mesma forma, dicionários podem produzir argumentos nomeados com o operador **:

```
>>> def parrot(voltage, state='a stiff', action='voom'):
...     print "-- This parrot wouldn't", action,
...     print "if you put", voltage, "volts through it.",
...     print "E's", state, "!"
...
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin'␣
↪demised !
```

4.7.5 Expressões lambda

Pequenas funções anônimas podem ser criadas com a palavra-chave `lambda`. Esta função retorna a soma de seus dois argumentos: `lambda a, b: a+b`. As funções lambda podem ser usadas sempre que objetos função forem necessários. Eles são sintaticamente restritos a uma única expressão. Semanticamente, eles são apenas açúcar sintático para uma definição de função normal. Como definições de funções aninhadas, as funções lambda podem referenciar variáveis contidas no escopo:

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

O exemplo acima usa uma expressão lambda para retornar uma função. Outro uso é passar uma pequena função como um argumento:

```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> pairs.sort(key=lambda pair: pair[1])
>>> pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

4.7.6 Strings de documentação

There are emerging conventions about the content and formatting of documentation strings.

A primeira linha deve sempre ser curta, um resumo conciso do propósito do objeto. Por brevidade, não deve explicitamente se referir ao nome ou tipo do objeto, uma vez que estas informações estão disponíveis por outros meios (exceto se o nome da função for o próprio verbo que descreve a finalidade da função). Essa linha deve começar com letra maiúscula e terminar com ponto.

Se existem mais linhas na string de documentação, a segunda linha deve estar em branco, separando visualmente o resumo do resto da descrição. As linhas seguintes devem conter um ou mais parágrafos descrevendo as convenções de chamada ao objeto, seus efeitos colaterais, etc.

O analisador Python não remove a indentação de literais string multi-linha. Portanto, ferramentas que processem strings de documentação precisam lidar com isso, quando desejável. Existe uma convenção para isso. A primeira linha não vazia após a linha de sumário determina a indentação para o resto da string de documentação. (Não podemos usar a primeira linha para isso porque ela em geral está adjacente às aspas que iniciam a string, portanto sua indentação real não fica aparente.) Espaços em branco "equivalentes" a essa indentação são então removidos do início das demais linhas da string. Linhas com indentação menor não devem ocorrer, mas se ocorrerem, todos os espaços à sua esquerda são removidos. A equivalência de espaços em branco deve ser testada após a expansão das tabulações (8 espaços, normalmente).

Eis um exemplo de uma string de documentação multilinha:

```
>>> def my_function():
...     """Do nothing, but document it.
...
...     No, really, it doesn't do anything.
...     """
...     pass
...
>>> print my_function.__doc__
Do nothing, but document it.

    No, really, it doesn't do anything.
```

4.8 Intermezzo: estilo de codificação

Agora que está prestes a escrever códigos mais longos e complexos em Python, é um bom momento para falar sobre *estilo de codificação*. A maioria das linguagens podem ser escritas (ou *formatadas*) em diferentes estilos; alguns são mais legíveis do que outros. Tornar o seu código mais fácil de ler, para os outros, é sempre uma boa ideia, e adotar um estilo de codificação agradável ajuda bastante.

Em Python, a **PEP 8** tornou-se o guia de estilo adotado pela maioria dos projetos; promove um estilo de codificação muito legível e visualmente agradável. Todo desenvolvedor Python deve lê-lo em algum momento; eis os pontos mais importantes, selecionados para você:

- Use indentação com 4 espaços, e não use tabulações.

  4 spaces are a good compromise between small indentation (allows greater nesting depth) and large indentation (easier to read). Tabs introduce confusion, and are best left out.

- Quebre as linhas de modo que não excedam 79 caracteres.

  Isso ajuda os usuários com telas pequenas e torna possível abrir vários arquivos de código lado a lado em telas maiores.

- Deixe linhas em branco para separar funções e classes, e blocos de código dentro de funções.

- Quando possível, coloque comentários em uma linha própria.

- Escreva strings de documentação.

- Use espaços ao redor de operadores e após vírgulas, mas não diretamente dentro de parênteses, colchetes e chaves: `a = f(1, 2) + g(3, 4)`.

- Name your classes and functions consistently; the convention is to use `CamelCase` for classes and `lower_case_with_underscores` for functions and methods. Always use `self` as the name for the first method argument (see *Uma primeira olhada nas classes* for more on classes and methods).

- Don't use fancy encodings if your code is meant to be used in international environments. Plain ASCII works best in any case.

CAPÍTULO 5

Estruturas de dados

Esse capítulo descreve algumas coisas que você já aprendeu em detalhes e adiciona algumas coisas novas também.

5.1 Mais sobre listas

O tipo de dado lista tem ainda mais métodos. Aqui estão apresentados todos os métodos de objetos do tipo lista:

`list.`**`append`**`(`*x*`)`
Add an item to the end of the list; equivalent to `a[len(a):] = [x]`.

`list.`**`extend`**`(`*L*`)`
Extend the list by appending all the items in the given list; equivalent to `a[len(a):] = L`.

`list.`**`insert`**`(`*i*, *x*`)`
Insere um item em uma dada posição. O primeiro argumento é o índice do elemento antes do qual será feita a inserção, assim `a.insert(0, x)` insere um elemento na frente da lista e `a.insert(len(a), x)` e equivale a `a.append(x)`.

`list.`**`remove`**`(`*x*`)`
Remove the first item from the list whose value is *x*. It is an error if there is no such item.

`list.`**`pop`**`(`[*i*]`)`
Remove um item em uma dada posição na lista e o retorna. Se nenhum índice é especificado, `a.pop()` remove e devolve o último item da lista. (Os colchetes ao redor do *i* na demonstração do método indica que o parâmetro é opcional, e não que é necessário escrever estes colchetes ao chamar o método. Você verá este tipo de notação frequentemente na Biblioteca de Referência Python.)

`list.`**`index`**`(`*x*`)`
Return the index in the list of the first item whose value is *x*. It is an error if there is no such item.

`list.`**`count`**`(`*x*`)`
Devolve o número de vezes em que *x* aparece na lista.

`list.sort`(*cmp=None*, *key=None*, *reverse=False*)
Ordena os itens na lista (os argumentos podem ser usados para personalizar a ordenação, veja a função `sorted()` para maiores explicações).

`list.reverse`()
Reverse the elements of the list, in place.

Um exemplo que usa a maior parte dos métodos das listas:

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.25), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
>>> a.pop()
1234.5
>>> a
[-1, 1, 66.25, 333, 333]
```

You might have noticed that methods like `insert`, `remove` or `sort` that only modify the list have no return value printed – they return the default `None`. This is a design principle for all mutable data structures in Python.

5.1.1 Usando listas como pilhas

Os métodos de lista tornam muito fácil utilizar listas como pilhas, onde o item adicionado por último é o primeiro a ser recuperado (política "último a entrar, primeiro a sair"). Para adicionar um item ao topo da pilha, use `append()`. Para recuperar um item do topo da pilha use `pop()` sem nenhum índice. Por exemplo:

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

5.1.2 Usando listas como filas

Você também pode usar uma lista como uma fila, onde o primeiro item adicionado é o primeiro a ser recuperado (política "primeiro a entrar, primeiro a sair"); porém, listas não são eficientes para esta finalidade. Embora *appends* e *pops* no final da lista sejam rápidos, fazer *inserts* ou *pops* no início da lista é lento (porque todos os demais elementos têm que ser deslocados).

Para implementar uma fila, use a classe `collections.deque` que foi projetada para permitir *appends* e *pops* eficientes nas duas extremidades. Por exemplo:

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")          # Graham arrives
>>> queue.popleft()                 # The first to arrive now leaves
'Eric'
>>> queue.popleft()                 # The second to arrive now leaves
'John'
>>> queue                           # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

5.1.3 Functional Programming Tools

There are three built-in functions that are very useful when used with lists: `filter()`, `map()`, and `reduce()`.

`filter(function, sequence)` returns a sequence consisting of those items from the sequence for which `function(item)` is true. If *sequence* is a `str`, `unicode` or `tuple`, the result will be of the same type; otherwise, it is always a `list`. For example, to compute a sequence of numbers divisible by 3 or 5:

```
>>> def f(x): return x % 3 == 0 or x % 5 == 0
...
>>> filter(f, range(2, 25))
[3, 5, 6, 9, 10, 12, 15, 18, 20, 21, 24]
```

`map(function, sequence)` calls `function(item)` for each of the sequence's items and returns a list of the return values. For example, to compute some cubes:

```
>>> def cube(x): return x*x*x
...
>>> map(cube, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

More than one sequence may be passed; the function must then have as many arguments as there are sequences and is called with the corresponding item from each sequence (or `None` if some sequence is shorter than another). For example:

```
>>> seq = range(8)
>>> def add(x, y): return x+y
...
>>> map(add, seq, seq)
[0, 2, 4, 6, 8, 10, 12, 14]
```

`reduce(function, sequence)` returns a single value constructed by calling the binary function *function* on the first two items of the sequence, then on the result and the next item, and so on. For example, to compute the sum of the numbers 1 through 10:

```
>>> def add(x,y): return x+y
...
>>> reduce(add, range(1, 11))
55
```

If there's only one item in the sequence, its value is returned; if the sequence is empty, an exception is raised.

A third argument can be passed to indicate the starting value. In this case the starting value is returned for an empty sequence, and the function is first applied to the starting value and the first sequence item, then to the result and the next item, and so on. For example,

```
>>> def sum(seq):
...     def add(x,y): return x+y
...     return reduce(add, seq, 0)
...
>>> sum(range(1, 11))
55
>>> sum([])
0
```

Don't use this example's definition of `sum()`: since summing numbers is such a common need, a built-in function `sum(sequence)` is already provided, and works exactly like this.

5.1.4 Compreensões de lista

Compreensões de lista fornece uma maneira concisa de criar uma lista. Aplicações comuns são criar novas listas onde cada elemento é o resultado de alguma operação aplicada a cada elemento de outra sequência ou iterável, ou criar uma subsequência de elementos que satisfaçam uma certa condição. (N.d.T. o termo original em inglês é *list comprehensions*, muito utilizado no Brasil; também se usa a abreviação *listcomp*).

Por exemplo, suponha que queremos criar uma lista de quadrados, assim:

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

We can obtain the same result with:

```
squares = [x**2 for x in range(10)]
```

This is also equivalent to `squares = map(lambda x: x**2, range(10))`, but it's more concise and readable.

A list comprehension consists of brackets containing an expression followed by a `for` clause, then zero or more `for` or `if` clauses. The result will be a new list resulting from evaluating the expression in the context of the `for` and `if` clauses which follow it. For example, this listcomp combines the elements of two lists if they are not equal:

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

and it's equivalent to:

```
>>> combs = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...         if x != y:
...             combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Note como a ordem das instruções `for` e `if` é a mesma em ambos os trechos.

Se a expressão é uma tupla (ex., `(x, y)` no exemplo anterior), ela deve ser inserida entre parênteses.

```
>>> vec = [-4, -2, 0, 2, 4]
>>> # create a new list with the values doubled
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filter the list to exclude negative numbers
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # apply a function to all the elements
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # call a method on each element
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # create a list of 2-tuples like (number, square)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # the tuple must be parenthesized, otherwise an error is raised
>>> [x, x**2 for x in range(6)]
  File "<stdin>", line 1, in <module>
    [x, x**2 for x in range(6)]
               ^
SyntaxError: invalid syntax
>>> # flatten a list using a listcomp with two 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Compreensões de lista podem conter expressões complexas e funções aninhadas:

```
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

Compreensões de lista aninhadas

A expressão inicial em uma list comprehension pode ser qualquer expressão arbitrária, incluindo outra list comprehension.

Observe este exemplo de uma matriz 3x4 implementada como uma lista de 3 listas de comprimento 4:

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
```

A compreensão de lista abaixo transpõe as linhas e colunas:

```
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Como vimos na seção anterior, a compreensão de lista aninhada é computada no contexto da cláusula `for` seguinte, portanto o exemplo acima equivale a:

```
>>> transposed = []
>>> for i in range(4):
...     transposed.append([row[i] for row in matrix])
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

e isso, por sua vez, faz o mesmo que isto:

```
>>> transposed = []
>>> for i in range(4):
...     # the following 3 lines implement the nested listcomp
...     transposed_row = []
...     for row in matrix:
...         transposed_row.append(row[i])
...     transposed.append(transposed_row)
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Na prática, você deve dar preferência a funções embutidas em vez de expressões complexas. A função `zip()` resolve muito bem este caso de uso:

```
>>> zip(*matrix)
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

Veja *Desempacotando listas de argumentos* para entender o uso do asterisco neste exemplo.

5.2 The `del` statement

There is a way to remove an item from a list given its index instead of its value: the `del` statement. This differs from the `pop()` method which returns a value. The `del` statement can also be used to remove slices from a list or clear the entire list (which we did earlier by assignment of an empty list to the slice). For example:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

`del` também pode ser usado para remover totalmente uma variável:

```
>>> del a
```

Referenciar a variável `a` depois de sua remoção constitui erro (pelo menos até que seja feita uma nova atribuição para ela). Encontraremos outros usos para a instrução `del` mais tarde.

5.3 Tuplas e sequências

Vimos que listas e strings têm muitas propriedades em comum, como indexação e operações de fatiamento. Elas são dois exemplos de *sequências* (veja typesseq). Como Python é uma linguagem em evolução, outros tipos de sequências podem ser adicionados. Existe ainda um outro tipo de sequência padrão na linguagem: a *tupla*.

Uma tupla consiste em uma sequência de valores separados por vírgulas, por exemplo:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tuples are immutable:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

Como você pode ver no trecho acima, na saída do console as tuplas são sempre envolvidas por parênteses, assim tuplas aninhadas podem ser lidas corretamente. Na criação, tuplas podem ser envolvidas ou não por parênteses, desde que o contexto não exija os parênteses (como no caso da tupla dentro de uma expressão maior). Não é possível atribuir itens individuais de uma tupla, contudo é possível criar tuplas que contenham objetos mutáveis, como listas.

Apesar de tuplas serem similares a listas, elas são frequentemente utilizadas em situações diferentes e com propósitos distintos. Tuplas são *imutáveis*, e usualmente contém uma sequência heterogênea de elementos que são acessados via desempacotamento (ver a seguir nessa seção) ou índice (ou mesmo por um atributo no caso de `namedtuples`). Listas são *mutáveis*, e seus elementos geralmente são homogêneos e são acessados iterando sobre a lista.

Um problema especial é a criação de tuplas contendo 0 ou 1 itens: a sintaxe usa certos truques para acomodar estes casos. Tuplas vazias são construídas por um par de parênteses vazios; uma tupla unitária é construída por um único valor e uma vírgula entre parênteses (não basta colocar um único valor entre parênteses). Feio, mas funciona. Por exemplo:

```
>>> empty = ()
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

A instrução `t = 12345, 54321, 'bom dia!'` é um exemplo de *empacotamento de tupla*: os valores `12345`, `54321` e `'bom dia!'` são empacotados em uma tupla. A operação inversa também é possível:

```
>>> x, y, z = t
```

This is called, appropriately enough, *sequence unpacking* and works for any sequence on the right-hand side. Sequence unpacking requires the list of variables on the left to have the same number of elements as the length of the sequence. Note that multiple assignment is really just a combination of tuple packing and sequence unpacking.

5.4 Conjuntos

Python também inclui um tipo de dados para conjuntos, chamado `set`. Um conjunto é uma coleção desordenada de elementos, sem elementos repetidos. Usos comuns para conjuntos incluem a verificação eficiente da existência de objetos e a eliminação de itens duplicados. Conjuntos também suportam operações matemáticas como união, interseção, diferença e diferença simétrica.

Chaves ou a função `set()` podem ser usados para criar conjuntos. Note: para criar um conjunto vazio você precisa usar `set()`, não `{}`; este último cria um dicionário vazio, uma estrutura de dados que discutiremos na próxima seção.

Uma pequena demonstração:

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> fruit = set(basket)               # create a set without duplicates
>>> fruit
set(['orange', 'pear', 'apple', 'banana'])
>>> 'orange' in fruit                 # fast membership testing
True
>>> 'crabgrass' in fruit
False

>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
set(['a', 'r', 'b', 'c', 'd'])
>>> a - b                              # letters in a but not in b
set(['r', 'd', 'b'])
```

(continua na próxima página)

(continuação da página anterior)

```
>>> a | b                              # letters in either a or b
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
>>> a & b                              # letters in both a and b
set(['a', 'c'])
>>> a ^ b                              # letters in a or b but not both
set(['r', 'd', 'b', 'm', 'z', 'l'])
```

Da mesma forma que *compreensão de listas*, compreensões de conjunto também são suportadas:

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
set(['r', 'd'])
```

5.5 Dicionários

Outra estrutura de dados muito útil embutida em Python é o *dicionário*, cujo tipo é `dict` (ver typesmapping). Dicionários são também chamados de "memória associativa" ou "vetor associativo" em outras linguagens. Diferente de sequências que são indexadas por inteiros, dicionários são indexados por chaves (*keys*), que podem ser de qualquer tipo imutável (como strings e inteiros). Tuplas também podem ser chaves se contiverem apenas strings, inteiros ou outras tuplas. Se a tupla contiver, direta ou indiretamente, qualquer valor mutável, não poderá ser chave. Listas não podem ser usadas como chaves porque podem ser modificadas *internamente* pela atribuição em índices ou fatias, e por métodos como `append()` e `extend()`.

It is best to think of a dictionary as an unordered set of *key: value* pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: `{}`. Placing a comma-separated list of key:value pairs within the braces adds initial key:value pairs to the dictionary; this is also the way dictionaries are written on output.

As principais operações em um dicionário são armazenar e recuperar valores a partir de chaves. Também é possível remover um par *chave:valor* com o comando `del`. Se você armazenar um valor utilizando uma chave já presente, o antigo valor será substituído pelo novo. Se tentar recuperar um valor usando uma chave inexistente, será gerado um erro.

The `keys()` method of a dictionary object returns a list of all the keys used in the dictionary, in arbitrary order (if you want it sorted, just apply the `sorted()` function to it). To check whether a single key is in the dictionary, use the `in` keyword.

A seguir, um exemplo de uso do dicionário:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
```

O construtor `dict()` produz dicionários diretamente de sequências de pares chave-valor:

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

Além disso, as compreensões de dicionários podem ser usadas para criar dicionários a partir de expressões arbitrárias de chave e valor:

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

Quando chaves são strings simples, é mais fácil especificar os pares usando argumentos nomeados no construtor:

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

5.6 Técnicas de iteração

Ao iterar sobre sequências, a posição e o valor correspondente podem ser obtidos simultaneamente usando a função `enumerate()`.

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print i, v
...
0 tic
1 tac
2 toe
```

Para percorrer duas ou mais sequências ao mesmo tempo, as entradas podem ser pareadas com a função `zip()`.

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print 'What is your {0}?  It is {1}.'.format(q, a)
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

Para percorrer uma sequência em ordem inversa, chame a função `reversed()` com a sequência na ordem original.

```
>>> for i in reversed(xrange(1,10,2)):
...     print i
...
9
7
5
3
1
```

Para percorrer uma sequência de maneira ordenada, use a função `sorted()`, que retorna uma lista ordenada com os itens, mantendo a sequência original inalterada.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
...     print f
```

(continua na próxima página)

(continuação da página anterior)

```
...
apple
banana
orange
pear
```

When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the `iteritems()` method.

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.iteritems():
...     print k, v
...
gallahad the pure
robin the brave
```

Às vezes é tentador alterar uma lista enquanto você itera sobre ela; porém, costuma ser mais simples e seguro criar uma nova lista.

```
>>> import math
>>> raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]
>>> filtered_data = []
>>> for value in raw_data:
...     if not math.isnan(value):
...         filtered_data.append(value)
...
>>> filtered_data
[56.2, 51.7, 55.3, 52.5, 47.8]
```

5.7 Mais sobre condições

As condições de controle usadas em `while` e `if` podem conter quaisquer operadores, não apenas comparações.

Os operadores de comparação `in` e `not in` verificam se um valor ocorre (ou não ocorre) em uma dada sequência. Os operadores `is` e `is not` comparam se dois objetos são na verdade o mesmo objeto; isto só é relevante no contexto de objetos mutáveis, como listas. Todos os operadores de comparação possuem a mesma precedência, que é menor do que a prioridade de todos os operadores numéricos.

Comparações podem ser encadeadas: Por exemplo `a < b == c` testa se `a` é menor que `b` e também se `b` é igual a `c`.

Comparações podem ser combinadas através de operadores booleanos `and` e `or`, e o resultado de uma comparação (ou de qualquer outra expressão), pode ter seu valor booleano negado através de `not`. Estes possuem menor prioridade que os demais operadores de comparação. Entre eles, `not` é o de maior prioridade e `or` o de menor. Dessa forma, a condição `A and not B or C` é equivalente a `(A and (not B)) or C`. Naturalmente, parênteses podem ser usados para expressar o agrupamento desejado.

Os operadores booleanos `and` e `or` são operadores *curto-circuito*: seus argumentos são avaliados da esquerda para a direita, e a avaliação encerra quando o resultado é determinado. Por exemplo, se `A` e `C` são expressões verdadeiras, mas `B` é falsa, então `A and B and C` não chega a avaliar a expressão `C`. Em geral, quando usado sobre valores genéricos e não como booleanos, o valor do resultado de um operador curto-circuito é o último valor avaliado na expressão.

É possível atribuir o resultado de uma comparação ou outra expressão booleana para uma variável. Por exemplo:

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

Note that in Python, unlike C, assignment cannot occur inside expressions. C programmers may grumble about this, but it avoids a common class of problems encountered in C programs: typing = in an expression when == was intended.

5.8 Comparando sequências e outros tipos

Sequence objects may be compared to other objects with the same sequence type. The comparison uses *lexicographical* ordering: first the first two items are compared, and if they differ this determines the outcome of the comparison; if they are equal, the next two items are compared, and so on, until either sequence is exhausted. If two items to be compared are themselves sequences of the same type, the lexicographical comparison is carried out recursively. If all items of two sequences compare equal, the sequences are considered equal. If one sequence is an initial sub-sequence of the other, the shorter sequence is the smaller (lesser) one. Lexicographical ordering for strings uses the ASCII ordering for individual characters. Some examples of comparisons between sequences of the same type:

```
(1, 2, 3)              < (1, 2, 4)
[1, 2, 3]              < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4)           < (1, 2, 4)
(1, 2)                 < (1, 2, -1)
(1, 2, 3)             == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'))   < (1, 2, ('abc', 'a'), 4)
```

Note that comparing objects of different types is legal. The outcome is deterministic but arbitrary: the types are ordered by their name. Thus, a list is always smaller than a string, a string is always smaller than a tuple, etc.[1] Mixed numeric types are compared according to their numeric value, so 0 equals 0.0, etc.

[1] The rules for comparing objects of different types should not be relied upon; they may change in a future version of the language.

CAPÍTULO 6

Módulos

Ao sair e entrar de novo no interpretador Python, as definições anteriores (funções e variáveis) são perdidas. Portanto, se quiser escrever um programa maior, será mais eficiente usar um editor de texto para preparar as entradas para o interpretador, e executá-lo usando o arquivo como entrada. Isso é conhecido como criar um *script*. Se o programa se torna ainda maior, é uma boa prática dividi-lo em arquivos menores, para facilitar a manutenção. Também é preferível usar um arquivo separado para uma função que você escreveria em vários programas diferentes, para não copiar a definição de função em cada um deles.

Para permitir isso, o Python tem uma maneira de colocar as definições em um arquivo e então usá-las em um script ou em uma execução interativa do interpretador. Tal arquivo é chamado de *módulo*; definições de um módulo podem ser *importadas* para outros módulos, ou para o módulo *principal* (a coleção de variáveis a que você tem acesso num script executado como um programa e no modo calculadora).

Um módulo é um arquivo contendo definições e instruções Python. O nome do arquivo é o nome do módulo acrescido do sufixo `.py`. Dentro de um módulo, o nome do módulo (como uma string) está disponível como o valor da variável global `__name__`. Por exemplo, use seu editor de texto favorito para criar um arquivo chamado `fibo.py` no diretório atual com o seguinte conteúdo:

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Agora entre no interpretador Python e importe o módulo com o seguinte comando:

```
>>> import fibo
```

Isso não coloca os nomes das funções definidas em `fibo` diretamente na tabela de símbolos atual; isso coloca somente o nome do módulo `fibo`. Usando o nome do módulo você pode acessar as funções:

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

Se pretender usar uma função muitas vezes, você pode atribui-lá a um nome local:

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

6.1 Mais sobre módulos

Um módulo pode conter tanto instruções executáveis quanto definições de funções e classes. Essas instruções servem para inicializar o módulo. Eles são executados somente na *primeira* vez que o módulo é encontrado em uma instrução de importação.[1] (Também rodam se o arquivo é executado como um script.)

Cada módulo tem sua própria tabela de símbolos privada, que é usada como tabela de símbolos global para todas as funções definidas no módulo. Assim, o autor de um módulo pode usar variáveis globais no seu módulo sem se preocupar com conflitos acidentais com as variáveis globais do usuário. Por outro lado, se você precisar usar uma variável global de um módulo, poderá fazê-lo com a mesma notação usada para se referir às suas funções, `nomemodulo.nomeitem`.

Módulos podem importar outros módulos. É costume, porém não obrigatório, colocar todos os comandos `import` no início do módulo (ou script , se preferir). As definições do módulo importado são colocadas na tabela de símbolos global do módulo que faz a importação.

Existe uma variante do comando `import` que importa definições de um módulo diretamente para a tabela de símbolos do módulo importador. Por exemplo:

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Isso não coloca o nome do módulo de onde foram feitas as importações na tabela de símbolos local (assim, no exemplo, `fibo` não está definido).

Existe ainda uma variante que importa todos os nomes definidos em um módulo:

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This imports all names except those beginning with an underscore (_).

Note que, em geral, a prática do `import *` de um módulo ou pacote é desaprovada, uma vez que muitas vezes dificulta a leitura do código. Contudo, é aceitável para diminuir a digitação em sessões interativas.

[1] [#] Na verdade, definições de funções também são 'instruções' que são 'executados'; a execução da definição de uma função coloca o nome da função na tabela de símbolos global do módulo.

If the module name is followed by `as`, then the name following `as` is bound directly to the imported module.

```
>>> import fibo as fib
>>> fib.fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Isto efetivamente importa o módulo, da mesma maneira que `import fibo` fará, com a única diferença de estar disponível com o nome `fib`.

Também pode ser utilizado com a palavra-chave `from`, com efeitos similares:

```
>>> from fibo import fib as fibonacci
>>> fibonacci(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Nota: For efficiency reasons, each module is only imported once per interpreter session. Therefore, if you change your modules, you must restart the interpreter – or, if it's just one module you want to test interactively, use `reload()`, e.g. `reload(modulename)`.

6.1.1 Executando módulos como scripts

Quando você rodar um módulo Python com

```
python fibo.py <arguments>
```

o código no módulo será executado, da mesma forma que quando é importado, mas com a variável `__name__` com valor `"__main__"`. Isto significa que adicionando este código ao final do seu módulo:

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

você pode tornar o arquivo utilizável tanto como script quanto como um módulo importável, porque o código que analisa a linha de comando só roda se o módulo é executado como arquivo "principal":

```
$ python fibo.py 50
1 1 2 3 5 8 13 21 34
```

Se o módulo é importado, o código não é executado:

```
>>> import fibo
>>>
```

Isso é frequentemente usado para fornecer uma interface de usuário conveniente para um módulo, ou para realizar testes (rodando o módulo como um script executa um conjunto de testes).

6.1.2 O caminho de busca dos módulos

Quando um módulo chamado `spam` é importado, o interpretador procura um módulo embutido com este nome. Se não encontra, procura um arquivo chamado `spam.py` em uma lista de diretórios incluídos na variável `sys.path`. A `sys.path` é inicializada com estes locais:

- the directory containing the input script (or the current directory).
- A variável de ambiente `PYTHONPATH` (uma lista de nomes de diretórios, com a mesma sintaxe da variável de ambiente `PATH`).
- the installation-dependent default.

Após a inicialização, programas Python podem modificar `sys.path`. O diretório que contém o script sendo executado é colocado no início da lista de caminhos, à frente do caminho da biblioteca padrão. Isto significa que módulos nesse diretório serão carregados, no lugar de módulos com o mesmo nome na biblioteca padrão. Isso costuma ser um erro, a menos que seja intencional. Veja a seção *Módulos padrões* para mais informações.

6.1.3 Arquivos Python "compilados"

As an important speed-up of the start-up time for short programs that use a lot of standard modules, if a file called `spam.pyc` exists in the directory where `spam.py` is found, this is assumed to contain an already-"byte-compiled" version of the module `spam`. The modification time of the version of `spam.py` used to create `spam.pyc` is recorded in `spam.pyc`, and the `.pyc` file is ignored if these don't match.

Normally, you don't need to do anything to create the `spam.pyc` file. Whenever `spam.py` is successfully compiled, an attempt is made to write the compiled version to `spam.pyc`. It is not an error if this attempt fails; if for any reason the file is not written completely, the resulting `spam.pyc` file will be recognized as invalid and thus ignored later. The contents of the `spam.pyc` file are platform independent, so a Python module directory can be shared by machines of different architectures.

Algumas dicas para especialistas:

- When the Python interpreter is invoked with the `-O` flag, optimized code is generated and stored in `.pyo` files. The optimizer currently doesn't help much; it only removes `assert` statements. When `-O` is used, *all bytecode* is optimized; `.pyc` files are ignored and `.py` files are compiled to optimized bytecode.
- Passing two `-O` flags to the Python interpreter (`-OO`) will cause the bytecode compiler to perform optimizations that could in some rare cases result in malfunctioning programs. Currently only `__doc__` strings are removed from the bytecode, resulting in more compact `.pyo` files. Since some programs may rely on having these available, you should only use this option if you know what you're doing.
- A program doesn't run any faster when it is read from a `.pyc` or `.pyo` file than when it is read from a `.py` file; the only thing that's faster about `.pyc` or `.pyo` files is the speed with which they are loaded.
- When a script is run by giving its name on the command line, the bytecode for the script is never written to a `.pyc` or `.pyo` file. Thus, the startup time of a script may be reduced by moving most of its code to a module and having a small bootstrap script that imports that module. It is also possible to name a `.pyc` or `.pyo` file directly on the command line.
- It is possible to have a file called `spam.pyc` (or `spam.pyo` when `-O` is used) without a file `spam.py` for the same module. This can be used to distribute a library of Python code in a form that is moderately hard to reverse engineer.
- The module `compileall` can create `.pyc` files (or `.pyo` files when `-O` is used) for all modules in a directory.

6.2 Módulos padrões

O Python traz uma biblioteca padrão de módulos, descrita em um documento em separado, a Referência da Biblioteca Python (doravante "Referência da Biblioteca"). Alguns módulos estão embutidos no interpretador; estes possibilitam acesso a operações que não são parte do núcleo da linguagem, mas estão no interpretador seja por eficiência ou para permitir o acesso a chamadas do sistema operacional. O conjunto destes módulos é uma opção de configuração que depende também da plataforma utilizada. Por exemplo, o módulo `winreg` só está disponível em sistemas Windows. Existe um módulo que requer especial atenção: `sys`, que é embutido em qualquer interpretador Python. As variáveis `sys.ps1` e `sys.ps2` definem as strings utilizadas como prompt primário e secundário:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print 'Yuck!'
Yuck!
C>
```

Essas variáveis só estão definidas se o interpretador está em modo interativo.

A variável `sys.path` contém uma lista de strings que determina os caminhos de busca de módulos conhecidos pelo interpretador. Ela é inicializada para um caminho padrão, determinado pela variável de ambiente `PYTHONPATH`, ou por um valor padrão interno, se `PYTHONPATH` não estiver definida. Você pode modificá-la com as operações típicas de lista, por exemplo:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

6.3 A função `dir()`

A função embutida `dir()` é usada para descobrir quais nomes são definidos por um módulo. Ela devolve uma lista ordenada de strings:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__package__',
 '__stderr__', '__stdin__', '__stdout__', '_clear_type_cache',
 '_current_frames', '_getframe', '_mercurial', 'api_version', 'argv',
 'builtin_module_names', 'byteorder', 'call_tracing', 'callstats',
 'copyright', 'displayhook', 'dont_write_bytecode', 'exc_clear', 'exc_info',
 'exc_traceback', 'exc_type', 'exc_value', 'excepthook', 'exec_prefix',
 'executable', 'exit', 'flags', 'float_info', 'float_repr_style',
 'getcheckinterval', 'getdefaultencoding', 'getdlopenflags',
 'getfilesystemencoding', 'getobjects', 'getprofile', 'getrecursionlimit',
 'getrefcount', 'getsizeof', 'gettotalrefcount', 'gettrace', 'hexversion',
 'long_info', 'maxint', 'maxsize', 'maxunicode', 'meta_path', 'modules',
 'path', 'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'ps1',
 'py3kwarning', 'setcheckinterval', 'setdlopenflags', 'setprofile',
 'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout', 'subversion',
 'version', 'version_info', 'warnoptions']
```

Sem argumentos, `dir()` lista os nomes atualmente definidos:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__name__', '__package__', 'a', 'fib', 'fibo', 'sys']
```

Observe que ela lista todo tipo de nomes: variáveis, módulos, funções, etc.

`dir()` does not list the names of built-in functions and variables. If you want a list of those, they are defined in the standard module `__builtin__`:

```
>>> import __builtin__
>>> dir(__builtin__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
 'BufferError', 'BytesWarning', 'DeprecationWarning', 'EOFError',
 'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FloatingPointError',
 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning',
 'IndentationError', 'IndexError', 'KeyError', 'KeyboardInterrupt',
 'LookupError', 'MemoryError', 'NameError', 'None', 'NotImplemented',
 'NotImplementedError', 'OSError', 'OverflowError',
 'PendingDeprecationWarning', 'ReferenceError', 'RuntimeError',
 'RuntimeWarning', 'StandardError', 'StopIteration', 'SyntaxError',
 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'True',
 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError',
 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError',
 'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning',
 'ZeroDivisionError', '_', '__debug__', '__doc__', '__import__',
 '__name__', '__package__', 'abs', 'all', 'any', 'apply', 'basestring',
 'bin', 'bool', 'buffer', 'bytearray', 'bytes', 'callable', 'chr',
 'classmethod', 'cmp', 'coerce', 'compile', 'complex', 'copyright',
 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval',
 'execfile', 'exit', 'file', 'filter', 'float', 'format', 'frozenset',
 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input',
 'int', 'intern', 'isinstance', 'issubclass', 'iter', 'len', 'license',
 'list', 'locals', 'long', 'map', 'max', 'memoryview', 'min', 'next',
 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit',
 'range', 'raw_input', 'reduce', 'reload', 'repr', 'reversed', 'round',
 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super',
 'tuple', 'type', 'unichr', 'unicode', 'vars', 'xrange', 'zip']
```

6.4 Pacotes

Os pacotes são uma maneira de estruturar o "espaço de nomes" dos módulos Python, usando "nomes de módulo com pontos". Por exemplo, o nome do módulo `A.B` designa um submódulo chamado `B`, em um pacote chamado `A`. Assim como o uso de módulos evita que os autores de módulos diferentes tenham que se preocupar com nomes de variáveis globais, o uso de nomes de módulos com pontos evita que os autores de pacotes com muitos módulos, como NumPy ou Pillow, tenham que se preocupar com os nomes dos módulos uns dos outros.

Suponha que você queira projetar uma coleção de módulos (um "pacote") para o gerenciamento uniforme de arquivos de som. Existem muitos formatos diferentes (normalmente identificados pela extensão do nome de arquivo, por exemplo `.wav`, `.aiff`, `.au`), de forma que você pode precisar criar e manter uma crescente coleção de módulos de conversão entre formatos. Ainda podem existir muitas operações diferentes, passíveis de aplicação sobre os arquivos de som (mixagem, eco, equalização, efeito stereo artificial). Logo, possivelmente você também estará escrevendo uma coleção sempre

crescente de módulos para aplicar estas operações. Eis uma possível estrutura para o seu pacote (expressa em termos de um sistema de arquivos hierárquico):

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

Ao importar esse pacote, Python busca pelo subdiretório com mesmo nome, nos diretórios listados em `sys.path`.

The `__init__.py` files are required to make Python treat the directories as containing packages; this is done to prevent directories with a common name, such as `string`, from unintentionally hiding valid modules that occur later on the module search path. In the simplest case, `__init__.py` can just be an empty file, but it can also execute initialization code for the package or set the `__all__` variable, described later.

Usuários do pacote podem importar módulos individuais, por exemplo:

```
import sound.effects.echo
```

Isso carrega o submódulo `sound.effects.echo`. Ele deve ser referenciado com seu nome completo, como em:

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

Uma maneira alternativa para a importação desse módulo é:

```
from sound.effects import echo
```

Isso carrega o submódulo `echo` sem necessidade de mencionar o prefixo do pacote no momento da utilização, assim:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

Também é possível importar diretamente uma única variável ou função:

```
from sound.effects.echo import echofilter
```

Novamente, isso carrega o submódulo `echo`, mas a função `echofilter()` está acessível diretamente sem prefixo:

```
echofilter(input, output, delay=0.7, atten=4)
```

Observe que ao utilizar `from pacote import item`, o item pode ser um subpacote, submódulo, classe, função ou variável. O comando `import` primeiro testa se o item está definido no pacote, senão assume que é um módulo e tenta carregá-lo. Se falhar em encontrar o módulo, uma exceção `ImportError` é levantada.

Em oposição, em uma construção como `import item.subitem.subsubitem`, cada item, com exceção do último, deve ser um pacote. O último pode ser também um pacote ou módulo, mas nunca uma classe, função ou variável contida em um módulo.

6.4.1 Importando * de um pacote

Agora, o que acontece quando um usuário escreve `from sound.effects import *` ? Idealmente, poderia se esperar que este comando vasculhasse o sistema de arquivos, encontrasse todos os submódulos presentes no pacote, e os importasse. Isso poderia demorar muito e a importação de submódulos pode ocasionar efeitos colaterais, que somente deveriam ocorrer quando o submódulo é explicitamente importado.

A única solução é o autor do pacote fornecer um índice explícito do pacote. O comando `import` usa a seguinte convenção: se o arquivo `__init__.py` do pacote define uma lista chamada `__all__`, então esta lista indica os nomes dos módulos a serem importados quando o comando `from pacote import *` é acionado. Fica a cargo do autor do pacote manter esta lista atualizada, inclusive fica a seu critério excluir inteiramente o suporte a importação direta de todo o pacote através de `from pacote import *`. Por exemplo, o arquivo `sounds/effects/__init__.py` poderia conter apenas:

```
__all__ = ["echo", "surround", "reverse"]
```

Isso significaria que `from sound.effects import *` importaria apenas os três submódulos especificados no pacote `sound`.

Se `__all__` não estiver definido, o comando `from sound.effects import *` não importa todos os submódulos do pacote `sound.effects` no espaço de nomes atual. Há apenas garantia que o pacote `sound.effects` foi importado (possivelmente executando qualquer código de inicialização em `__init__.py`) juntamente com os nomes definidos no pacote. Isso inclui todo nome definido em `__init__.py` bem como em qualquer submódulo importado a partir deste. Também inclui quaisquer submódulos do pacote que tenham sido carregados explicitamente por comandos `import` anteriores. Considere o código abaixo:

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

Nesse exemplo, os nomes `echo` e `surround` são importados no espaço de nomes atual, no momento em que o comando `from...import` é executado, pois estão definidos no pacote `sound.effects`. (Isso também funciona quando `__all__` estiver definida.)

Apesar de que certos módulos são projetados para exportar apenas nomes conforme algum critério quando se faz `import *`, ainda assim essa sintaxe é considerada uma prática ruim em código de produção.

Lembre-se, não há nada errado em usar `from pacote import submodulo_especifico`! De fato, essa é a notação recomendada, a menos que o módulo importado necessite usar submódulos com o mesmo nome, de diferentes pacotes.

6.4.2 Referências em um mesmo pacote

The submodules often need to refer to each other. For example, the `surround` module might use the `echo` module. In fact, such references are so common that the `import` statement first looks in the containing package before looking in the standard module search path. Thus, the `surround` module can simply use `import echo` or `from echo import echofilter`. If the imported module is not found in the current package (the package of which the current module is a submodule), the `import` statement looks for a top-level module with the given name.

Quando pacotes são estruturados em subpacotes (como no pacote `sound` do exemplo), pode-se usar a sintaxe de importações absolutas para se referir aos submódulos de pacotes irmãos (o que na prática é uma forma de fazer um import relativo, a partir da base do pacote). Por exemplo, se o módulo `sound.filters.vocoder` precisa usar o módulo `echo` do pacote `sound.effects`, é preciso importá-lo com `from sound.effects import echo`.

Starting with Python 2.5, in addition to the implicit relative imports described above, you can write explicit relative imports with the `from module import name` form of import statement. These explicit relative imports use leading dots to indicate the current and parent packages involved in the relative import. From the `surround` module for example, you might use:

```
from . import echo
from .. import formats
from ..filters import equalizer
```

Note that both explicit and implicit relative imports are based on the name of the current module. Since the name of the main module is always `"__main__"`, modules intended for use as the main module of a Python application should always use absolute imports.

6.4.3 Pacotes em múltiplos diretórios

Pacotes possuem mais um atributo especial, `__path__`. Inicializado como uma lista contendo o nome do diretório onde está o arquivo `__init__.py` do pacote, antes do código naquele arquivo ser executado. Esta variável pode ser modificada; isso afeta a busca futura de módulos e subpacotes contidos no pacote.

Apesar de não ser muito usado, esse mecanismo permite estender o conjunto de módulos encontrados em um pacote.

CAPÍTULO 7

Entrada e saída

Existem várias maneiras de apresentar a saída de um programa; os dados podem ser exibidos em forma legível para seres humanos, ou escritos em arquivos para uso posterior. Este capítulo apresentará algumas das possibilidades.

7.1 Refinando a formatação de saída

So far we've encountered two ways of writing values: *expression statements* and the `print` statement. (A third way is using the `write()` method of file objects; the standard output file can be referenced as `sys.stdout`. See the Library Reference for more information on this.)

Often you'll want more control over the formatting of your output than simply printing space-separated values. There are two ways to format your output; the first way is to do all the string handling yourself; using string slicing and concatenation operations you can create any layout you can imagine. The string types have some methods that perform useful operations for padding strings to a given column width; these will be discussed shortly. The second way is to use the `str.format()` method.

The `string` module contains a `Template` class which offers yet another way to substitute values into strings.

One question remains, of course: how do you convert values to strings? Luckily, Python has ways to convert any value to a string: pass it to the `repr()` or `str()` functions.

The `str()` function is meant to return representations of values which are fairly human-readable, while `repr()` is meant to generate representations which can be read by the interpreter (or will force a `SyntaxError` if there is no equivalent syntax). For objects which don't have a particular representation for human consumption, `str()` will return the same value as `repr()`. Many values, such as numbers or structures like lists and dictionaries, have the same representation using either function. Strings and floating point numbers, in particular, have two distinct representations.

Alguns exemplos:

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
```

(continua na próxima página)

(continuação da página anterior)

```
>>> str(1.0/7.0)
'0.142857142857'
>>> repr(1.0/7.0)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
>>> print s
The value of x is 32.5, and y is 40000...
>>> # The repr() of a string adds string quotes and backslashes:
... hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print hellos
'hello, world\n'
>>> # The argument to repr() may be any Python object:
... repr((x, y, ('spam', 'eggs')))
"(32.5, 40000, ('spam', 'eggs'))"
```

Here are two ways to write a table of squares and cubes:

```
>>> for x in range(1, 11):
...     print repr(x).rjust(2), repr(x*x).rjust(3),
...     # Note trailing comma on previous line
...     print repr(x*x*x).rjust(4)
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000

>>> for x in range(1,11):
...     print '{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x)
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

(Note that in the first example, one space between each column was added by the way `print` works: by default it adds spaces between its arguments.)

This example demonstrates the `str.rjust()` method of string objects, which right-justifies a string in a field of a given width by padding it with spaces on the left. There are similar methods `str.ljust()` and `str.center()`. These methods do not write anything, they just return a new string. If the input string is too long, they don't truncate it, but return it unchanged; this will mess up your column lay-out but that's usually better than the alternative, which would

be lying about a value. (If you really want truncation you can always add a slice operation, as in `x.ljust(n)[:n]`.)

Existe ainda o método `str.zfill()` que preenche uma string numérica com zeros à esquerda, e sabe lidar com sinais positivos e negativos:

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

Um uso básico do método `str.format()` tem esta forma:

```
>>> print 'We are the {} who say "{}!"'.format('knights', 'Ni')
We are the knights who say "Ni!"
```

The brackets and characters within them (called format fields) are replaced with the objects passed into the `str.format()` method. A number in the brackets refers to the position of the object passed into the `str.format()` method.

```
>>> print '{0} and {1}'.format('spam', 'eggs')
spam and eggs
>>> print '{1} and {0}'.format('spam', 'eggs')
eggs and spam
```

Se argumentos nomeados são passados para o método `str.format()`, seus valores serão referenciados usando o nome do argumento:

```
>>> print 'This {food} is {adjective}.'.format(
...       food='spam', adjective='absolutely horrible')
This spam is absolutely horrible.
```

Argumentos posicionais e nomeados podem ser combinados à vontade:

```
>>> print 'The story of {0}, {1}, and {other}.'.format('Bill', 'Manfred',
...                                                   other='Georg')
The story of Bill, Manfred, and Georg.
```

`'!s'` (apply `str()`) and `'!r'` (apply `repr()`) can be used to convert the value before it is formatted.

```
>>> import math
>>> print 'The value of PI is approximately {}.'.format(math.pi)
The value of PI is approximately 3.14159265359.
>>> print 'The value of PI is approximately {!r}.'.format(math.pi)
The value of PI is approximately 3.141592653589793.
```

An optional `':'` and format specifier can follow the field name. This allows greater control over how the value is formatted. The following example rounds Pi to three places after the decimal.

```
>>> import math
>>> print 'The value of PI is approximately {0:.3f}.'.format(math.pi)
The value of PI is approximately 3.142.
```

Passing an integer after the `':'` will cause that field to be a minimum number of characters wide. This is useful for making tables pretty.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print '{0:10} ==> {1:10d}'.format(name, phone)
...
Jack       ==>       4098
Dcab       ==>       7678
Sjoerd     ==>       4127
```

Se uma string de formatação é muito longa, e não se deseja quebrá-la, pode ser bom referir-se aos valores a serem formatados por nome, em vez de posição. Isto pode ser feito passando um dicionário usando colchetes `[]` para acessar as chaves:

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print ('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; '
...        'Dcab: {0[Dcab]:d}'.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

Isto também pode ser feito passando o dicionário como argumento do método, usando a notação **:

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print 'Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format(**table)
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

Isto é particularmente útil em conjunto com a função embutida `vars()`, que devolve um dicionário contendo todas as variáveis locais.

Para uma visão completa da formatação de strings com `str.format()`, veja a seção formatstrings.

7.1.1 Formatação de strings à moda antiga

O operador `%` também pode ser usado para formatação de strings. O argumento da esquerda é interpretado de forma semelhante ao estilo de formatação da função `sprintf()` da linguagem C, aplicando a formatação ao argumento da direita, e devolvendo a string resultante. Por exemplo:

```
>>> import math
>>> print 'The value of PI is approximately %5.3f.' % math.pi
The value of PI is approximately 3.142.
```

More information can be found in the string-formatting section.

7.2 Leitura e escrita de arquivos

`open()` returns a file object, and is most commonly used with two arguments: `open(filename, mode)`.

```
>>> f = open('workfile', 'w')
>>> print f
<open file 'workfile', mode 'w' at 80a0960>
```

O primeiro argumento é uma string contendo o nome do arquivo. O segundo argumento é outra string, contendo alguns caracteres que descrevem o modo como o arquivo será usado. *modo* pode ser `'r'` quando o arquivo será apenas lido, `'w'` para escrever (se o arquivo já existir seu conteúdo prévio será apagado), e `'a'` para abrir o arquivo para adição; qualquer escrita será adicionada ao final do arquivo. A opção `'r+'` abre o arquivo tanto para leitura como para escrita. O argumento *modo* é opcional, em caso de omissão será assumido `'r'`.

On Windows, 'b' appended to the mode opens the file in binary mode, so there are also modes like 'rb', 'wb', and 'r+b'. Python on Windows makes a distinction between text and binary files; the end-of-line characters in text files are automatically altered slightly when data is read or written. This behind-the-scenes modification to file data is fine for ASCII text files, but it'll corrupt binary data like that in JPEG or EXE files. Be very careful to use binary mode when reading and writing such files. On Unix, it doesn't hurt to append a 'b' to the mode, so you can use it platform-independently for all binary files.

7.2.1 Métodos de objetos arquivo

Para simplificar, o resto dos exemplos nesta seção assumem que um objeto arquivo chamado f já foi criado.

To read a file's contents, call f.read(size), which reads some quantity of data and returns it as a string. *size* is an optional numeric argument. When *size* is omitted or negative, the entire contents of the file will be read and returned; it's your problem if the file is twice as large as your machine's memory. Otherwise, at most *size* bytes are read and returned. If the end of the file has been reached, f.read() will return an empty string ("").

```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

f.readline() reads a single line from the file; a newline character (\n) is left at the end of the string, and is only omitted on the last line of the file if the file doesn't end in a newline. This makes the return value unambiguous; if f.readline() returns an empty string, the end of the file has been reached, while a blank line is represented by '\n', a string containing only a single newline.

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

Uma maneira alternativa de ler linhas do arquivo é iterar diretamente pelo objeto arquivo. É eficiente, rápido e resulta em código mais simples:

```
>>> for line in f:
        print line,

This is the first line of the file.
Second line of the file
```

Se desejar ler todas as linhas de um arquivo em uma lista, pode-se usar list(f) ou f.readlines().

f.write(string) writes the contents of *string* to the file, returning None.

```
>>> f.write('This is a test\n')
```

To write something other than a string, it needs to be converted to a string first:

```
>>> value = ('the answer', 42)
>>> s = str(value)
>>> f.write(s)
```

f.tell() returns an integer giving the file object's current position in the file, measured in bytes from the beginning of the file. To change the file object's position, use f.seek(offset, from_what). The position is computed from adding *offset* to a reference point; the reference point is selected by the *from_what* argument. A *from_what* value of 0

measures from the beginning of the file, 1 uses the current file position, and 2 uses the end of the file as the reference point. *from_what* can be omitted and defaults to 0, using the beginning of the file as the reference point.

```
>>> f = open('workfile', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5)      # Go to the 6th byte in the file
>>> f.read(1)
'5'
>>> f.seek(-3, 2)  # Go to the 3rd byte before the end
>>> f.read(1)
'd'
```

When you're done with a file, call `f.close()` to close it and free up any system resources taken up by the open file. After calling `f.close()`, attempts to use the file object will automatically fail.

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file
```

It is good practice to use the `with` keyword when dealing with file objects. This has the advantage that the file is properly closed after its suite finishes, even if an exception is raised on the way. It is also much shorter than writing equivalent `try-finally` blocks:

```
>>> with open('workfile', 'r') as f:
...     read_data = f.read()
>>> f.closed
True
```

Objetos arquivo tem alguns método adicionais, como `isatty()` e `truncate()` que não são usados com frequência; consulte a Biblioteca de Referência para um guia completo de objetos arquivo.

7.2.2 Gravando dados estruturados com `json`

Strings podem ser facilmente gravadas e lidas em um arquivo. Números dão um pouco mais de trabalho, já que o método `read()` só retorna strings, que terão que ser passadas para uma função como `int()`, que pega uma string como `'123'` e retorna seu valor numérico 123. Quando você deseja salvar tipos de dados mais complexos, como listas e dicionários aninhados, a análise e serialização manual tornam-se complicadas.

Ao invés de ter usuários constantemente escrevendo e depurando código para gravar tipos complicados de dados em arquivos, o Python permite que se use o popular formato de troca de dados chamado JSON (JavaScript Object Notation). O módulo padrão chamado `json` pode pegar hierarquias de dados em Python e convertê-las em representações de strings; esse processo é chamado *serialização*. Reconstruir os dados estruturados da representação string é chamado *desserialização*. Entre serializar e desserializar, a string que representa o objeto pode ser armazenada em um arquivo, ou estrutura de dados, ou enviada por uma conexão de rede para alguma outra máquina.

Nota: O formato JSON é comumente usado por aplicativos modernos para permitir troca de dados. Pessoas que programam já estão familiarizadas com esse formato, o que o torna uma boa opção para interoperabilidade.

Um objeto `x`, pode ser visualizado na sua representação JSON com uma simples linha de código:

```
>>> import json
>>> json.dumps([1, 'simple', 'list'])
'[1, "simple", "list"]'
```

Another variant of the `dumps()` function, called `dump()`, simply serializes the object to a file. So if `f` is a *file object* opened for writing, we can do this:

```
json.dump(x, f)
```

To decode the object again, if `f` is a *file object* which has been opened for reading:

```
x = json.load(f)
```

Essa técnica de serialização simples pode manipular listas e dicionários, mas a serialização de instâncias de classes arbitrárias no JSON requer um pouco mais de esforço. A referência para o módulo `json` contém uma explicação disso.

Ver também:

O módulo `pickle`

Ao contrário do *JSON*, *pickle* é um protocolo que permite a serialização de objetos Python arbitrariamente complexos. Por isso, é específico do Python e não pode ser usado para se comunicar com aplicativos escritos em outros linguagens. Também é inseguro por padrão: desserializar dados de pickle, provenientes de uma fonte não confiável, pode executar código arbitrário, se os dados foram criados por um invasor habilidoso.

CAPÍTULO 8

Erros e exceções

Até agora mensagens de erro foram apenas mencionadas, mas se você testou os exemplos, talvez tenha esbarrado em algumas. Existem pelo menos dois tipos distintos de erros: *erros de sintaxe* e *exceções*.

8.1 Erros de sintaxe

Erros de sintaxe, também conhecidos como erros de parse, são provavelmente os mais frequentes entre aqueles que ainda estão aprendendo Python:

```
>>> while True print 'Hello world'
  File "<stdin>", line 1
    while True print 'Hello world'
                   ^
SyntaxError: invalid syntax
```

The parser repeats the offending line and displays a little 'arrow' pointing at the earliest point in the line where the error was detected. The error is caused by (or at least detected at) the token *preceding* the arrow: in the example, the error is detected at the keyword `print`, since a colon (`':'`) is missing before it. File name and line number are printed so you know where to look in case the input came from a script.

8.2 Exceções

Mesmo que um comando ou expressão estejam sintaticamente corretos, talvez ocorra um erro na hora de sua execução. Erros detectados durante a execução são chamados *exceções* e não são necessariamente fatais: logo veremos como tratá-las em programas Python. A maioria das exceções não são tratadas pelos programas e acabam resultando em mensagens de erro:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

(continua na próxima página)

(continuação da página anterior)

```
ZeroDivisionError: integer division or modulo by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

A última linha da mensagem de erro indica o que aconteceu. Exceções surgem com diferentes tipos, e o tipo é exibido como parte da mensagem: os tipos no exemplo são `ZeroDivisionError`, `NameError` e `TypeError`. A string exibida como sendo o tipo da exceção é o nome da exceção embutida que ocorreu. Isso é verdade para todas exceções pré-definidas em Python, mas não é necessariamente verdade para exceções definidas pelo usuário (embora seja uma convenção útil). Os nomes das exceções padrões são identificadores embutidos (não palavras reservadas).

O resto da linha é um detalhamento que depende do tipo da exceção ocorrida e sua causa.

A parte anterior da mensagem de erro apresenta o contexto onde ocorreu a exceção. Essa informação é denominada *stack traceback* (situação da pilha de execução). Em geral, contém uma lista de linhas do código-fonte, sem apresentar, no entanto, linhas lidas da entrada padrão.

bltin-exceptions lista as exceções pré-definidas e seus significados.

8.3 Tratamento de exceções

É possível escrever programas que tratam exceções específicas. Observe o exemplo seguinte, que pede dados ao usuário até que um inteiro válido seja fornecido, ainda permitindo que o programa seja interrompido (utilizando `Control-C` ou seja lá o que for que o sistema operacional suporte); note que uma interrupção gerada pelo usuário será sinalizada pela exceção `KeyboardInterrupt`.

```
>>> while True:
...     try:
...         x = int(raw_input("Please enter a number: "))
...         break
...     except ValueError:
...         print "Oops!  That was no valid number.  Try again..."
...
```

A instrução `try` funciona da seguinte maneira:

- Primeiramente, a *cláusula try* (o conjunto de instruções entre as palavras reservadas `try` e `except`) é executada.
- Se nenhuma exceção ocorrer, a *cláusula except* é ignorada e a execução da instrução `try` é finalizada.
- Se ocorrer uma execução durante a execução da cláusula try, as instruções remanescentes na cláusula são ignoradas. Se o tipo da exceção ocorrida tiver sido previsto em algum `except`, então essa cláusula será executada. Depois disso, a execução continua após a instrução `try`.
- Se a exceção levantada não foi corresponder a nenhuma exceção listada na cláusula de exceção, então ela é entregue a uma instrução `try` mais externa. Se não existir nenhum tratador previsto para tal exceção, trata-se de uma *exceção não tratada* e a execução do programa termina com uma mensagem de erro.

A `try` statement may have more than one except clause, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding try clause, not in other handlers of the same `try` statement. An except clause may name multiple exceptions as a parenthesized tuple, for example:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

Note that the parentheses around this tuple are required, because `except ValueError, e:` was the syntax used for what is normally written as `except ValueError as e:` in modern Python (described below). The old syntax is still supported for backwards compatibility. This means `except RuntimeError, TypeError` is not equivalent to `except (RuntimeError, TypeError):` but to `except RuntimeError as TypeError:` which is not what you want.

A última cláusula de exceção pode omitir o nome da exceção, funcionando como um curinga. Utilize esse recurso com extrema cautela, uma vez que isso pode esconder erros do programador e do usuário! Também pode ser utilizado para exibir uma mensagem de erro e então levantar novamente a exceção (permitindo que o invocador da função atual também possa tratá-la):

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as e:
    print "I/O error({0}): {1}".format(e.errno, e.strerror)
except ValueError:
    print "Could not convert data to an integer."
except:
    print "Unexpected error:", sys.exc_info()[0]
    raise
```

A construção `try` ... `except` possui uma *cláusula else* opcional, que quando presente, deve ser colocada depois de todas as outras cláusulas. É útil para um código que precisa ser executado se nenhuma exceção foi levantada. Por exemplo:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print 'cannot open', arg
    else:
        print arg, 'has', len(f.readlines()), 'lines'
        f.close()
```

The use of the `else` clause is better than adding additional code to the `try` clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the `try` ... `except` statement.

Quando uma exceção ocorre, ela pode estar associada a um valor chamado *argumento* da exceção. A presença e o tipo do argumento dependem do tipo da exceção.

The except clause may specify a variable after the exception name (or tuple). The variable is bound to an exception instance with the arguments stored in `instance.args`. For convenience, the exception instance defines `__str__()` so the arguments can be printed directly without having to reference `.args`.

One may also instantiate an exception first before raising it and add any attributes to it as desired.

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print type(inst)     # the exception instance
...     print inst.args      # arguments stored in .args
...     print inst           # __str__ allows args to be printed directly
```

(continua na próxima página)

(continuação da página anterior)

```
...     x, y = inst.args
...     print 'x =', x
...     print 'y =', y
...
<type 'exceptions.Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

If an exception has an argument, it is printed as the last part ('detail') of the message for unhandled exceptions.

Além disso, tratadores de exceção são capazes de capturar exceções que tenham sido levantadas no interior de funções invocadas (mesmo que indiretamente) na cláusula try. Por exemplo:

```
>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError as detail:
...     print 'Handling run-time error:', detail
...
Handling run-time error: integer division or modulo by zero
```

8.4 Levantando exceções

A instrução `raise` permite ao programador forçar a ocorrência de um determinado tipo de exceção. Por exemplo:

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: HiThere
```

The sole argument to `raise` indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from `Exception`).

Caso você precise determinar se uma exceção foi levantada ou não, mas não quer manipular o erro, uma forma simples de instrução `raise` permite que você levante-a novamente:

```
>>> try:
...     raise NameError('HiThere')
... except NameError:
...     print 'An exception flew by!'
...     raise
...
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: HiThere
```

8.5 Exceções definidas pelo usuário

Programs may name their own exceptions by creating a new exception class (see *Classes* for more about Python classes). Exceptions should typically be derived from the `Exception` class, either directly or indirectly. For example:

```
>>> class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return repr(self.value)
...
>>> try:
...     raise MyError(2*2)
... except MyError as e:
...     print 'My exception occurred, value:', e.value
...
My exception occurred, value: 4
>>> raise MyError('oops!')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.MyError: 'oops!'
```

In this example, the default `__init__()` of `Exception` has been overridden. The new behavior simply creates the *value* attribute. This replaces the default behavior of creating the *args* attribute.

Classes de exceções podem ser definidas para fazer qualquer coisa que qualquer outra classe faz, mas em geral são bem simples, frequentemente oferecendo apenas alguns atributos que fornecem informações sobre o erro que ocorreu. Ao criar um módulo que pode gerar diversos erros, uma prática comum é criar uma classe base para as exceções definidas por aquele módulo, e as classes específicas para cada condição de erro como subclasses dela:

```
class Error(Exception):
    """Base class for exceptions in this module."""
    pass

class InputError(Error):
    """Exception raised for errors in the input.

    Attributes:
        expr -- input expression in which the error occurred
        msg  -- explanation of the error
    """

    def __init__(self, expr, msg):
        self.expr = expr
        self.msg = msg

class TransitionError(Error):
    """Raised when an operation attempts a state transition that's not
    allowed.

    Attributes:
        prev -- state at beginning of transition
        next -- attempted new state
        msg  -- explanation of why the specific transition is not allowed
    """

    def __init__(self, prev, next, msg):
```

(continua na próxima página)

(continuação da página anterior)

```
        self.prev = prev
        self.next = next
        self.msg = msg
```

É comum que novas exceções sejam definidas com nomes terminando em "Error", semelhante a muitas exceções embutidas.

Muitos módulos padrão definem novas exceções para reportar erros que ocorrem no interior das funções que definem. Mais informações sobre classes aparecem no capítulo *Classes*.

8.6 Definindo ações de limpeza

A instrução `try` possui outra cláusula opcional, cuja finalidade é permitir a implementação de ações de limpeza, que sempre devem ser executadas independentemente da ocorrência de exceções. Como no exemplo:

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print 'Goodbye, world!'
...
Goodbye, world!
KeyboardInterrupt
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
```

A *finally clause* is always executed before leaving the `try` statement, whether an exception has occurred or not. When an exception has occurred in the `try` clause and has not been handled by an `except` clause (or it has occurred in an `except` or `else` clause), it is re-raised after the `finally` clause has been executed. The `finally` clause is also executed "on the way out" when any other clause of the `try` statement is left via a `break`, `continue` or `return` statement. A more complicated example (having `except` and `finally` clauses in the same `try` statement works as of Python 2.5):

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print "division by zero!"
...     else:
...         print "result is", result
...     finally:
...         print "executing finally clause"
...
>>> divide(2, 1)
result is 2
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

As you can see, the `finally` clause is executed in any event. The `TypeError` raised by dividing two strings is not handled by the `except` clause and therefore re-raised after the `finally` clause has been executed.

Em aplicação do mundo real, a cláusula `finally` é útil para liberar recursos externos (como arquivos ou conexões de rede), independentemente do uso do recurso ter sido bem sucedido ou não.

8.7 Ações de limpeza predefinidas

Alguns objetos definem ações de limpeza padrões para serem executadas quando o objeto não é mais necessário, independentemente da operação que estava usando o objeto ter sido ou não bem sucedida. Veja o exemplo a seguir, que tenta abrir um arquivo e exibir seu conteúdo na tela.

```
for line in open("myfile.txt"):
    print line,
```

The problem with this code is that it leaves the file open for an indeterminate amount of time after the code has finished executing. This is not an issue in simple scripts, but can be a problem for larger applications. The `with` statement allows objects like files to be used in a way that ensures they are always cleaned up promptly and correctly.

```
with open("myfile.txt") as f:
    for line in f:
        print line,
```

After the statement is executed, the file *f* is always closed, even if a problem was encountered while processing the lines. Other objects which provide predefined clean-up actions will indicate this in their documentation.

CAPÍTULO 9

Classes

Em comparação com outras linguagens de programação, o mecanismo de classes de Python introduz a programação orientada a objetos sem acrescentar muitas novidades de sintaxe ou semântica. É uma mistura de mecanismos equivalentes encontrados em C++ e Modula-3. As classes em Python oferecem todas as características tradicionais da programação orientada a objetos: o mecanismo de herança permite múltiplas classes base (herança múltipla), uma classe derivada pode sobrescrever quaisquer métodos de uma classe ancestral, e um método pode invocar outro método homônimo de uma classe ancestral. Objetos podem armazenar uma quantidade arbitrária de dados de qualquer tipo. Assim como acontece com os módulos, as classes fazem parte da natureza dinâmica de Python: são criadas em tempo de execução, e podem ser alteradas após sua criação.

Usando a terminologia de C++, todos os membros de uma classe (incluindo dados) são públicos (veja exceção abaixo *Private Variables and Class-local References*), e todos as funções membro são "virtuais". Como em Modula-3, não existem atalhos para referenciar membros do objeto de dentro dos seus métodos: o método (função definida em uma classe) é declarado com um primeiro argumento explícito representando o objeto (instância da classe), que é fornecido implicitamente pela chamada ao método. Como em Smalltalk, classes são objetos. Isso fornece uma semântica para importar e renomear. Ao contrário de C++ ou Modula-3, tipos pré-definidos podem ser utilizados como classes base para extensões por herança pelo usuário. Também, como em C++, a maioria dos operadores (aritméticos, indexação, etc) podem ser redefinidos por instâncias de classe.

(Na falta de uma terminologia universalmente aceita para falar sobre classes, ocasionalmente farei uso de termos comuns em Smalltalk ou C++. Eu usaria termos de Modula-3, já que sua semântica de orientação a objetos é mais próxima da de Python, mas creio que poucos leitores já ouviram falar dessa linguagem.)

9.1 Uma palavra sobre nomes e objetos

Objetos têm individualidade, e vários nomes (em diferentes escopos) podem ser vinculados a um mesmo objeto. Isso é chamado de "aliasing" em outras linguagens. (N.d.T. *aliasing* é, literalmente, "apelidar": um mesmo objeto pode ter vários nomes.) Geralmente, esta característica não é muito apreciada, e pode ser ignorada com segurança ao lidar com tipos imutáveis (números, strings, tuplas). Entretanto, "aliasing" pode ter um efeito surpreendente na semântica do código Python envolvendo objetos mutáveis como listas, dicionários e a maioria dos outros tipos. Isso pode ser usado em benefício do programa, porque os apelidos funcionam de certa forma como ponteiros. Por exemplo, passar um objeto como argumento é barato, pois só um ponteiro é passado na implementação; e se uma função modifica um objeto passado

como argumento, o invocador verá a mudança — isso elimina a necessidade de ter dois mecanismos de passagem de parâmetros como em Pascal.

9.2 Escopos e espaços de nomes do Python

Antes de introduzir classes, é preciso falar das regras de escopo em Python. Definições de classe fazem alguns truques com espaços de nomes. Portanto, primeiro é preciso entender claramente como escopos e espaços de nomes funcionam, para entender o que está acontecendo. Esse conhecimento é muito útil para qualquer programador avançado em Python.

Vamos começar com algumas definições.

Um espaço de nomes é um mapeamento que associa nomes a objetos. Atualmente, são implementados como dicionários em Python, mas isso não é perceptível (a não ser pelo desempenho), e pode mudar no futuro. Exemplos de espaços de nomes são: o conjunto de nomes pré-definidos (funções como `abs()` e as exceções pré-definidas); nomes globais em um módulo; e nomes locais na invocação de uma função. De uma certa forma, os atributos de um objeto também formam um espaço de nomes. O mais importante é saber que não existe nenhuma relação entre nomes em "espaços de nomes" distintos. Por exemplo, dois módulos podem definir uma função de nome `maximize` sem confusão — usuários dos módulos devem prefixar a função com o nome do módulo, para evitar colisão.

A propósito, utilizo a palavra *atributo* para qualquer nome depois de um ponto. Na expressão `z.real`, por exemplo, `real` é um atributo do objeto `z`. Estritamente falando, referências para nomes em módulos são atributos: na expressão `modname.funcname`, `modname` é um objeto módulo e `funcname` é um de seus atributos. Neste caso, existe um mapeamento direto entre os atributos de um módulo e os nomes globais definidos no módulo: eles compartilham o mesmo espaço de nomes![1]

Atributos podem ser somente-leitura ou para leitura e escrita. No segundo caso, é possível atribuir um novo valor ao atributo. Atributos de módulos são passíveis de atribuição: você pode escrever `modname.the_answer = 42`. Atributos que aceitam escrita também podem ser apagados através da instrução `del`. Por exemplo, `del modname.the_answer` removerá o atributo `the_answer` do objeto referenciado por `modname`.

Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted. The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits. The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called `__main__`, so they have their own global namespace. (The built-in names actually also live in a module; this is called `__builtin__`.)

O espaço de nomes local de uma função é criado quando a função é invocada, e apagado quando a função retorna ou levanta uma exceção que não é tratada na própria função. (Na verdade, uma forma melhor de descrever o que realmente acontece é que o espaço de nomes local é "esquecido" quando a função termina.) Naturalmente, cada invocação recursiva de uma função tem seu próprio espaço de nomes.

Um *escopo* é uma região textual de um programa Python onde um espaço de nomes é diretamente acessível. Aqui, "diretamente acessível" significa que uma referência sem um prefixo qualificador permite o acesso ao nome.

Ainda que escopos sejam determinados estaticamente, eles são usados dinamicamente. A qualquer momento durante a execução, existem no mínimo três escopos diretamente acessíveis:

- o escopo mais interno, que é acessado primeiro, contendo nomes locais
- os escopos das funções que envolvem a função atual, que são acessados a partir do escopo mais próximo, contém nomes não-locais mas também não-globais
- o penúltimo escopo contém os nomes globais do módulo atual

[1] Exceto por uma coisa. Os objetos módulo têm um atributo secreto e somente para leitura chamado `__dict__` que retorna o dicionário usado para implementar o espaço de nomes do módulo; o nome `__dict__` é um atributo, mas não um nome global. Obviamente, usar isso viola a abstração da implementação do espaço de nomes, e deve ser restrito a coisas como depuradores post-mortem.

- e o escopo mais externo (acessado por último) contém os nomes das funções embutidas e demais objetos pré-definidos do interpretador

If a name is declared global, then all references and assignments go directly to the middle scope containing the module's global names. Otherwise, all variables found outside of the innermost scope are read-only (an attempt to write to such a variable will simply create a *new* local variable in the innermost scope, leaving the identically named outer variable unchanged).

Normalmente, o escopo local referencia os nomes locais da função corrente no texto do programa. Fora de funções, o escopo local referencia os nomes do escopo global: espaço de nomes do módulo. Definições de classes adicionam um outro espaço de nomes ao escopo local.

É importante perceber que escopos são determinados estaticamente, pelo texto do código-fonte: o escopo global de uma função definida em um módulo é o espaço de nomes deste módulo, sem importar de onde ou por qual apelido a função é invocada. Por outro lado, a busca de nomes é dinâmica, ocorrendo durante a execução. Porém, a evolução da linguagem está caminhando para uma resolução de nomes estática, em "tempo de compilação", portanto não conte com a resolução dinâmica de nomes! (De fato, variáveis locais já são resolvidas estaticamente.)

A special quirk of Python is that – if no `global` statement is in effect – assignments to names always go into the innermost scope. Assignments do not copy data — they just bind names to objects. The same is true for deletions: the statement `del x` removes the binding of `x` from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, `import` statements and function definitions bind the module or function name in the local scope. (The `global` statement can be used to indicate that particular variables live in the global scope.)

9.3 Uma primeira olhada nas classes

As classes introduzem um pouco de nova sintaxe, três novos tipos de objeto e algumas semânticas novas.

9.3.1 Sintaxe da definição de classe

A forma mais simples de definição de classe se parece com isto:

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

Definições de classe, assim como definições de função (instruções `def`), precisam ser executadas antes que tenham qualquer efeito. (Você pode colocar uma definição de classe dentro do teste condicional de um `if` ou dentro de uma função.)

Na prática, as instruções dentro da definição de classe geralmente serão definições de funções, mas outras instruções são permitidas, e às vezes são bem úteis — voltaremos a este tema depois. Definições de funções dentro da classe normalmente têm um forma peculiar de lista de argumentos, determinada pela convenção de chamada a métodos — isso também será explicado mais tarde.

Quando se inicia a definição de classe, um novo espaço de nomes é criado, e usado como escopo local — assim, todas atribuições a variáveis locais ocorrem nesse espaço de nomes. Em particular, funções definidas aqui são vinculadas a nomes nesse escopo.

Quando uma definição de classe é completado (normalmente, sem erros), um *objeto classe* é criado. Este objeto encapsula o conteúdo do espaço de nomes criado pela definição da classe; aprenderemos mais sobre objetos classe na próxima seção. O escopo local que estava vigente antes da definição da classe é reativado, e o objeto classe é vinculado ao identificador da classe nesse escopo (`ClassName` no exemplo).

9.3.2 Objetos classe

Objetos classe suportam dois tipos de operações: *referências a atributos* e *instanciação*.

Referências a atributos de classe utilizam a sintaxe padrão utilizada para quaisquer referências a atributos em Python: `obj.nome`. Nomes de atributos válidos são todos os nomes presentes dentro do espaço de nomes da classe, quando o objeto classe foi criado. Portanto, se a definição de classe tem esta forma:

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

então `MyClass.i` e `MyClass.f` são referências a atributo válidas, retornando, respectivamente, um inteiro e um objeto função. Atributos de classe podem receber valores, pode-se modificar o valor de `MyClass.i` num atribuição. `__doc__` também é um atributo válido da classe, retornando a *documentação* associada: `"A simple example class"`.

Para *instanciar* uma classe, usa-se a mesma sintaxe de invocar uma função. Apenas finja que o objeto classe do exemplo é uma função sem parâmetros, que devolve uma nova instância da classe. Por exemplo (assumindo a classe acima):

```
x = MyClass()
```

cria uma nova *instância* da classe e atribui o objeto resultante à variável local `x`.

A operação de instanciação ("invocar" um objeto classe) cria um objeto vazio. Muitas classes preferem criar novos objetos com um estado inicial predeterminado. Para tanto, a classe pode definir um método especial chamado `__init__()`, assim:

```
def __init__(self):
    self.data = []
```

Quando uma classe define um método `__init__()`, o processo de instanciação automaticamente invoca `__init__()` sobre a instância recém criada. Em nosso exemplo, uma nova instância já inicializada pode ser obtida desta maneira:

```
x = MyClass()
```

Naturalmente, o método `__init__()` pode ter parâmetros para maior flexibilidade. Neste caso, os argumentos fornecidos na invocação da classe serão passados para o método `__init__()`. Por exemplo,

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

9.3.3 Objetos instância

Now what can we do with instance objects? The only operations understood by instance objects are attribute references. There are two kinds of valid attribute names, data attributes and methods.

Atributos de dados correspondem a "variáveis de instância" em Smalltalk, e a "membros de dados" em C++. Atributos de dados não precisam ser declarados. Assim como variáveis locais, eles passam a existir na primeira vez em que é feita uma atribuição. Por exemplo, se `x` é uma instância da `MyClass` criada acima, o próximo trecho de código irá exibir o valor `16`, sem deixar nenhum rastro:

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print x.counter
del x.counter
```

O outro tipo de referências a atributos de instância é o "método". Um método é uma função que "pertence" a um objeto instância. (Em Python, o termo método não é aplicado exclusivamente a instâncias de classes definidas pelo usuário: outros tipos de objetos também podem ter métodos. Por exemplo, listas possuem os métodos append, insert, remove, sort, entre outros. Porém, na discussão a seguir, usaremos o termo método apenas para se referir a métodos de classes definidas pelo usuário. Seremos explícitos ao falar de outros métodos.)

Nomes de métodos válidos de uma instância dependem de sua classe. Por definição, cada atributo de uma classe que é uma função corresponde a um método das instâncias. Em nosso exemplo, `x.f` é uma referência de método válida já que `MyClass.f` é uma função, enquanto `x.i` não é, já que `MyClass.i` não é uma função. Entretanto, `x.f` não é o mesmo que `MyClass.f`. A referência `x.f` acessa um objeto método e a `MyClass.f` acessa um objeto função.

9.3.4 Objetos método

Normalmente, um método é chamado imediatamente após ser referenciado:

```
x.f()
```

No exemplo `MyClass` o resultado da expressão acima será a string `'hello world'`. No entanto, não é obrigatório invocar o método imediatamente: como `x.f` é também um objeto ele pode ser atribuído a uma variável e invocado depois. Por exemplo:

```
xf = x.f
while True:
    print xf()
```

exibirá o texto `hello world` até o mundo acabar.

O que ocorre precisamente quando um método é invocado? Você deve ter notado que `x.f()` foi chamado sem nenhum argumento, porém a definição da função `f()` especificava um argumento. O que aconteceu com esse argumento? Certamente Python levanta uma exceção quando uma função que declara um argumento é invocada sem nenhum argumento — mesmo que o argumento não seja usado no corpo da função…

Actually, you may have guessed the answer: the special thing about methods is that the object is passed as the first argument of the function. In our example, the call `x.f()` is exactly equivalent to `MyClass.f(x)`. In general, calling a method with a list of *n* arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method's object before the first argument.

Se você ainda não entende como os métodos funcionam, dê uma olhada na implementação para esclarecer as coisas. Quando um atributo de uma instância, não relacionado a dados, é referenciado, a classe da instância é pesquisada. Se o nome é um atributo de classe válido, e é o nome de uma função, um método é criado, empacotando a instância e a função, que estão juntos num objeto abstrato: este é o método. Quando o método é invocado com uma lista de argumentos, uma

nova lista de argumentos é criada inserindo a instância na posição 0 da lista. Finalmente, o objeto função — empacotado dentro do objeto método — é invocado com a nova lista de argumentos.

9.3.5 Variáveis de classe e instância

De forma geral, variáveis de instância são variáveis que indicam dados que são únicos a cada instância individual, e variáveis de classe são variáveis de atributos e de métodos que são comuns a todas as instâncias de uma classe:

```
class Dog:

    kind = 'canine'         # class variable shared by all instances

    def __init__(self, name):
        self.name = name    # instance variable unique to each instance

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                  # shared by all dogs
'canine'
>>> e.kind                  # shared by all dogs
'canine'
>>> d.name                  # unique to d
'Fido'
>>> e.name                  # unique to e
'Buddy'
```

Como vimos em *Uma palavra sobre nomes e objetos*, dados compartilhados podem causar efeitos inesperados quando envolvem objetos mutáveis (*mutable*), como listas ou dicionários. Por exemplo, a lista *tricks* do código abaixo não deve ser usada como variável de classe, pois assim seria compartilhada por todas as instâncias de *Dog*:

```
class Dog:

    tricks = []             # mistaken use of a class variable

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks                # unexpectedly shared by all dogs
['roll over', 'play dead']
```

Em vez disso, o modelo correto da classe deve usar uma variável de instância:

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each dog

    def add_trick(self, trick):
```

(continua na próxima página)

(continuação da página anterior)

```
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```

9.4 Observações aleatórias

Data attributes override method attributes with the same name; to avoid accidental name conflicts, which may cause hard-to-find bugs in large programs, it is wise to use some kind of convention that minimizes the chance of conflicts. Possible conventions include capitalizing method names, prefixing data attribute names with a small unique string (perhaps just an underscore), or using verbs for methods and nouns for data attributes.

Atributos de dados podem ser referenciados por métodos da própria instância, bem como por qualquer outro usuário do objeto (também chamados "clientes" do objeto). Em outras palavras, classes não servem para implementar tipos puramente abstratos de dados. De fato, nada em Python torna possível assegurar o encapsulamento de dados — tudo é baseado em convenção. (Por outro lado, a implementação de Python, escrita em C, pode esconder completamente detalhes de um objeto e controlar o acesso ao objeto, se necessário; isto pode ser utilizado por extensões de Python escritas em C.)

Clientes devem utilizar atributos de dados com cuidado, pois podem bagunçar invariantes assumidas pelos métodos ao esbarrar em seus atributos de dados. Note que clientes podem adicionar atributos de dados a suas próprias instâncias, sem afetar a validade dos métodos, desde que seja evitado o conflito de nomes. Novamente, uma convenção de nomenclatura poupa muita dor de cabeça.

Não existe atalho para referenciar atributos de dados (ou outros métodos!) de dentro de um método. Isso aumenta a legibilidade dos métodos: não há como confundir variáveis locais com variáveis da instância quando lemos rapidamente um método.

Frequentemente, o primeiro argumento de um método é chamado `self`. Isso não passa de uma convenção: o identificador `self` não é uma palavra reservada nem possui qualquer significado especial em Python. Mas note que, ao seguir essa convenção, seu código se torna legível por uma grande comunidade de desenvolvedores Python e é possível que alguma *IDE* dependa dessa convenção para analisar seu código.

Qualquer objeto função que é atributo de uma classe, define um método para as instâncias dessa classe. Não é necessário que a definição da função esteja textualmente embutida na definição da classe. Atribuir um objeto função a uma variável local da classe é válido. Por exemplo:

```
# Function defined outside the class
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1

    def g(self):
        return 'hello world'

    h = g
```

Agora `f`, `g` e `h` são todos atributos da classe `C` que referenciam funções, e consequentemente são todos métodos de instâncias da classe `C`, onde `h` é exatamente equivalente a `g`. No entanto, essa prática serve apenas para confundir o leitor do programa.

Métodos podem invocar outros métodos usando atributos de método do argumento `self`:

```
class Bag:
    def __init__(self):
        self.data = []

    def add(self, x):
        self.data.append(x)

    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

Métodos podem referenciar nomes globais da mesma forma que funções comuns. O escopo global associado a um método é o módulo contendo sua definição na classe (a classe propriamente dita nunca é usada como escopo global!). Ainda que seja raro justificar o uso de dados globais em um método, há diversos usos legítimos do escopo global. Por exemplo, funções e módulos importados no escopo global podem ser usados por métodos, bem como as funções e classes definidas no próprio escopo global. Provavelmente, a classe contendo o método em questão também foi definida neste escopo global. Na próxima seção veremos razões pelas quais um método pode querer referenciar sua própria classe.

Cada valor é um objeto e, portanto, tem uma *classe* (também chamada de *tipo*). Ela é armazenada como `object.__class__`.

9.5 Herança

Obviamente, uma característica não seria digna do nome "classe" se não suportasse herança. A sintaxe para uma classe derivada é assim:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

O identificador `BaseClassName` deve estar definido no escopo que contém a definição da classe derivada. No lugar do nome da classe base, também são aceitas outras expressões. Isso é muito útil, por exemplo, quando a classe base é definida em outro módulo:

```
class DerivedClassName(modname.BaseClassName):
```

A execução de uma definição de classe derivada procede da mesma forma que a de uma classe base. Quando o objeto classe é construído, a classe base é lembrada. Isso é utilizado para resolver referências a atributos. Se um atributo requisitado não for encontrado na classe, ele é procurado na classe base. Essa regra é aplicada recursivamente se a classe base por sua vez for derivada de outra.

Não há nada de especial sobre instanciação de classes derivadas: `DerivedClassName()` cria uma nova instância da classe. Referências a métodos são resolvidas da seguinte forma: o atributo correspondente é procurado através da cadeia de classes base, e referências a métodos são válidas se essa procura produzir um objeto função.

Classes derivadas podem sobrescrever métodos das suas classes base. Uma vez que métodos não possuem privilégios especiais quando invocam outros métodos no mesmo objeto, um método na classe base que invoca um outro método

da mesma classe base pode, efetivamente, acabar invocando um método sobreposto por uma classe derivada. (Para programadores C++ isso significa que todos os métodos em Python são realmente virtuais.)

Um método sobrescrito em uma classe derivada, de fato, pode querer estender, em vez de simplesmente substituir, o método da classe base, de mesmo nome. Existe uma maneira simples de chamar diretamente o método da classe base: apenas chame `BaseClassName.methodname(self, arguments)`. Isso é geralmente útil para os clientes também. (Note que isto só funciona se a classe base estiver acessível como `BaseClassName` no escopo global).

Python tem duas funções embutidas que trabalham com herança:

- Use `isinstance()` para verificar o tipo de uma instância: `isinstance(obj, int)` será `True` somente se `obj.__class__` é a classe `int` ou alguma classe derivada de `int`.
- Use `issubclass()` to check class inheritance: `issubclass(bool, int)` is `True` since `bool` is a subclass of `int`. However, `issubclass(unicode, str)` is `False` since `unicode` is not a subclass of `str` (they only share a common ancestor, `basestring`).

9.5.1 Herança múltipla

Python supports a limited form of multiple inheritance as well. A class definition with multiple base classes looks like this:

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

For old-style classes, the only rule is depth-first, left-to-right. Thus, if an attribute is not found in `DerivedClassName`, it is searched in `Base1`, then (recursively) in the base classes of `Base1`, and only if it is not found there, it is searched in `Base2`, and so on.

(To some people breadth first — searching `Base2` and `Base3` before the base classes of `Base1` — looks more natural. However, this would require you to know whether a particular attribute of `Base1` is actually defined in `Base1` or in one of its base classes before you can figure out the consequences of a name conflict with an attribute of `Base2`. The depth-first rule makes no differences between direct and inherited attributes of `Base1`.)

For *new-style class*es, the method resolution order changes dynamically to support cooperative calls to `super()`. This approach is known in some other multiple-inheritance languages as call-next-method and is more powerful than the super call found in single-inheritance languages.

With new-style classes, dynamic ordering is necessary because all cases of multiple inheritance exhibit one or more diamond relationships (where at least one of the parent classes can be accessed through multiple paths from the bottommost class). For example, all new-style classes inherit from `object`, so any case of multiple inheritance provides more than one path to reach `object`. To keep the base classes from being accessed more than once, the dynamic algorithm linearizes the search order in a way that preserves the left-to-right ordering specified in each class, that calls each parent only once, and that is monotonic (meaning that a class can be subclassed without affecting the precedence order of its parents). Taken together, these properties make it possible to design reliable and extensible classes with multiple inheritance. For more detail, see https://www.python.org/download/releases/2.3/mro/.

9.6 Private Variables and Class-local References

Variáveis de instância "privadas", que não podem ser acessadas, exceto em métodos do próprio objeto, não existem em Python. No entanto, existe uma convenção que é seguida pela maioria dos programas em Python: um nome prefixado com um sublinhado (por exemplo: `_spam`) deve ser tratado como uma parte não-pública da API (seja uma função, um método ou um atributo de dados). Tais nomes devem ser considerados um detalhe de implementação e sujeito a alteração sem aviso prévio.

Uma vez que existe um caso de uso válido para a definição de atributos privados em classes (especificamente para evitar conflitos com nomes definidos em subclasses), existe um suporte limitado a identificadores privados em classes, chamado *name mangling*, desfiguração de nomes. Qualquer identificador no formato `__spam` (pelo menos dois sublinhados no início, e no máximo um sublinhado no final) é textualmente substituído por `_classname__spam`, onde `classname` é o nome da classe atual com sublinhado(s) iniciais omitidos. Essa desfiguração independe da posição sintática do identificador, desde que ele apareça dentro da definição de uma classe.

A desfiguração de nomes é útil para que subclasses possam sobrescrever métodos sem quebrar invocações de métodos dentro de outra classe. Por exemplo:

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update   # private copy of original update() method

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # provides new signature for update()
        # but does not break __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

O exemplo acima deve funcionar mesmo se `MappingSubclass` introduzisse um identificador `__update` uma vez que é substituído por `_Mapping__update` na classe `Mapping` e `_MappingSubclass__update` na classe `MappingSubclass`, respectivamente.

Note que as regras de desfiguração de nomes foram projetadas para evitar acidentes; ainda é possível acessar ou modificar uma variável que é considerada privada. Isso pode ser útil em certas circunstâncias especiais, como depuração de código.

Notice that code passed to `exec`, `eval()` or `execfile()` does not consider the classname of the invoking class to be the current class; this is similar to the effect of the `global` statement, the effect of which is likewise restricted to code that is byte-compiled together. The same restriction applies to `getattr()`, `setattr()` and `delattr()`, as well as when referencing `__dict__` directly.

9.7 Curiosidades e conclusões

Às vezes, é útil ter um tipo semelhante ao "record" de Pascal ou ao "struct" de C, para agrupar alguns itens de dados. Uma definição de classe vazia funciona bem para este fim:

```
class Employee:
    pass

john = Employee()  # Create an empty employee record

# Fill the fields of the record
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

Um trecho de código Python que espera um tipo de dado abstrato em particular, pode receber, ao invés disso, uma classe que imita os métodos que aquele tipo suporta. Por exemplo, se você tem uma função que formata dados obtidos de um objeto do tipo "arquivo", pode definir, e passar como argumento, uma classe com métodos `read()` e `readline()` que obtém os dados de um "buffer de caracteres".

Instance method objects have attributes, too: `m.im_self` is the instance object with the method `m()`, and `m.im_func` is the function object corresponding to the method.

9.8 Exceptions Are Classes Too

User-defined exceptions are identified by classes as well. Using this mechanism it is possible to create extensible hierarchies of exceptions.

There are two new valid (semantic) forms for the `raise` statement:

```
raise Class, instance

raise instance
```

In the first form, `instance` must be an instance of `Class` or of a class derived from it. The second form is a shorthand for:

```
raise instance.__class__, instance
```

A class in an `except` clause is compatible with an exception if it is the same class or a base class thereof (but not the other way around — an except clause listing a derived class is not compatible with a base class). For example, the following code will print B, C, D in that order:

```
class B:
    pass
class C(B):
    pass
class D(C):
    pass

for c in [B, C, D]:
    try:
        raise c()
    except D:
```

(continua na próxima página)

(continuação da página anterior)

```
        print "D"
    except C:
        print "C"
    except B:
        print "B"
```

Note that if the except clauses were reversed (with `except B` first), it would have printed B, B, B — the first matching except clause is triggered.

When an error message is printed for an unhandled exception, the exception's class name is printed, then a colon and a space, and finally the instance converted to a string using the built-in function `str()`.

9.9 Iteradores

Você já deve ter notado que pode usar laços `for` com a maioria das coleções em Python:

```
for element in [1, 2, 3]:
    print element
for element in (1, 2, 3):
    print element
for key in {'one':1, 'two':2}:
    print key
for char in "123":
    print char
for line in open("myfile.txt"):
    print line,
```

This style of access is clear, concise, and convenient. The use of iterators pervades and unifies Python. Behind the scenes, the `for` statement calls `iter()` on the container object. The function returns an iterator object that defines the method `next()` which accesses elements in the container one at a time. When there are no more elements, `next()` raises a `StopIteration` exception which tells the `for` loop to terminate. This example shows how it all works:

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> it.next()
'a'
>>> it.next()
'b'
>>> it.next()
'c'
>>> it.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    it.next()
StopIteration
```

Having seen the mechanics behind the iterator protocol, it is easy to add iterator behavior to your classes. Define an `__iter__()` method which returns an object with a `next()` method. If the class defines `next()`, then `__iter__()` can just return `self`:

```
class Reverse:
    """Iterator for looping over a sequence backwards."""
```

(continua na próxima página)

(continuação da página anterior)

```
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def next(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

```
>>> rev = Reverse('spam')
>>> iter(rev)
<__main__.Reverse object at 0x00A1DB50>
>>> for char in rev:
...     print char
...
m
a
p
s
```

9.10 Geradores

*Generator*s – geradores – são uma ferramenta simples e poderosa para criar iteradores. São escritos como funções normais mas usam a instrução `yield` quando retornam dados. Cada vez que `next()` é chamado, o gerador volta ao ponto onde parou (lembrando todos os valores de dados e qual instrução foi executada pela última vez). Um exemplo mostra como geradores podem ser trivialmente fáceis de criar:

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]
```

```
>>> for char in reverse('golf'):
...     print char
...
f
l
o
g
```

Anything that can be done with generators can also be done with class-based iterators as described in the previous section. What makes generators so compact is that the `__iter__()` and `next()` methods are created automatically.

Outro ponto chave é que as variáveis locais e o estado da execução são preservados automaticamente entre as chamadas. Isto torna a função mais fácil de escrever e muito mais clara do que uma implementação usando variáveis de instância como `self.index` e `self.data`.

Além disso, quando geradores terminam, eles levantam `StopIteration` automaticamente. Combinados, todos estes aspectos tornam a criação de iteradores tão fácil quanto escrever uma função normal.

9.11 Expressões geradoras

Alguns geradores simples podem ser codificados, de forma sucinta, como expressões, usando uma sintaxe semelhante a compreensões de lista, mas com parênteses em vez de colchetes. Essas expressões são projetadas para situações em que o gerador é usado imediatamente, pela função que o engloba. As expressões geradoras são mais compactas, mas menos versáteis do que as definições completas do gerador, e tendem a usar menos memória do que as compreensões de lista equivalentes.

Exemplos:

```
>>> sum(i*i for i in range(10))                 # sum of squares
285

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))         # dot product
260

>>> from math import pi, sin
>>> sine_table = dict((x, sin(x*pi/180)) for x in range(0, 91))

>>> unique_words = set(word  for line in page  for word in line.split())

>>> valedictorian = max((student.gpa, student.name) for student in graduates)

>>> data = 'golf'
>>> list(data[i] for i in range(len(data)-1,-1,-1))
['f', 'l', 'o', 'g']
```

CAPÍTULO 10

Um breve passeio pela biblioteca padrão

10.1 Interface com o sistema operacional

O módulo `os` fornece dúzias de funções para interagir com o sistema operacional:

```
>>> import os
>>> os.getcwd()      # Return the current working directory
'C:\\Python26'
>>> os.chdir('/server/accesslogs')   # Change current working directory
>>> os.system('mkdir today')   # Run the command mkdir in the system shell
0
```

Certifique-se de usar a forma `import os` ao invés de `from os import *`. Isso evitará que `os.open()` oculte a função `open()` que opera de forma muito diferente.

As funções embutidas `dir()` e `help()` são úteis como um sistema de ajuda interativa para lidar com módulos grandes como `os`:

```
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's docstrings>
```

Para tarefas de gerenciamento cotidiano de arquivos e diretórios, o módulo `shutil` fornece uma interface de alto nível que é mais simples de usar:

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
>>> shutil.move('/build/executables', 'installdir')
```

10.2 Caracteres curinga

O módulo `glob` fornece uma função para criar listas de arquivos a partir de buscas em diretórios usando caracteres curinga:

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

10.3 Argumentos de linha de comando

Scripts geralmente precisam processar argumentos passados na linha de comando. Esses argumentos são armazenados como uma lista no atributo *argv* do módulo `sys`. Por exemplo, teríamos a seguinte saída executando `python demo.py one two three` na linha de comando:

```
>>> import sys
>>> print sys.argv
['demo.py', 'one', 'two', 'three']
```

The `getopt` module processes *sys.argv* using the conventions of the Unix `getopt()` function. More powerful and flexible command line processing is provided by the `argparse` module.

10.4 Redirecionamento de erros e encerramento do programa

O módulo `sys` também possui atributos para *stdin*, *stdout* e *stderr*. O último é usado para emitir avisos e mensagens de erros visíveis mesmo quando *stdout* foi redirecionado:

```
>>> sys.stderr.write('Warning, log file not found starting a new one\n')
Warning, log file not found starting a new one
```

A forma mais direta de encerrar um script é usando `sys.exit()`.

10.5 Reconhecimento de padrões em strings

O módulo `re` fornece ferramentas para lidar com processamento de strings através de expressões regulares. Para reconhecimento de padrões complexos, expressões regulares oferecem uma solução sucinta e eficiente:

```
>>> import re
>>> re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

Quando as exigências são simples, métodos de strings são preferíveis por serem mais fáceis de ler e depurar:

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

10.6 Matemática

O módulo `math` oferece acesso às funções da biblioteca C para matemática de ponto flutuante:

```
>>> import math
>>> math.cos(math.pi / 4.0)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

O módulo `random` fornece ferramentas para gerar seleções aleatórias:

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(xrange(100), 10)   # sampling without replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()    # random float
0.17970987693706186
>>> random.randrange(6)    # random integer chosen from range(6)
4
```

10.7 Acesso à internet

There are a number of modules for accessing the internet and processing internet protocols. Two of the simplest are `urllib2` for retrieving data from URLs and `smtplib` for sending mail:

```
>>> import urllib2
>>> for line in urllib2.urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl'):
...     if 'EST' in line or 'EDT' in line:  # look for Eastern Time
...         print line

<BR>Nov. 25, 09:43:32 PM EST

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
... """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Beware the Ides of March.
... """)
>>> server.quit()
```

(Note que o segundo exemplo precisa de um servidor de email rodando em localhost.)

10.8 Data e hora

O módulo `datetime` fornece classes para manipulação de datas e horas nas mais variadas formas. Apesar da disponibilidade de aritmética com data e hora, o foco da implementação é na extração eficiente dos membros para formatação e manipulação. O módulo também oferece objetos que levam os fusos horários em consideração.

```
>>> # dates are easily constructed and formatted
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'

>>> # dates support calendar arithmetic
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

10.9 Compressão de dados

Common data archiving and compression formats are directly supported by modules including: `zlib`, `gzip`, `bz2`, `zipfile` and `tarfile`.

```
>>> import zlib
>>> s = 'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```

10.10 Medição de desempenho

Alguns usuários de Python desenvolvem um interesse profundo pelo desempenho relativo de diferentes abordagens para o mesmo problema. Python oferece uma ferramenta de medição que esclarece essas dúvidas rapidamente.

Por exemplo, pode ser tentador usar o empacotamento e desempacotamento de tuplas ao invés da abordagem tradicional de permutar os argumentos. O módulo `timeit` rapidamente mostra uma modesta vantagem de desempenho:

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

Em contraste com a granulidade fina do módulo `timeit`, os módulos `profile` e `pstats` oferecem ferramentas para identificar os trechos mais críticos em grandes blocos de código.

10.11 Controle de qualidade

Uma das abordagens usadas no desenvolvimento de software de alta qualidade é escrever testes para cada função à medida que é desenvolvida e executar esses testes frequentemente durante o processo de desenvolvimento.

O módulo `doctest` oferece uma ferramenta para realizar um trabalho de varredura e validação de testes escritos nas strings de documentação (docstrings) de um programa. A construção dos testes é tão simples quanto copiar uma chamada típica juntamente com seus resultados e colá-los na docstring. Isto aprimora a documentação, fornecendo ao usuário um exemplo real, e permite que o módulo doctest verifique se o código continua fiel à documentação:

```
def average(values):
    """Computes the arithmetic mean of a list of numbers.

    >>> print average([20, 30, 70])
    40.0
    """
    return sum(values, 0.0) / len(values)

import doctest
doctest.testmod()   # automatically validate the embedded tests
```

O módulo `unittest` não é tão simples de usar quanto o módulo `doctest`, mas permite que um conjunto muito maior de testes seja mantido em um arquivo separado:

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        with self.assertRaises(ZeroDivisionError):
            average([])
        with self.assertRaises(TypeError):
            average(20, 30, 70)

unittest.main()  # Calling from the command line invokes all tests
```

10.12 Baterias incluídas

Python tem uma filosofia de "baterias incluídas". Isso fica mais evidente através da sofisticação e robustez dos seus maiores pacotes. Por exemplo:

- The `xmlrpclib` and `SimpleXMLRPCServer` modules make implementing remote procedure calls into an almost trivial task. Despite the modules names, no direct knowledge or handling of XML is needed.
- The `email` package is a library for managing email messages, including MIME and other RFC 2822-based message documents. Unlike `smtplib` and `poplib` which actually send and receive messages, the email package has a complete toolset for building or decoding complex message structures (including attachments) and for implementing internet encoding and header protocols.

- The `xml.dom` and `xml.sax` packages provide robust support for parsing this popular data interchange format. Likewise, the `csv` module supports direct reads and writes in a common database format. Together, these modules and packages greatly simplify data interchange between Python applications and other tools.
- Internacionalização está disponível através de diversos módulos, como `gettext`, `locale`, e o pacote `codecs`.

CAPÍTULO 11

Um breve passeio pela biblioteca padrão — parte II

Este segundo passeio apresenta alguns módulos avançados que atendem necessidades de programação profissional. Estes módulos raramente aparecem em scripts pequenos.

11.1 Formatando a saída

The `repr` module provides a version of `repr()` customized for abbreviated displays of large or deeply nested containers:

```
>>> import repr
>>> repr.repr(set('supercalifragilisticexpialidocious'))
"set(['a', 'c', 'd', 'e', 'f', 'g', ...])"
```

O módulo `pprint` oferece um controle mais sofisticado na exibição tanto de objetos embutidos quanto aqueles criados pelo usuário de maneira que fique legível para o interpretador. Quando o resultado é maior que uma linha, o "pretty printer" acrescenta quebras de linha e indentação para revelar as estruturas de maneira mais clara:

```
>>> import pprint
>>> t = [[[['black', 'cyan'], 'white', ['green', 'red']], [['magenta',
...     'yellow'], 'blue']]]
...
>>> pprint.pprint(t, width=30)
[[[['black', 'cyan'],
   'white',
   ['green', 'red']],
  [['magenta', 'yellow'],
   'blue']]]
```

O módulo `textwrap` formata parágrafos de texto para que caibam em uma dada largura de tela:

```
>>> import textwrap
>>> doc = """The wrap() method is just like fill() except that it returns
... a list of strings instead of one big string with newlines to separate
... the wrapped lines."""
```

(continua na próxima página)

(continuação da página anterior)

```
...
>>> print textwrap.fill(doc, width=40)
The wrap() method is just like fill()
except that it returns a list of strings
instead of one big string with newlines
to separate the wrapped lines.
```

O módulo `locale` acessa uma base de dados de formatos específicos a determinada cultura. O atributo de agrupamento da função "format" oferece uma forma direta de formatar números com separadores de grupo:

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'English_United States.1252')
'English_United States.1252'
>>> conv = locale.localeconv()          # get a mapping of conventions
>>> x = 1234567.8
>>> locale.format("%d", x, grouping=True)
'1,234,567'
>>> locale.format_string("%s%.*f", (conv['currency_symbol'],
...                      conv['frac_digits'], x), grouping=True)
'$1,234,567.80'
```

11.2 Usando templates

módulo `string` inclui a versátil classe `Template` com uma sintaxe simplificada, adequada para ser editada por usuários finais. Isso permite que usuários personalizem suas aplicações sem a necessidade de alterar a aplicação.

Em um template são colocadas marcações indicando o local onde o texto variável deve ser inserido. Uma marcação é formada por `$` seguido de um identificador Python válido (caracteres alfanuméricos e underscores). Envolvendo-se o identificador da marcação entre chaves, permite que ele seja seguido por mais caracteres alfanuméricos sem a necessidade de espaços. Escrevendo-se `$$` cria-se um único `$`:

```
>>> from string import Template
>>> t = Template('${village}folk send $$10 to $cause.')
>>> t.substitute(village='Nottingham', cause='the ditch fund')
'Nottinghamfolk send $10 to the ditch fund.'
```

O método `substitute()` levanta uma exceção `KeyError` quando o identificador de uma marcação não é fornecido em um dicionário ou em um argumento nomeado (*keyword argument*). Para aplicações que podem receber dados incompletos fornecidos pelo usuário, o método `safe_substitute()` pode ser mais apropriado — deixará os marcadores intactos se os dados estiverem faltando:

```
>>> t = Template('Return the $item to $owner.')
>>> d = dict(item='unladen swallow')
>>> t.substitute(d)
Traceback (most recent call last):
  ...
KeyError: 'owner'
>>> t.safe_substitute(d)
'Return the unladen swallow to $owner.'
```

Subclasses de Template podem especificar um delimitador personalizado. Por exemplo, um utilitário para renomeação em lote de fotos pode usar o sinal de porcentagem para marcações como a data atual, número sequencial da imagem ou formato do aquivo:

```
>>> import time, os.path
>>> photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class BatchRename(Template):
...     delimiter = '%'
>>> fmt = raw_input('Enter rename style (%d-date %n-seqnum %f-format):  ')
Enter rename style (%d-date %n-seqnum %f-format):  Ashley_%n%f

>>> t = BatchRename(fmt)
>>> date = time.strftime('%d%b%y')
>>> for i, filename in enumerate(photofiles):
...     base, ext = os.path.splitext(filename)
...     newname = t.substitute(d=date, n=i, f=ext)
...     print '{0} --> {1}'.format(filename, newname)

img_1074.jpg --> Ashley_0.jpg
img_1076.jpg --> Ashley_1.jpg
img_1077.jpg --> Ashley_2.jpg
```

Outra aplicação para templates é separar a lógica da aplicação dos detalhes de múltiplos formatos de saída. Assim é possível usar templates personalizados para gerar arquivos XML, relatórios em texto puro e relatórios web em HTML.

11.3 Trabalhando com formatos binários de dados

O módulo `struct` oferece as funções `pack()` e `unpack()` para trabalhar com registros binários de tamanho variável. O exemplo a seguir mostra como iterar através do cabeçalho de informação num aquivo ZIP sem usar o módulo `zipfile`. Os códigos de empacotamento `"H"` e `"I"` representam números sem sinal de dois e quatro bytes respectivamente. O `"<"` indica que os números têm tamanho padrão e são little-endian (bytes menos significativos primeiro):

```
import struct

data = open('myfile.zip', 'rb').read()
start = 0
for i in range(3):                      # show the first 3 file headers
    start += 14
    fields = struct.unpack('<IIIHH', data[start:start+16])
    crc32, comp_size, uncomp_size, filenamesize, extra_size = fields

    start += 16
    filename = data[start:start+filenamesize]
    start += filenamesize
    extra = data[start:start+extra_size]
    print filename, hex(crc32), comp_size, uncomp_size

    start += extra_size + comp_size     # skip to the next header
```

11.4 Multi-threading

O uso de threads é uma técnica para desacoplar tarefas que não são sequencialmente dependentes. Threads podem ser usadas para melhorar o tempo de resposta de aplicações que aceitam entradas do usuário enquanto outras tarefas são executadas em segundo plano. Um caso relacionado é executar ações de entrada e saída (I/O) em uma thread paralelamente a cálculos em outra thread.

O código a seguir mostra como o módulo de alto nível `threading` pode executar tarefas em segundo plano enquanto o programa principal continua a sua execução:

```
import threading, zipfile

class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile

    def run(self):
        f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.infile)
        f.close()
        print 'Finished background zip of: ', self.infile

background = AsyncZip('mydata.txt', 'myarchive.zip')
background.start()
print 'The main program continues to run in foreground.'

background.join()    # Wait for the background task to finish
print 'Main program waited until background was done.'
```

O principal desafio para as aplicações que usam múltiplas threads é coordenar as threads que compartilham dados ou outros recursos. Para esta finalidade, o módulo threading oferece alguns mecanismos primitivos de sincronização, como travas (locks), eventos, variáveis de condição e semáforos.

While those tools are powerful, minor design errors can result in problems that are difficult to reproduce. So, the preferred approach to task coordination is to concentrate all access to a resource in a single thread and then use the `Queue` module to feed that thread with requests from other threads. Applications using `Queue.Queue` objects for inter-thread communication and coordination are easier to design, more readable, and more reliable.

11.5 Gerando logs

O módulo `logging` oferece um completo e flexível sistema de log. Da maneira mais simples, mensagens de log são enviadas para um arquivo ou para `sys.stderr`:

```
import logging
logging.debug('Debugging information')
logging.info('Informational message')
logging.warning('Warning:config file %s not found', 'server.conf')
logging.error('Error occurred')
logging.critical('Critical error -- shutting down')
```

Isso produz a seguinte saída:

```
WARNING:root:Warning:config file server.conf not found
ERROR:root:Error occurred
CRITICAL:root:Critical error -- shutting down
```

Por padrão, mensagens informativas e de depuração são suprimidas e a saída é enviada para a saída de erros padrão (stderr). Outras opções de saída incluem envio de mensagens através de correio eletrônico, datagramas, sockets ou para um servidor HTTP. Novos filtros podem selecionar diferentes formas de envio de mensagens, baseadas na prioridade da mensagem: `DEBUG`, `INFO`, `WARNING`, `ERROR` e `CRITICAL`.

O sistema de log pode ser configurado diretamente do Python ou pode ser carregado a partir de um arquivo de configuração editável pelo usuário para logs personalizados sem a necessidade de alterar a aplicação.

11.6 Referências fracas

Python faz gerenciamento automático de memória (contagem de referências para a maioria dos objetos e *garbage collection* [coleta de lixo] para eliminar ciclos). A memória ocupada por um objeto é liberada logo depois da última referência a ele ser eliminada.

Essa abordagem funciona bem para a maioria das aplicações, mas ocasionalmente surge a necessidade de rastrear objetos apenas enquanto estão sendo usados por algum outro. Infelizmente rastreá-los cria uma referência, e isso os fazem permanentes. O módulo `weakref` oferece ferramentas para rastrear objetos sem criar uma referência. Quando o objeto não é mais necessário, ele é automaticamente removido de uma tabela de referências fracas e uma chamada (*callback*) é disparada. Aplicações típicas incluem cacheamento de objetos que são muito custosos para criar:

```
>>> import weakref, gc
>>> class A:
...     def __init__(self, value):
...         self.value = value
...     def __repr__(self):
...         return str(self.value)
...
>>> a = A(10)                   # create a reference
>>> d = weakref.WeakValueDictionary()
>>> d['primary'] = a            # does not create a reference
>>> d['primary']                # fetch the object if it is still alive
10
>>> del a                       # remove the one reference
>>> gc.collect()                # run garbage collection right away
0
>>> d['primary']                # entry was automatically removed
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    d['primary']                # entry was automatically removed
  File "C:/python26/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
KeyError: 'primary'
```

11.7 Ferramentas para trabalhar com listas

Muitas necessidades envolvendo estruturas de dados podem ser satisfeitas com o tipo embutido lista. Entretanto, algumas vezes há uma necessidade por implementações alternativas que sacrificam algumas facilidades em nome de melhor desempenho.

O módulo `array` oferece uma classe `array`, semelhante a uma lista, mas que armazena apenas dados homogêneos e de maneira mais compacta. O exemplo a seguir mostra um vetor de números armazenados como números binários de dois bytes sem sinal (código de tipo `"H"`) ao invés dos 16 bytes usuais para cada item em uma lista de `int`:

```
>>> from array import array
>>> a = array('H', [4000, 10, 700, 22222])
>>> sum(a)
26932
>>> a[1:3]
array('H', [10, 700])
```

O módulo `collections` oferece um objeto `deque()` que comporta-se como uma lista mas com *appends* e *pops* pela esquerda mais rápidos, porém mais lento ao percorrer o meio da sequência. Esses objetos são adequados para implementar filas e buscas de amplitude em árvores de dados (*breadth first tree searches*):

```
>>> from collections import deque
>>> d = deque(["task1", "task2", "task3"])
>>> d.append("task4")
>>> print "Handling", d.popleft()
Handling task1
```

```
unsearched = deque([starting_node])
def breadth_first_search(unsearched):
    node = unsearched.popleft()
    for m in gen_moves(node):
        if is_goal(m):
            return m
        unsearched.append(m)
```

Além de implementações alternativas de listas, a biblioteca também oferece outras ferramentas como o módulo `bisect` com funções para manipulação de listas ordenadas:

```
>>> import bisect
>>> scores = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500, 'python')]
>>> bisect.insort(scores, (300, 'ruby'))
>>> scores
[(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400, 'lua'), (500, 'python')]
```

O módulo `heapq` oferece funções para implementação de *heaps* baseadas em listas normais. O valor mais baixo é sempre mantido na posição zero. Isso é útil para aplicações que acessam repetidamente o menor elemento, mas não querem reordenar a lista toda a cada acesso:

```
>>> from heapq import heapify, heappop, heappush
>>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(data)                      # rearrange the list into heap order
>>> heappush(data, -5)                 # add a new entry
>>> [heappop(data) for i in range(3)]  # fetch the three smallest entries
[-5, 0, 1]
```

11.8 Aritmética decimal com ponto flutuante

O módulo `decimal` oferece o tipo `Decimal` para aritmética decimal com ponto flutuante. Comparado a implementação embutida `float` que usa aritmética binária de ponto flutuante, a classe é especialmente útil para:

- aplicações financeiras que requerem representação decimal exata,
- controle sobre a precisão,
- controle sobre arredondamento para satisfazer requisitos legais,
- rastreamento de casas decimais significativas, ou
- aplicações onde o usuário espera que os resultados sejam os mesmos que os dos cálculos feitos à mão.

Por exemplo, calcular um imposto de 5% sobre uma chamada telefônica de 70 centavos devolve diferentes resultados com aritmética de ponto flutuante decimal ou binária. A diferença torna-se significativa se os resultados são arredondados para o centavo mais próximo:

```
>>> from decimal import *
>>> x = Decimal('0.70') * Decimal('1.05')
>>> x
Decimal('0.7350')
>>> x.quantize(Decimal('0.01'))  # round to nearest cent
Decimal('0.74')
>>> round(.70 * 1.05, 2)         # same calculation with floats
0.73
```

O resultado de `Decimal` considera zeros à direita, automaticamente inferindo quatro casas decimais a partir de multiplicandos com duas casas decimais. O módulo Decimal reproduz a aritmética como fazemos à mão e evita problemas que podem ocorrer quando a representação binária do ponto flutuante não consegue representar quantidades decimais com exatidão.

A representação exata permite à classe `Decimal` executar cálculos de módulo e testes de igualdade que não funcionam bem em ponto flutuante binário:

```
>>> Decimal('1.00') % Decimal('.10')
Decimal('0.00')
>>> 1.00 % 0.10
0.09999999999999995

>>> sum([Decimal('0.1')]*10) == Decimal('1.0')
True
>>> sum([0.1]*10) == 1.0
False
```

O módulo `decimal` implementa a aritmética com tanta precisão quanto necessária:

```
>>> getcontext().prec = 36
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857')
```

CAPÍTULO 12

E agora?

Ler este tutorial provavelmente reforçou seu interesse em usar Python — você deve estar ansioso para aplicar Python para resolver problemas do mundo real. Aonde você deveria ir para aprender mais?

Este tutorial é parte do conjunto de documentação da linguagem Python. Alguns outros documentos neste conjunto são:

- library-index:

  Você deveria navegar através deste manual, que lhe dará material completo (ainda que breve) de referência sobre tipos, funções e módulos na biblioteca padrão. A distribuição padrão do Python inclui *muito* código adicional. Há módulos para ler caixa de correio Unix, baixar documentos via HTTP, gerar números aleatórios, processar opções de linha de comando, escrever programas CGI, comprimir dados a muitas outras tarefas. Uma lida rápida da Referência da Biblioteca lhe dará uma ideia do que está disponível.

- install-index explains how to install external modules written by other Python users.

- reference-index: Uma explicação detalhada da sintaxe e da semântica do Python. É uma leitura pesada, mas é útil como um guia completo da linguagem propriamente dita.

Mais recursos Python:

- https://www.python.org: O principal website sobre Python. Ele contém código, documentação e aponta para páginas na Web relacionadas ao Python. Esse site é espelhado em vários lugares do mundo como Europa, Japão e Austrália; um espelho pode ser mais rápido que o site principal, dependendo da sua localização geográfica.

- https://docs.python.org: Acesso rápido à documentação Python.

- https://pypi.org: The Python Package Index, previously also nicknamed the Cheese Shop, is an index of user-created Python modules that are available for download. Once you begin releasing code, you can register it here so that others can find it.

- https://code.activestate.com/recipes/langs/python/: O Python Cookbook (livro de receitas de Python) é uma grande coleção de exemplos de código, módulos maiores e scripts úteis.Contribuições particularmente notáveis são coletadas em um livro também chamado Python Cookbook (O'Reilly & Associates, ISBN 0-596-00797-3.)

For Python-related questions and problem reports, you can post to the newsgroup `comp.lang.python`, or send them to the mailing list at python-list@python.org. The newsgroup and mailing list are gatewayed, so messages posted to one will automatically be forwarded to the other. There are around 120 postings a day (with peaks up to several hundred), asking (and answering) questions, suggesting new features, and announcing new modules. Before posting, be sure to check

the list of Frequently Asked Questions (also called the FAQ). Mailing list archives are available at https://mail.python.org/pipermail/. The FAQ answers many of the questions that come up again and again, and may already contain the solution for your problem.

CAPÍTULO 13

Edição de entrada interativa e substituição de histórico

Some versions of the Python interpreter support editing of the current input line and history substitution, similar to facilities found in the Korn shell and the GNU Bash shell. This is implemented using the GNU Readline library, which supports Emacs-style and vi-style editing. This library has its own documentation which I won't duplicate here; however, the basics are easily explained. The interactive editing and history described here are optionally available in the Unix and Cygwin versions of the interpreter.

This chapter does *not* document the editing facilities of Mark Hammond's PythonWin package or the Tk-based environment, IDLE, distributed with Python. The command line history recall which operates within DOS boxes on NT and some other DOS and Windows flavors is yet another beast.

13.1 Line Editing

If supported, input line editing is active whenever the interpreter prints a primary or secondary prompt. The current line can be edited using the conventional Emacs control characters. The most important of these are: `C-A` (Control-A) moves the cursor to the beginning of the line, `C-E` to the end, `C-B` moves it one position to the left, `C-F` to the right. Backspace erases the character to the left of the cursor, `C-D` the character to its right. `C-K` kills (erases) the rest of the line to the right of the cursor, `C-Y` yanks back the last killed string. `C-underscore` undoes the last change you made; it can be repeated for cumulative effect.

13.2 History Substitution

History substitution works as follows. All non-empty input lines issued are saved in a history buffer, and when a new prompt is given you are positioned on a new line at the bottom of this buffer. `C-P` moves one line up (back) in the history buffer, `C-N` moves one down. Any line in the history buffer can be edited; an asterisk appears in front of the prompt to mark a line as modified. Pressing the `Return` key passes the current line to the interpreter. `C-R` starts an incremental reverse search; `C-S` starts a forward search.

13.3 Key Bindings

The key bindings and some other parameters of the Readline library can be customized by placing commands in an initialization file called `~/.inputrc`. Key bindings have the form

```
key-name: function-name
```

or

```
"string": function-name
```

and options can be set with

```
set option-name value
```

For example:

```
# I prefer vi-style editing:
set editing-mode vi

# Edit using a single line:
set horizontal-scroll-mode On

# Rebind some keys:
Meta-h: backward-kill-word
"\C-u": universal-argument
"\C-x\C-r": re-read-init-file
```

Note that the default binding for `Tab` in Python is to insert a `Tab` character instead of Readline's default filename completion function. If you insist, you can override this by putting

```
Tab: complete
```

in your `~/.inputrc`. (Of course, this makes it harder to type indented continuation lines if you're accustomed to using `Tab` for that purpose.)

Automatic completion of variable and module names is optionally available. To enable it in the interpreter's interactive mode, add the following to your startup file:[1]

```
import rlcompleter, readline
readline.parse_and_bind('tab: complete')
```

This binds the `Tab` key to the completion function, so hitting the `Tab` key twice suggests completions; it looks at Python statement names, the current local variables, and the available module names. For dotted expressions such as `string.a`, it will evaluate the expression up to the final `'.'` and then suggest completions from the attributes of the resulting object. Note that this may execute application-defined code if an object with a `__getattr__()` method is part of the expression.

A more capable startup file might look like this example. Note that this deletes the names it creates once they are no longer needed; this is done since the startup file is executed in the same namespace as the interactive commands, and removing the names avoids creating side effects in the interactive environment. You may find it convenient to keep some of the imported modules, such as `os`, which turn out to be needed in most sessions with the interpreter.

[1] Python will execute the contents of a file identified by the `PYTHONSTARTUP` environment variable when you start an interactive interpreter. To customize Python even for non-interactive mode, see *Módulos de customização*.

```
# Add auto-completion and a stored history file of commands to your Python
# interactive interpreter. Requires Python 2.0+, readline. Autocomplete is
# bound to the Esc key by default (you can change it - see readline docs).
#
# Store the file in ~/.pystartup, and set an environment variable to point
# to it:  "export PYTHONSTARTUP=~/.pystartup" in bash.

import atexit
import os
import readline
import rlcompleter

historyPath = os.path.expanduser("~/.pyhistory")

def save_history(historyPath=historyPath):
    import readline
    readline.write_history_file(historyPath)

if os.path.exists(historyPath):
    readline.read_history_file(historyPath)

atexit.register(save_history)
del os, atexit, readline, rlcompleter, save_history, historyPath
```

13.4 Alternativas ao interpretador interativo

Esta facilidade é um enorme passo em frente em comparação com as versões anteriores do intérpretador; No entanto, alguns desejos são deixados: seria bom se a indentação adequada fosse sugerida nas linhas de continuação (o analisador sabe se é necessário um token de recuo). O mecanismo de conclusão pode usar a tabela de símbolos do interpretador. Um comando para verificar (ou mesmo sugerir) parênteses, citações, etc., também seria útil.

Um interpretador interativo aprimorado e alternativo que existe há algum tempo é o IPython, que apresenta a conclusão da guia, a exploração de objetos e o gerenciamento de histórico avançado. Também pode ser completamente personalizado e incorporado em outras aplicações. Outro ambiente interativo aprimorado similar é bpython.

CAPÍTULO 14

Aritmética de ponto flutuante: problemas e limitações

Números de ponto flutuante são representados no hardware do computador como frações binárias (base 2). Por exemplo, a fração decimal:

```
0.125
```

tem o valor 1/10 + 2/100 + 5/1000, e da mesma maneira a fração binária:

```
0.001
```

tem o valor 0/2 + 0/4 + 1/8. Essas duas frações têm valores idênticos, a única diferença real é que a primeira está representada na forma de frações base 10, e a segunda na base 2.

Infelizmente, muitas frações decimais não podem ser representadas precisamente como frações binárias. O resultado é que, em geral, os números decimais de ponto flutuante que você digita acabam sendo armazenados de forma apenas aproximada, na forma de números binários de ponto flutuante.

O problema é mais fácil de entender primeiro em base 10. Considere a fração 1/3. Podemos representá-la aproximadamente como uma fração base 10:

```
0.3
```

ou melhor,

```
0.33
```

ou melhor,

```
0.333
```

e assim por diante. Não importa quantos dígitos você está disposto a escrever, o resultado nunca será exatamente 1/3, mas será uma aproximação de cada vez melhor de 1/3.

Da mesma forma, não importa quantos dígitos de base 2 estejas disposto a usar, o valor decimal 0.1 não pode ser representado exatamente como uma fração de base 2. No sistema de base 2, 1/10 é uma fração binária que se repete infinitamente:

```
0.0001100110011001100110011001100110011001100110011...
```

Stop at any finite number of bits, and you get an approximation.

On a typical machine running Python, there are 53 bits of precision available for a Python float, so the value stored internally when you enter the decimal number `0.1` is the binary fraction

```
0.0001100110011001100110011001100110011001100110011010
```

which is close to, but not exactly equal to, 1/10.

It's easy to forget that the stored value is an approximation to the original decimal fraction, because of the way that floats are displayed at the interpreter prompt. Python only prints a decimal approximation to the true decimal value of the binary approximation stored by the machine. If Python were to print the true decimal value of the binary approximation stored for 0.1, it would have to display

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

Contém muito mais dígitos do que é o esperado e utilizado pela grande maioria dos desenvolvedores, portanto, o Python limita o número de dígitos exibidos, apresentando um valor arredondado, ao invés de mostrar todas as casas decimais:

```
>>> 0.1
0.1
```

It's important to realize that this is, in a real sense, an illusion: the value in the machine is not exactly 1/10, you're simply rounding the *display* of the true machine value. This fact becomes apparent as soon as you try to do arithmetic with these values

```
>>> 0.1 + 0.2
0.30000000000000004
```

Note que essa é a própria natureza do ponto flutuante binário: não é um bug do Python, e nem é um bug do seu código. Essa situação pode ser observada em todas as linguagens que usam as instruções aritméticas de ponto flutuante do hardware (apesar de algumas linguagens não *mostrarem* a diferença, por padrão, ou em todos os modos de saída).

Other surprises follow from this one. For example, if you try to round the value 2.675 to two decimal places, you get this

```
>>> round(2.675, 2)
2.67
```

The documentation for the built-in `round()` function says that it rounds to the nearest value, rounding ties away from zero. Since the decimal fraction 2.675 is exactly halfway between 2.67 and 2.68, you might expect the result here to be (a binary approximation to) 2.68. It's not, because when the decimal string `2.675` is converted to a binary floating-point number, it's again replaced with a binary approximation, whose exact value is

```
2.674999999999999822364316059974953532218933105468 75
```

Since this approximation is slightly closer to 2.67 than to 2.68, it's rounded down.

If you're in a situation where you care which way your decimal halfway-cases are rounded, you should consider using the `decimal` module. Incidentally, the `decimal` module also provides a nice way to "see" the exact value that's stored in any particular Python float

```
>>> from decimal import Decimal
>>> Decimal(2.675)
Decimal('2.674999999999999822364316059974953532218933105468 75')
```

Another consequence is that since 0.1 is not exactly 1/10, summing ten values of 0.1 may not yield exactly 1.0, either:

```
>>> sum = 0.0
>>> for i in range(10):
...     sum += 0.1
...
>>> sum
0.9999999999999999
```

A aritmética de ponto flutuante binário traz muitas surpresas como essas. O problema do "0.1" é explicado em detalhes precisos abaixo, na seção "Erro de Representação". Para uma descrição mais completa de outras surpresas que comumente nos deparamos, veja a seção The Perils of Floating Point que contém diversos exemplos distintos.

As that says near the end, "there are no easy answers." Still, don't be unduly wary of floating-point! The errors in Python float operations are inherited from the floating-point hardware, and on most machines are on the order of no more than 1 part in 2**53 per operation. That's more than adequate for most tasks, but you do need to keep in mind that it's not decimal arithmetic, and that every float operation can suffer a new rounding error.

While pathological cases do exist, for most casual use of floating-point arithmetic you'll see the result you expect in the end if you simply round the display of your final results to the number of decimal digits you expect. For fine control over how a float is displayed see the `str.format()` method's format specifiers in formatstrings.

14.1 Erro de representação

Esta seção explica o exemplo do "0,1" em detalhes, e mostra como poderás realizar uma análise exata de casos semelhantes. Assumimos que tenhas uma familiaridade básica com a representação binária de ponto flutuante.

Representation error refers to the fact that some (most, actually) decimal fractions cannot be represented exactly as binary (base 2) fractions. This is the chief reason why Python (or Perl, C, C++, Java, Fortran, and many others) often won't display the exact decimal number you expect:

```
>>> 0.1 + 0.2
0.30000000000000004
```

Why is that? 1/10 and 2/10 are not exactly representable as a binary fraction. Almost all machines today (July 2010) use IEEE-754 floating point arithmetic, and almost all platforms map Python floats to IEEE-754 "double precision". 754 doubles contain 53 bits of precision, so on input the computer strives to convert 0.1 to the closest fraction it can of the form *J*/2***N* where *J* is an integer containing exactly 53 bits. Rewriting

```
1 / 10 ~= J / (2**N)
```

como

```
J ~= 2**N / 10
```

e recordando que *J* tenha exatamente 53 bits (é `>=  2**52`, mas `<  2**53`), o melhor valor para *N* é 56:

```
>>> 2**52
4503599627370496
>>> 2**53
9007199254740992
>>> 2**56/10
7205759403792793
```

That is, 56 is the only value for *N* that leaves *J* with exactly 53 bits. The best possible value for *J* is then that quotient rounded:

```
>>> q, r = divmod(2**56, 10)
>>> r
6
```

Uma vez que o resto seja maior do que a metade de 10, a melhor aproximação que poderá ser obtida se arredondarmos para cima:

```
>>> q+1
7205759403792794
```

Therefore the best possible approximation to 1/10 in 754 double precision is that over 2**56, or

```
7205759403792794 / 72057594037927936
```

Note que, como arredondamos para cima, esse valor é, de fato, um pouco maior que 1/10; se não tivéssemos arredondado para cima, o quociente teria sido um pouco menor que 1/10. Mas em nenhum caso seria possível obter *exatamente* o valor 1/10!

Por isso, o computador nunca "vê" 1/10: o que ele vê é exatamente a fração que é obtida pra cima, a melhor aproximação "IEEE-754 double" possível é:

```
>>> .1 * 2**56
7205759403792794.0
```

If we multiply that fraction by 10**30, we can see the (truncated) value of its 30 most significant decimal digits:

```
>>> 7205759403792794 * 10**30 // 2**56
100000000000000005551115123125L
```

meaning that the exact number stored in the computer is approximately equal to the decimal value 0.100000000000000005551115123125. In versions prior to Python 2.7 and Python 3.1, Python rounded this value to 17 significant digits, giving '0.10000000000000001'. In current versions, Python displays a value based on the shortest decimal fraction that rounds correctly back to the true binary value, resulting simply in '0.1'.

CAPÍTULO 15

Anexo

15.1 Modo interativo

15.1.1 Tratamento de erros

Quando um erro ocorre, o interpretador exibe uma mensagem de erro e um *stack trace* (rastreamento de pilha). Se estiver no modo interativo, ele volta para o prompt primário; se a entrada veio de um arquivo, a execução termina com um status de saída *nonzero* (diferente de zero) após a exibição do *stack trace*. (Exceções tratadas por uma cláusula `except` numa declaração `try` não são consideradas erros, nesse contexto.) Alguns erros são irremediavelmente graves e causam términos de execução com status de saída *nonzero*; isso pode acontecer devido a inconsistências internas e em alguns casos por falta de memória. Todas as mensagens de erro são escritas no fluxo de erros padrão; a saída normal resultante da execução de comandos é escrita no canal de saída padrão.

Digitar o caractere de interrupção (geralmente `Control-C` ou `Delete`) em prompts primários ou secundários causam a interrupção da entrada de dados e o retorno ao prompt primário.[1] Digitar a interrupção durante a execução de um comando levanta a exceção `KeyboardInterrupt`, que pode ser tratada por uma declaração `try`.

15.1.2 Scripts Python executáveis

Em sistemas Unix similares ao BSD, scripts Python podem ser executados diretamente, tal como scripts shell, se tiverem a linha de código

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

(assumindo que o interpretador está na `PATH` do usuário) no começo do script e configurando o arquivo no modo executável. Os dois primeiros caracteres do arquivo devem ser `#!`. Em algumas plataformas, essa primeira linha deve terminar com uma quebra de linha em estilo Unix (`'\n'`), e não em estilo windows (`'\r\n'`). Note que o caractere `'#'` (em inglês chamado de *hash*, ou *pound* etc.), é usado em Python para marcar o início de um comentário.

O script pode receber a permissão para atuar em modo executável através do comando **`chmod`**.

[1] Um problema com a package GNU Readline pode impedir que isso aconteça.

```
$ chmod +x myscript.py
```

Em sistemas Windows, não existe a noção de um "modo executável". O instalador Python associa automaticamente os arquivos `.py` com o `python.exe`, de forma que um clique duplo num arquivo Python o executará como um script. A extensão pode ser também `.pyw`, o que omite a janela de console que normalmente aparece.

15.1.3 Arquivo de inicialização do modo interativo

Quando se usa o Python no modo interativo, pode ser útil definir alguns comandos que sejam executados automaticamente toda vez que o interpretador for inicializado. Isso pode ser feito configurando-se uma variável de ambiente chamada `PYTHONSTARTUP` para que ela aponte para o arquivo contendo esses comandos. Isso é similar ao recurso `.profile` das shells Unix.

Esse arquivo será lido apenas em sessões do modo interativo, e não quando Python lê comandos de um script, tampouco quando `/dev/tty` é passado explicitamente como a origem dos comandos (neste caso, teremos um comportamento similar a uma sessão interativa padrão). Ele é executado no mesmo *namespace* (espaço de nomes) em que os comandos interativos são executados, de modo que os objetos que ele define ou importa possam ser usados sem qualificação na sessão interativa. Também é possível alterar os *prompts* `sys.ps1` e `sys.ps2` no mesmo arquivo.

Caso deseje usar um arquivo de inicialização adicional a partir do atual diretório de trabalho, você pode programá-lo no arquivo de inicialização global usando um código parecido com `if os.path.isfile('.pythonrc.py'): exec(open('.pythonrc.py').read())`. Se quiser usar o arquivo de inicialização num script, será necessário fazê-lo explicitamente no script:

```
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
    with open(filename) as fobj:
        startup_file = fobj.read()
    exec(startup_file)
```

15.1.4 Módulos de customização

Python oferece dois *hooks* que permitem sua customização: `sitecustomize` e `usercustomize`. Para entender como funcionam, primeiro você deve localizar o diretório site-packages do usuário. Inicie o Python e execute este código:

```
>>> import site
>>> site.getusersitepackages()
'/home/user/.local/lib/python2.7/site-packages'
```

Agora você pode criar um arquivo chamado `usercustomize.py` neste diretório e colocar qualquer coisa que quiser dentro. Isto vai afetar toda invocação do Python, a menos que seja iniciado com a opção `-s` para desabilitar a importação automática.

`sitecustomize` funciona da mesma forma, mas normalmente é criado por um administrador do computador no diretório site-packages global e é importado antes de `usercustomize`. Veja a documentação do módulo `site` para mais detalhes.

Notas de Rodapé

APÊNDICE A

Glossário

>>> O prompt padrão do shell interativo do Python. Normalmente visto em exemplos de código que podem ser executados interativamente no interpretador.

... The default Python prompt of the interactive shell when entering code for an indented code block, when within a pair of matching left and right delimiters (parentheses, square brackets, curly braces or triple quotes), or after specifying a decorator.

2to3 Uma ferramenta que tenta converter código Python 2.x em código Python 3.x tratando a maioria das incompatibilidades que podem se detectadas com análise do código-fonte e navegação na árvore sintática.

O 2to3 está disponível na biblioteca padrão como `lib2to3`; um ponto de entrada é disponibilizado como `Tools/scripts/2to3`. Veja 2to3-reference.

classe base abstrata Abstract base classes complement *duck-typing* by providing a way to define interfaces when other techniques like `hasattr()` would be clumsy or subtly wrong (for example with magic methods). ABCs introduce virtual subclasses, which are classes that don't inherit from a class but are still recognized by `isinstance()` and `issubclass()`; see the `abc` module documentation. Python comes with many built-in ABCs for data structures (in the `collections` module), numbers (in the `numbers` module), and streams (in the `io` module). You can create your own ABCs with the `abc` module.

argumento A value passed to a *function* (or *method*) when calling the function. There are two types of arguments:

- *argumento nomeado*: um argumento precedido por um identificador (por exemplo, `nome=`) na chamada de uma função ou passada como um valor em um dicionário precedido por `**`. Por exemplo, `3` e `5` são ambos argumentos nomeados na chamada da função `complex()` a seguir:

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- *argumento posicional*: um argumento que não é um argumento nomeado. Argumentos posicionais podem aparecer no início da lista de argumentos e/ou podem ser passados com elementos de um *iterável* precedido por `*`. Por exemplo, `3` e `5` são ambos argumentos posicionais nas chamadas a seguir:

```
complex(3, 5)
complex(*(3, 5))
```

Argumentos são atribuídos às variáveis locais nomeadas no corpo da função. Veja a seção calls para as regras de atribuição. Sintaticamente, qualquer expressão pode ser usada para representar um argumento; avaliada a expressão, o valor é atribuído à variável local.

See also the *parameter* glossary entry and the FAQ question on the difference between arguments and parameters.

atributo Um valor associado a um objeto que é referenciado pelo nome separado por um ponto. Por exemplo, se um objeto *o* tem um atributo *a* esse seria referenciado como *o.a*.

BDFL Benevolent Dictator For Life, a.k.a. Guido van Rossum, Python's creator.

objeto byte ou similar An object that supports the buffer protocol, like `str`, `bytearray` or `memoryview`. Bytes-like objects can be used for various operations that expect binary data, such as compression, saving to a binary file or sending over a socket. Some operations need the binary data to be mutable, in which case not all bytes-like objects can apply.

bytecode Python source code is compiled into bytecode, the internal representation of a Python program in the CPython interpreter. The bytecode is also cached in `.pyc` and `.pyo` files so that executing the same file is faster the second time (recompilation from source to bytecode can be avoided). This "intermediate language" is said to run on a *virtual machine* that executes the machine code corresponding to each bytecode. Do note that bytecodes are not expected to work between different Python virtual machines, nor to be stable between Python releases.

Uma lista de instruções bytecode pode ser encontrada na documentação para o módulo dis.

Classe Um modelo para criação de objetos definidos pelo usuário. Definições de classe normalmente contém definições de métodos que operam sobre instâncias da classe.

classic class Any class which does not inherit from `object`. See *new-style class*. Classic classes have been removed in Python 3.

coerção The implicit conversion of an instance of one type to another during an operation which involves two arguments of the same type. For example, `int(3.15)` converts the floating point number to the integer `3`, but in `3+4.5`, each argument is of a different type (one int, one float), and both must be converted to the same type before they can be added or it will raise a `TypeError`. Coercion between two operands can be performed with the `coerce` built-in function; thus, `3+4.5` is equivalent to calling `operator.add(*coerce(3, 4.5))` and results in `operator.add(3.0, 4.5)`. Without coercion, all arguments of even compatible types would have to be normalized to the same value by the programmer, e.g., `float(3)+4.5` rather than just `3+4.5`.

número complexo Uma extensão ao familiar sistema de números reais em que todos os números são expressos como uma soma de uma parte real e uma parte imaginária. Números imaginários são múltiplos reais da unidade imaginária (a raiz quadrada de -1), normalmente escrita como `i` em matemática ou `j` em engenharia. O Python tem suporte nativo para números complexos, que são escritos com esta última notação; a parte imaginária escrita com um sufixo `j`, p.ex., `3+1j`. Para ter acesso aos equivalentes para números complexos do módulo `math`, utilize `cmath`. O uso de números complexos é uma funcionalidade matemática bastante avançada. Se você não sabe se irá precisar deles, é quase certo que você pode ignorá-los sem problemas.

gerenciador de contexto Um objeto que controla o ambiente visto numa instrução `with` por meio da definição dos métodos `__enter__()` e `__exit__()`. Veja **PEP 343**.

CPython A implementação canônica da linguagem de programação Python, como disponibilizada pelo python.org. O termo "CPython" é quando for necessário distinguir esta implementação de outras como Jython ou IronPython.

decorador Uma função que retorna outra função, geralmente aplicada como uma transformação de função usando a sintaxe `@wrapper`. Exemplos comuns para decoradores são `classmethod()` e `staticmethod()`.

A sintaxe do decorador é meramente um açúcar-sintático, as duas definições de funções a seguir são semanticamente equivalentes:

```
def f(...):
    ...
f = staticmethod(f)
```

(continua na próxima página)

(continuação da página anterior)

```
@staticmethod
def f(...):
    ...
```

O mesmo conceito existe para as classes, mas não é comumente utilizado. Veja a documentação de function definitions e class definitions para obter mais informações sobre decoradores.

descritor Any *new-style* object which defines the methods `__get__()`, `__set__()`, or `__delete__()`. When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, using *a.b* to get, set or delete an attribute looks up the object named *b* in the class dictionary for *a*, but if *b* is a descriptor, the respective descriptor method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.

Para obter mais informações sobre os métodos dos descritores, veja: descriptors.

dicionário An associative array, where arbitrary keys are mapped to values. The keys can be any object with `__hash__()` and `__eq__()` methods. Called a hash in Perl.

visualização de dicionário The objects returned from `dict.viewkeys()`, `dict.viewvalues()`, and `dict.viewitems()` are called dictionary views. They provide a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes. To force the dictionary view to become a full list use `list(dictview)`. See dict-views.

docstring Uma string literal que aparece como primeira expressão numa classe, função ou módulo. Ainda que sejam ignoradas quando a suíte é executada, é reconhecida pelo compilador que a coloca no atributo `__doc__` da classe, função ou módulo que a encapsula. Como ficam disponíveis por meio de introspecção, docstrings são o lugar canônico para documentação do objeto.

duck-typing (tipagem pato) Um estilo de programação que não verifica o tipo do objeto para determinar se ele possui a interface correta; em vez disso, o método ou atributo é simplesmente chamado ou utilizado ("Se se parece com um pato e grasna como um pato, então deve ser um pato.") Enfatizando interfaces ao invés de tipos específicos, o código bem desenvolvido aprimora sua flexibilidade por permitir substituição polimórfica. Tipagem pato evita necessidade de testes que usem `type()` ou `isinstance()`. (Note, porém, que a a tipagem pato pode ser complementada com o uso de *classes base abstratas*.) Ao invés disso, são normalmente empregados testes `hasattr()` ou programação *EAFP*.

EAFP Iniciais da expressão em inglês "easier to ask for forgiveness than permission" que significa "é mais fácil pedir perdão que permissão". Este estilo de codificação comum em Python assume a existência de chaves ou atributos válidos e captura exceções caso essa premissa se prove falsa. Este estilo limpo e rápido se caracteriza pela presença de várias instruções `try` e `except`. A técnica diverge do estilo *LBYL*, comum em outras linguagens como C, por exemplo.

expressão A piece of syntax which can be evaluated to some value. In other words, an expression is an accumulation of expression elements like literals, names, attribute access, operators or function calls which all return a value. In contrast to many other languages, not all language constructs are expressions. There are also *statement*s which cannot be used as expressions, such as `print` or `if`. Assignments are also statements, not expressions.

extension module (módulo de extensão) Um módulo escrito em C ou C++, usando a API C de Python para interagir tanto com código de usuário quanto do núcleo.

objeto arquivo Um objeto que expõe uma API orientada a arquivos (com métodos tais como `read()` ou `write()`) para um recurso subjacente. Dependendo da maneira como foi criado, um objeto arquivo pode mediar o acesso a um arquivo real no disco ou outro tipo de dispositivo de armazenamento ou de comunicação (por exemplo a entrada/saída padrão, buffers em memória, soquetes, pipes, etc.). Objetos arquivo também são chamados de *file-like objects* ou *streams*.

There are actually three categories of file objects: raw binary files, buffered binary files and text files. Their interfaces are defined in the `io` module. The canonical way to create a file object is by using the `open()` function.

objeto como-arquivo Um sinônimo do termo *file object*.

localizador An object that tries to find the *loader* for a module. It must implement a method named `find_module()`. See **PEP 302** for details.

divisão pelo piso Divisão matemática que arredonda para baixo para o inteiro mais próximo. O operador de divisão pelo piso é `//`. Por exemplo, a expressão `11 // 4` retorna o valor `2` ao invés de `2.75`, que seria retornado pela divisão de ponto flutuante. Note que `(-11) // 4` é `-3` porque é `-2.75` arredondado *para baixo*. Consulte a **PEP 238**.

função Uma série de instruções que retorna algum valor para um chamador. Também pode ser passado zero ou mais *argumentos* que podem ser usados na execução do corpo. Veja também *parâmetro*, *métodos* e a seção function.

__future__ A pseudo-module which programmers can use to enable new language features which are not compatible with the current interpreter. For example, the expression `11/4` currently evaluates to `2`. If the module in which it is executed had enabled *true division* by executing:

```
from __future__ import division
```

the expression `11/4` would evaluate to `2.75`. By importing the `__future__` module and evaluating its variables, you can see when a new feature was first added to the language and when it will become the default:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

garbage collection (coletor de lixo) The process of freeing memory when it is not used anymore. Python performs garbage collection via reference counting and a cyclic garbage collector that is able to detect and break reference cycles.

gerador A function which returns an iterator. It looks like a normal function except that it contains `yield` statements for producing a series of values usable in a for-loop or that can be retrieved one at a time with the `next()` function. Each `yield` temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the generator resumes, it picks up where it left off (in contrast to functions which start fresh on every invocation).

generator expression An expression that returns an iterator. It looks like a normal expression followed by a `for` expression defining a loop variable, range, and an optional `if` expression. The combined expression generates values for an enclosing function:

```
>>> sum(i*i for i in range(10))         # sum of squares 0, 1, 4, ... 81
285
```

GIL Veja *global interpreter lock*.

global interpreter lock (bloqueio global do interpretador) O mecanismo utilizado pelo interpretador *CPython* para garantir que apenas uma thread execute o *bytecode* Python por vez. Isto simplifica a implementação do CPython ao fazer com que o modelo de objetos (incluindo tipos internos críticos como o `dict`) ganhem segurança implícita contra acesso concorrente. Travar todo o interpretador facilita que o interpretador em si seja multitarefa, às custas de muito do paralelismo já provido por máquinas multiprocessador.

No entanto, alguns módulos de extensão, tanto da biblioteca padrão quanto de terceiros, são desenvolvidos de forma a liberar o GIL ao realizar tarefas computacionalmente muito intensas, como compactação ou cálculos de hash. Além disso, o GIL é sempre liberado nas operações de E/S.

No passado, esforços para criar um interpretador que lidasse plenamente com threads (travando dados compartilhados numa granularidade bem mais fina) não foram bem sucedidos devido a queda no desempenho ao serem

executados em processadores de apenas um núcleo. Acredita-se que superar essa questão de desempenho acabaria tornando a implementação muito mais complicada e bem mais difícil de manter.

hasheável An object is *hashable* if it has a hash value which never changes during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an `__eq__()` or `__cmp__()` method). Hashable objects which compare equal must have the same hash value.

A hashabilidade faz com que um objeto possa ser usado como chave de um dicionário e como membro de um conjunto, pois estas estruturas de dados utilizam os valores de hash internamente.

All of Python's immutable built-in objects are hashable, while no mutable containers (such as lists or dictionaries) are. Objects which are instances of user-defined classes are hashable by default; they all compare unequal (except with themselves), and their hash value is derived from their `id()`.

IDLE Um ambiente de desenvolvimento integrado para Python. IDLE é um editor básico e um ambiente interpretador que vem junto com a distribuição padrão do Python.

imutável Um objeto que possui um valor fixo. Objetos imutáveis incluem números, strings e tuplas. Estes objetos não podem ser alterados. Um novo objeto deve ser criado se um valor diferente tiver de ser armazenado. Objetos imutáveis têm um papel importante em lugares onde um valor constante de hash seja necessário, como por exemplo uma chave em um dicionário.

integer division Mathematical division discarding any remainder. For example, the expression `11/4` currently evaluates to `2` in contrast to the `2.75` returned by float division. Also called *floor division*. When dividing two integers the outcome will always be another integer (having the floor function applied to it). However, if one of the operands is another numeric type (such as a `float`), the result will be coerced (see *coercion*) to a common type. For example, an integer divided by a float will result in a float value, possibly with a decimal fraction. Integer division can be forced by using the `//` operator instead of the `/` operator. See also *__future__*.

importing (importando) O processo pelo qual o código Python em um módulo é disponibilizado para o código Python em outro módulo.

importer Um objeto que localiza e carrega um módulo; Tanto um *finder* e o objeto *loader*.

interactive Python tem um interpretador interativo, o que significa que você pode digitar comandos e expressões no prompt do interpretador, executá-los imediatamente e ver seus resultados. Apenas execute `python` sem argumentos (possivelmente selecionando-o a partir do menu de aplicações de seu sistema operacional). O interpretador interativo é uma maneira poderosa de testar novas ideias ou aprender mais sobre módulos e pacotes (lembre-se do comando `help(x)`).

interpreted Python é uma linguagem interpretada, em oposição àquelas que são compiladas, embora esta distinção possa ser nebulosa devido à presença do compilador de bytecode. Isto significa que os arquivos-fontes podem ser executados diretamente sem necessidade explícita de se criar um arquivo executável. Linguagens interpretadas normalmente têm um ciclo de desenvolvimento/depuração mais curto que as linguagens compiladas, apesar de seus programas geralmente serem executados mais lentamente. Veja também *interativo*.

iterável An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list`, `str`, and `tuple`) and some non-sequence types like `dict` and `file` and objects of any classes you define with an `__iter__()` or `__getitem__()` method. Iterables can be used in a `for` loop and in many other places where a sequence is needed (`zip()`, `map()`, ...). When an iterable object is passed as an argument to the built-in function `iter()`, it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call `iter()` or deal with iterator objects yourself. The `for` statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also *iterator*, *sequence*, and *generator*.

iterador An object representing a stream of data. Repeated calls to the iterator's `next()` method return successive items in the stream. When no more data are available a `StopIteration` exception is raised instead. At this point, the iterator object is exhausted and any further calls to its `next()` method just raise `StopIteration` again. Iterators are required to have an `__iter__()` method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code

which attempts multiple iteration passes. A container object (such as a `list`) produces a fresh new iterator each time you pass it to the `iter()` function or use it in a `for` loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.

Mais informações podem ser encontradas em typeiter.

key function (função chave) Uma função chave ou função colação é algo que retorna um valor utilizado para ordenação ou classificação. Por exemplo, `locale.strxfrm()` é usada para produzir uma chave de ordenação que leva o locale em consideração para fins de ordenação.

A number of tools in Python accept key functions to control how elements are ordered or grouped. They include `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.nsmallest()`, `heapq.nlargest()`, and `itertools.groupby()`.

There are several ways to create a key function. For example. the `str.lower()` method can serve as a key function for case insensitive sorts. Alternatively, an ad-hoc key function can be built from a `lambda` expression such as `lambda r: (r[0], r[2])`. Also, the `operator` module provides three key function constructors: `attrgetter()`, `itemgetter()`, and `methodcaller()`. See the Sorting HOW TO for examples of how to create and use key functions.

argumento nomeado Veja o *argument*.

lambda Uma função de linha anônima consistindo de uma única *expression*, que é avaliada quando a função é chamada. A sintaxe para criar uma função lambda é `lambda [parameters]: expression`

LBYL Iniciais da expressão em inglês "look before you leap", que significa algo como "olhe antes de pisar". Este estilo de codificação testa as pré-condições explicitamente antes de fazer chamadas ou buscas. Este estilo contrasta com a abordagem *EAFP* e é caracterizada pela presença de muitos comandos `if`.

Em um ambiente multithread, a abordagem LBYL pode arriscar a introdução de uma condição de corrida entre "o olhar" e "o pisar". Por exemplo, o código `if key in mapping: return mapping[key]` pode falhar se outra thread remover *key* do *mapping* após o teste, mas antes da olhada. Esse problema pode ser resolvido com bloqueios ou usando a abordagem EAFP.

lista Uma *sequence* embutida no Python. Apesar do seu nome, é mais próximo de um vetor em outras linguagens do que uma lista encadeada, como o acesso aos elementos é da ordem O(1).

list comprehension A compact way to process all or part of the elements in a sequence and return a list with the results. `result = ["0x%02x" % x for x in range(256) if x % 2 == 0]` generates a list of strings containing even hex numbers (0x..) in the range from 0 to 255. The `if` clause is optional. If omitted, all elements in `range(256)` are processed.

loader An object that loads a module. It must define a method named `load_module()`. A loader is typically returned by a *finder*. See **PEP 302** for details.

método mágico Um sinônimo informal para um *special method*.

mapeamento A container object that supports arbitrary key lookups and implements the methods specified in the `Mapping` or `MutableMapping` abstract base classes. Examples include `dict`, `collections.defaultdict`, `collections.OrderedDict` and `collections.Counter`.

metaclass A classe de uma classe. Definições de classe criam um nome de classe, um dicionário de classe e uma lista de classes base. A metaclasse é responsável por receber estes três argumentos e criar a classe. A maioria das linguagens de programação orientadas a objetos provê uma implementação default. O que torna o Python especial é o fato de ser possível criar metaclasses personalizadas. A maioria dos usuários nunca vai precisar deste recurso, mas quando houver necessidade, metaclasses possibilitam soluções poderosas e elegantes. Metaclasses têm sido utilizadas para gerar registros de acesso a atributos, para incluir proteção contra acesso concorrente, rastrear a criação de objetos, implementar singletons, dentre muitas outras tarefas.

More information can be found in metaclasses.

method (método) Uma função que é definida dentro do corpo de uma classe. Se chamada como um atributo de uma instância daquela classe, o método receberá a instância do objeto como seu primeiro *argumento* (que comumente é chamado de `self`). Veja *função* e *nested scope*.

method resolution order (ordem de resolução de método) Ordem de resolução de métodos é a ordem em que os membros de uma classe base são buscados durante a pesquisa. Veja A ordem de resolução de métodos do Python 2.3.

módulo Um objeto que serve como uma unidade organizacional de código Python. Os módulos têm um namespace contendo objetos Python arbitrários. Os módulos são carregados pelo Python através do processo de *importação*.

Veja também *pacote*.

MRO Veja *method resolution order*.

mutável Objeto mutável é aquele que pode modificar seus valor mas manter seu `id()`. Veja também *immutable*.

tupla nomeada Any tuple-like class whose indexable elements are also accessible using named attributes (for example, `time.localtime()` returns a tuple-like object where the *year* is accessible either with an index such as `t[0]` or with a named attribute like `t.tm_year`).

A named tuple can be a built-in type such as `time.struct_time`, or it can be created with a regular class definition. A full featured named tuple can also be created with the factory function `collections.namedtuple()`. The latter approach automatically provides extra features such as a self-documenting representation like `Employee(name='jones', title='programmer')`.

namespace The place where a variable is stored. Namespaces are implemented as dictionaries. There are the local, global and built-in namespaces as well as nested namespaces in objects (in methods). Namespaces support modularity by preventing naming conflicts. For instance, the functions `__builtin__.open()` and `os.open()` are distinguished by their namespaces. Namespaces also aid readability and maintainability by making it clear which module implements a function. For instance, writing `random.seed()` or `itertools.izip()` makes it clear that those functions are implemented by the `random` and `itertools` modules, respectively.

nested scope (escopo aninhado) The ability to refer to a variable in an enclosing definition. For instance, a function defined inside another function can refer to variables in the outer function. Note that nested scopes work only for reference and not for assignment which will always write to the innermost scope. In contrast, local variables both read and write in the innermost scope. Likewise, global variables read and write to the global namespace.

new-style class (novo estilo de classes) Any class which inherits from `object`. This includes all built-in types like `list` and `dict`. Only new-style classes can use Python's newer, versatile features like `__slots__`, descriptors, properties, and `__getattribute__()`.

More information can be found in newstyle.

object (objeto) Qualquer dado que tenha estado (atributos ou valores) e comportamento definidos (métodos). Também a última classe base de qualquer *new-style class*.

pacote Um *module* Python é capaz de conter submódulos ou recursivamente, sub-pacotes. Tecnicamente, um pacote é um módulo Python com um atributo `__path__`.

parâmetro A named entity in a *function* (or method) definition that specifies an *argument* (or in some cases, arguments) that the function can accept. There are four types of parameters:

- *posicional-ou-nomeado*: especifica um argumento que pode ser tanto *posicional* quanto *nomeado*. Esse é o tipo padrão de parâmetro, por exemplo *foo* e *bar* a seguir:

```
def func(foo, bar=None): ...
```

- *positional-only*: specifies an argument that can be supplied only by position. Python has no syntax for defining positional-only parameters. However, some built-in functions have positional-only parameters (e.g. `abs()`).

- *var-posicional*: especifica quem uma sequência arbitrária de argumentos posicionais pode ser fornecida (em adição a qualquer argumento posicional já aceito por outros parâmetros). Tal parâmetro pode ser definido colocando um `*` antes do nome, por exemplo *args* a seguir:

```
def func(*args, **kwargs): ...
```

- *var-nomeado*: especifica que, arbitrariamente, muitos argumentos nomeados podem ser fornecidos (em adição a qualquer argumento nomeado já aceito por outros parâmetros). Tal parâmetro pode definido colocando-se `**` antes do nome, por exemplo *kwargs* no exemplo acima.

Parâmetros podem especificar tanto argumentos opcionais quanto obrigatórios, assim como valores padrões para alguns argumentos opcionais.

See also the *argument* glossary entry, the FAQ question on the difference between arguments and parameters, and the function section.

PEP Proposta de melhoria do Python. Uma PEP é um documento de design que fornece informação para a comunidade Python, ou descreve uma nova funcionalidade para o Python ou seus predecessores ou ambientes. PEPs devem prover uma especificação técnica concisa e um racional para funcionalidades propostas.

PEPs tem a intenção de ser os mecanismos primários para propor novas funcionalidades significativas, para coletar opiniões da comunidade sobre um problema, e para documentar as decisões de design que foram adicionadas ao Python. O autor da PEP é responsável por construir um consenso dentro da comunidade e documentar opiniões dissidentes.

Veja **PEP 1**.

positional argument (argumento posicional) Veja o *argument*.

Python 3000 Apelido para a versão do Python 3.x linha de lançamento (cunhado há muito tempo, quando o lançamento da versão 3 era algo em um futuro muito distante.) Esse termo possui a seguinte abreviação: “Py3k”.

Pythonic Uma ideia ou um pedaço de código que segue de perto os idiomas mais comuns da linguagem Python, ao invés de implementar códigos usando conceitos comuns a outros idiomas. Por exemplo, um idioma comum em Python é fazer um loop sobre todos os elementos de uma iterável usando a instrução `for`. Muitas outras linguagens não têm esse tipo de construção, então as pessoas que não estão familiarizadas com o Python usam um contador numérico:

```
for i in range(len(food)):
    print food[i]
```

Ao contrário do método limpo, ou então, Pythônico:

```
for piece in food:
    print piece
```

reference count O número de referências para um objeto. Quando a contagem de referências de um objeto atinge zero, ele é desalocado. Contagem de referências geralmente não é visível no código Python, mas é um elemento chave da implementação *CPython*. O módulo `sys` define a função `getrefcount()` que programadores podem chamar para retornar a contagem de referências para um objeto em particular.

__slots__ A declaration inside a *new-style class* that saves memory by pre-declaring space for instance attributes and eliminating instance dictionaries. Though popular, the technique is somewhat tricky to get right and is best reserved for rare cases where there are large numbers of instances in a memory-critical application.

sequência An *iterable* which supports efficient element access using integer indices via the `__getitem__()` special method and defines a `len()` method that returns the length of the sequence. Some built-in sequence types are `list`, `str`, `tuple`, and `unicode`. Note that `dict` also supports `__getitem__()` and `__len__()`, but is considered a mapping rather than a sequence because the lookups use arbitrary *immutable* keys rather than integers.

fatia An object usually containing a portion of a *sequence*. A slice is created using the subscript notation, `[]` with colons between numbers when several are given, such as in `variable_name[1:3:5]`. The bracket (subscript) notation uses `slice` objects internally (or in older versions, `__getslice__()` and `__setslice__()`).

método especial Um método que é chamado implicitamente pelo Python para executar uma certa operação em um tipo, como uma adição por exemplo. Tais métodos tem nomes iniciando e terminando com dois underscores. Métodos especiais estão documentados em specialnames.

instrução Uma instrução é parte de uma suíte (um "bloco" de código). Uma instrução é ou uma *expression* ou uma de várias construções com uma palavra-chave, tal como `if`, `while` ou `for`.

struct sequence A tuple with named elements. Struct sequences expose an interface similiar to *named tuple* in that elements can be accessed either by index or as an attribute. However, they do not have any of the named tuple methods like `_make()` or `_asdict()`. Examples of struct sequences include `sys.float_info` and the return value of `os.stat()`.

aspas triplas Uma string que está definida com três ocorrências de aspas duplas (") ou apóstrofes ('). Enquanto elas não fornecem nenhuma funcionalidade não disponível com strings de aspas simples, elas são úteis para inúmeras razões. Elas permitem que você inclua aspas simples e duplas não encerradas dentro de uma string, e elas podem utilizar múltiplas linhas sem o uso de caractere de continuação, fazendo-as especialmente úteis quando escrevemos documentação em docstrings.

type O tipo de um objeto Python determina qual tipo de objeto ele é; cada objeto tem um tipo. Um tipo de objeto é acessível pelo atributo `__class__` ou pode ser recuperado com `type(obj)`.

Novas linhas universais A manner of interpreting text streams in which all of the following are recognized as ending a line: the Unix end-of-line convention `'\n'`, the Windows convention `'\r\n'`, and the old Macintosh convention `'\r'`. See **PEP 278** and **PEP 3116**, as well as `str.splitlines()` for an additional use.

ambiente virtual Um ambiente de execução isolado que permite usuários Python e aplicações instalarem e atualizarem pacotes Python sem interferir no comportamento de outras aplicações Python em execução no mesmo sistema.

máquina virtual Um computador definido inteiramente em software. A máquina virtual de Python executa o *bytecode* emitido pelo compilador de bytecode.

Zen of Python Lista de princípios de projeto e filosofias do Python que são úteis para a compreensão e uso da linguagem. A lista é exibida quando se digita "`import this`" no console interativo.

APÊNDICE B

Sobre esses documentos

Esses documentos são gerados a partir de reStructuredText pelo Sphinx, um processador de documentos especificamente escrito para documentação do Python.

O desenvolvimento da documentação e de suas ferramentas é um esforço totalmente voluntário, como o Python em si. Se você quer contribuir, por favor dê uma olhada na página reporting-bugs para informações sobre como fazer. Novos voluntários são sempre bem vindos!

Agradecimentos especiais para:

- Fred L. Drake, Jr., o criador do primeiro conjunto de ferramentas para documentar o Python e escritor de boa parte do conteúdo;
- O projeto Docutils para criar reStructuredText e o pacote Docutils;
- Fredrik Lundh por seu projeto Referência Alternativa para Python do qual Sphinx teve muitas ideias boas.

B.1 Contribuidores da Documentação do Python

Muitas pessoas tem contribuído para a linguagem Python, sua biblioteca padrão e sua documentação. Veja Misc/ACKS na distribuição do código do Python para ver uma lista parcial de contribuidores.

Tudo isso só foi possível com o esforço e a contribuição da comunidade Python, por isso temos essa maravilhosa documentação – Obrigado a todos!

APÊNDICE C

História e Licença

C.1 História do software

O Python foi criado no início dos anos 1990 por Guido van Rossum na Stichting Mathematisch Centrum (CWI, veja https://www.cwi.nl/) na Holanda como um sucessor de uma linguagem chamada ABC. Guido continua a ser o principal autor de Python, embora inclua muitas contribuições de outros.

Em 1995, Guido continuou seu trabalho em Python na Corporação para Iniciativas Nacionais de Pesquisa (CNRI, veja https://www.cnri.reston.va.us/) em Reston, Virgínia, onde lançou várias versões do software.

Em maio de 2000, Guido e a equipe principal de desenvolvimento do Python mudaram-se para o BeOpen.com para formar a equipe BeOpen PythonLabs. Em outubro do mesmo ano, a equipe da PythonLabs mudou para a Digital Creations (agora Zope Corporation; veja https://www.zope.org/). Em 2001, formou-se a Python Software Foundation (PSF, ver https://www.python.org/psf/), uma organização sem fins lucrativos criada especificamente para possuir propriedade intelectual relacionada a Python. A Zope Corporation é um membro patrocinador do PSF.

Todas as versões do Python são de código aberto (consulte https://opensource.org/ para a definição de código aberto). Historicamente, a maioria, mas não todas, versões do Python também são compatíveis com GPL; a tabela abaixo resume os vários lançamentos.

Release	Derivado de	Ano	Proprietário	GPL compatível?
0.9.0 a 1.2	n/a	1991-1995	CWI	sim
1.3 a 1.5.2	1.2	1995-1999	CNRI	sim
1.6	1.5.2	2000	CNRI	não
2.0	1.6	2000	BeOpen.com	não
1.6.1	1.6	2001	CNRI	não
2.1	2.0+1.6.1	2001	PSF	não
2.0.1	2.0+1.6.1	2001	PSF	sim
2.1.1	2.1+2.0.1	2001	PSF	sim
2.1.2	2.1.1	2002	PSF	sim
2.1.3	2.1.2	2002	PSF	sim
2.2 e acima	2.1.1	2001-agora	PSF	sim

Nota: Compatível com GPL não significa que estamos distribuindo Python sob a GPL. Todas as licenças do Python, ao contrário da GPL, permitem distribuir uma versão modificada sem fazer alterações em código aberto. As licenças compatíveis com GPL possibilitam combinar o Python com outro software lançado sob a GPL; os outros não.

Graças aos muitos voluntários externos que trabalharam sob a direção de Guido para tornar esses lançamentos possíveis.

C.2 Termos e condições para acessar ou usar Python

C.2.1 PSF LICENSE AGREEMENT FOR PYTHON 2.7.18

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"),␣
↪and
   the Individual or Organization ("Licensee") accessing and otherwise using␣
↪Python
   2.7.18 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to␣
↪reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 2.7.18 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice␣
↪of
   copyright, i.e., "Copyright © 2001-2020 Python Software Foundation; All␣
↪Rights
   Reserved" are retained in Python 2.7.18 alone or in any derivative version
   prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 2.7.18 or any part thereof, and wants to make the
   derivative work available to others as provided herein, then Licensee␣
↪hereby
   agrees to include in any such work a brief summary of the changes made to␣
↪Python
   2.7.18.

4. PSF is making Python 2.7.18 available to Licensee on an "AS IS" basis.
   PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION␣
↪OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT␣
↪THE
   USE OF PYTHON 2.7.18 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.7.18
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT␣
↪OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.7.18, OR ANY␣
↪DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
```

```
6. This License Agreement will automatically terminate upon a material breach␣
↪of
   its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any␣
↪relationship
   of agency, partnership, or joint venture between PSF and Licensee.  This␣
↪License
   Agreement does not grant permission to use PSF trademarks or trade name in␣
↪a
   trademark sense to endorse or promote products or services of Licensee, or␣
↪any
   third party.

8. By copying, installing or otherwise using Python 2.7.18, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.
```

C.2.2 ACORDO DE LICENÇA DA BEOPEN.COM PARA PYTHON 2.0

CONTRATO DE LICENÇA DE FONTE ABERTA DO BEOPEN PYTHON VERSÃO 1

```
1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at
   160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization
   ("Licensee") accessing and otherwise using this software in source or binary
   form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement,
   BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license
   to reproduce, analyze, test, perform and/or display publicly, prepare derivative
   works, distribute, and otherwise use the Software alone or in any derivative
   version, provided, however, that the BeOpen Python License is retained in the
   Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis.
   BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING,
   MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF
   ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects
   by the law of the State of California, excluding conflict of law provisions.
   Nothing in this License Agreement shall be deemed to create any relationship of
   agency, partnership, or joint venture between BeOpen and Licensee.  This License
   Agreement does not grant permission to use BeOpen trademarks or trade names in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.  As an exception, the "BeOpen Python" logos available at
```

(continua na próxima página)

(continuação da página anterior)

```
   http://www.pythonlabs.com/logos.html may be used according to the permissions
   granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be
   bound by the terms and conditions of this License Agreement.
```

C.2.3 CONTRATO DE LICENÇA DA CNRI PARA O PYTHON 1.6.1

```
1. This LICENSE AGREEMENT is between the Corporation for National Research
   Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191
   ("CNRI"), and the Individual or Organization ("Licensee") accessing and
   otherwise using Python 1.6.1 software in source or binary form and its
   associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 1.6.1 alone or in any derivative version,
   provided, however, that CNRI's License Agreement and CNRI's notice of copyright,
   i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All
   Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version
   prepared by Licensee.  Alternately, in lieu of CNRI's License Agreement,
   Licensee may substitute the following text (omitting the quotes): "Python 1.6.1
   is made available subject to the terms and conditions in CNRI's License
   Agreement.  This Agreement together with Python 1.6.1 may be located on the
   Internet using the following unique, persistent identifier (known as a handle):
   1895.22/1013.  This Agreement may also be obtained from a proxy server on the
   Internet using the following URL: http://hdl.handle.net/1895.22/1013."

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 1.6.1 or any part thereof, and wants to make the derivative
   work available to others as provided herein, then Licensee hereby agrees to
   include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis.  CNRI
   MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF EXAMPLE,
   BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY
   OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF
   PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property
   law of the United States, including without limitation the federal copyright
   law, and, to the extent such U.S. federal law does not apply, by the law of the
   Commonwealth of Virginia, excluding Virginia's conflict of law provisions.
   Notwithstanding the foregoing, with regard to derivative works based on Python
   1.6.1 that incorporate non-separable material that was previously distributed
```

(continua na próxima página)

(continuação da página anterior)

```
   under the GNU General Public License (GPL), the law of the Commonwealth of
   Virginia shall govern this License Agreement only as to issues arising under or
   with respect to Paragraphs 4, 5, and 7 of this License Agreement.  Nothing in
   this License Agreement shall be deemed to create any relationship of agency,
   partnership, or joint venture between CNRI and Licensee.  This License Agreement
   does not grant permission to use CNRI trademarks or trade name in a trademark
   sense to endorse or promote products or services of Licensee, or any third
   party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing
   or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and
   conditions of this License Agreement.
```

C.2.4 ACORDO DE LICENÇA DA CWI PARA PYTHON 0.9.0 A 1.2

```
Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The
Netherlands.  All rights reserved.

Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted, provided that
the above copyright notice appear in all copies and that both that copyright
notice and this permission notice appear in supporting documentation, and that
the name of Stichting Mathematisch Centrum or CWI not be used in advertising or
publicity pertaining to distribution of the software without specific, written
prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT
OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE,
DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS
SOFTWARE.
```

C.3 Licenças e Reconhecimentos para Software Incorporado

Esta seção é uma lista incompleta, mas crescente, de licenças e confirmações para softwares de terceiros incorporados na distribuição do Python.

C.3.1 Mersenne Twister

O módulo `_random` inclui código baseado em um download de http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html. A seguir estão os comentários literais do código original:

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).
```

(continua na próxima página)

(continuação da página anterior)

```
Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 3. The names of its contributors may not be used to endorse or promote
    products derived from this software without specific prior written
    permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


Any feedback is very welcome.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
```

C.3.2 Sockets

O módulo `socket` usa as funções `getaddrinfo()` e `getnameinfo()`, que são codificadas em arquivos de origem separados do Projeto WIDE, http://www.wide.ad.jp/.

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.
```

(continua na próxima página)

(continuação da página anterior)

```
THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 Floating point exception control

The source for the `fpectl` module includes the following notice:

```
  ---------------------------------------------------------------------
 /                      Copyright (c) 1996.                            \
|          The Regents of the University of California.                 |
|                        All rights reserved.                           |
|                                                                       |
|   Permission to use, copy, modify, and distribute this software for   |
|   any purpose without fee is hereby granted, provided that this en-   |
|   tire notice is included in all copies of any software which is or   |
|   includes  a  copy  or  modification  of  this software and in all   |
|   copies of the supporting documentation for such software.           |
|                                                                       |
|   This  work was produced at the University of California, Lawrence   |
|   Livermore National Laboratory under  contract  no.  W-7405-ENG-48   |
|   between  the  U.S.  Department  of  Energy and The Regents of the   |
|   University of California for the operation of UC LLNL.              |
|                                                                       |
|                              DISCLAIMER                               |
|                                                                       |
|   This  software was prepared as an account of work sponsored by an   |
|   agency of the United States Government. Neither the United States   |
|   Government  nor the University of California nor any of their em-   |
|   ployees, makes any warranty, express or implied, or  assumes  any   |
|   liability  or  responsibility  for the accuracy, completeness, or   |
|   usefulness of any information,  apparatus,  product,  or  process   |
|   disclosed,   or  represents  that  its  use  would  not  infringe   |
|   privately-owned rights. Reference herein to any specific  commer-   |
|   cial  products,  process,  or  service  by trade name, trademark,   |
|   manufacturer, or otherwise, does not  necessarily  constitute  or   |
|   imply  its endorsement, recommendation, or favoring by the United   |
|   States Government or the University of California. The views  and   |
|   opinions  of authors expressed herein do not necessarily state or   |
|   reflect those of the United States Government or  the  University   |
|   of  California,  and shall not be used for advertising or product   |
 \  endorsement purposes.                                              /
  ---------------------------------------------------------------------
```

C.3.4 MD5 message digest algorithm

The source code for the `md5` module contains the following notice:

```
Copyright (C) 1999, 2002 Aladdin Enterprises.  All rights reserved.

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

L. Peter Deutsch
ghost@aladdin.com

Independent implementation of MD5 (RFC 1321).

This code implements the MD5 Algorithm defined in RFC 1321, whose
text is available at
      http://www.ietf.org/rfc/rfc1321.txt
The code is derived from the text of the RFC, including the test suite
(section A.5) but excluding the rest of Appendix A.  It does not include
any code or documentation that is identified in the RFC as being
copyrighted.

The original and principal author of md5.h is L. Peter Deutsch
<ghost@aladdin.com>.  Other authors are noted in the change history
that follows (in reverse chronological order):

2002-04-13 lpd Removed support for non-ANSI compilers; removed
      references to Ghostscript; clarified derivation from RFC 1321;
      now handles byte order either statically or dynamically.
1999-11-04 lpd Edited comments slightly for automatic TOC extraction.
1999-10-18 lpd Fixed typo in header comment (ansi2knr rather than md5);
      added conditionalization for C++ compilation from Martin
      Purschke <purschke@bnl.gov>.
1999-05-03 lpd Original version.
```

C.3.5 Serviços de soquete assíncrono

Os módulos `asynchat` e `asyncore` contêm o seguinte aviso:

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

C.3.6 Gerenciamento de cookies

The `Cookie` module contains the following notice:

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

               All Rights Reserved

Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Timothy O'Malley  not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3.7 Rastreamento de execução

O módulo `trace` contém o seguinte aviso:

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err...  reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.


Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

C.3.8 Funções UUencode e UUdecode

O módulo `uu` contém o seguinte aviso:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                       All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
```

(continua na próxima página)

(continuação da página anterior)

```
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.9 Chamadas de Procedimento Remoto XML

The `xmlrpclib` module contains the following notice:

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice appears in
all copies, and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
ABILITY AND FITNESS.  IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

C.3.10 test_epoll

The `test_epoll` contains the following notice:

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
```

(continua na próxima página)

(continuação da página anterior)

```
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.11 Selecione o kqueue

The `select` and contains the following notice for the kqueue interface:

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.12 strtod e dtoa

O arquivo `Python/dtoa.c`, que fornece as funções C dtoa e strtod para conversão de duplas de C para e de strings, é derivado do arquivo com o mesmo nome de David M. Gay, atualmente disponível em http://www.netlib.org/fp/. O arquivo original, conforme recuperado em 16 de março de 2009, contém os seguintes avisos de direitos autorais e de licenciamento:

```
/****************************************************************
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
```

(continua na próxima página)

(continuação da página anterior)

```
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 ***********************************************************/
```

C.3.13 OpenSSL

Os módulos `hashlib`, `posix`, `ssl`, `crypt` usam a biblioteca OpenSSL para desempenho adicional se forem disponibilizados pelo sistema operacional. Além disso, os instaladores do Windows e do Mac OS X para Python podem incluir uma cópia das bibliotecas do OpenSSL, portanto incluímos uma cópia da licença do OpenSSL aqui:

```
LICENSE ISSUES
==============

The OpenSSL toolkit stays under a dual license, i.e. both the conditions of
the OpenSSL License and the original SSLeay license apply to the toolkit.
See below for the actual license texts. Actually both licenses are BSD-style
Open Source licenses. In case of any license issues related to OpenSSL
please contact openssl-core@openssl.org.

OpenSSL License
---------------

  /* ====================================================================
   * Copyright (c) 1998-2008 The OpenSSL Project.  All rights reserved.
   *
   * Redistribution and use in source and binary forms, with or without
   * modification, are permitted provided that the following conditions
   * are met:
   *
   * 1. Redistributions of source code must retain the above copyright
   *    notice, this list of conditions and the following disclaimer.
   *
   * 2. Redistributions in binary form must reproduce the above copyright
   *    notice, this list of conditions and the following disclaimer in
   *    the documentation and/or other materials provided with the
   *    distribution.
   *
   * 3. All advertising materials mentioning features or use of this
   *    software must display the following acknowledgment:
   *    "This product includes software developed by the OpenSSL Project
   *    for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
   *
   * 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
   *    endorse or promote products derived from this software without
   *    prior written permission. For written permission, please contact
   *    openssl-core@openssl.org.
   *
   * 5. Products derived from this software may not be called "OpenSSL"
   *    nor may "OpenSSL" appear in their names without prior written
   *    permission of the OpenSSL Project.
   *
   * 6. Redistributions of any form whatsoever must retain the following
```

(continua na próxima página)

(continuação da página anterior)

```
   *    acknowledgment:
   *    "This product includes software developed by the OpenSSL Project
   *    for use in the OpenSSL Toolkit (http://www.openssl.org/)"
   *
   * THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
   * EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
   * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
   * PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE OpenSSL PROJECT OR
   * ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
   * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
   * NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
   * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
   * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
   * STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
   * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
   * OF THE POSSIBILITY OF SUCH DAMAGE.
   * ====================================================================
   *
   * This product includes cryptographic software written by Eric Young
   * (eay@cryptsoft.com).  This product includes software written by Tim
   * Hudson (tjh@cryptsoft.com).
   *
   */

Original SSLeay License
-----------------------

  /* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
   * All rights reserved.
   *
   * This package is an SSL implementation written
   * by Eric Young (eay@cryptsoft.com).
   * The implementation was written so as to conform with Netscapes SSL.
   *
   * This library is free for commercial and non-commercial use as long as
   * the following conditions are aheared to.  The following conditions
   * apply to all code found in this distribution, be it the RC4, RSA,
   * lhash, DES, etc., code; not just the SSL code.  The SSL documentation
   * included with this distribution is covered by the same copyright terms
   * except that the holder is Tim Hudson (tjh@cryptsoft.com).
   *
   * Copyright remains Eric Young's, and as such any Copyright notices in
   * the code are not to be removed.
   * If this package is used in a product, Eric Young should be given attribution
   * as the author of the parts of the library used.
   * This can be in the form of a textual message at program startup or
   * in documentation (online or textual) provided with the package.
   *
   * Redistribution and use in source and binary forms, with or without
   * modification, are permitted provided that the following conditions
   * are met:
   * 1. Redistributions of source code must retain the copyright
   *    notice, this list of conditions and the following disclaimer.
   * 2. Redistributions in binary form must reproduce the above copyright
   *    notice, this list of conditions and the following disclaimer in the
   *    documentation and/or other materials provided with the distribution.
   * 3. All advertising materials mentioning features or use of this software
```

(continua na próxima página)

(continuação da página anterior)

```
   *    must display the following acknowledgement:
   *    "This product includes cryptographic software written by
   *     Eric Young (eay@cryptsoft.com)"
   *    The word 'cryptographic' can be left out if the rouines from the library
   *    being used are not cryptographic related :-).
   * 4. If you include any Windows specific code (or a derivative thereof) from
   *    the apps directory (application code) you must include an acknowledgement:
   *    "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
   *
   * THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
   * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
   * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
   * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
   * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
   * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
   * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
   * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
   * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
   * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
   * SUCH DAMAGE.
   *
   * The licence and distribution terms for any publically available version or
   * derivative of this code cannot be changed.  i.e. this code cannot simply be
   * copied and put under another distribution licence
   * [including the GNU Public Licence.]
   */
```

C.3.14 expat

A extensão `pyexpat` é construída usando uma cópia incluída das fontes de expatriadas, a menos que a compilação esteja configurada `--with-system-expat`:

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                               and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.15 libffi

A extensão `_ctypes` é construída usando uma cópia incluída das fontes libffi, a menos que a compilação esteja configurada `--with-system-libffi`:

```
Copyright (c) 1996-2008  Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
``Software''), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ``AS IS'', WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT.  IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

C.3.16 zlib

A extensão `zlib` é construída usando uma cópia incluída das fontes zlib se a versão do zlib encontrada no sistema for muito antiga para ser usada na construção:

```
Copyright (C) 1995-2010 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

APÊNDICE D

Copyright

Python e essa documentação é:

Veja: *História e Licença* para informações completas de licença e permissões.

Índice

Não alfabético

A

B

C

D

E

F

G

H

S

T

U

V

Z