Python Tutorial

Release 3.13.2

Guido van Rossum and the Python development team

março 11, 2025

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 sintaxe elegante e a tipagem dinâmica do Python, aliadas com sua natureza interpretativa, o tornam uma linguagem ideal para fazer scripts e desenvolvimento de aplicações rápidas em diversas áreas e na maioria das plataformas.

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

O interpretador do Python pode ser facilmente estendido com novas funções e tipos de dados implementados em C ou C++ (ou outras linguagens chamáveis a partir de C). Python também é adequado 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 bastante com computadores, eventualmente perceberá que há alguma tarefa que gostaria de automatizar. Por exemplo, pode querer realizar uma busca e substituição em um grande número de arquivos de texto ou renomear e reorganizar um conjunto de fotos de maneira complexa. Talvez deseje criar um pequeno banco de dados personalizado, uma aplicação GUI especializada 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 pode escrever um script shell do Unix ou arquivos batch do Windows para algumas dessas tarefas, mas os scripts shell são melhores para mover arquivos e alterar dados de texto, não são adequados para aplicações GUI ou jogos. Você pode escrever um programa C/C++/Java, mas pode levar muito tempo de desenvolvimento para obter até mesmo um primeiro rascunho de programa. O Python é mais simples de usar, está disponível nos sistemas operacionais Windows, macOS e Unix e ajudará você a terminar 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 ligação (*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 módulos ou funções embutidas 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 se entusiasmou com o Python, vai querer conhecê-lo com mais detalhes. Partindo do princípio que a melhor maneira de aprender uma linguagem é usando-a, este tutorial lhe convida para fazê-lo com o interpretador do Python enquanto você o lê.

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, instruções, 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

O interpretador Python é frequentemente instalado como `/usr/local/bin/python3.13` nas máquinas onde está disponível; adicionando `/usr/local/bin` ao caminho de busca da shell de seu Unix torna-se possível iniciá-lo digitando o comando:

```
python3.13
```

no shell.[1] Considerando que a escolha do diretório onde o interpretador está é uma opção de instalação, outros locais são possíveis; verifique com seu guru local de Python ou administrador do sistema local. (Por exemplo, `/usr/local/python` é um local alternativo popular.)

Em máquinas Windows onde você instalou Python a partir da Microsoft Store, o comando `python3.13` estará disponível. Se você tem o lançador py.exe instalado, você pode usar o comando `py`. Veja setting-envvars para outras maneiras de executar o Python.

Digitar 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()`.

Os recursos de edição de linha do interpretador incluem edição interativa, substituição de histórico e complemento de código, em sistemas com suporte à biblioteca GNU Readline. Talvez a verificação mais rápida para ver se o suporte à edição de linha de comando está disponível é digitando `Control-P` no primeiro prompt oferecido pelo Python. Se for emitido um bipe, você terá a edição da linha de comando; veja Apêndice *Edição de entrada interativa e substituição de histórico* para uma introdução às combinações. Se nada acontecer, ou se `^P` aparecer na tela, a edição da linha de comando não está disponível; você só poderá usar backspace para remover caracteres da linha atual.

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 iniciar o interpretador é `python -c comando [arg] ...`, que executa uma ou mais instruções especificadas 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 todo o *comando* esteja entre as aspas.

[1] No Unix, o interpretador Python 3.x não é instalado por padrão com o executável nomeado `python`, então não vai conflitar com um executável Python 2.x instalado simultaneamente.

Alguns módulos Python são também úteis como scripts. Estes podem ser chamados usando `python -m módulo [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.

Todas as opções de linha de comando são descritas em 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:

```
$ python3.13
Python 3.13 (default, April 4 2023, 09:25:04)
[GCC 10.2.0] on linux
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`:

```
>>> o_mundo_é_plano = True
>>> if o_mundo_é_plano:
...     print("Cuidado para não cair da borda!")
...
Cuidado para não cair da borda!
```

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

2.2 O interpretador e seu ambiente

2.2.1 Edição de código-fonte

Por padrão, arquivos fonte de Python são tratados com codificação UTF-8. Nessa codificação, caracteres de muitos idiomas no mundo podem ser usados simultaneamente em literais string, identificadores e comentários — embora a biblioteca padrão use apenas caracteres ASCII para identificadores, uma convenção que qualquer código portável deve seguir. Para exibir todos esses caracteres corretamente, seu editor deve reconhecer que o arquivo é UTF-8 e deve usar uma fonte com suporte a todos os caracteres no arquivo.

Para declarar uma codificação diferente da padrão, uma linha de comentário especial deve ser adicionada como *primeira* linha do arquivo. A sintaxe é essa:

```
# -*- 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 python3
# -*- 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 exemplos neste manual, mesmo aqueles inscritos na linha de comando interativa, incluem comentários. Comentários em Python começam com o caractere cerquilha `#` e estende até o final da linha. Um comentário pode aparecer no inicio da linha ou após espaço em branco ou código, mas não dentro de uma string literal. O caractere cerquilha dentro de uma string literal é apenas uma cerquilha. Como os comentários são para esclarecer o código e não são interpretados pelo Python, eles podem ser omitidos ao digitar exemplos.

Alguns exemplos:

```
# este é o primeiro comentário
spam = 1  # e este é o segundo comentário
          # ... e agora um terceiro!
texto = "# Este não é um comentário por estar entre aspas."
```

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 `/` podem ser usadas para realizar operações aritméticas; parênteses (`()`) podem ser usados para agrupar expressões. Por exemplo:

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

(continua na próxima página)

(continuação da página anterior)

```
>>> 8 / 5  # divisão sempre retorna um número de ponto flutuante
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.

Divisão (`/`) sempre retorna ponto flutuante (float). Para fazer uma *divisão pelo piso* e receber um inteiro como resultado você pode usar o operador `//`; para calcular o resto você pode usar o `%`:

```
>>> 17 / 3  # divisão clássica retorna um ponto flutuante
5.666666666666667
>>>
>>> 17 // 3  # divisão pelo piso descarta a parte fracionária
5
>>> 17 % 3  # o operador % retorna o resto da divisão
2
>>> 5 * 3 + 2  # quociente do piso * divisor + restante
17
```

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

```
>>> 5 ** 2  # 5 ao quadrado
25
>>> 2 ** 7  # 2 à potência de 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:

```
>>> largura = 20
>>> altura = 5 * 9
>>> largura * altura
900
```

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

```
>>> n  # tenta acessar uma variável não definida
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:

```
>>> 4 * 3.75 - 1
14.0
```

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:

```
>>> taxa = 12.5 / 100
>>> preço = 100.50
>>> preço * taxa
12.5625
>>> preço + _
113.0625
```

(continua na próxima página)

[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`.

(continuação da página anterior)

```
>>> 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 Texto

Python pode manipular texto (representado pelo tipo `str`, também chamado de "strings"), bem como números. Isso inclui caracteres "`!`", palavras "`coelho`", nomes "`Paris`", frases "`Eu te protejo.`", etc. "`Oba! :)`". Eles podem ser colocados entre aspas simples (`'...'`) ou aspas duplas (`"..."`) com o mesmo resultado[2].

```
>>> 'spam eggs'  # single quotes
'spam eggs'
>>> "Paris rabbit got your back :)! Yay!"  # double quotes
'Paris rabbit got your back :)! Yay!'
>>> '1975'  # digits and numerals enclosed in quotes are also strings
'1975'
```

Para colocar aspas entre aspas, precisamos "escapá-la", precedendo-as com `\`. Alternativamente, podemos usar o outro tipo de aspas:

```
>>> 'd\'água'  # use \' para escapar a aspa simples...
"d'água"
>>> "d'água"  # ...ou use aspas duplas
"d'água"
>>> '"Sim", eles disseram.'
'"Sim", eles disseram.'
>>> "\"Sim\", eles disseram."
'"Sim", eles disseram.'
>>> '"Copo d\'água", eles pediram.'
'"Copo d'água", eles pediram.'
```

No shell do Python, a definição de string e a string de saída podem parecer diferentes. A função `print()` produz uma saída mais legível, omitindo as aspas delimitadoras e imprimindo caracteres de escape e especiais:

```
>>> s = 'Primeira linha.\nSegunda linha.'  # \n significa nova linha
>>> s  # sem print(), caracteres especiais são incluídos na string
'Primeira linha.\nSegunda linha.'
>>> print(s)  # com print(), caracteres especiais são interpretados, então \n␣
↪produz nova linha
Primeira linha.
Segunda linha.
```

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:\algum\nome')  # aqui \n significa nova linha!
C:\algum
```

(continua na próxima página)

[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.

(continuação da página anterior)

```
ome
>>> print(r'C:\algum\nome')  # observe o r antes das aspas
C:\algum\nome
```

Há um aspecto sutil nas strings brutas: uma string bruta não pode terminar em um número ímpar de caracteres \; consulte o FAQ relacionado para mais informações e soluções alternativas.

As strings literais podem abranger várias linhas. Uma maneira é usar as aspas triplas: """...""" ou '''...'''. Caracteres de fim de linha são incluídos automaticamente na string, mas é possível evitar isso adicionando uma \ no final. No exemplo a seguir, a nova linha no início não é incluída:

```
>>> print("""\
... Uso: coisinha [OPÇÕES]
...      -h                        Exibe esta mensagem de uso
...      -H hostname               Hostname para se conectar
... """)
Usage: coisinha [OPÇÕES]
     -h                        Exibe esta mensagem de ajuda
     -H hostname               Hostname para se conectar

>>>
```

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

```
>>> # 3 vezes 'un', seguido por '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:

```
>>> texto = ('Coloque várias strings dentro de parênteses '
...          'para fazer com que elas sejam concatenadas.')
>>> texto
'Coloque várias strings dentro de parênteses para fazer com que elas sejam␣
↪concatenadas.'
```

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

```
>>> prefixo = 'Py'
>>> prefixo 'thon'  # não é possível concatenar uma variável e um literal de string
  File "<stdin>", line 1
    prefixi 'thon'
            ^^^^^^
SyntaxError: invalid syntax
>>> ('un' * 3) 'ium'
  File "<stdin>", line 1
    ('un' * 3) 'ium'
               ^^^^^
SyntaxError: invalid syntax
```

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

```
>>> prefixo + '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:

```
>>> palavra = 'Python'
>>> palavra[0]  # caractere na posição 0
'P'
>>> palavra[5]  # caractere na posição 5
'n'
```

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

```
>>> palavra[-1]  # último caractere
'n'
>>> palavra[-2]  # penúltimo caractere
'o'
>>> palavra[-6]
'P'
```

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

Além da indexação, o *fatiamento* também é permitido. Embora a indexação seja usada para obter caracteres individuais, *fatiar* permite que você obtenha uma substring:

```
>>> palavra[0:2]  # caracteres da posição 0 (incluída) até 2 (excluída)
'Py'
>>> palavra[2:5]  # caracteres da posição 2 (incluída) até 5 (excluída)
'tho'
```

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:

```
>>> palavra[:2]   # caracteres do início até a posição 2 (excluída)
'Py'
>>> palavra[4:]   # caracteres da posição 4 (incluída) até o fim
'on'
>>> palavra[-2:]  # caracteres da penúltima (incluída) até o fim
'on'
```

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`:

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

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:

```
>>> palavra[42]  # a palavra só tem 6 caracteres
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:

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

```
>>> palavra[0] = 'J'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> palavra[2:] = 'py'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

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

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

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

```
>>> s = 'supercalifragilisticexpialidoce'
>>> len(s)
31
```

Ver também

textseq
: As strings são exemplos de *tipos de sequências* e suportam as operações comumente suportadas por esses tipos.

string-methods
: As strings suportam uma grande quantidade de métodos para transformações básicas e busca.

f-strings
: Strings literais que possuem expressões embutidas.

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

old-string-formatting
As antigas operações de formatação invocadas quando as strings são o operando esquerdo do operador `%` são descritas com mais detalhes aqui.

3.1.3 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.

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

Como strings (e todos os tipos embutidos de *sequência*), listas pode ser indexados e fatiados:

```
>>> quadrados[0]  # indexação retorna o item
1
>>> quadrados[-1]
25
>>> quadrados[-3:]  # fatiamento retorna uma nova lista
[9, 16, 25]
```

As listas também suportam operações como concatenação:

```
>>> quadrados + [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:

```
>>> cubos = [1, 8, 27, 65, 125]  # algo errado aqui
>>> 4 ** 3  # o cubo de 4 é 64, não 65!
64
>>> cubos[3] = 64  # substitui o valor errado
>>> cubos
[1, 8, 27, 64, 125]
```

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

```
>>> cubos.append(216)  # adiciona o cubo de 6
>>> cubos.append(7 ** 3)  # e o cubo de 7
>>> cubos
[1, 8, 27, 64, 125, 216, 343]
```

A atribuição simples em Python nunca copia dados. Quando você atribui uma lista a uma variável, a variável se refere à *lista existente*. Quaisquer alterações que você fizer na lista por meio de uma variável serão vistas por todas as outras variáveis que se referem a ela:

```
>>> rgb = ["Vermelho", "Verde", "Azul"]
>>> rgba = rgb
>>> id(rgb) == id(rgba)  # elas referenciam o mesmo objeto
True
>>> rgba.append("Alf")
```

(continua na próxima página)

(continuação da página anterior)

```
>>> rgb
["Vermelho", "Verde", "Azul", "Alf"]
```

Todas as operações de fatiamento devolvem uma nova lista contendo os elementos solicitados. Isso significa que o seguinte fatiamento devolve uma cópia rasa da lista:

```
>>> rgba_correto = rgba[:]
>>> rgba_correto[-1] = "Alfa"
>>> rgba_correto
["Vermelho", "Verde", "Azul", "Alfa"]
>>> rgba
["Vermelho", "Verde", "Azul", "Alf"]
```

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

```
>>> letras = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letras
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # substitui alguns valores
>>> letras[2:5] = ['C', 'D', 'E']
>>> letras
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # agora remove-os
>>> letras[2:5] = []
>>> letras
['a', 'b', 'f', 'g']
>>> # limpa a lista substituindo todos os elementos por uma lista vazia
>>> letras[:] = []
>>> letras
[]
```

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

```
>>> letras = ['a', 'b', 'c', 'd']
>>> len(letras)
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]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

3.2 Primeiros passos para a programação

Claro, podemos usar o Python para tarefas mais complicadas do que somar 2+2. Por exemplo, podemos escrever o início da sequência de Fibonacci assim:

```
>>> # Sequência de Fibonacci:
>>> # a soma de dois elementos define a próxima
```

(continua na próxima página)

(continuação da página anterior)

```
>>> a, b = 0, 1
>>> while a < 10:
...     print(a)
...     a, b = b, a+b
...
0
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.
- O laço de repetição `while` executa enquanto a condição (aqui: `a < 10`) permanece verdadeira. Em Python, como em C, qualquer valor inteiro que não seja zero é considerado verdadeiro; zero é considerado falso. A condição pode também ser uma cadeia de caracteres ou uma lista, ou qualquer sequência; qualquer coisa com um tamanho maior que zero é verdadeiro, enquanto sequências vazias são falsas. O teste usado no exemplo é uma comparação simples. Os operadores padrões de comparação são os mesmos de C: < (menor que), > (maior que), == (igual), <= (menor ou igual), >= (maior ou igual) e != (diferente).
- 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.
- A função `print()` escreve o valor dos argumentos fornecidos. É diferente de apenas escrever a expressão no interpretador (como fizemos anteriormente nos exemplos da calculadora) pela forma como lida com múltiplos argumentos, quantidades de ponto flutuante e strings. As strings são impressas sem aspas, e um espaço é inserido entre os itens, assim você pode formatar bem o resultado, dessa forma:

```
>>> i = 256*256
>>> print('O valor de i é', i)
O valor de i é 65536
```

O argumento nomeado *end* pode ser usado para evitar uma nova linha após a saída ou finalizar a saída com uma string diferente:

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

CAPÍTULO 4

Mais ferramentas de controle de fluxo

Assim como a instrução `while` que acabou de ser apresentada, o Python usa mais algumas que encontraremos neste capítulo.

4.1 Instruções `if`

Provavelmente a mais conhecida instrução de controle de fluxo é o `if`. Por exemplo:

```
>>> x = int(input("Insira um número inteiro: "))
Insira um número inteiro: 42
>>> if x < 0:
...     x = 0
...     print('Negativo alterado para zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Um')
... else:
...     print('Mais')
...
Mais
```

Pode haver zero ou mais partes `elif`, e a parte `else` é opcional. A palavra-chave '`elif`' é uma abreviação para 'else if', e é útil para evitar indentação excessiva. Uma sequência `if` … `elif` … `elif` … substitui as instruções `switch` ou `case`, encontrados em outras linguagens.

Se você está comparando o mesmo valor com várias constantes, ou verificando por tipos ou atributos específicos, você também pode achar a instrução `match` útil. Para mais detalhes veja *Instruções match*.

4.2 Instruções `for`

A instrução `for` em Python é um pouco diferente do que costuma ser em C ou Pascal. Ao invés de sempre iterar sobre uma progressão aritmética de números (como no Pascal), ou permitir ao usuário definir o passo de iteração e a condição de parada (como C), a instrução `for` do Python itera sobre os itens de qualquer sequência (seja uma lista ou uma string), na ordem que aparecem na sequência. Por exemplo:

```
>>> # Mede algumas strings:
>>> palavras = ['gato', 'janela', 'defenestrar']
>>> for p in palavras:
...     print(p, len(p))
...
gato 4
janela 6
defenestrar 11
```

Código que modifica uma coleção sobre a qual está iterando pode ser inseguro. No lugar disso, usualmente você deve iterar sobre uma cópia da coleção ou criar uma nova coleção:

```
# Cria uma amostra de coleção
users = {'Hans': 'active', 'Éléonore': 'inactive', '□□□': 'active'}

# Estratégia: iterar por uma cópia
for user, status in users.copy().items():
    if status == 'inactive':
        del users[user]

# Estratégia: criar uma nova coleção
active_users = {}
for user, status in users.items():
    if status == 'active':
        active_users[user] = status
```

4.3 A função `range()`

Se você precisa iterar sobre sequências numéricas, a função embutida `range()` é a resposta. Ela gera progressões aritméticas:

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

O ponto de parada fornecido nunca é incluído na lista; `range(10)` gera uma lista com 10 valores, exatamente os índices válidos para uma sequência de comprimento 10. É possível iniciar o intervalo com outro número, ou alterar a razão da progressão (inclusive com passo negativo):

```
>>> list(range(5, 10))
[5, 6, 7, 8, 9]

>>> list(range(0, 10, 3))
[0, 3, 6, 9]

>>> list(range(-10, -100, -30))
[-10, -40, -70]
```

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

```
>>> a = ['Maria', 'tinha', 'um', 'carneirinho']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Maria
1 tinha
2 um
3 carneirinho
```

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

Uma coisa estranha acontece se você imprime um intervalo:

```
>>> range(10)
range(0, 10)
```

Em muitos aspectos, o objeto retornado pela função `range()` se comporta como se fosse uma lista, mas na verdade não é. É um objeto que retorna os itens sucessivos da sequência desejada quando você itera sobre a mesma, mas na verdade ele não gera a lista, economizando espaço.

Dizemos que um objeto é *iterável*, isso é, candidato a ser alvo de uma função ou construção que espera alguma coisa capaz de retornar sucessivamente seus elementos um de cada vez. Nós vimos que a instrução `for` é um exemplo de construção, enquanto que um exemplo de função que recebe um iterável é `sum()`:

```
>>> sum(range(4))  # 0 + 1 + 2 + 3
6
```

Mais tarde, veremos mais funções que retornam iteráveis e tomam iteráveis como argumentos. No capítulo *Estruturas de dados*, iremos discutir em mais detalhes sobre `list()`.

4.4 Instruções `break` e `continue`

A instrução `break` sai imediatamente do laço de repetição mais interno, seja `for` ou `while`:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(f"{n} igual a {x} * {n//x}")
...             break
...
4 igual a 2 * 2
6 igual a 2 * 3
8 igual a 2 * 4
9 equals 3 * 3
```

A instrução `continue` continua com a próxima iteração do laço:

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print(f"Encontrado um número par {num}")
...         continue
...     print(f"Encontrado um número ímpar {num}")
...
Encontrado um número par 2
Encontrado um número ímpar 3
Encontrado um número par 4
Encontrado um número ímpar 5
Encontrado um número par 6
```

(continua na próxima página)

(continuação da página anterior)

```
Encontrado um número ímpar 7
Encontrado um número par 8
Encontrado um número ímpar 9
```

4.5 Cláusulas `else` em laços

Em um laço `for` ou `while` a instrução `break` pode ser pareada com uma cláusula `else`. Se o laço terminar sem executar o `break`, a cláusula `else` será executada.

Em um laço `for`, a cláusula `else` é executada após o laço finalizar sua iteração final, ou seja, se não ocorrer nenhuma interrupção.

Em um laço `while`, ele é executado após a condição do laço se tornar falsa.

Em qualquer tipo de laço, a cláusula `else` **não** é executada se o laço foi encerrado por um `break`. Claro, outras maneiras de encerrar o laço mais cedo, como um `return` ou uma exceção levantada, também pularão a execução da cláusula `else`.

Isso é exemplificado no seguinte laço `for`, que procura por números primos:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'igual a', x, '*', n//x)
...             break
...     else:
...         # a iteração passou direto sem encontrar um fator
...         print(n, 'é um número primo')
...
2 é um número primo
3 é um número primo
4 igual a 2 * 2
5 é um número primo
6 igual a 2 * 3
7 é um número primo
8 igual a 2 * 4
9 igual a 3 * 3
```

(Sim, este é o código correto. Observe atentamente: a cláusula `else` pertence ao laço `for`, **não** à instrução `if`.)

Uma maneira de pensar na cláusula else é imaginá-la pareada com o `if` dentro do laço. Conforme o laço é executado, ele executará uma sequência como if/if/if/else. O `if` está dentro do laço, encontrado várias vezes. Se a condição for verdadeira, um `break` acontecerá. Se a condição nunca for verdadeira, a cláusula `else` fora do laçoserá executada.

Quando usado em um laço, a cláusula `else` tem mais em comum com a cláusula `else` de uma instrução `try` do que com a de instruções `if`: a cláusula `else` de uma instrução `try` é executada quando não ocorre exceção, e a cláusula `else` de um laço é executada quando não ocorre um `break`. Para mais informações sobre instrução `try` e exceções, veja *Tratamento de exceções*.

4.6 Instruções `pass`

A instrução `pass` não faz nada. Pode ser usada quando a sintaxe exige uma instrução, mas a semântica do programa não requer nenhuma ação. Por exemplo:

```
>>> while True:
...     pass  # Ocupado, aguardando interrupção por teclado (Ctrl+C)
...
```

Isto é usado muitas vezes para se definir classes mínimas:

```
>>> class MinhaClasseVazia:
...     pass
...
```

Outra ocasião em que o `pass` pode ser usado é como um substituto temporário para uma função ou bloco condicional, quando se está trabalhando com código novo, ainda indefinido, permitindo que mantenha-se o pensamento num nível mais abstrato. O `pass` é silenciosamente ignorado:

```
>>> def initlog(*args):
...     pass   # Lembre-se de implementar isso!
...
```

4.7 Instruções `match`

Uma instrução `match` pega uma expressão e compara seu valor com padrões sucessivos fornecidos como um ou mais blocos de case. Isso é superficialmente semelhante a uma instrução switch em C, Java ou JavaScript (e muitas outras linguagens), mas também pode extrair componentes (elementos de sequência ou atributos de objeto) do valor em variáveis, mas muito mais parecido com a correspondência de padrões em linguages como Rust ou Haskell. Apenas o primeiro padrão que corresponder será executado, podendo também extrair componentes (elementos de sequência ou atributos de objetos) do valor para variáveis.

A forma mais simples compara um valor de assunto com um ou mais literais:

```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
        case _:
            return "Something's wrong with the internet"
```

Observe o último bloco: o "nome da variável" _ atua como um *curinga* e nunca falha em corresponder. Se nenhum caso corresponder, nenhuma das ramificações será executada.

Você pode combinar vários literais em um único padrão usando `|` ("ou"):

```
case 401 | 403 | 404:
    return "Não permitido"
```

Os padrões podem se parecer com atribuições de desempacotamento e podem ser usados para vincular variáveis:

```
# ponto é uma tupla (x, y)
match ponto:
    case (0, 0):
        print("Origem")
    case (0, y):
        print(f"Y={y}")
    case (x, 0):
        print(f"X={x}")
    case (x, y):
        print(f"X={x}, Y={y}")
    case _:
        raise ValueError("Não é um ponto")
```

Estude isso com cuidado! O primeiro padrão tem dois literais e pode ser considerado uma extensão do padrão literal mostrado acima. Mas os próximos dois padrões combinam um literal e uma variável, e a variável *vincula* um valor do assunto (`ponto`). O quarto padrão captura dois valores, o que o torna conceitualmente semelhante à atribuição de desempacotamento `(x, y) = ponto`.

Se você estiver usando classes para estruturar seus dados, você pode usar o nome da classe seguido por uma lista de argumentos semelhante a um construtor, mas com a capacidade de capturar atributos em variáveis:

```
class Ponto:
    def __init__(self, x, y):
        self.x = x
        self.y = y

def onde_está(ponto):
    match ponto:
        case Ponto(x=0, y=0):
            print("Origem")
        case Ponto(x=0, y=y):
            print(f"Y={y}")
        case Ponto(x=x, y=0):
            print(f"X={x}")
        case Ponto():
            print("Em outro lugar")
        case _:
            print("Não é um ponto")
```

Você pode usar parâmetros posicionais com algumas classes embutidas que fornecem uma ordem para seus atributos (por exemplo, classes de dados). Você também pode definir uma posição específica para atributos em padrões configurando o atributo especial `__match_args__` em suas classes. Se for definido como ("x", "y"), os seguintes padrões são todos equivalentes (e todos vinculam o atributo `y` à variável `var`):

```
Ponto(1, var)
Ponto(1, y=var)
Ponto(x=1, y=var)
Ponto(y=var, x=1)
```

Uma maneira recomendada de ler padrões é vê-los como uma forma estendida do que você colocaria à esquerda de uma atribuição, para entender quais variáveis seriam definidas para quê. Apenas os nomes autônomos (como `var` acima) são atribuídos por uma instrução de correspondência. Nomes pontilhados (como `foo.bar`), nomes de atributos (o `x=` e `y=` acima) ou nomes de classes (reconhecidos pelo "(…)" próximo a eles, como `Ponto` acima) nunca são atribuídos.

Os padrões podem ser aninhados arbitrariamente. Por exemplo, se tivermos uma pequena lista de Pontos, com `__match_args__` adicionado, poderíamos correspondê-la assim:

```
class Ponto:
    __match_args__ = ('x', 'y')
    def __init__(self, x, y):
        self.x = x
        self.y = y

match pontos:
    case []:
        print("Sem pontos")
    case [Ponto(0, 0)]:
        print("A origem")
    case [Ponto(x, y)]:
        print(f"Ponto único {x}, {y}")
    case [Ponto(0, y1), Ponto(0, y2)]:
```

(continua na próxima página)

(continuação da página anterior)

```
        print(f"Dois do eixo Y em {y1}, {y2}")
    case _:
        print("Outra coisa")
```

Podemos adicionar uma cláusula `if` a um padrão, conhecido como "guarda". Se a guarda for falsa, `match` continua para tentar o próximo bloco de caso. Observe que a captura de valor ocorre antes que a guarda seja avaliada:

```
match ponto:
    case Ponto(x, y) if x == y:
        print(f"Y=X at {x}")
    case Ponto(x, y):
        print(f"Não está na diagonal")
```

Vários outros recursos importantes desta instrução:

- Assim como desempacotar atribuições, os padrões de tupla e lista têm exatamente o mesmo significado e realmente correspondem a sequências arbitrárias. Uma exceção importante é que eles não correspondem a iteradores ou strings.
- Os padrões de sequência têm suporte ao desempacotamento estendido: `[x, y, *rest]` e `(x, y, *rest)` funcionam de forma semelhante ao desempacotamento de atribuições. O nome depois de `*` também pode ser `_`, então `(x, y, *_)` corresponde a uma sequência de pelo menos dois itens sem ligar os itens restantes.
- Padrões de mapeamento: `{"bandwidth": b, "latency": l}` captura os valores `"bandwidth"` e `"latency"` de um dicionário. Diferente dos padrões de sequência, chaves extra são ignoradas. Um desempacotamento como `**rest` também é permitido. (Mas `**_` seria redundante, então não é permitido.)
- Subpadrões podem ser capturados usando a palavra reservada `as`:

  ```
  case (Ponto(x1, y1), Ponto(x2, y2) as p2): ...
  ```

  Vai capturar o segundo elemento da entrada como `p2` (se a entrada for uma sequência de dois pontos)
- A maioria dos literais são comparados por igualdade, no entando os singletons `True`, `False` e `None` são comparados por identidade.
- Padrões podem usar constantes nomeadas. Estas devem ser nomes pontilhados para prevenir que sejam interpretadas como variáveis de captura:

  ```
  from enum import Enum
  class Cor(Enum):
      VERMELHO = 'vermelho'
      VERDE = 'verde'
      AZUL = 'azul'

  cor = Cor(input("Insira sua escolha de 'vermelho, 'azul' ou 'verde': "))

  match color:
      case Cor.VERMELHO:
          print("Eu vejo vermelho!")
      case Cor.VERDE:
          print("Grama é verde")
      case Cor.AZUL:
          print("O céu é azul :)")
  ```

Para uma explicação mais detalhada e exemplos adicionais, você pode olhar **PEP 636** que foi escrita em formato de tutorial.

4.8 Definindo funções

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

```
>>> def fib(n):    # escreve série de Fibonacci menor que n
...     """Imprime uma série de Fibonacci menor que n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
>>> # Agora chamamos a função que acabamos de definir:
>>> 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. As instruções 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.

A *execução* de uma função cria uma nova tabela de símbolos para as variáveis locais da função. Mais precisamente, todas as atribuições de variáveis numa função são armazenadas na tabela de símbolos local; referências a variáveis são buscadas primeiro na tabela de símbolos local, em seguida na tabela de símbolos locais de funções delimitadoras ou circundantes, depois na tabela de símbolos global e, finalmente, na tabela de nomes da própria linguagem. Embora possam ser referenciadas, variáveis globais e de funções externas não podem ter atribuições (a menos que seja utilizada a instrução `global`, para variáveis globais, ou `nonlocal`, para variáveis de funções externas).

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, ou chama a si mesma recursivamente, uma nova tabela de símbolos é criada para tal chamada.

Uma definição de função associa o nome da função com o objeto função na tabela de símbolos atual. O interpretador reconhece o objeto apontado pelo nome como uma função definida pelo usuário. Outros nomes também podem apontar para o mesmo objeto função e também pode ser usados pra acessar a função:

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

Conhecendo outras linguagens, pode-se questionar que `fib` não é uma função, mas um procedimento, pois ela não devolve um valor. Na verdade, mesmo funções que não usam a instrução `return` devolvem um valor, ainda que pouco interessante. Esse valor é chamado `None` (é um nome embutido). O interpretador interativo evita escrever `None` quando ele é o único resultado de uma expressão. Mas se quiser vê-lo pode usar a função `print()`:

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

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

[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).

```
>>> def fib2(n):  # retorna série de  Fibonacci até n
...     """Retorna uma lista contendo a série de Fibonacci até n."""
...     resultado = []
...     a, b = 0, 1
...     while a < n:
...         resultado.append(a)     # veja abaixo
...         a, b = b, a+b
...     return resultado
...
>>> f100 = fib2(100)     # chama-o
>>> f100                 # escreve o resultado
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Este exemplo demonstra novos recursos de Python:

- A instrução `return` finaliza a execução e retorna um valor da função. `return` sem qualquer expressão como argumento retorna `None`. Atingir o final da função também retorna `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.

4.9 Mais sobre definição de funções

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

4.9.1 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 pergunta_ok(mensagem, tentativas=4, lembrete='Por favor, tente novamente!'):
    while True:
        resposta = input(mensagem)
        if resposta in {'s', 'sim', 'é}:
            return True
        if resposta in {'n', 'não', 'nah'}:
            return False
        tentativas = tentativas - 1
        if tentativas < 0:
            raise ValueError('resposta inválida de usuário')
        print(lembrete)
```

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 a palavra-chave `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
```

4.9.2 Argumentos nomeados

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

```
def papagaio(voltagem, estado='é um cadáver', ação='voar', tipo='Azul Norueguês'):
    print("-- Este papagaio não conseguiria", ação, end=' ')
    print("nem se você desse um choque de", voltagem, "de volts nele.")
    print("-- Plumagem formosa, o", tipo)
    print("-- Ele", estado, "!")
```

aceita um argumento obrigatório (`voltagem`) e três argumentos opcionais (`estado`, `ação`, e `tipo`). Esta função pode ser chamada de qualquer uma dessas formas:

```
papagaio(1000)                                  # 1 argumento posicional
papagaio(voltagem=1000)                         # 1 argumento nomeado
papagaio(voltagem=1000000, ação='fazer VOOOOOM')      # 2 argumentos nomeados
papagaio(ação='fazer VOOOOOM', voltagem=1000000)      # 2 argumentos nomeados
papagaio('um milhão', 'sem vida', 'pular')      # 3 argumentos posicionais
papagaio('mil', estado='estaria no céu')        # 1 posicional, 1 argumento
```

mas todas as formas a seguir seriam inválidas:

```
papagaio()                          # faltando argumento obrigatório
papagaio(voltagem=5.0, 'morto')     # argumento não nomeado após um argumento nomeado
papagaio(110, voltagem=220)         # valor duplicado para o mesmo argumento
papagaio(ator='John Cleese')        # argumento nomeado desconhecido
```

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. `ator` não é um argumento nomeado válido para a função `papagaio`), mas sua ordem é irrelevante. Isto também inclui argumentos obrigatórios (ex.: `papagaio(voltagem=1000)` funciona). Nenhum argumento pode receber mais de um valor. Eis um exemplo que não funciona devido a esta restrição:

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

Quando um último parâmetro formal no formato `**nome` está presente, ele recebe um dicionário (veja typesmapping) contendo todos os argumentos nomeados, exceto aqueles que correspondem a um parâmetro formal. Isto pode ser combinado com um parâmetro formal no formato `*nome` (descrito na próxima subseção) que recebe uma *tupla* contendo os argumentos posicionais, além da lista de parâmetros formais. (`*nome` deve ocorrer antes de `**nome`.) Por exemplo, se definirmos uma função assim:

```
def loja_de_queijos(tipo, *argumentos, **argumentos_nomeados):
    print("-- Você tem algum", tipo, "?")
    print("-- Lamento, acabou o", tipo)
    for arg in argumentos:
        print(arg)
    print("-" * 40)
    for kw in argumentos_nomeados:
        print(kw, ":", argumentos_nomeados[kw])
```

Pode ser chamada assim:

```
loja_de_queijos("Limburger", "Está muito mole, senhor",
           "Está realmente muito, MUITO mole, senhor.",
           vendedor="Michael Palin",
           cliente="John Cleese",
           sketch="Sketch da Loja de Queijos")
```

e, claro, exibiria:

```
-- Você tem algum Limburger ?
-- Lamento, acabou o Limburger
Está muito mole, senhor.
Está realmente muito, MUITO mole, senhor.
----------------------------------------
vendedor : Michael Palin
cliente : John Cleese
sketch : Cheese Shop Sketch
```

Observe que a ordem em que os argumentos nomeados são exibidos é garantida para corresponder à ordem em que foram fornecidos na chamada da função.

4.9.3 Parâmetros especiais

Por padrão, argumentos podem ser passadas para uma função Python tanto por posição quanto explicitamente pelo nome. Para uma melhor legibilidade e desempenho, faz sentido restringir a maneira pelo qual argumentos possam ser passados, assim um desenvolvedor precisa apenas olhar para a definição da função para determinar se os itens são passados por posição, por posição e nome, ou por nome.

A definição de uma função pode parecer com:

```
def f(pos1, pos2, /, pos_ou_kwd, *, kwd1, kwd2):
      -----------    ----------     ----------
        |             |                  |
        |        Posicional ou nomeado   |
        |                                - Somente nomeado
         -- Somente posicional
```

onde `/` e `*` são opcionais. Se usados, esses símbolos indicam o tipo de parâmetro pelo qual os argumentos podem ser passados para as funções: somente-posicional, posicional-ou-nomeado, e somente-nomeado. Parâmetros nomeados são também conhecidos como parâmetros palavra-chave.

Argumentos posicional-ou-nomeados

Se `/` e `*` não estão presentes na definição da função, argumentos podem ser passados para uma função por posição ou por nome.

Parâmetros somente-posicionais

Olhando com um pouco mais de detalhes, é possível definir certos parâmetros como *somente-posicional*. Se *somente-posicional*, a ordem do parâmetro importa, e os parâmetros não podem ser passados por nome. Parâmetros somente-posicional são colocados antes de `/` (barra). A `/` é usada para logicamente separar os argumentos somente-posicional dos demais parâmetros. Se não existe uma `/` na definição da função, não existe parâmetros somente-posicionais.

Parâmetros após a `/` podem ser *posicional-ou-nomeado* ou *somente-nomeado*.

Argumentos somente-nomeados

Para definir parâmetros como *somente-nomeado*, indicando que o parâmetro deve ser passado por argumento nomeado, colocamos um `*` na lista de argumentos imediatamente antes do primeiro parâmetro *somente-nomeado*.

Exemplos de funções

Considere o seguinte exemplo de definição de função com atenção redobrada para os marcadores `/` e `*`:

```
>>> def arg_padrão(arg):
...     print(arg)
...
>>> def arg_somente_pos(arg, /):
...     print(arg)
...
>>> def arg_somente_nom(*, arg):
...     print(arg)
...
>>> def exemplo_combinado(somente_pos, /, padrão, *, somente_nom):
...     print(somente_pos, padrão, somente_nom)
```

A definição da primeira função, `arg_padrão`, a forma mais familiar, não coloca nenhuma restrição para a chamada da função e argumentos podem ser passados por posição ou nome:

```
>>> arg_padrão(2)
2
```

(continua na próxima página)

(continuação da página anterior)

```
>>> arg_padrão(arg=2)
2
```

A segunda função `arg_somente_pos` está restrita ao uso de parâmetros somente posicionais, uma vez que existe uma `/` na definição da função:

```
>>> arg_somente_pos(1)
1

>>> arg_somente_pos(arg=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: arg_somente_pos() got some positional-only arguments passed as keyword␣
↪arguments: 'arg'
```

A terceira função `arg_somente_nom` permite somente argumentos nomeados como indicado pelo `*` na definição da função:

```
>>> arg_somente_nom(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: arg_somente_nom() takes 0 positional arguments but 1 was given

>>> arg_somente_nom(arg=3)
3
```

E a última usa as três convenções de chamada na mesma definição de função:

```
>>> exemplo_combinado(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: exemplo_combinado() takes 2 positional arguments but 3 were given

>>> exemplo_combinado(1, 2, somente_nom=3)
1 2 3

>>> exemplo_combinado(1, padrão=2, somente_nom=3)
1 2 3

>>> exemplo_combinado(somente_pos=1, padrão=2, somente_nom=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: exemplo_combinado() got some positional-only arguments passed as␣
↪keyword arguments: 'somente_pos'
```

Finalmente, considere essa definição de função que possui uma potencial colisão entre o argumento posicional `nome` e `**kwds` que possui `nome` como uma chave:

```
def foo(nome, **kwds):
    return 'nome' in kwds
```

Não é possível essa chamada devolver `True`, uma vez que o argumento nomeado `'nome'` sempre será aplicado para o primeiro parâmetro. Por exemplo:

```
>>> foo(1, **{'nome': 2})
Traceback (most recent call last):
```

(continua na próxima página)

(continuação da página anterior)

```
  File "<stdin>", line 1, in <module>
TypeError: foo() got multiple values for argument 'nome'
>>>
```

Mas usando / (argumentos somente-posicionais), isso é possível já que permite `nome` como um argumento posicional e `'nome'` como uma chave nos argumentos nomeados:

```
>>> def foo(nome, /, **kwds):
...     return 'nome' in kwds
...
>>> foo(1, **{'nome': 2})
True
```

Em outras palavras, o nome de parâmetros somente-posicional podem ser usados em `**kwds` sem ambiguidade.

Recapitulando

A situação irá determinar quais parâmetros usar na definição da função:

```
def f(pos1, pos2, /, pos_ou_nom *, nom1, nom2):
```

Como guia:

- Use somente-posicional se você não quer que o nome do parâmetro esteja disponível para o usuário. Isso é útil quando nomes de parâmetros não tem um significado real, se você quer forçar a ordem dos argumentos da função quando ela é chamada ou se você precisa ter alguns parâmetros posicionais e alguns nomeados.
- Use somente-nomeado quando os nomes tem significado e a definição da função fica mais clara deixando esses nomes explícitos ou se você quer evitar que usuários confiem na posição dos argumentos que estão sendo passados.
- Para uma API, use somente-posicional para evitar quebras na mudança da API se os nomes dos parâmetros forem alterados no futuro.

4.9.4 Listas de argumentos arbitrárias

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 escreve_vários_itens(arquivo, separador, *args):
    file.write(separador.join(args))
```

Normalmente, esses argumentos *variádicos* estarão no final da lista de parâmetros formais, porque eles englobam todos os argumentos de entrada restantes, que são passados para a função. Quaisquer parâmetros formais que ocorrem após o parâmetro `*args` são argumentos 'somente-nomeados' , o que significa que eles só podem ser usados como chave-valor, em vez de argumentos posicionais:

```
>>> def concat(*args, sep="/"):
...     return sep.join(args)
...
>>> concat("terra", "marte", "vênus")
'terra/marte/vênus'
>>> concat("terra", "marte", "vênus", sep=".")
'terra.marte.vênus'
```

4.9.5 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:

```
>>> list(range(3, 6))            # chamada normal com argumentos separados
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))           # chamada com argumentos desempacotados a partir␣
↪de uma lista
[3, 4, 5]
```

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

```
>>> def papagaio(voltagem, estado='um cadáver', ação='voar'):
...     print("-- Este papagaio não conseguiria", ação, end=' ')
...     print("nem se você desse um choque de", voltagem, "de volts nele.", end='
↪')
...     print("Ele", estado, "!")
...
>>> d = {"voltagem": "quatro milhões", "estado": "está realmente morto", "ação":
↪"voar"}
>>> papagaio(**d)
-- Este papagaio não conseguiria voar nem se você desse um choque de quatro␣
↪milhões de volts nele. Ele está realmente morto !
```

4.9.6 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 cria_incrementador(n):
...     return lambda x: x + n
...
>>> f = cria_incrementador(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, 'um'), (2, 'dois'), (3, 'três'), (4, 'quatro')]
>>> pairs.sort(key=lambda pair: pair[1])
>>> pairs
[(4, 'quatro'), (1, 'um'), (3, 'três'), (2, 'dois')]
```

4.9.7 Strings de documentação

Aqui estão algumas convenções sobre o conteúdo e formatação de strings de documentação, também conhecidas como docstrings.

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 multilinha. 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 minha_função():
...     """Faz nada, mas documenta-a.
...
...     Não, é sério, ela faz nada mesmo.
...     """
...     pass
...
>>> print(minha_função.__doc__)
Faz nada, mas documenta-a.

    Não, é sério, ela faz nada mesmo.
```

4.9.8 Anotações de função

Anotações de função são informações de metadados completamente opcionais sobre os tipos usados pelas funções definidas pelo usuário (veja **PEP 3107** e **PEP 484** para mais informações).

Anotações são armazenadas no atributo `__annotations__` da função como um dicionário e não tem nenhum efeito em qualquer outra parte da função. Anotações de parâmetro são definidas por dois-pontos (":") após o nome do parâmetro, seguida por uma expressão que quando avaliada determina o valor da anotação. Anotações do tipo do retorno são definidas por um literal `->`, seguida por uma expressão, entre a lista de parâmetro e os dois-pontos que marcam o fim da instrução `def` . O exemplo a seguir possui um argumento obrigatório, um argumento opcional e o valor de retorno anotados:

```
>>> def f(ham: str, ovos: str = 'ovos') -> str:
...     print("Anotações:", f.__annotations__)
...     print("Argumentos:", ham, ovos)
...     return ham + ' e ' + ovos
...
>>> f('spam')
Anotações: {'ham': <class 'str'>, 'return': <class 'str'>, 'ovos': <class 'str'>}
Argumentos: spam ovos
'spam e ovos'
```

4.10 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 espaços são um bom meio termo entre indentação estreita (permite maior profundidade de aninhamento) e indentação larga (mais fácil de ler). Tabulações trazem complicações, e o melhor é não usar.

- 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)`.
- Nomeie suas classes e funções de forma consistente; a convenção é usar `MaiusculoCamelCase` para classes e `minusculo_com_sublinhado` para funções e métodos. Sempre use `self` como o nome para o primeiro argumento dos métodos (veja *Uma primeira olhada nas classes* para mais informações sobre classes e métodos).
- Não use codificações exóticas se o seu código é feito para ser usado em um contexto internacional. O padrão do Python, UTF-8, ou mesmo ASCII puro funciona bem em qualquer caso.
- Da mesma forma, não use caracteres não-ASCII em identificadores se houver apenas a menor chance de pessoas falando um idioma diferente ler ou manter o código.

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*`)`

Adiciona um item ao fim da lista. Similar a `a[len(a):] = [x]`.

`list.`**`extend`**`(`*iterable*`)`

Estende a lista, adicionando no fim todos os elementos do argumento iterável passado como parâmetro. Similar a `a[len(a):] = iterable`.

`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 o primeiro item encontrado na lista cujo valor é igual a *x*. Se não existir valor igual, uma exceção `ValueError` é levantada.

`list.`**`pop`**`(`[*i*]`)`

Remove o item na posição fornecida na lista e retorna. Se nenhum índice for especificado, `a.pop()` remove e retorna o último item da lista. Levanta um `IndexError` se a lista estiver vazia ou o índice estiver fora do intervalo da lista.

`list.`**`clear`**`()`

Remove todos os itens de uma lista. Similar a `del a[:]`.

`list.`**`index`**`(`*x*[, *start*[, *end*]]`)`

Devolve o índice base-zero do primeiro item cujo valor é igual a *x*, levantando `ValueError` se este valor não existe.

Os argumentos opcionais *start* e *end* são interpretados como nas notações de fatiamento e são usados para limitar a busca para uma subsequência específica da lista. O índice retornado é calculado relativo ao começo da sequência inteira e não referente ao argumento *start*.

`list.count(x)`

Devolve o número de vezes em que *x* aparece na lista.

`list.sort(* (Separador de parâmetros somente-nomeados (PEP 3102)), 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()`

Inverte a ordem dos elementos na lista.

`list.copy()`

Devolve uma cópia rasa da lista. Similar a `a[:]`.

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

```
>>> frutas = ['laranja', 'maçã', 'pera', 'banana', 'kiwi', 'maçã', 'banana']
>>> frutas.count('maçã')
2
>>> frutas.count('tangerina')
0
>>> frutas.index('banana')
3
>>> frutas.index('banana', 4) # Encontra a próxima banana iniciando da posição 4
6
>>> frutas.reverse()
>>> frutas
['banana', 'maçã', 'kiwi', 'banana', 'pera', 'maçã', 'laranja']
>>> frutas.append('uva')
>>> frutas
['banana', 'maçã', 'kiwi', 'banana', 'pera', 'maçã', 'laranja', 'uva']
>>> frutas.sort()
>>> frutas
['maçã', 'maçã', 'banana', 'banana', 'uva', 'kiwi', 'laranja', 'pera']
>>> frutas.pop()
'pera'
```

Você pode ter percebido que métodos como `insert`, `remove` ou `sort`, que apenas modificam a lista, não têm valor de retorno impresso – eles retornam o `None` padrão.[1] Isto é um princípio de design para todas as estruturas de dados mutáveis em Python.

Outro aspecto que você pode notar é que nem todos os dados podem ser classificados ou comparados. Por exemplo, `[None, 'hello', 10]` não é ordenável porque os inteiros não podem ser comparados a strings e `None` não pode ser comparado a outros tipos. Além disso, há alguns tipos que não têm uma relação de ordenação definida. Por exemplo, `3+4j < 5+7j` não é uma comparação válida.

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:

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

(continua na próxima página)

[1] Outras linguagens podem retornar o objeto modificado, o que permite encadeamento de métodos, como `d->insert("a")->remove("b")->sort();`.

(continuação da página anterior)

```
7
>>> pilha
[3, 4, 5, 6]
>>> pilha.pop()
6
>>> pilha.pop()
5
>>> pilha
[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
>>> fila = deque(["Erik", "João", "Miguel"])
>>> fila.append("Tiago")            # Tiago chega
>>> fila.append("George")           # George chega
>>> fila.popleft()                  # O primeiro a chegar agora sai
'Erik'
>>> fila.popleft()                  # O segundo a chegar agora sai
'João'
>>> fila                            # A fila restante na ordem de chegada
deque(['Miguel', 'Tiago', 'George'])
```

5.1.3 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:

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

Note que isto cria (ou sobrescreve) uma variável chamada `x` que ainda existe após o término do laço. Podemos calcular a lista dos quadrados sem qualquer efeito colateral usando:

```
quadrados = list(map(lambda x: x**2, range(10)))
```

ou, de maneira equivalente:

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

que é mais conciso e legível.

Um compreensão de lista consiste de um par de colchetes contendo uma expressão seguida de uma cláusula `for`, e então zero ou mais cláusulas `for` ou `if`. O resultado será uma nova lista resultante da avaliação da expressão no contexto das cláusulas `for` e `if`. Por exemplo, essa compreensão combina os elementos de duas listas se eles forem diferentes:

```
>>> [(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)]
```

e é equivalente a:

```
>>> combos = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...         if x != y:
...             combos.append((x, y))
...
>>> combos
[(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]
>>> # cria uma nova lista com os valores dobrados
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filtra a lista para excluir números negativos
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # aplica uma função para todos os elementos
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # chama um método em cada elemento
>>> frutafresca = ['  banana', '  baga-de-logan ', 'maracujá  ']
>>> [arma.strip() for arma in frutafresca]
['banana', 'baga-de-logan', 'maracujá']
>>> # cria uma lista de tuplas de 2 elementos como (número, quadrado)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # a tupla deve estar entre parênteses, do contrário um erro é levantado
>>> [x, x**2 for x in range(6)]
  File "<stdin>", line 1
    [x, x**2 for x in range(6)]
     ^^^^^^^
SyntaxError: did you forget parentheses around the comprehension target?
>>> # achatamento de uma lista usando uma compreensão de lista com dois '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']
```

5.1.4 Compreensões de lista aninhadas

A expressão inicial em uma compreensão de lista pode ser qualquer expressão arbitrária, incluindo outra compreensão de lista.

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

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

A compreensão de lista abaixo transpõe as linhas e colunas:

```
>>> [[linha[i] for linha in matriz] 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 interna é computada no contexto da cláusula `for` seguinte, portanto o exemplo acima equivale a:

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

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

```
>>> transposta = []
>>> for i in range(4):
...     # as 3 linhas a seguir implementam uma compreensão de lista aninhada
...     linha_transposta = []
...     for linha in matriz:
...         linha_transposta.append(linha[i])
...     transposta.append(linha_transposta)
...
>>> transposta
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

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

```
>>> list(zip(*matriz))
[(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 A instrução `del`

Existe uma maneira de remover um item de uma lista usando seu índice no lugar do seu valor: a instrução `del`. Ele difere do método `pop()` que devolve um valor. A instrução `del` pode também ser utilizada para remover fatias de uma lista ou limpar a lista inteira (que fizemos antes atribuindo uma lista vazia à fatia `a[:]`). Por exemplo:

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

(continua na próxima página)

(continuação da página anterior)

```
>>> 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, 'olá!'
>>> t[0]
12345
>>> t
(12345, 54321, 'olá!')
>>> # Tuplas pode ser aninhadas:
>>> u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'olá!'), (1, 2, 3, 4, 5))
>>> # Tuplas são imutáveis:
>>> t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # mas elas podem conter objetos mutáveis:
>>> 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 seguido de uma vírgula (não basta colocar um único valor entre parênteses). Feio, mas funciona. Por exemplo:

```
>>> vazio = ()
>>> unitário = 'olá', # <!- note a vírgula ao final
```

(continua na próxima página)

(continuação da página anterior)

```
>>> len(vazio)
0
>>> len(unitário)
1
>>> unitário
('olá',)
```

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
```

Isso é chamado, apropriadamente, de *desempacotamento de sequência* e funciona para qualquer sequência no lado direito. O desempacotamento de sequência requer que haja tantas variáveis no lado esquerdo do sinal de igual, quanto existem de elementos na sequência. Observe que a atribuição múltipla é, na verdade, apenas uma combinação de empacotamento de tupla e desempacotamento de sequência.

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:

```
>>> cesta = {'maçã', 'laranja', 'maçã', 'pera', 'laranja', 'banana'}
>>> print(cesta)                      # mostra que itens duplicados foram␣
↪removidos
{'laranja', 'banana', 'pera', 'maçã'}
>>> 'laranja' in cesta                # teste de pertinência rápido
True
>>> 'crabgrass' in cesta
False

>>> # Demonstra operações de conjunto em letras únicas de duas palavras
>>>
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                 # letras únicas em a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                             # letras em a, mas não em b
{'r', 'd', 'b'}
>>> a | b                             # letras em a ou em b ou ambos
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                             # letras em ambos a e b
{'a', 'c'}
>>> a ^ b                             # letras em a ou b, mas não em ambos
{'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
```

(continua na próxima página)

(continuação da página anterior)

```
{'r', 'd'}
```

5.5 Dicionários

Outra estrutura de dados embutida muito útil 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()`.

Um bom modelo mental é imaginar um dicionário como um conjunto não-ordenado de pares *chave:valor*, onde as chaves são únicas em uma dada instância do dicionário. Dicionários são delimitados por chaves: `{}`, e contém uma lista de pares chave:valor separada por vírgulas. Dessa forma também será exibido o conteúdo de um dicionário no console do Python. O dicionário vazio é `{}`.

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.

Executar `list(d)` em um dicionário devolve a lista de todas as chaves presentes no dicionário, na ordem de inserção (se desejar ordená-las basta usar a função `sorted(d)`). Para verificar a existência de uma chave, use o operador `in`.

A seguir, um exemplo de uso do dicionário:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)
['jack', 'guido', 'irv']
>>> sorted(tel)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

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

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

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, 'guido': 4127, 'jack': 4098}
```

5.6 Técnicas de iteração

Ao iterar sobre dicionários, a chave e o valor correspondente podem ser obtidos simultaneamente usando o método `items()`.

```
>>> cavaleiros = {'gallahad': 'o puro', 'robin': 'o bravo'}
>>> for k, v in cavaleiros.items():
...     print(k, v)
...
gallahad o puro
robin o bravo
```

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

```
>>> for i, v in enumerate(['jogo', 'da', 'velha']):
...     print(i, v)
...
0 jogo
1 da
2 velha
```

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

```
>>> perguntas = ['Nome', 'Missão', 'Cor favorita']
>>> respostas = ['Lancelot', 'o santo graal', 'azul']
>>> for q, a in zip(perguntas, respostas):
...     print('{0}? É {1}.'.format(q, a))
...
Nome? É Lancelot.
Missão? É o santo graal.
Cor favorita? É azul.
```

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

```
>>> for i in reversed(range(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.

```
>>> cesta = ['maçã', 'laranja', 'maçã', 'pera', 'laranja', 'banana']
>>> for i in sorted(cesta):
...     print(i)
...
maçã
maçã
banana
```

(continua na próxima página)

(continuação da página anterior)

```
laranja
laranja
pera
```

Usar `set()` em uma sequência elimina elementos duplicados. O uso de `sorted()` em combinação com `set()` sobre uma sequência é uma maneira idiomática de fazer um loop sobre elementos exclusivos da sequência na ordem de classificação.

```
>>> cesta = ['maçã', 'laranja', 'maçã', 'pera', 'laranja', 'banana']
>>> for i in sorted(cesta):
...     print(i)
...
maçã
maçã
banana
laranja
laranja
pera
```

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

5.7 Mais sobre condições

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

Os operadores de comparação `in` e `not in` fazem testes de inclusão que determinam se um valor está (ou não está) em um contêiner. Os operadores `is` e `is not` comparam se dois objetos são realmente o mesmo objeto. Todos os operadores de comparação possuem a mesma prioridade, que é menor 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'
```

Observe que no Python, ao contrário de C, a atribuição dentro de expressões deve ser feita explicitamente com o operador morsa `:=`. Isso evita uma classe comum de problemas encontrados nos programas C: digitar = em uma expressão quando == era o planejado.

5.8 Comparando sequências e outros tipos

Objetos sequência podem ser comparados com outros objetos sequência, desde que o tipo das sequências seja o mesmo. A comparação utiliza a ordem lexicográfica: primeiramente os dois primeiros itens são comparados, e se diferirem isto determinará o resultado da comparação, caso contrário os próximos dois itens serão comparados, e assim por diante até que se tenha exaurido alguma das sequências. Se em uma comparação de itens, os mesmos forem também sequências (aninhadas), então é disparada recursivamente outra comparação lexicográfica. Se todos os itens da sequência forem iguais, então as sequências são ditas iguais. Se uma das sequências é uma subsequência da outra, então a subsequência é a menor. A comparação lexicográfica de strings utiliza codificação Unicode para definir a ordenação. Alguns exemplos de comparações entre sequências do mesmo tipo:

```
(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 que comparar objetos de tipos diferentes com < ou > é permitido desde que os objetos possuam os métodos de comparação apropriados. Por exemplo, tipos numéricos mistos são comparados de acordo com os seus valores numéricos, portanto 0 é igual a 0.0, etc. Em caso contrário, ao invés de fornecer uma ordenação arbitrária, o interpretador levantará um `TypeError`.

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, 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:

```
# Módulo de números de Fibonacci

def fib(n):    # escreve a série de Fibonacci até n
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fib2(n):   # retorna a série de Fibonacci até n
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result
```

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

```
>>> import fibo
```

Isso não adiciona os nomes das funções definidas em `fibo` diretamente ao *espaço de nomes* atual (veja *Escopos e espaços de nomes do Python* para mais detalhes); isso adiciona somente o nome do módulo `fibo`. Usando o nome do módulo você pode acessar as funções:

```
>>> fibo.fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

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

```
>>> fib = fibo.fib
>>> fib(500)
0 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 seu próprio espaço de nomes privado, que é usado como espaço de nomes 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 todas as instruções `import` no início do módulo (ou script , se preferir). As definições do módulo importado, se colocados no nível de um módulo (fora de quaisquer funções ou classes), elas são adicionadas a espaço de nomes global da módulo.

Existe uma variante da instrução `import` que importa definições de um módulo diretamente para o espaço de nomes do módulo importador. Por exemplo:

```
>>> from fibo import fib, fib2
>>> fib(500)
0 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 no espaço de nomes 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)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Isso importa todos as declarações de nomes, exceto aqueles que iniciam com um sublinhado (_). Na maioria dos casos, programadores Python não usam esta facilidade porque ela introduz um conjunto desconhecido de nomes no ambiente, podendo esconder outros nomes previamente definidos.

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.

Se o nome do módulo é seguido pela palavra-chave `as`, o nome a seguir é vinculado diretamente ao módulo importado.

[1] [#] Na verdade, definições de funções também são 'instruções' que são 'executados'; a execução da definição de uma função adiciona o nome da função no espaço de nomes global do módulo.

```
>>> 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 `importlib.reload()`, e.g. `import importlib; importlib.reload(modulename)`.

6.1.1 Executando módulos como scripts

Quando você rodar um módulo Python com

```
python fibo.py <argumentos>
```

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
0 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. Estes nomes de módulo são listados em `sys.builtin_module_names`. 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:

- O diretório que contém o script importador (ou o diretório atual quando nenhum arquivo é especificado).
- A variável de ambiente `PYTHONPATH` (uma lista de nomes de diretórios, com a mesma sintaxe da variável de ambiente `PATH`).
- O padrão dependente da instalação (por convenção, incluindo um diretório `site-packages`, tratado pelo módulo `site`).

Mais detalhes em sys-path-init.

> **Nota**
>
> Nos sistemas de arquivos que suportam links simbólicos, o diretório contendo o script de entrada é resultante do diretório apontado pelo link simbólico. Em outras palavras o diretório que contém o link simbólico **não** é adicionado ao caminho de busca de módulos.

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"

Para acelerar o carregamento de módulos, o Python guarda versões compiladas de cada módulo no diretório `__pycache__` com o nome `modulo.versão.pyc`, onde a versão corresponde ao formato do arquivo compilado; geralmente contêm o número da versão Python utilizada. Por exemplo, no CPython release 3.3 a versão compilada de spam.py será guardada como `__pycache__/spam.cpython-33.pyc`. Esta convenção de nomes permite a coexistência de módulos compilados de diferentes releases e versões de Python.

O Python verifica a data de modificação do arquivo fonte mediante a versão compilada, para ver se está desatualizada e precisa ser recompilada. É um processo completamente automático. Além disso, os módulos compilados são independentes de plataforma, portanto a mesma biblioteca pode ser compartilhada entre sistemas de arquiteturas diferentes.

O Python não verifica as versões compiladas em duas circunstâncias. Primeiro, sempre recompila e não armazena o resultado para módulos carregados diretamente da linha de comando. Segundo, não verifica se não houver um módulo fonte. Para suportar uma distribuição sem fontes (somente as versões compiladas), o módulo compilado deve estar no diretório de fontes, e não deve haver um módulo fonte.

Algumas dicas para especialistas:

- Você pode usar as opções `-O` ou `-OO` no comando Python para reduzir o tamanho de um módulo compilado. A opção `-O` remove as instruções assert, e a opção `-OO` remove, além das instruções assert, as strings de documentações. Como alguns programas podem contar com essa disponibilidade, só use essa opção se souber o que está fazendo. Módulos "otimizados" tem uma marcação `opt-` e são geralmente de menor tamanho. Futuros releases podem mudar os efeitos da otimização.
- Um programa não roda mais rápido quando é lido de um arquivo `.pyc` do que quando lido de um arquivo `.py`; a única coisa que é mais rápida com arquivos `.pyc` é sua velocidade de carregamento.
- O módulo `compileall` pode criar arquivos .pyc para todos os módulos de um diretório.
- Há mais detalhes desse processo, incluindo um fluxograma de decisões, no **PEP 3147**.

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
'>>> '
```

(continua na próxima página)

(continuação da página anterior)

```
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print('ECA!')
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 embutido, 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)
['__breakpointhook__', '__displayhook__', '__doc__', '__excepthook__',
 '__interactivehook__', '__loader__', '__name__', '__package__', '__spec__',
 '__stderr__', '__stdin__', '__stdout__', '__unraisablehook__',
 '_clear_type_cache', '_current_frames', '_debugmallocstats', '_framework',
 '_getframe', '_git', '_home', '_xoptions', 'abiflags', 'addaudithook',
 'api_version', 'argv', 'audit', 'base_exec_prefix', 'base_prefix',
 'breakpointhook', 'builtin_module_names', 'byteorder', 'call_tracing',
 'callstats', 'copyright', 'displayhook', 'dont_write_bytecode', 'exc_info',
 'excepthook', 'exec_prefix', 'executable', 'exit', 'flags', 'float_info',
 'float_repr_style', 'get_asyncgen_hooks', 'get_coroutine_origin_tracking_depth',
 'getallocatedblocks', 'getdefaultencoding', 'getdlopenflags',
 'getfilesystemencodeerrors', 'getfilesystemencoding', 'getprofile',
 'getrecursionlimit', 'getrefcount', 'getsizeof', 'getswitchinterval',
 'gettrace', 'hash_info', 'hexversion', 'implementation', 'int_info',
 'intern', 'is_finalizing', 'last_traceback', 'last_type', 'last_value',
 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path', 'path_hooks',
 'path_importer_cache', 'platform', 'prefix', 'ps1', 'ps2', 'pycache_prefix',
 'set_asyncgen_hooks', 'set_coroutine_origin_tracking_depth', 'setdlopenflags',
 'setprofile', 'setrecursionlimit', 'setswitchinterval', 'settrace', 'stderr',
 'stdin', 'stdout', 'thread_info', 'unraisablehook', '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__', 'a', 'fib', 'fibo', 'sys']
```

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

`dir()` não lista os nomes de variáveis e funções embutidas. Esta lista está disponível no módulo padrão `builtins`:

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning',
 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
 'FileExistsError', 'FileNotFoundError', 'FloatingPointError',
 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError',
 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError',
 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError',
 'MemoryError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented',
 'NotImplementedError', 'OSError', 'OverflowError',
 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError',
 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning',
 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError',
 'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError',
 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError',
 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning',
 'ValueError', 'Warning', 'ZeroDivisionError', '_', '__build_class__',
 '__debug__', '__doc__', '__import__', '__name__', '__package__', 'abs',
 'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'callable',
 'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits',
 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit',
 'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr',
 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass',
 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview',
 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property',
 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice',
 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars',
 '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/                          pacote de nível superior
      __init__.py               Inicializa o pacote de som
      formats/                  Subpacote para as conversões entre formatos de␣
↪arquivos
              __init__.py
              wavread.py
              wavwrite.py
```

(continua na próxima página)

(continuação da página anterior)

```
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                Subpacote para efeitos de som
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                Subpacote para filtros
              __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`.

Os arquivos `__init__.py` são necessários para que o Python trate diretórios contendo o arquivo como pacotes (a menos que se esteja usando um *pacote de espaço de nomes*, um recurso relativamente avançado). Isso impede que diretórios com um nome comum, como `string`, ocultem, involuntariamente, módulos válidos que ocorrem posteriormente no caminho de busca do módulo. No caso mais simples, `__init__.py` pode ser apenas um arquivo vazio, mas pode também executar código de inicialização do pacote, ou configurar a variável `__all__`, descrita mais adiante.

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. A instrução `import` primeiro testa se o item está definido no pacote, senão presume 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. A instrução `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 a instrução `from pacote import *` é acionada. 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 os três submódulos nomeados do pacote `sound.effects`.

Esteja ciente de que os submódulos podem ficar sobrepostos por nomes definidos localmente. Por exemplo, se você adicionou uma função `reverse` ao arquivo `sound/effects/__init__.py`, usar `from sound.effects import *` só importaria os dois submódulos `echo` e `surround`, mas *não* o submódulo `reverse`, porque ele fica sobreposto pela função `reverse` definida localmente:

```
__all__ = [
    "echo",      # refere-se ao arquivo 'echo.py'
    "surround",  # refere-se ao arquivo 'surround.py'
    "reverse",   # !!! refere-se à função 'reverse' agora !!!
]

def reverse(msg: str):  # <-- este nome ofusca o submódulo 'reverse.py'
    return msg[::-1]    #     no caso de uma importação 'from sound.effects import␣
↪*'
```

Se `__all__` não estiver definido, a instrução `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 instruções `import` anteriores. Considere o código abaixo:

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

Nesse exemplo, os módulos `echo` e `surround` são importados no espaço de nomes atual, no momento em que a instrução `from...import` é executada, 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

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`.

Também é possível escrever imports relativos, com a forma `from module import name`. Esses imports usam pontos para indicar o pacote pai e o atual, envolvidos no import relativo. Do módulo `surround`, por exemplo, pode-se usar:

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

Note que imports relativos são baseados no nome do módulo atual. Uma vez que o nome do módulo principal é sempre `"__main__"`, módulos destinados ao uso como módulo principal de um aplicativo Python devem sempre usar imports absolutos.

6.4.3 Pacotes em múltiplos diretórios

Pacotes possuem mais um atributo especial, `__path__`. Inicializado como uma *sequência* de strings 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

Até agora vimos duas maneiras de exibir valores: *expressões* e a função `print()`. (Uma outra maneira é utilizar o método `write()` de objetos do tipo arquivo; o arquivo saída padrão pode ser referenciado como `sys.stdout`. Veja a Referência da Biblioteca Python para mais informações sobre isso.)

Muitas vezes se deseja mais controle sobre a formatação da saída do que simplesmente exibir valores separados por espaço. Existem várias maneiras de formatar a saída.

- Para usar *strings literais formatadas*, comece uma string com `f` ou `F`, antes de abrir as aspas ou aspas triplas. Dentro dessa string, pode-se escrever uma expressão Python entre caracteres `{` e `}`, que podem se referir a variáveis, ou valores literais.

```
>>> ano = 2016
>>> evento = 'Referendo'
>>> f'Resultados do {evento} {ano}'
'Resultados do Referendo 2016'
```

- O método de strings `str.format()` requer mais esforço manual. Ainda será necessário usar `{` e `}` para marcar onde a variável será substituída e pode-se incluir diretivas de formatação detalhadas, mas também precisará incluir a informação a ser formatada. No bloco de código a seguir há dois exemplos de como formatar variáveis:

```
>>> votos_sim = 42_572_654
>>> votos_totais = 85_705_149
>>> porcentagem = votos_sim / votos_totais
>>> '{:-9} votos SIM  {:2.2%}'.format(votos_sim, porcentagem)
' 42572654 votos SIM  49.67%'
```

  Observe como `yes_votes` são preenchidos com espaços e um sinal negativo apenas para números negativos. O exemplo também imprime `percentage` multiplicado por 100, com 2 casas decimais e seguido por um sinal de porcentagem (veja formatspec para detalhes).

- Finalmente, pode-se fazer todo o tratamento da saída usando as operações de fatiamento e concatenação de strings para criar qualquer layout que se possa imaginar. O tipo string possui alguns métodos que realizam operações úteis para preenchimento de strings para uma determinada largura de coluna.

Quando não é necessário sofisticar a saída, mas apenas exibir algumas variáveis com propósito de depuração, pode-se converter qualquer valor para uma string com as funções `repr()` ou `str()`.

A função `str()` serve para retornar representações de valores que sejam legíveis para as pessoas, enquanto `repr()` é para gerar representações que o interpretador Python consegue ler (ou levantará uma exceção `SyntaxError`, se não houver sintaxe equivalente). Para objetos que não têm uma representação adequada para consumo humano, `str()` devolve o mesmo valor que `repr()`. Muitos valores, tal como números ou estruturas, como listas e dicionários, têm a mesma representação usando quaisquer das funções. Strings, em particular, têm duas representações distintas.

Alguns exemplos:

```
>>> s = 'Olá, mundo.'
>>> str(s)
'Olá, mundo.'
>>> repr(s)
"'Olá, mundo.'"
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'O valor de x é ' + repr(x) + ' e de y é ' + repr(y) + '...'
>>> print(s)
O valor de x é 32.5 e de y é 40000...
>>> # A repr() da string adiciona aspas e contrabarras:
>>> olá = 'olá, mundo\n'
>>> olás = repr(olá)
>>> print(olás)
'olá, mundo\n'
>>> # O argumento de repr() pode ser qualquer objeto Python:
>>> repr((x, y, ('spam', 'ovos')))
"(32.5, 40000, ('spam', 'ovos'))"
```

O módulo `string` contém uma classe `Template` que oferece ainda outra maneira de substituir valores em strings, usando espaços reservados como `$x` e substituindo-os por valores de um dicionário, mas oferece muito menos controle da formatação.

7.1.1 Strings literais formatadas

Strings literais formatadas (também chamadas f-strings, para abreviar) permite que se inclua o valor de expressões Python dentro de uma string, prefixando-a com `f` ou `F` e escrevendo expressões na forma `{expression}`.

Um especificador opcional de formato pode ser incluído após a expressão. Isso permite maior controle sobre como o valor é formatado. O exemplo a seguir arredonda pi para três casas decimais:

```
>>> import math
>>> print(f'O valor de pi é aproximadamente {math.pi:.3f}.')
O valor de pi é aproximadamente 3.142.
```

Passando um inteiro após o `':'` fará com que o campo tenha um número mínimo de caracteres de largura. Isso é útil para alinhar colunas.

```
>>> tabela = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for nome, telefone in tabela.items():
...     print(f'{n9me:10} ==> {telefone:10d}')
...
Sjoerd     ==>       4127
Jack       ==>       4098
Dcab       ==>       7678
```

Outros modificadores podem ser usados para converter o valor antes de ser formatado. `'!a'` aplica a função `ascii()`, `'!s'` aplica a função `str()` e `'!r'` aplica a função `repr()`

```
>>> animais = 'enguias'
>>> print(f'Meu hovercraft está cheio de {animais}.')
Meu hovercraft está cheio de enguias.
>>> print(f'Meu hovercraft está cheio de {animais!r}.')
Meu hovercraft está cheio de 'enguias'.
```

O especificador = pode ser usado para expandir uma expressão para o texto da expressão, um sinal de igual e, então, a representação da expressão avaliada:

```
>>> bugs = 'roaches'
>>> count = 13
>>> area = 'living room'
>>> print(f'Debugging {bugs=} {count=} {area=}')
Debugging bugs='roaches' count=13 area='living room'
```

Veja expressões autodocumentadas para mais informações sobre o especificador =. Para obter uma referência sobre essas especificações de formato, consulte o guia de referência para a formatspec.

7.1.2 O método format()

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

```
>>> print('Nós somos os {} que dizem "{}!"'.format('cavaleiros', 'Ni'))
Nós somos os cavaleiros que dizem "Ni!"
```

As chaves e seus conteúdos (chamados campos de formatação) são substituídos pelos objetos passados para o método `str.format()`. Um número nas chaves pode ser usado para referenciar a posição do objeto passado no método `str.format()`.

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

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

```
>>> print('Este {comida} está {adjetivo}.'.format(
...       comida='spam', adjetivo='absolutamente horrível'))
Este spam está absolutamente horrível.
```

Argumentos posicionais e nomeados podem ser combinados à vontade:

```
>>> print('A história de {0}, {1} e {outro}.'.format('Bill', 'Manfred',
...                                                 outro='Georg'))
A história de Bill, Manfred e Georg.
```

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

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

Isto também pode ser feito passando o dicionário `table` como argumentos nomeados com a notação `**`.

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

```
>>> tabela = {k: str(v) for k, v in vars().items()}
>>> mensagem = " ".join([f'{k}: ' + '{' + k +'};' for k in tabela.keys()])
>>> print(mensagem.format(**tabela))
__name__: __main__; __doc__: None; __package__: None; __loader__: ...
```

Como exemplo, as linhas seguintes produzem um conjunto de colunas alinhadas, com alguns inteiros e seus quadrados e cubos:

```
>>> 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
```

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

7.1.3 Formatação manual de string

Aqui está a mesma tabela de quadrados e cubos, formatados manualmente:

```
>>> for x in range(1, 11):
...     print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
...     # Observe o uso do 'end' na linha anterior
...     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
```

(Note que o espaço entre cada coluna foi adicionado pela forma que a função `print()` funciona: sempre adiciona espaços entre seus argumentos.)

O método `str.rjust()` justifica uma string à direita, num campo de tamanho definido, acrescentando espaços à esquerda. De forma similar, os métodos `str.ljust()`, justifica à esquerda, e `str.center()`, para centralizar. Esses métodos não escrevem nada, apenas retornam uma nova string. Se a string de entrada é muito longa, os métodos não truncarão a saída, e retornarão a mesma string, sem mudança; isso vai atrapalhar o layout da coluna,

mas geralmente é melhor do que a alternativa, que estaria distorcendo o valor. (Se realmente quiser truncar, sempre se pode adicionar uma operação de fatiamento, como em `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'
```

7.1.4 Formatação de strings à moda antiga

O operador % (módulo) também pode ser usado para formatação de string. Dado `formato % valores` (onde *formato* é uma string), as instâncias de `%` em `formato` serão substituídas por zero ou mais elementos de `valores`. Essa operação é conhecida como interpolação de string. Por exemplo:

```
>>> import math
>>> print('O valor de pi é aproximadamente %5.3f.' % math.pi)
O valor de pi é aproximadamente 3.142.
```

Mais informação pode ser encontrada na seção old-string-formatting.

7.2 Leitura e escrita de arquivos

`open()` retorna um *objeto arquivo*, e é mais utilizado com dois argumentos posicionais e um argumento nomeado: `open(filename, mode, encoding=None)`

```
>>> f = open('arquivo_de_trabalho', 'w', encoding="utf-8")
```

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 presumido `'r'`.

Normalmente, arquivos são abertos no *modo texto*, o que significa que você lê strings de e para o arquivo, o qual está em um codificação específica. Se a *codificação* não for especificada, o padrão irá depender da plataforma (veja `open()`). Como o UTF-8 é o padrão mais moderno, `encoding="utf-8"` é recomendado a não ser que você precise utilizar uma *codificação* diferente. Adicionando `'b'` ao modo irá abrir o o arquivo em *modo binário*. Dados no modo binário são lidos e escritos como objetos `bytes`. Você não pode especificar a *codificação* quando estiver abrindo os arquivos em modo binário.

Em modo texto, o padrão durante a leitura é converter terminadores de linha específicos da plataforma (`\n` no Unix, `\r\n` no Windows) para apenas `\n`. Ao escrever no modo de texto, o padrão é converter as ocorrências de `\n` de volta para os finais de linha específicos da plataforma. Essa modificação de bastidores nos dados do arquivo é adequada para arquivos de texto, mas corromperá dados binários, como arquivos `JPEG` ou `EXE`. Tenha muito cuidado para só usar o modo binário, ao ler e gravar esses arquivos.

É uma boa prática usar a palavra-chave `with` ao lidar com arquivos. A vantagem é que o arquivo é fechado corretamente após o término de sua utilização, mesmo que uma exceção seja levantada em algum momento. Usar `with` também é muito mais curto que escrever seu bloco equivalente `try-finally`:

```
>>> with open('arquivo_de_trabalho', encoding="utf-8") as f:
...     read_data = f.read()

>>> # Podemos verificar se o arquivo foi fechado automaticamente.
```

(continua na próxima página)

(continuação da página anterior)

```
>>> f.closed
True
```

Se você não está usando a palavra reservada `with`, então você deveria chamar `f.close()` para fechar o arquivo e imediatamente liberar qualquer recurso do sistema usado por ele.

> **Aviso**
>
> Chamar `f.write()` sem usar a palavra reservada `with` ou chamar `f.close()` **pode** resultar nos argumentos de `f.write()` não serem completamente escritos no disco, mesmo se o programa for encerrado com êxito.

Depois que um arquivo é fechado, seja por uma instrução `with` ou chamando `f.close()`, as tentativas de usar o arquivo falharão automaticamente.

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

7.2.1 Métodos de objetos arquivo

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

Para ler o conteúdo de um arquivo, chame `f.read(tamanho)`, que lê um punhado de dados devolvendo-os como uma string (em modo texto) ou bytes (em modo binário). *tamanho* é um argumento numérico opcional. Quando *tamanho* é omitido ou negativo, todo o conteúdo do arquivo é lido e devolvido; se o arquivo é duas vezes maior que memória da máquina, o problema é seu. Caso contrário, no máximo *tamanho* caracteres (em modo texto) ou *tamanho* bytes (em modo binário) são lidos e devolvidos. Se o fim do arquivo for atingido, `f.read()` devolve uma string vazia (`''`).

```
>>> f.read()
'Este é o arquivo inteiro.\n'
>>> f.read()
''
```

O método `f.readline()` lê uma única linha do arquivo; o caractere de quebra de linha (`\n`) é mantido ao final da string, e só é omitido na última linha do arquivo, se o arquivo não terminar com uma quebra de linha. Isso elimina a ambiguidade do valor retornado; se `f.readline()` retorna uma string vazia, o fim do arquivo foi atingido. Linhas em branco são representadas por um `'\n'` – uma string contendo apenas o caractere terminador de linha.

```
>>> f.readline()
'Está é a primeira linha do arquivo.\n'
>>> f.readline()
'Segunda linha do arquivo\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, end='')
...
Esta é a primeira linha do arquivo.
Segunda linha do arquivo
```

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

`f.write(string)` escreve o conteúdo de *string* para o arquivo, retornando o número de caracteres escritos.

```
>>> f.write('Este é um teste\n')
15
```

Outros tipos de objetos precisam ser convertidos – seja para uma string (em modo texto) ou para bytes (em modo binário) – antes de escrevê-los:

```
>>> value = ('a resposta', 42)
>>> s = str(value)  # converte a tupla para string
>>> f.write(s)
18
```

`f.tell()` retorna um inteiro dando a posição atual do objeto arquivo, no arquivo representado, como número de bytes desde o início do arquivo, no modo binário, e um número ininteligível, quando no modo de texto.

Para mudar a posição, use `f.seek(offset, de_onde)`. A nova posição é computada pela soma do deslocamento *offset* a um ponto de referência especificado pelo argumento *de-onde*. Se o valor de *de_onde* é 0,a referência é o início do arquivo, 1 refere-se à posição atual, e 2 refere-se ao fim do arquivo. Este argumento pode ser omitido e o valor padrão é 0, usando o início do arquivo como referência.

```
>>> f = open('arquivo_de_trabalho', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)      # Vai até o 6º byte no arquivo
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2)  # Vai até o 3º byte antes do fim
13
>>> f.read(1)
b'd'
```

Em arquivos texto (abertos sem um `b`, em modo string), somente *seeks* relativos ao início do arquivo serão permitidos (exceto se for indicado o final do arquivo, com `seek(0, 2)`) e o único valor válido para *offset* são aqueles retornados por chamada à `f.tell()`, ou zero. Qualquer outro valor para *offset* produz um comportamento indefinido.

Objetos arquivo tem alguns métodos 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
>>> x = [1, 'lista', 'simples']
>>> json.dumps(x)
'[1, "lista", "simples"]'
```

Outra variação da função `dumps()`, chamada `dump()`, serializa o objeto para um *arquivo texto*. Se `f` é um *arquivo texto* aberto para escrita, podemos fazer isto:

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

Para decodificar o objeto novamente, se `f` é um objeto *arquivo binário* ou *arquivo texto* que foi aberto para leitura:

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

Nota

Arquivos JSON devem ser codificados em UTF-8. Use `encoding="utf-8"` quando abrir um arquivo JSON como um *arquivo texto* tanto para leitura quanto para escrita.

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('Olá mundo')
  File "<stdin>", line 1
    while True print('Olá mundo')
               ^^^^^
SyntaxError: invalid syntax
```

O analisador sintático repete a linha inválida e mostra pequenas setas apontando para o ponto da linha em que o erro foi detectado. O erro deve ter sido causado pela ausência do símbolo que *precede* a seta. No exemplo, o erro foi detectado na função `print()`, uma vez que um dois-pontos (`':'`) está faltando antes dela. O nome de arquivo e número de linha são exibidos para que você possa rastrear o erro no texto do 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>
    10 * (1/0)
         ~^~
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
```

(continua na próxima página)

(continuação da página anterior)

```
  File "<stdin>", line 1, in <module>
    4 + spam*3
        ^^^^
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    '2' + 2
    ~~~~^~~
TypeError: can only concatenate str (not "int") to str
```

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(input("Insira um número: "))
...         break
...     except ValueError:
...         print("Ops! Esse não é um número válido. Tente novamente...")
...
```

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 exceção durante a execução de uma cláusura `try`, as instruções remanescentes na cláusula são ignoradas. Se o tipo da exceção ocorrida tiver sido previsto em algum `except`, essa *cláusura except* é executada, e então depois a execução continua após o bloco try/except.
- Se a exceção levantada não 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 instrução `try` pode ter uma ou mais *cláusula de exceção*, para especificar múltiplos tratadores para diferentes exceções. No máximo um único tratador será executado. Tratadores só são sensíveis às exceções levantadas no interior da *cláusula de tentativa*, e não às que tenham ocorrido no interior de outro tratador numa mesma instrução `try`. Uma *cláusula de exceção* pode ser sensível a múltiplas exceções, desde que as especifique em uma tupla, por exemplo:

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

Uma classe em uma cláusula `except` corresponde a exceções que são instâncias da própria classe ou de uma de suas classes derivadas (mas o contrário não é válido — uma *cláusula except* listando uma classe derivada não corresponde a instâncias de suas classes base). Por exemplo, o seguinte código irá mostrar B, C, D nessa ordem:

```
class B(Exception):
    pass

class C(B):
    pass

class D(C):
    pass

for cls in [B, C, D]:
    try:
        raise cls()
    except D:
        print("D")
    except C:
        print("C")
    except B:
        print("B")
```

Se a ordem das *cláusulas de exceção* fosse invertida (`except B` no início), seria exibido B, B, B — somente a primeira *cláusula de exceção* compatível é ativada.

Quando uma exceção ocorre, ela pode estar associada a valores chamados *argumentos* da exceção. A presença e os tipos dos argumentos dependem do tipo da exceção.

A *cláusula except* pode especificar uma variável após o nome da exceção. A variável está vinculada à instância de exceção que normalmente possui um atributo `args` que armazena os argumentos. Por conveniência, os tipos de exceção embutidos definem `__str__()` para exibir todos os argumentos sem acessar explicitamente `.args`.

```
>>> try:
...     raise Exception('spam', 'ovos')
... except Exception as inst:
...     print(type(inst))    # o tipo da exceção
...     print(inst.args)     # argumentos armazenados em .args
...     print(inst)          # __str__ permite imprimir args diretamente,
...                          # mas pode ser substituído em subclasses de exceção
...     x, y = inst.args     # desempacota args
...     print('x =', x)
...     print('y =', y)
...
<class 'Exception'>
('spam', 'ovos')
('spam', 'ovos')
x = spam
y = ovos
```

A saída `__str__()` da exceção é exibida como a última parte ("detalhe") da mensagem para exceções não tratadas.

`BaseException` é a classe base comum de todas as exceções. Uma de suas subclasses, `Exception`, é a classe base de todas as exceções não fatais. Exceções que não são subclasses de `Exception` normalmente não são tratadas, pois são usadas para indicar que o programa deve terminar. Elas incluem `SystemExit` que é kevantada por `sys.exit()` e `KeyboardInterrupt` que é levantada quando um usuário deseja interromper o programa.

`Exception` pode ser usada como um curinga que captura (quase) tudo. No entanto, é uma boa prática ser o mais específico possível com os tipos de exceções que pretendemos manipular e permitir que quaisquer exceções inesperadas se propaguem.

O padrão mais comum para lidar com `Exception` é imprimir ou registrar a exceção e então levantá-la novamente (permitindo que um chamador lide com a exceção também):

```
import sys

try:
    f = open('meuarquivo.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("Erro de E/S:", err)
except ValueError:
    print("Não foi possível converter dados para um inteiro.")
except Exception as err:
    print(f"Não esperava {err=}, {type(err)=}")
    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 OSError:
        print('não foi possível abrir', arg)
    else:
        print(arg, 'tem', len(f.readlines()), 'linhas')
        f.close()
```

É melhor usar a cláusula `else` do que adicionar código adicional à cláusula `try` porque ela evita que acidentalmente seja tratada uma exceção que não foi levantada pelo código protegido pela construção com as instruções `try` … `except`.

Os manipuladores de exceção não tratam apenas exceções que ocorrem imediatamente na *cláusula try*, mas também aquelas que ocorrem dentro de funções que são chamadas (mesmo indiretamente) na *cláusula try*. Por exemplo:

```
>>> def this_fails():
...     x = 1/0
...
>>> try:
...     isso_aqui_falha()
... except ZeroDivisionError as err:
...     print('Tratando o erro de tempo execução:', err)
...
Tratando o erro de tempo execução: division 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('Olá')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

(continua na próxima página)

(continuação da página anterior)

```
    raise NameError('Olá')
NameError: Olá
```

O argumento de `raise` indica a exceção a ser levantada. Esse argumento deve ser uma instância de exceção ou uma classe de exceção (uma classe que deriva de `BaseException`, tal como `Exception` ou uma de suas subclasses). Se uma classe de exceção for passada, será implicitamente instanciada invocando o seu construtor sem argumentos:

```
raise ValueError  # abreviação para 'raise ValueError()'
```

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('Olá')
... except NameError:
...     print('Uma exceção passou por aqui!')
...     raise
...
Uma exceção passou por aqui!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    raise NameError('Olá')
NameError: Olá
```

8.5 Encadeamento de exceções

Se uma exceção não tratada ocorrer dentro de uma seção `except`, ela terá a exceção sendo tratada anexada a ela e incluída na mensagem de erro:

```
>>> try:
...     open("database.sqlite")
... except OSError:
...     raise RuntimeError("unable to handle error")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    open("database.sqlite")
    ~~~~^^^^^^^^^^^^^^^^^^^
FileNotFoundError: [Errno 2] No such file or directory: 'database.sqlite'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
    raise RuntimeError("unable to handle error")
RuntimeError: unable to handle error
```

Para indicar que uma exceção é uma consequência direta de outra, a instrução `raise` permite uma cláusula opcional `from`:

```
# exc deve ser uma instância de exceção ou None.
raise RuntimeError from exc
```

Isso pode ser útil quando você está transformando exceções. Por exemplo:

```
>>> def func():
...     raise ConnectionError
...
>>> try:
...     func()
... except ConnectionError as exc:
...     raise RuntimeError('Failed to open database') from exc
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    func()
    ~~~~^^
  File "<stdin>", line 2, in func
ConnectionError

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
    raise RuntimeError('Failed to open database') from exc
RuntimeError: Failed to open database
```

Ele também permite desabilitar o encadeamento automático de exceções usando o idioma `from None`:

```
>>> try:
...     open('bancodedados.sqlite')
... except OSError:
...     raise RuntimeError from None
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
    raise RuntimeError from None
RuntimeError
```

Para mais informações sobre os mecanismos de encadeamento, veja bltin-exceptions.

8.6 Exceções definidas pelo usuário

Programas podem definir novos tipos de exceções, através da criação de uma nova classe (veja *Classes* para mais informações sobre classes Python). Exceções devem ser derivadas da classe `Exception`, direta ou indiretamente.

As classes de exceção podem ser definidas para fazer qualquer coisa que qualquer outra classe pode fazer, mas geralmente são mantidas simples, geralmente oferecendo apenas um número de atributos que permitem que informações sobre o erro sejam extraídas por manipuladores para a exceção.

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

Muitos módulos padrão definem suas próprias exceções para relatar erros que podem ocorrer nas funções que definem.

8.7 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
```

(continua na próxima página)

(continuação da página anterior)

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

Se uma cláusula `finally` estiver presente, a cláusula `finally` será executada como a última tarefa antes da conclusão da instrução `try`. A cláusula `finally` executa se a instrução `try` produz uma exceção. Os pontos a seguir discutem casos mais complexos quando ocorre uma exceção:

- Se ocorrer uma exceção durante a execução da cláusula `try`, a exceção poderá ser tratada por uma cláusula `except`. Se a exceção não for tratada por uma cláusula `except`, a exceção será gerada novamente após a execução da cláusula `finally`.
- Uma exceção pode ocorrer durante a execução de uma cláusula `except` ou `else`. Novamente, a exceção é re-levantada depois que `finally` é executada.
- Se a cláusula `finally` executa uma instrução `break`, `continue` ou `return`, as exceções não são levantadas novamente.
- Se a instrução `try` atingir uma instrução `break`, `continue` ou `return`, a cláusula `finally` será executada imediatamente antes da execução da instrução `break`, `continue` ou `return`.
- Se uma cláusula `finally` incluir uma instrução `return`, o valor retornado será aquele da instrução `return` da cláusula `finally`, não o valor da instrução `return` da cláusula `try`.

Por exemplo:

```
>>> def retorna_booleano():
...     try:
...         return True
...     finally:
...         return False
...
>>> retorna_booleano()
False
```

Um exemplo mais complicado:

```
>>> 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 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
```

(continua na próxima página)

(continuação da página anterior)

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    divide("2", "1")
    ~~~~~~^^^^^^^^^^
  File "<stdin>", line 3, in divide
    result = x / y
             ~~^~~
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Como você pode ver, a cláusula `finally` é executada em todos os casos. A exceção `TypeError` levantada pela divisão de duas strings não é tratada pela cláusula `except` e portanto é re-levantada depois que a cláusula `finally` é executada.

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.

8.8 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 linha in open("meuarquivo.txt"):
    print(linha, end="")
```

O problema com esse código é que ele deixa o arquivo aberto um período indeterminado depois que o código é executado. Isso não chega a ser problema em scripts simples, mas pode ser um problema para grandes aplicações. A palavra reservada `with` permite que objetos como arquivos sejam utilizados com a certeza de que sempre serão prontamente e corretamente finalizados.

```
with open("meuarquivo.txt") as f:
    for linha in f:
        print(linha, end="")
```

Depois que a instrução é executada, o arquivo *f* é sempre fechado, mesmo se ocorrer um problema durante o processamento das linhas. Outros objetos que, como arquivos, fornecem ações de limpeza predefinidas as indicarão em suas documentações.

8.9 Criando e tratando várias exceções não relacionadas

Existem situações em que é necessário relatar várias exceções que ocorreram. Isso geralmente ocorre em estruturas de simultaneidade, quando várias tarefas podem ter falhado em paralelo, mas também há outros casos de uso em que é desejável continuar a execução e coletar vários erros em vez de levantar a primeira exceção.

O `ExceptionGroup` integrado envolve uma lista de instâncias de exceção para que elas possam ser levantadas juntas. É uma exceção em si, portanto, pode ser capturada como qualquer outra exceção.

```
>>> def f():
...     excs = [OSError('error 1'), SystemError('error 2')]
...     raise ExceptionGroup('houve problemas', excs)
...
>>> f()
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 1, in <module>
  |     f()
  |     ~^^
```

(continua na próxima página)

(continuação da página anterior)

```
  |   File "<stdin>", line 3, in f
  | ExceptionGroup: houve problemas (2 sub-exceptions)
  +-+---------------- 1 ----------------
    | OSError: error 1
    +---------------- 2 ----------------
    | SystemError: error 2
    +------------------------------------
>>> try:
...     f()
... except Exception as e:
...     print(f'capturada {type(e)}: e')
...
capturada <class 'ExceptionGroup'>: e
>>>
```

Usando `except*` em vez de `except`, podemos manipular seletivamente apenas as exceções no grupo que correspondem a um determinado tipo. No exemplo a seguir, que mostra um grupo de exceção aninhado, cada cláusula `except*` extrai do grupo exceções de um certo tipo enquanto permite que todas as outras exceções se propaguem para outras cláusulas e eventualmente sejam levantadas novamente.

```
>>> def f():
...     raise ExceptionGroup(
...         "group1",
...         [
...             OSError(1),
...             SystemError(2),
...             ExceptionGroup(
...                 "group2",
...                 [
...                     OSError(3),
...                     RecursionError(4)
...                 ]
...             )
...         ]
...     )
...
>>> try:
...     f()
... except* OSError as e:
...     print("Houve OSErrors")
... except* SystemError as e:
...     print("Houve SystemErrors")
...
Houve OSErrors
Houve SystemErrors
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 2, in <module>
  |     f()
  |     ~^^
  |   File "<stdin>", line 2, in f
  |     raise ExceptionGroup(
  |     ...<12 lines>...
  |     )
  | ExceptionGroup: group1
  +-+---------------- 1 ----------------
    | ExceptionGroup: group2
```

(continua na próxima página)

(continuação da página anterior)

```
    +-+---------------- 1 ----------------
      | RecursionError: 4
      +------------------------------------
>>>
```

Observe que as exceções aninhadas em um grupo de exceções devem ser instâncias, não tipos. Isso ocorre porque, na prática, as exceções normalmente seriam aquelas que já foram levantadas e capturadas pelo programa, seguindo o seguinte padrão:

```
>>> excs = []
... for test in tests:
...     try:
...         test.run()
...     except Exception as e:
...         excs.append(e)
...
>>> if excs:
...    raise ExceptionGroup("Falhas no teste", excs)
...
```

8.10 Enriquecendo exceções com notas

Quando uma exceção é criada para ser levantada, geralmente é inicializada com informações que descrevem o erro ocorrido. Há casos em que é útil adicionar informações após a captura da exceção. Para este propósito, as exceções possuem um método `add_note(note)` que aceita uma string e a adiciona à lista de notas da exceção. A renderização de traceback padrão inclui todas as notas, na ordem em que foram adicionadas, após a exceção.

```
>>> try:
...     raise TypeError('tipo inválido')
... except Exception as e:
...     e.add_note('Adiciona algumas informações')
...     e.add_note('Adiciona mais algumas informações')
...     raise
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    raise TypeError('tipo inválido')
TypeError: tipo inválido
Adiciona algumas informações
Adiciona mais algumas informações
>>>
```

Por exemplo, ao coletar exceções em um grupo de exceções, podemos querer adicionar informações de contexto para os erros individuais. A seguir, cada exceção no grupo tem uma nota indicando quando esse erro ocorreu.

```
>>> def f():
...     raise OSError('operação falhou')
...
>>> excs = []
>>> for i in range(3):
...     try:
...         f()
...     except Exception as e:
...         e.add_note(f'Aconteceu na iteração {i+1}')
...         excs.append(e)
...
```

(continua na próxima página)

(continuação da página anterior)

```
>>> raise ExceptionGroup('Nós temos alguns problemas', excs)
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 1, in <module>
  |     raise ExceptionGroup('Nós temos alguns problemas', excs)
  | ExceptionGroup: Nós temos alguns problemas (3 sub-exceptions)
  +-+---------------- 1 ----------------
    | Traceback (most recent call last):
    |   File "<stdin>", line 3, in <module>
    |     f()
    |     ~^^
    |   File "<stdin>", line 2, in f
    |     raise OSError('operação falhou')
    | OSError: operação falhou
    | Aconteceu na iteração 1
    +---------------- 2 ----------------
    | Traceback (most recent call last):
    |   File "<stdin>", line 3, in <module>
    |     f()
    |     ~^^
    |   File "<stdin>", line 2, in f
    |     raise OSError('operação falhou')
    | OSError: operação falhou
    | Aconteceu na iteração 2
    +---------------- 3 ----------------
    | Traceback (most recent call last):
    |   File "<stdin>", line 3, in <module>
    |     f()
    |     ~^^
    |   File "<stdin>", line 2, in f
    |     raise OSError('operação falhou')
    | OSError: operação falhou
    | Aconteceu na iteração 3
    +------------------------------------
>>>
```

CAPÍTULO 9

Classes

Classes proporcionam uma forma de organizar dados e funcionalidades juntos. Criar uma nova classe cria um novo "tipo" de objeto, permitindo que novas "instâncias" desse tipo sejam produzidas. Cada instância da classe pode ter atributos anexados a ela, para manter seu estado. Instâncias da classe também podem ter métodos (definidos pela classe) para modificar seu estado.

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 *Variáveis privadas*), 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 apelidamento em outras linguagens. Geralmente, esta característica não é muito apreciada, e pode ser ignorada com segurança ao lidar com tipos imutáveis (números, strings, tuplas). Entretanto, apelidamento 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 Python avançado.

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 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`.

Espaços de nomes são criados em momentos diferentes e possuem diferentes ciclos de vida. O espaço de nomes que contém os nomes embutidos é criado quando o interpretador inicializa e nunca é removido. O espaço de nomes global de um módulo é criado quando a definição do módulo é lida, e normalmente duram até a terminação do interpretador. Os comandos executados pela invocação do interpretador, pela leitura de um script com programa principal, ou interativamente, são parte do módulo chamado `__main__`, e portanto possuem seu próprio espaço de nomes. (Os nomes embutidos possuem seu próprio espaço de nomes no módulo chamado `builtins`.).

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 3 ou 4 escopos aninhados cujos espaços de nomes são diretamente acessíveis:

- o escopo mais interno, que é acessado primeiro, contem os nomes locais
- os escopos das funções que envolvem a função atual, que são acessados a partir do escopo mais próximo, contêm nomes não-locais, mas também não-globais
- o penúltimo escopo contém os nomes globais do módulo atual
- e o escopo mais externo (acessado por último) contém os nomes das funções embutidas e demais objetos pré-definidos do interpretador

Se um nome é declarado no escopo global, então todas as referências e atribuições de valores vão diretamente para o penúltimo escopo, que contém os nomes globais do módulo. Para alterar variáveis declaradas fora do escopo mais interno, a instrução `nonlocal` pode ser usada; caso contrário, todas essas variáveis serão apenas para leitura (a tentativa de atribuir valores a essas variáveis simplesmente criará uma *nova* variável local, no escopo interno, não alterando nada na variável de nome idêntico fora dele).

[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.

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.)

Uma peculiaridade especial do Python é que – se nenhuma instrução `global` ou `nonlocal` estiver em vigor – as atribuições de nomes sempre entram no escopo mais interno. As atribuições não copiam dados — elas apenas vinculam nomes aos objetos. O mesmo vale para exclusões: a instrução `del x` remove a ligação de `x` do espaço de nomes referenciado pelo escopo local. De fato, todas as operações que introduzem novos nomes usam o escopo local: em particular, instruções `import` e definições de funções ligam o módulo ou o nome da função no escopo local.

A instrução `global` pode ser usada para indicar que certas variáveis residem no escopo global ao invés do local; a instrução `nonlocal` indica que variáveis particulares estão em um espoco mais interno e devem ser recuperadas lá.

9.2.1 Exemplo de escopos e espaço de nomes

Este é um exemplo que demonstra como se referir aos diferentes escopos e aos espaços de nomes, e como `global` e `nonlocal` pode afetar ligação entre as variáveis:

```
def teste_de_escopo():
    def faz_local():
        spam = "spam local"

    def faz_nonlocal():
        nonlocal spam
        spam = "spam não-local"

    def faz_global():
        global spam
        spam = "spam global"

    spam = "spam teste"
    faz_local()
    print("Após atribuição local:", spam)
    faz_nonlocal()
    print("Após atribuição não-local:", spam)
    faz_global()
    print("Após atribuição global:", spam)

teste_de_escopo()
print("No escopo global:", spam)
```

A saída do código de exemplo é:

```
Após atribuição local: spam teste
Após atribuição não-local: spam não-local
Após atribuição global: spam não-local
No escopo global: spam global
```

Observe como uma atribuição *local* (que é o padrão) não altera o vínculo de *teste_de_escopo* a *spam*. A instrução `nonlocal` mudou o vínculo de *teste_de_escopo* de *spam* e a atribuição `global` alterou a ligação para o nível do módulo.

Você também pode ver que não havia nenhuma ligação anterior para *spam* antes da atribuição `global`.

9.3 Uma primeira olhada nas classes

Classes introduzem novidades sintáticas, três novos tipos de objetos, e também alguma semântica nova.

9.3.1 Sintaxe da definição de classe

A forma mais simples de definir uma classe é:

```
class NomeClasse:
    <instrução-1>
    .
    .
    .
    <instrução-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 finalizada normalmente (até o fim), 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 (`NomeClasse` 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 MinhaClasse:
    """Um exemplo de classe simples"""
    i = 12345

    def f(self):
        return 'olá mundo'
```

então `MinhaClasse.i` e `MinhaClasse.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 `MinhaClasse.i` num atribuição. `__doc__` também é um atributo válido da classe, retornando a docstring associada à classe: `"Um exemplo de classe simples"`.

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 (presumindo a classe acima):

```
x = MinhaClasse()
```

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 = MinhaClasse()
```

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

Agora o que podemos fazer com objetos de instância? As únicas operações compreendidas por objetos de instância são os atributos de referência. Existem duas maneiras válidas para nomear atributos: atributos de dados e métodos.

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 `MinhaClasse` criada acima, o próximo trecho de código irá exibir o valor `16`, sem deixar nenhum rastro:

```
x.contador = 1
while x.contador < 10:
    x.contador = x.contador * 2
print(x.contador)
del x.contador
```

O outro tipo de referência a um atributo de instância é um *método*. Um método é uma função que "pertence a" um objeto.

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 `MinhaClasse.f` é uma função, enquanto `x.i` não é, já que `MinhaClasse.i` não é uma função. Entretanto, `x.f` não é o mesmo que `MinhaClasse.f`. A referência `x.f` acessa um objeto método e a `MinhaClasse.f` acessa um objeto função.

9.3.4 Objetos método

Normalmente, um método é chamado imediatamente após ser referenciado:

```
x.f()
```

No exemplo `MinhaClasse`, o resultado da expressão acima será a string `"olá mundo"`. 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 `olá mundo` 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…

Na verdade, pode-se supor a resposta: a particularidade sobre os métodos é que o objeto da instância é passado como o primeiro argumento da função. Em nosso exemplo, a chamada `x.f()` é exatamente equivalente a `MinhaClasse.f(x)`. Em geral, chamar um método com uma lista de *n* argumentos é equivalente a chamar a função correspondente com uma lista de argumentos que é criada inserindo o objeto de instância do método antes do primeiro argumento.

Em geral, os métodos funcionam da seguinte forma. 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 que é um objeto função, referências ao objeto de instância e ao objeto função serão empacotadas em um objeto método. Quando o objeto método é chamado com uma lista de argumentos, uma nova lista de argumentos é construída a partir do objeto de instância e da lista de argumentos, e o objeto função é chamado com essa 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 Cachorro:

    tipo = 'canino'         # variável de classe compartilhada por todas as␣
↪instâncias

    def __init__(self, nome):
        self.nome = nome    # variável de instância única para cada instância

>>> d = Cachorro('Fido')
>>> e = Cachorro('Buddy')
>>> d.tipo                  # compartilhada por todos os cachorros
'canino'
>>> e.tipo                  # compartilhada por todos os cachorros
'canino'
>>> d.nome                  # exclusiva do d
'Fido'
>>> e.nome                  # exclusiva do e
'Buddy'
```

Como vimos em *Uma palavra sobre nomes e objetos*, dados compartilhados podem causar efeitos inesperados quando envolvem objetos (*mutáveis*), 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 *Cachorro*:

```
class Cachorro:

    truques = []             # uso incorreto de uma variável de classe

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

    def adicionar_truque(self, truque):
        self.truques.append(truque)
```

(continua na próxima página)

(continuação da página anterior)

```
>>> d = Cachorro('Fido')
>>> e = Cachorro('Buddy')
>>> d.adicionar_truque('rolar')
>>> e.adicionar_truque('fingir de morto')
>>> d.truques                # inesperadamente compartilhado por todos os cães
['rolar', 'fingir de morto']
```

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

```
class Cachorro:

    def __init__(self, nome):
        self.nome = nome
        self.truques = []     # cria uma nova lista vazia para cada cachorro

    def adicionar_truque(self, truque):
        self.truques.append(truque)

>>> d = Cachorro('Fido')
>>> e = Cachorro('Buddy')
>>> d.adicionar_truque('rolar')
>>> e.adicionar_truque('fingir de morto')
>>> d.truques
['rolar']
>>> e.truques
['fingir de morto']
```

9.4 Observações aleatórias

Se um mesmo nome de atributo ocorre tanto na instância quanto na classe, a busca pelo atributo prioriza a instância:

```
>>> class Armazém:
...     propósito = 'armazenar'
...     região = 'oeste'
...
>>> a1 = Armazém()
>>> print(a1.propósito, a1.região)
armazenar oeste
>>> a2 = Armazém()
>>> a2.região = 'leste'
>>> print(a2.propósito, a2.região)
armazenar leste
```

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:

```
# Função definida fora da classe
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1

    def g(self):
        return 'olá mundo'

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

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

    def adicionar_em_dobro(self, x):
        self.adicionar(x)
        self.adicionar(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 da linguagem não seria digna do nome "classe" se não suportasse herança. A sintaxe para uma classe derivada é assim:

```
class NomeClasseDerivada(NomeClasseBase):
    <instrução-1>
```

(continua na próxima página)

(continuação da página anterior)

```
    .
    .
    .
    <instrução-N>
```

O identificador `BaseClassName` deve estar definido em um espaço de nomes acessível do 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 NomeClasseDerivada(nomemódulo.NomeClasseBase):
```

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: `NomeClasseDerivada()` 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()` para verificar herança entre classes: `issubclass(bool, int)` é `True` porque `bool` é uma subclasse de `int`. Porém, `issubclass(float, int)` é `False` porque `float` não é uma subclasse de `int`.

9.5.1 Herança múltipla

Python também suporta uma forma de herança múltipla. Uma definição de classe com várias classes bases tem esta forma:

```
class NomeClasseDerivada(Base1, Base2, Base3):
    <instrução-1>
    .
    .
    .
    <instrução-N>
```

Para a maioria dos casos mais simples, pense na pesquisa de atributos herdados de uma classe pai como o primeiro nível de profundidade, da esquerda para a direita, não pesquisando duas vezes na mesma classe em que há uma sobreposição na hierarquia. Assim, se um atributo não é encontrado em `DerivedClassName`, é procurado em `Base1`, depois, recursivamente, nas classes base de `Base1`, e se não for encontrado lá, é pesquisado em `Base2` e assim por diante.

De fato, é um pouco mais complexo que isso; a ordem de resolução de métodos muda dinamicamente para suportar chamadas cooperativas para `super()`. Essa abordagem é conhecida em outras linguagens de herança múltipla como chamar-o-próximo-método, e é mais poderosa que a chamada à função super, encontrada em linguagens de herança única.

A ordenação dinâmica é necessária porque todos os casos de herança múltipla exibem um ou mais relacionamentos de diamante (em que pelo menos uma das classes bases pode ser acessada por meio de vários caminhos da classe mais inferior). Por exemplo, todas as classes herdam de `object`, portanto, qualquer caso de herança múltipla fornece mais de um caminho para alcançar `object`. Para evitar que as classes base sejam acessadas mais de uma vez, o algoritmo dinâmico lineariza a ordem de pesquisa, de forma a preservar a ordenação da esquerda para a direita, especificada em cada classe, que chama cada pai apenas uma vez, e que é monotônica (significando que uma classe pode ser subclassificada sem afetar a ordem de precedência de seus pais). Juntas, essas propriedades tornam possível projetar classes confiáveis e extensíveis com herança múltipla. Para mais detalhes, veja python_2.3_mro.

9.6 Variáveis privadas

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 *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.

Ver também

O especificações de desfiguração de nome privado para detalhes e casos especiais.

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

```
class Mapeamento:
    def __init__(self, iterável):
        self.lista_itens = []
        self.__atualizar(iterável)

    def atualizar(self, iterável):
        for item in iterável:
            self.lista_itens.append(item)

    __atualizar = atualizar   # cópia privada do método atualizar() original

class SubclasseMapeamento(Mapeamento):

    def update(self, chaves, valores):
        # fornece nova assinatura para atualizar(),
        # mas não quebra __init__()
        for item in zip(chaves, valores):
            self.lista_itens.append(item)
```

O exemplo acima deve funcionar mesmo se `SubclasseMapeamento` introduzisse um identificador `__atualizar` uma vez que é substituído por `_Mapeamento__atualizar` na classe `Mapeamento` e `_SubclasseMapeamento__atualizar` na classe `SubclasseMapeamento`, 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.

Código passado para `exec()` ou `eval()` não considera o nome da classe que invocou como sendo a classe corrente; isso é semelhante ao funcionamento da instrução `global`, cujo efeito se aplica somente ao código que é compilado junto. A mesma restrição se aplica às funções `getattr()`, `setattr()` e `delattr()`, e quando acessamos diretamente o `__dict__` da classe.

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. A maneira pythônica para este fim é usar `dataclasses`:

```
from dataclasses import dataclass

@dataclass
class Empregado:
    nome: str
    dept: str
    salário: int
```

```
>>> joão = Empregado('joão', 'lab de computadores', 1000)
>>> joão.dept
'lab de computadores'
>>> joão.salário
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 uma classe com métodos `read()` e `readline()` que obtém os dados de um "buffer de caracteres" e passar como argumento.

Métodos de instância têm atributos também: `m.__self__` é o objeto instância com o método `m()`, e `m.__func__` é o objeto função correspondente ao método.

9.8 Iteradores

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

```
for elemento in [1, 2, 3]:
    print(elemento)
for elemento in (1, 2, 3):
    print(elemento)
for chave in {'um':1, 'dois':2}:
    print(chave)
for char in "123":
    print(char)
for linha in open("meuarquivo.txt"):
    print(linha, end='')
```

Esse estilo de acesso é claro, conciso e conveniente. O uso de iteradores permeia e unifica o Python. Nos bastidores, a instrução `for` chama `iter()` no objeto contêiner. A função retorna um objeto iterador que define o método `__next__()` que acessa elementos no contêiner, um de cada vez. Quando não há mais elementos, `__next__()` levanta uma exceção `StopIteration` que informa ao `for` para terminar. Você pode chamar o método `__next__()` usando a função embutida `next()`; este exemplo mostra como tudo funciona:

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
```

(continua na próxima página)

(continuação da página anterior)

```
<str_iterator object at 0x10c90e650>
>>> next(it)
'a'
>>> next(it)
'b'
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    next(it)
StopIteration
```

Observando o mecanismo por trás do protocolo dos iteradores, fica fácil adicionar esse comportamento às suas classes. Defina um método `__iter__()` que retorna um objeto que tenha um método `__next__()`. Se uma classe já define `__next__()`, então `__iter__()` pode simplesmente retornar `self`:

```
class Inverter:
    """Iterador para fazer um laço em uma sequência de trás para frente."""
    def __init__(self, dados):
        self.dados = dados
        self.índice = len(dados)

    def __iter__(self):
        return self

    def __next__(self):
        if self.índice == 0:
            raise StopIteration
        self.índice = self.índice - 1
        return self.dados[self.índice]
```

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

9.9 Geradores

Geradores são uma ferramenta simples e poderosa para criar iteradores. São escritos como funções normais mas usam a instrução `yield` quando precisam retornar 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 inverter(dados):
    for índice in range(len(dados)-1, -1, -1):
        yield dados[índice]
```

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

Qualquer coisa que possa ser feita com geradores também pode ser feita com iteradores baseados numa classe, como descrito na seção anterior. O que torna geradores tão compactos é que os métodos `__iter__()` e `__next__()` são criados automaticamente.

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.

9.10 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))                 # soma dos quadrados
285

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

>>> palavras_únicas = set(palavra for linha página for palavra in linha.split())

>>> orador_da_turma = max((estudante.gpa, estudante.nome) for estudante in␣
↪graduados)

>>> dados = 'golf'
>>> list(dados[i] for i in range(len(dados)-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()      # Retorna o diretório de trabalho atual
'C:\\Python313'
>>> os.chdir('/server/accesslogs')   # Altera o diretório de trabalho atual
>>> os.system('mkdir hoje')   # Executa o comando mkdir no shell do sistema
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)
<retorna uma lista de todas as funções do módulo>
>>> help(os)
<retorna uma página de manual extensa criada a partir das docstrings do módulo>
```

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('dados.db', 'arquivo.db')
'arquivo.db'
>>> shutil.move('/build/executáveis', 'dir_instal')
'dir_instal'
```

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')
['primos.py', 'aleatorizar.py', 'aspas.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, consideremos o arquivo `demo.py` a seguir:

```
# Arquivo demo.py
import sys
print(sys.argv)
```

Aqui está a saída da execução `python demo.py um dois três` na linha de comando:

```
['demo.py', 'um', 'dois, 'três']
```

O módulo `argparse` fornece um mecanismo mais sofisticado para processar argumentos de linha de comando. O script seguinte extrai e exibe um ou mais nomes de arquivos e um número de linhas opcional:

```
import argparse

parser = argparse.ArgumentParser(
    prog='topo',
    description='Mostra as primeiras linhas de cada arquivo')
parser.add_argument('arquivos, nargs='+')
parser.add_argument('-l', '--linhas', type=int, default=10)
args = parser.parse_args()
print(args)
```

Quando executada na linha de comando `python topo.py --linhas=5 alfa.txt beta.txt`, o script define `args.linhas` para `5` e `args.arquivos` para `['alfa.txt', 'beta.txt']`.

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('Aviso, arquivo log não encontrado, iniciando um novo\n')
Aviso, arquivo log não encontrado, iniciando um novo
```

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'\br[a-z]*', 'o rato roeu a roupa do rei de roma')
['rato', 'roeu', 'roupa', 'rei', 'roma']
```

(continua na próxima página)

(continuação da página anterior)

```
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'o gato de de chapéu')
'o gato de chapéu'
```

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

```
>>> 'xá para dois'.replace('xá', 'chá')
'chá para dois'
```

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)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

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

```
>>> import random
>>> random.choice(['maçã', 'pera', 'banana'])
'maçã'
>>> random.sample(range(100), 10)   # amostragem sem reposição
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()    # ponto flutuante aleatório do intervalo [0.0, 1.0)
0.17970987693706186
>>> random.randrange(6)    # inteiro aleatório escolhido de range(6)
4
```

O módulo `statistics` calcula as propriedades estatísticas básicas (a média, a mediana, a variação, etc.) de dados numéricos:

```
>>> import statistics
>>> dados = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> statistics.mean(dados)
1.6071428571428572
>>> statistics.median(dados)
1.25
>>> statistics.variance(dados)
1.3720238095238095
```

O projeto SciPy <https://scipy.org> tem muitos outros módulos para cálculos numéricos.

10.7 Acesso à internet

Há diversos módulos para acesso e processamento de protocolos da internet. Dois dos mais simples são `urllib.request` para efetuar download de dados a partir de URLs e `smtplib` para enviar mensagens de correio eletrônico:

```
>>> from urllib.request import urlopen
>>> with urlopen('http://worldtimeapi.org/api/timezone/etc/UTC.txt') as response:
...     for line in response:
...         line = line.decode()             # Converte bytes para str
...         if line.startswith('datetime'):
...             print(line.rstrip())         # Remove nova linha ao final
```

(continua na próxima página)

(continuação da página anterior)

```
...
datetime: 2022-01-01T01:36:47.689215+00:00

>>> 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.

```
>>> # datas são facilmente construídas e formatadas
>>> 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.'

>>> # datas têm suporte a matemática de calendário
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

10.9 Compressão de dados

Formatos comuns de arquivamento e compressão de dados estão disponíveis diretamente através de alguns módulos, entre eles: `zlib`, `gzip`, `bz2`, `lzma`, `zipfile` and `tarfile`.

```
>>> import zlib
>>> s = b'A vaca malhada foi molhada por outra vaca molhada e malhada.'
>>> len(s)
60
>>> t = zlib.compress(s)
>>> len(t)
47
>>> zlib.decompress(t)
b'A vaca malhada foi molhada por outra vaca molhada e malhada.'
>>> zlib.crc32(s)
2444551089
```

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 granularidade 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):
    """Calcula a média aritmética da lista de números.

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

import doctest
doctest.testmod()   # valida automaticamente os testes incorporados
```

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()  # Chamar a partir da linha de comando invoca todos os testes
```

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:

- Os módulos `xmlrpc.client` e `xmlrpc.server` tornam a implementação de chamadas remotas (remote procedure calls) em uma tarefa quase trivial. Apesar dos nomes dos módulos, nenhum conhecimento direto ou manipulação de XML é necessário.
- O pacote `email` é uma biblioteca para gerenciamento de mensagens de correio eletrônico, incluindo MIME e outros baseados no **RFC 2822**. Diferente dos módulos `smtplib` e `poplib` que apenas enviam e recebem mensagens, o pacote de email tem um conjunto completo de ferramentas para construir ou decodificar a estrutura de mensagens complexas (incluindo anexos) e para implementação de protocolos de codificação e cabeçalhos.
- O pacote `json` oferece um suporte robusto para analisar este popular formato para troca de dados. O módulo `csv` oferece suporte para leitura e escrita direta em arquivos no formato Comma-Separated Value, comumente suportado por bancos de dados e planilhas. O processamento XML é fornecido pelos pacotes `xml.etree.ElementTree`, `xml.dom` e `xml.sax`. Juntos, esses módulos e pacotes simplificam muito a troca de informações entre aplicativos Python e outras ferramentas.
- O módulo `sqlite3` é um wrapper para a biblioteca de banco de dados SQLite, fornecendo um banco de dados persistente que pode ser atualizado e acessado usando sintaxe SQL ligeiramente fora do padrão.
- 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

O módulo `reprlib` fornece uma versão de `repr()` personalizado para exibições abreviadas de contêineres grandes ou profundamente aninhados:

```
>>> import reprlib
>>> reprlib.repr(set('supercalifragilisticexpialidoce'))
"{'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 = [[[['preto', 'ciano'], 'branco', ['verde', 'vermelho']], [['magenta',
...     'amarelo'], 'azul']]]
>>>
>>> pprint.pprint(t, width=30)
[[[['preto', 'ciano'],
   'branco',
   ['verde', 'vermelho']],
  [['magenta', 'amarelo'],
   'azul']]]
```

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

```
>>> import textwrap
>>> doc = """O método wrap() é como fill(), exceto que ele retorna
... uma lista de strings em vez de uma string grande com quebras
... de linha para separar as linhas quebradas."""
...
>>> print(textwrap.fill(doc, width=40))
```

(continua na próxima página)

(continuação da página anterior)

```
O método wrap() é como fill(), exceto
que ele retorna uma lista de strings em
vez de uma string grande com quebras de
linha para separar as linhas quebradas.
```

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()          # obtém mapeamento de convenções
>>> x = 1234567.8
>>> locale.format_string("%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('O povo de ${vila} envia $$10 para $causa.')
>>> t.substitute(vila='Nottingham', causa='o fundo de recuperação')
'O povo de Nottingham envia $10 para o fundo de recuperação.'
```

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('Devolva $item para $dono.')
>>> d = dict(item='a andorinha descarregada')
>>> t.substitute(d)
Traceback (most recent call last):
  ...
KeyError: 'dono'
>>> t.safe_substitute(d)
'Devolva a andorinha descarregada para $dono.'
```

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
>>> arquivos_fotos = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class RenomeiaEmLote(Template):
...     delimiter = '%'
```

(continua na próxima página)

(continuação da página anterior)

```
...
>>> formato = input('Informe o estilo de renomeação (%d-data %n-sequencia %f-
↪formato):  ')
Informe o estilo de renomeação (%d-data %n-sequencia %f-formato):  Ashley_%n%f

>>> t = RenomeiaEmLote(formato)
>>> data = time.strftime('%d%b%y')
>>> for i, nome_arquivo in enumerate(arquivos_fotos):
...     base, extensão = os.path.splitext(nome_arquivo)
...     novo_nome = t.substitute(d=data, n=i, f=extensão)
...     print('{0} --> {1}'.format(nome_arquivo, novo_nome))

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

with open('meuarquivo.zip', 'rb') as arq:
    dados = arq.read()

inicio = 0
for i in range(3):                      # mostra o cabeçalho dos 3 primeiros␣
↪arquivos
    inicio += 14
    campos = struct.unpack('<IIIHH', dados[inicio:inicio+16])
    crc32, tamanho_comprimido, tamanho_descomprimido, tamanho_nome_arquivo,␣
↪tamanho_extra = campos

    inicio += 16
    nome_arquivo = dados[inicio:inicio+tamanho_nome_arquivo]
    inicio += tamanho_nome_arquivo
    extra = dados[inicio:inicio+tamanho_extra]
    print(nome_arquivo, hex(crc32), tamanho_comprimido, tamanho_descomprimido)

    inicio += tamanho_extra + tamanho_comprimido     # pula para o próximo␣
↪cabeçalho
```

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, arquivo_entrada, arquivo_saida):
        threading.Thread.__init__(self)
        self.arquivo_entrada = arquivo_entrada
        self.arquivo_saida = arquivo_saida

    def run(self):
        arq = zipfile.ZipFile(self.arquivo_saida, 'w', zipfile.ZIP_DEFLATED)
        arq.write(self.)
        arq.close()
        print('Finalizou compactação em background de:', self.arquivo_entrada)

background = AsyncZip('meusdados.txt', 'meuarquivo.zip')
background.start()
print('O programa original continua a executar em primeiro plano.')

background.join()    # Aguarda a tarefa em background finalizar
print('Programa principal aguardou até a tarefa em background estar finalizada.')
```

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, eventos, variáveis de condição e semáforos.

Ainda que todas essas ferramentas sejam poderosas, pequenos erros de design podem resultar em problemas difíceis de serem diagnosticados. Por isso, a abordagem preferida para a coordenação da tarefa é concentrar todo o acesso a um recurso em um único tópico e, em seguida, usar o módulo `queue` para alimentar esse segmento com solicitações de outros tópicos. Aplicações que utilizam objetos `Queue` para comunicação e coordenação inter-thread são mais fáceis de serem projetados, mais legíveis e mais confiáveis.

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('Depurando informações')
logging.info('Mensagem informática')
logging.warning('Alerta: arquivo de configuração  %s não encontrado', 'server.conf
↪')
logging.error('Ocorreu um erro')
logging.critical('Erro crítico -- desligando')
```

Isso produz a seguinte saída:

```
WARNING:root:Alerta:arquivo de configuração server.conf não encontrado
ERROR:root:Ocorreu um erro
CRITICAL:root:Erro crítico -- 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 *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 armazenamento em cache de objetos que são muito custosos para criar:

```
>>> import weakref, gc
>>> class A:
...     def __init__(self, valor):
...         self.valor = valor
...     def __repr__(self):
...         return str(self.valor)
...
>>> a = A(10)                    # cria uma referência
>>> d = weakref.WeakValueDictionary()
>>> d['primaria'] = a            # não cria uma referência
>>> d['primaria']                # busca o objeto se ele estiver ativo
10
>>> del a                        # remove a referência criada
>>> gc.collect()                 # roda imediatamente a coleta de lixo
0
>>> d['primaria']                # o acesso foi automaticamente removido
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    d['primaria']               # o acesso foi automaticamente removido
  File "C:/python313/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
KeyError: 'primaria'
```

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 em amplitude em árvores de dados:

```
>>> from collections import deque
>>> d = deque(["tarefa1", "tarefa2", "tarefa3"])
>>> d.append("tarefa4")
>>> print("Tratando", d.popleft())
Tratando tarefa1
```

```
nao_pesquisada = deque([no_inicial])
def busca_em_profundidade(nao_pesquisada):
    no = nao_pesquisada.popleft()
    for m in gen_moves(no):
        if eh_o_alvo(m):
            return m
        nao_pesquisada.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
>>> notas = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500, 'python')]
>>> bisect.insort(notas, (300, 'ruby'))
>>> notas
[(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
>>> dados = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(dados)                      # reorganiza a lista como um heap
>>> heappush(dados, -5)                 # adiciona o valor -5 ao heap
>>> [heappop(dados) for i in range(3)]  # encontra as 3 menores entradas
[-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 *
>>> round(Decimal('0.70') * Decimal('1.05'), 2)
Decimal('0.74')
>>> round(.70 * 1.05, 2)
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
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 == 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

Ambientes virtuais e pacotes

12.1 Introdução

Aplicações em Python normalmente usam pacotes e módulos que não vêm como parte da instalação padrão. Aplicações às vezes necessitam uma versão específica de uma biblioteca, porque ela requer que algum problema em particular tenha sido consertado ou foi escrita utilizando-se de uma versão obsoleta da interface da biblioteca.

Isso significa que talvez não seja possível que uma instalação Python preencha os requisitos de qualquer aplicação. Se uma aplicação A necessita a versão 1.0 de um módulo particular mas a aplicação B necessita a versão 2.0, os requisitos entrarão em conflito e instalar qualquer uma das duas versões 1.0 ou 2.0 fará com que uma das aplicações não consiga executar.

A solução para este problema é criar um *ambiente virtual*, uma árvore de diretórios que contém uma instalação Python para uma versão particular do Python, além de uma série de pacotes adicionais.

Diferentes aplicações podem então usar diferentes ambientes virtuais. Para resolver o exemplo anterior de requisitos conflitantes, a aplicação A deve ter seu próprio ambiente virtual com a versão 1.0 instalada enquanto a aplicação B vai possuir outro ambiente virtual com a versão 2.0. Se a aplicação B precisar fazer uma atualização para a versão 3.0, isso não afetará o ambiente da aplicação A.

12.2 Criando ambientes virtuais

O módulo usado para criar e gerenciar ambientes virtuais é chamado `venv`. `venv` instalará a versão do Python a partir da qual o comando foi executado (conforme relatado pela opção `--version`). Por exemplo, executar o comando com `python3.12` instalará a versão 3.12.

Para criar um ambiente virtual, escolha um diretório onde deseja colocá-lo e execute o módulo `venv` como um script com o caminho do diretório:

```
python -m venv tutorial-env
```

Isso irá criar o diretório `tutorial-env` se ele não existir, e também criará diretórios dentro dele contendo uma cópia do interpretador Python, a biblioteca padrão e diversos arquivos de suporte.

Um diretório de localização comum para um ambiente virtual é `.venv`. Esse nome tipicamente mantém o diretório oculto em seu ambiente, portanto é transparente, ao menos tempo que explica o motivo desse diretório existir. Também previne conflitos com `.env`, arquivos de definição de variáveis de ambiente que algumas ferramentas utilizam.

Uma vez criado seu ambiente virtual, você deve ativá-lo.

No Windows, execute:

```
tutorial-env\Scripts\activate
```

No Unix ou no MacOS, executa:

```
source tutorial-env/bin/activate
```

(Este script é escrito para o shell bash. Se você usa shells **csh** ou **fish**, existem scripts alternativos `activate.csh` e `activate.fish` para utilização.)

Ao ativar seu ambiente virtual haverá uma mudança no prompt do shell para mostrar qual ambiente virtual você está usando e modificará o ambiente para que quando você executar `python` ativar a versão e instalação do Python particular àquele ambiente. Por exemplo:

```
$ source ~/envs/tutorial-env/bin/activate
(tutorial-env) $ python
Python 3.5.1 (default, May  6 2016, 10:59:36)
  ...
>>> import sys
>>> sys.path
['', '/usr/local/lib/python35.zip', ...,
'~/envs/tutorial-env/lib/python3.5/site-packages']
>>>
```

Para desativar um ambiente virtual, digite:

```
deactivate
```

no terminal.

12.3 Gerenciando pacotes com o pip

Você pode instalar, atualizar e remover pacotes usando um programa chamado **pip**. Por padrão `pip` irá instalar pacotes do Python Package Index. Você pode navegar pelo Python Package Index através do seu navegador web.

`pip` tem uma série de subcomandos: "install", "uninstall", "freeze", etc. (Consulte o guia installing-index para a documentação completa do `pip`.)

Você pode instalar a última versão de um pacote apenas especificando nome do pacote:

```
(tutorial-env) $ python -m pip install novas
Collecting novas
  Downloading novas-3.1.1.3.tar.gz (136kB)
Installing collected packages: novas
  Running setup.py install for novas
Successfully installed novas-3.1.1.3
```

Você também pode instalar uma versão específica de um pacote dando o nome do pacote seguido por == e o número da versão:

```
(tutorial-env) $ python -m pip install requests==2.6.0
Collecting requests==2.6.0
  Using cached requests-2.6.0-py2.py3-none-any.whl
Installing collected packages: requests
Successfully installed requests-2.6.0
```

Se você executar novamente o comando, `pip` será notificado de que a versão já está instalada, e não fará nada. Você pode fornecer um número de versão diferente para obter essa versão ou pode executar `python -m pip install --upgrade` para atualizar o pacote para a versão mais recente:

```
(tutorial-env) $ python -m pip install --upgrade requests
Collecting requests
Installing collected packages: requests
  Found existing installation: requests 2.6.0
    Uninstalling requests-2.6.0:
      Successfully uninstalled requests-2.6.0
Successfully installed requests-2.7.0
```

`python -m pip uninstall` seguido por um ou mais nomes de pacote removerá os pacotes do ambiente virtual.

`python -m pip show` exibirá informações sobre um pacote específico:

```
(tutorial-env) $ python -m pip show requests
---
Metadata-Version: 2.0
Name: requests
Version: 2.7.0
Summary: Python HTTP for Humans.
Home-page: http://python-requests.org
Author: Kenneth Reitz
Author-email: me@kennethreitz.com
License: Apache 2.0
Location: /Users/akuchling/envs/tutorial-env/lib/python3.4/site-packages
Requires:
```

`python -m pip list` exibirá todos os pacotes instalados no ambiente virtual:

```
(tutorial-env) $ python -m pip list
novas (3.1.1.3)
numpy (1.9.2)
pip (7.0.3)
requests (2.7.0)
setuptools (16.0)
```

`python -m pip freeze` produzirá uma lista semelhante dos pacotes instalados, mas a saída usa o formato que `python -m pip install` espera. Uma convenção comum é colocar esta lista em um arquivo `requirements.txt`:

```
(tutorial-env) $ python -m pip freeze > requirements.txt
(tutorial-env) $ cat requirements.txt
novas==3.1.1.3
numpy==1.9.2
requests==2.7.0
```

O arquivo `requirements.txt` pode ser submetido no controle de versão e adicionado como parte da aplicação. Usuários poderão então instalar todos os pacotes necessários com um `install -r`:

```
(tutorial-env) $ python -m pip install -r requirements.txt
Collecting novas==3.1.1.3 (from -r requirements.txt (line 1))
  ...
Collecting numpy==1.9.2 (from -r requirements.txt (line 2))
  ...
Collecting requests==2.7.0 (from -r requirements.txt (line 3))
  ...
Installing collected packages: novas, numpy, requests
  Running setup.py install for novas
Successfully installed novas-3.1.1.3 numpy-1.9.2 requests-2.7.0
```

`pip` tem inúmeras outras opções. Consulte o guia installing-index para a documentação completa do `pip`. Quando

você escrever um pacote e desejar deixá-lo disponível no Python Package Index, consulte o guia de usuário para empacotamento de Python.

CAPÍTULO 13

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, comprimir dados a muitas outras tarefas. Uma lida rápida da Referência da Biblioteca lhe dará uma ideia do que está disponível.

- installing-index explica como instalar módulos adicionais escritos por outros usuários de Python.

- reference-index: Uma explicação detalhada da sintaxe e da semântica de 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 web site sobre Python. Contém código, documentação, e referências para outras páginas relacionadas a Python ao redor da web.

- https://docs.python.org: acesso rápido à documentação Python.

- https://pypi.org: O Python Package Index, anteriormente apelidado de Cheese Shop[1], é um índice de módulos Python criados pelos usuários. Uma vez que você começa a publicar código, pode registrar seus pacotes aqui para que outros possam encontrá-los.

- 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.)

- https://pyvideo.org coleta links para vídeo relacionados a Python de conferências e reuniões de grupos de usuários.

- https://scipy.org: o projeto Scientific Python (Python Científico) inclui módulos para computação e manipulação rápida de vetores e também hospeda pacotes para coisas similares como álgebra linear, transformações de Fourier, resolvedores não-lineares, distribuições de números aleatórios, análise estatística e afins.

[1] "Cheese Shop" é o título de um quadro do grupo Monty Python: um freguês entra em uma loja especializada em queijos, mas qualquer queijo que ele pede, o balconista diz que está em falta.

Para reportar questões e problemas relacionadas a Python, você pode postar no newsgroup *comp.lang.python* ou enviá-los para o grupo de e-mail em python-list@python.org. O newsgroup e a lista são conectados, então mensagens postadas em um são automaticamente encaminhadas ao outro. Há centenas de postagem diárias perguntando (e respondendo) perguntas, sugerindo novas funcionalidades e anunciando novos módulos. E-mails arquivados estão disponíveis em https://mail.python.org/pipermail/. Existe também o grupo de discussão da comunidade brasileira de Python: python-brasil no Google Groups (http://groups.google.com/group/python-brasil), com discussões de ótimo nível técnico.

Antes de postar, certifique-se de checar a lista de perguntas frequentes (também chamada de FAQ). O FAQ responde muitas das perguntas que aparecem com frequência e pode já conter a solução para o seu problema.

CAPÍTULO 14

Edição de entrada interativa e substituição de histórico

Algumas versões do interpretador Python suportam a edição da linha de entrada atual e a substituição da história, semelhante às habilidade encontradas no shell Korn e no shell GNU Bash. Isso é implementado usando a biblioteca GNU Readline, que oferece suporte a vários estilos de edição. Esta biblioteca possui sua própria documentação, que não vamos duplicar aqui.

14.1 Tab Completion e Histórico de Edição

A conclusão dos nomes de variáveis e módulos é ativado automaticamente na inicialização do interpretador para que a tecla `Tab` invoque a função de conclusão. Ele analisa os nomes das instruções Python, as variáveis locais atuais e os nomes dos módulos disponíveis. Para expressões pontilhadas como `string.a`, ele avaliará a expressão até o `'.'` final e então sugerirá conclusões dos atributos do objeto resultante. Observe que isso pode executar o código definido pelo aplicativo se um objeto com um método `__getattr__()` faz parte da expressão. A configuração padrão também guarda seu histórico em um arquivo chamado `.python_history` no seu diretório de usuário. O histórico estará disponível novamente durante a próxima sessão de interpretação interativa.

14.2 Alternativas ao interpretador interativo

Esta facilidade é um enorme passo em frente em comparação com as versões anteriores do interpretador; 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, aspas, 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 15

Aritmética de ponto flutuante: problemas e limitações

Os números de ponto flutuante são representados no hardware do computador como frações de base 2 (binárias). Por exemplo, a fração **decimal** `0.625` tem valor 6/10 + 2/100 + 5/1000, e da mesma forma a fração **binária** `0.101` tem valor 0/2 + 0/4 + 1/8. Essas duas frações têm valores idênticos, a única diferença real é que a primeira é escrita em notação fracionária de base 10 e a segunda em 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:

```
0.0001100110011001100110011001100110011001100110011001100110011...
```

Se parares em qualquer número finito de bits, obterás uma aproximação. Hoje em dia, na maioria dos computadores, as casas decimais são aproximados usando uma fração binária onde o numerado utiliza os primeiros 53 bits iniciando no bit mais significativo e tendo como denominador uma potência de dois. No caso de 1/10, a fração binária seria `602879701896397 / 2 ** 55` o que chega bem perto, mas mesmo assim, não é igual ao valor original de 1/10.

É fácil esquecer que o valor armazenado é uma aproximação da fração decimal original, devido à forma como os pontos flutuantes são exibidos no interpretador interativo. O Python exibe apenas uma aproximação decimal do

verdadeiro valor decimal da aproximação binária armazenada pela máquina. Se o Python exibisse o verdadeiro valor decimal da aproximação binária que representa o decimal 0.1, seria necessário mostrar:

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

Isto 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:

```
>>> 1 / 10
0.1
```

Lembre-se, mesmo que o resultado impresso seja o valor exato de 1/10, o valor que verdadeiramente estará armazenado será uma fração binária representável que mais se aproxima.

Curiosamente, existem muitos números decimais diferentes que compartilham a mesma fração binária aproximada. Por exemplo, os números `0.1` ou o `0.10000000000000001` e `0.1000000000000000055511151231257827021181583404541015625` são todos aproximações de `3602879701896397 / 2 ** 55`. Como todos esses valores decimais compartilham um mesma de aproximação, qualquer um poderá ser exibido enquanto for preservado o invariante `eval(repr(x)) == x`.

Historicamente, o prompt do Python e a função embutida `repr()` utilizariam o que contivesse 17 dígitos significativos, `0.10000000000000001`. Desde a versão do Python 3.1, o Python (na maioria dos sistemas) agora é possível optar pela forma mais reduzida, exibindo simplesmente o número `0.1`.

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).

Para obter um valor mais agradável, poderás utilizar a formatação de sequência de caracteres para produzir um número limitado de dígitos significativos:

```
>>> format(math.pi, '.12g')  # ponto flutuante fornece 12 dígitos significativos
'3.14159265359'

>>> format(math.pi, '.2f')   # ponto flutuante fornece 2 dígitos significativos
'3.14'

>>> repr(math.pi)
'3.141592653589793'
```

É importante perceber que tudo não passa de pura ilusão: estas simplesmente arredondando a *exibição* da verdadeira maquinaria do valor.

Uma ilusão pode gerar outra. Por exemplo, uma vez que 0.1 não é exatamente 1/10, somar três vezes o valor 0.1, não garantirá que o resultado seja exatamente 0.3, isso porque:

```
>>> 0.1 + 0.1 + 0.1 == 0.3
False
```

Inclusive, uma vez que o 0.1 não consegue aproximar-se do valor exato de 1/10 e 0.3 não pode se aproximar mais do valor exato de 3/10, temos então que o pré-arredondamento com a função `round()` não servirá como ajuda:

```
>>> round(0.1, 1) + round(0.1, 1) + round(0.1, 1) == round(0.3, 1)
False
```

Embora os números não possam se aproximar mais dos exatos valores que desejamos, a função `math.isclose()` poderá ser útil para comparar valores inexatos:

```
>>> math.isclose(0.1 + 0.1 + 0.1, 0.3)
True
```

Alternativamente, a função `round()` pode ser usada para comparar aproximações aproximadas:

```
>>> round(math.pi, ndigits=2) == round(22 / 7, ndigits=2)
True
```

A aritmética binária de ponto flutuante contém muitas surpresas como essa. O problema com "0.1" é explicado em detalhes precisos abaixo, na seção "Erro de representação". Consulte Exemplos de problemas de ponto flutuante (em inglês) para obter um resumo agradável de como o ponto flutuante binário funciona e os tipos de problemas comumente encontrados na prática. Veja também Os perigos do ponto flutuante (em inglês) para um relato mais completo de outras surpresas comuns.

Como dizemos perto do final, "não há respostas fáceis". Ainda assim, não se percam indevidamente no uso do ponto flutuante! Os erros nas operações do tipo float do Python são heranças do hardware de ponto flutuante e, a maioria dos computadores estão na ordem de não mais do que 1 parte em 2**53 por operação. Isso é mais do que o suficiente para a maioria das tarefas, portanto, é importante lembrar que não se trata de uma aritmética decimal e que toda operação com o tipo float poderá via a apresentar novos problemas referentes ao arredondamento.

Embora existam casos patológicos, na maior parte das vezes, terás como resultado final o valor esperado, se simplesmente arredondares a exibição final dos resultados para a quantidade de dígitos decimais que esperas a função `str()` geralmente será o suficiente, e , para seja necessário um valor refinado, veja os especificadores de formato `str.format()` contido na seção formatstrings.

Para as situações que exijam uma representação decimal exata, experimente o módulo `decimal` que possui, a implementação de uma adequada aritmética decimal bastante utilizada nas aplicações contábeis e pelas aplicações que demandam alta precisão.

Uma outra forma de obter uma aritmética exata tem suporte pelo módulo `fracções` que implementa a aritmética baseada em números racionais (portanto, os números fracionários como o 1/3 conseguem uma representação precisa).

Caso necessites fazer um intenso uso das operações de ponto flutuante, é importante que conheças o pacote NumPy e, também é importante dizer, que existem diversos pacotes destinados ao trabalho intenso com operações matemáticas e estatísticas que são fornecidas pelo projeto SciPy. Veja <https://scipy.org>.

O Python fornece ferramentas que podem ajudar nessas raras ocasiões em que realmente *faz* necessário conhecer o valor exato de um ponto flutuante. O método `float.as_integer_ratio()` expressa o valor de um número do tipo float em sua forma fracionária:

```
>>> x = 3.14159
>>> x.as_integer_ratio()
(3537115888337719, 1125899906842624)
```

Uma vez que a relação seja exata, será possível utiliza-la para obter, sem que haja quaisquer perda do valor original:

```
>>> x == 3537115888337719 / 1125899906842624
True
```

O método `float.hex()` expressa um ponto flutuante em hexadecimal (base 16), o mesmo também retornará o valor exato pelo computador:

```
>>> x.hex()
'0x1.921f9f01b866ep+1'
```

Sua precisa representação hexadecimal poderá ser utilizada para reconstruir o valor exato do ponto flutuante:

```
>>> x == float.fromhex('0x1.921f9f01b866ep+1')
True
```

Como a representação será exata, é interessante utilizar valores confiáveis em diferentes versões do Python (independente da plataforma) e a troca de dados entre idiomas diferentes que forneçam o mesmo formato (como o Java e o C99).

Uma outra ferramenta que poderá ser útil é a função `sum()` que ajuda a mitigar a perda de precisão durante a soma. Ele usa precisão estendida para etapas intermediárias de arredondamento, pois os valores serão adicionados a um

total em execução. Isso poderá fazer a diferença na precisão geral de forma que os erros não se acumulem chegando ao ponto de afetar o resultado final:

```
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 == 1.0
False
>>> sum([0.1] * 10) == 1.0
True
```

A função `math.fsum()` vai além e rastreia todos os "dígitos perdidos" à medida que os valores são adicionados a um total contínuo, de modo que o resultado tenha apenas um único arredondamento. Isso é mais lento que `sum()`, mas será mais preciso em casos incomuns em que entradas de grande magnitude se cancelam, deixando uma soma final próxima de zero:

```
>>> arr = [-0.10430216751806065, -266310978.67179024, 143401161448607.16,
...        -143401161400469.7, 266262841.31058735, -0.003244936839808227]
>>> float(sum(map(Fraction, arr)))   # Soma exata com arredondamento simples
8.042173697819788e-13
>>> math.fsum(arr)                   # Arredondamento simples
8.042173697819788e-13
>>> sum(arr)                         # Arredondamentos múltiplos em precisão␣
↪estendida
8.042178034628478e-13
>>> total = 0.0
>>> for x in arr:
...     total += x                   # Arredondamentos múltiplos em precisão padrão
...
>>> total                            # A adição direta não tem dígitos corretos!
-0.0051575902860057365
```

15.1 Erro de representação

Esta seção explica o exemplo do "0.1" em detalhes, e mostra como poderás realizar uma análise exata de casos semelhantes. Presumimos que tenhas uma familiaridade básica com a representação binária de ponto flutuante.

Erro de representação refere-se ao fato de que algumas frações decimais (a maioria, na verdade) não podem ser representadas exatamente como frações binárias (base 2). Esta é a principal razão por que o Python (ou Perl, C, C++, Java, Fortran, e muitas outras) frequentemente não exibe o número decimal exato conforme o esperado:

Por que isso acontece? 1/10 não é representado exatamente sendo fração binária. Desde pelo menos 2000, quase todas as máquinas usam aritmética binária de ponto flutuante da IEEE 754 e quase todas as plataformas representam pontos flutuante do Python como valores binary64 de "double precision" (dupla precisão) da IEEE 754. Os valores binary64 da IEEE 754 têm 53 bits de precisão, por isso na entrada o computador se esforça para converter "0.1" para a fração mais próxima que puder, na forma *J*/2***N* onde *J* é um número inteiro contendo exatamente 53 bits. Reescrevendo:

```
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 <=  2**56 // 10  < 2**53
True
```

Ou seja, 56 é o único valor de *N* que deixa *J* com exatamente 53 bits. Portanto, o melhor valor que conseguimos obter pra *J* será aquele que possui o quociente arredondado:

```
>>> 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
```

Portanto, a melhor aproximação possível de 1/10 como um "IEEE 754 double precision" é:

```
7205759403792794 / 2 ** 56
```

Dividir o numerador e o denominador por dois reduzirá a fração para:

```
3602879701896397 / 2 ** 55
```

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:

```
>>> 0.1 * 2 ** 55
3602879701896397.0
```

Se multiplicarmos essa fração por 10**55, podemos ver o valor contendo os 55 dígitos mais significativos:

```
>>> 3602879701896397 * 10 ** 55 // 2 ** 55
1000000000000000055511151231257827021181583404541015625
```

o que significa que o número exato armazenado no computador é igual ao valor decimal 0.1000000000000000055511151231257827021181583404541015625. Em vez de exibir o valor decimal completo, muitas linguagens (incluindo versões mais antigas do Python), arredondam o resultado para 17 dígitos significativos:

```
>>> format(0.1, '.17f')
'0.10000000000000001'
```

Módulos como o `fractions` e o `decimal` tornam esses cálculos muito mais fáceis:

```
>>> from decimal import Decimal
>>> from fractions import Fraction

>>> Fraction.from_float(0.1)
Fraction(3602879701896397, 36028797018963968)

>>> (0.1).as_integer_ratio()
(3602879701896397, 36028797018963968)

>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')

>>> format(Decimal.from_float(0.1), '.17')
'0.10000000000000001'
```

CAPÍTULO 16

Anexo

16.1 Modo interativo

Existem duas variantes do *REPL* interativo. O interpretador básico clássico é compatível com todas as plataformas com recursos mínimos de controle de linha.

No Windows, ou em sistemas do tipo Unix com suporte a `curses`, um novo console interativo é usado por padrão. Este oferece suporte a cores, edição multilinha, navegação no histórico e modo de colagem. Para desabilitar a cor, veja using-on-controlling-color para detalhes. As teclas de função fornecem algumas funcionalidades adicionais. `F1` entra no navegador de ajuda interativo `pydoc`. `F2` permite navegar no histórico da linha de comando sem saída nem os prompts `»>` e `...`. `F3` entra no "modo de colagem", o que torna mais fácil colar blocos maiores de código. Pressione `F3` para retornar ao prompt normal.

Ao usar o novo console interativo, saia do console digitando `exit` ou `quit`. Não é necessário adicionar parênteses de chamada após esses comandos.

Se o novo shell interativo não for desejado, ele pode ser desabilitado através da variável de ambiente `PYTHON_BASIC_REPL`.

16.1.1 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 instrução `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`.

[1] Um problema com a package GNU Readline pode impedir que isso aconteça.

16.1.2 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 python3
```

(presumindo 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`**.

```
$ chmod +x meuscript.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.

16.1.3 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)
```

16.1.4 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/python3.x/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.

APÊNDICE A

Glossário

>>>
: O prompt padrão do console *interativo* do Python. Normalmente visto em exemplos de código que podem ser executados interativamente no interpretador.

...
: Pode se referir a:

- O prompt padrão do console *interativo* do Python ao inserir o código para um bloco de código recuado, quando dentro de um par de delimitadores correspondentes esquerdo e direito (parênteses, colchetes, chaves ou aspas triplas) ou após especificar um decorador.
- A constante embutida `Ellipsis`.

classe base abstrata
: Classes bases abstratas complementam *tipagem pato*, fornecendo uma maneira de definir interfaces quando outras técnicas, como `hasattr()`, seriam desajeitadas ou sutilmente erradas (por exemplo, com métodos mágicos). ABCs introduzem subclasses virtuais, classes que não herdam de uma classe mas ainda são reconhecidas por `isinstance()` e `issubclass()`; veja a documentação do módulo `abc`. Python vem com muitas ABCs embutidas para estruturas de dados (no módulo `collections.abc`), números (no módulo `numbers`), fluxos (no módulo `io`), localizadores e carregadores de importação (no módulo `importlib.abc`). Você pode criar suas próprias ABCs com o módulo `abc`.

anotação
: Um rótulo associado a uma variável, um atributo de classe ou um parâmetro de função ou valor de retorno, usado por convenção como *dica de tipo*.

Anotações de variáveis locais não podem ser acessadas em tempo de execução, mas anotações de variáveis globais, atributos de classe e funções são armazenadas no atributo especial `__annotations__` de módulos, classes e funções, respectivamente.

Veja *anotação de variável*, *anotação de função*, **PEP 484** e **PEP 526**, que descrevem esta funcionalidade. Veja também annotations-howto para as melhores práticas sobre como trabalhar com anotações.

argumento
: Um valor passado para uma *função* (ou *método*) ao chamar a função. Existem dois tipos de argumento:

- *argumento nomeado*: um argumento precedido por um identificador (por exemplo, `name=`) 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.

Veja também o termo *parâmetro* no glossário, a pergunta no FAQ sobre a diferença entre argumentos e parâmetros e **PEP 362**.

gerenciador de contexto assíncrono
Um objeto que controla o ambiente visto numa instrução `async with` por meio da definição dos métodos `__aenter__()` e `__aexit__()`. Introduzido pela **PEP 492**.

gerador assíncrono
Uma função que retorna um *iterador gerador assíncrono*. É parecida com uma função de corrotina definida com `async def` exceto pelo fato de conter instruções `yield` para produzir uma série de valores que podem ser usados em um laço `async for`.

Normalmente se refere a uma função geradora assíncrona, mas pode se referir a um *iterador gerador assíncrono* em alguns contextos. Em casos em que o significado não esteja claro, usar o termo completo evita a ambiguidade.

Uma função geradora assíncrona pode conter expressões `await` e também as instruções `async for` e `async with`.

iterador gerador assíncrono
Um objeto criado por uma função *geradora assíncrona*.

Este é um *iterador assíncrono* que, quando chamado usando o método `__anext__()`, retorna um objeto aguardável que executará o corpo da função geradora assíncrona até a próxima expressão `yield`.

Cada `yield` suspende temporariamente o processamento, lembrando o estado de execução (incluindo variáveis locais e instruções `try` pendentes). Quando o *iterador gerador assíncrono* é efetivamente retomado com outro aguardável retornado por `__anext__()`, ele inicia de onde parou. Veja **PEP 492** e **PEP 525**.

iterável assíncrono
Um objeto que pode ser usado em uma instrução `async for`. Deve retornar um *iterador assíncrono* do seu método `__aiter__()`. Introduzido por **PEP 492**.

iterador assíncrono
Um objeto que implementa os métodos `__aiter__()` e `__anext__()`. `__anext__()` deve retornar um objeto *aguardável*. `async for` resolve os aguardáveis retornados por um método `__anext__()` do iterador assíncrono até que ele levante uma exceção `StopAsyncIteration`. Introduzido pela **PEP 492**.

atributo
Um valor associado a um objeto que é geralmente referenciado pelo nome separado por um ponto. Por exemplo, se um objeto *o* tem um atributo *a* esse seria referenciado como *o.a*.

É possível dar a um objeto um atributo cujo nome não seja um identificador conforme definido por identifiers, por exemplo usando `setattr()`, se o objeto permitir. Tal atributo não será acessível usando uma expressão pontilhada e, em vez disso, precisaria ser recuperado com `getattr()`.

aguardável
Um objeto que pode ser usado em uma expressão `await`. Pode ser uma *corrotina* ou um objeto com um método `__await__()`. Veja também a **PEP 492**.

BDFL
Abreviação da expressão da língua inglesa "Benevolent Dictator for Life" (em português, "Ditador Benevolente Vitalício"), referindo-se a Guido van Rossum, criador do Python.

arquivo binário
Um *objeto arquivo* capaz de ler e gravar em *objetos bytes ou similar*. Exemplos de arquivos binários são arquivos abertos no modo binário (`'rb'`, `'wb'` ou `'rb+'`), `sys.stdin.buffer`, `sys.stdout.buffer`, e instâncias de `io.BytesIO` e `gzip.GzipFile`.

Veja também *arquivo texto* para um objeto arquivo capaz de ler e gravar em objetos `str`.

referência emprestada
Na API C do Python, uma referência emprestada é uma referência a um objeto que não é dona da referência. Ela se torna um ponteiro solto se o objeto for destruído. Por exemplo, uma coleta de lixo pode remover a última *referência forte* para o objeto e assim destruí-lo.

Chamar `Py_INCREF()` na *referência emprestada* é recomendado para convertê-lo, internamente, em uma *referência forte*, exceto quando o objeto não pode ser destruído antes do último uso da referência emprestada. A função `Py_NewRef()` pode ser usada para criar uma nova *referência forte*.

objeto byte ou similar
Um objeto com suporte ao o bufferobjects e que pode exportar um buffer C *contíguo*. Isso inclui todos os objetos `bytes`, `bytearray` e `array.array`, além de muitos objetos `memoryview` comuns. Objetos byte ou similar podem ser usados para várias operações que funcionam com dados binários; isso inclui compactação, salvamento em um arquivo binário e envio por um soquete.

Algumas operações precisam que os dados binários sejam mutáveis. A documentação geralmente se refere a eles como "objetos byte ou similar para leitura e escrita". Exemplos de objetos de buffer mutável incluem `bytearray` e um `memoryview` de um `bytearray`. Outras operações exigem que os dados binários sejam armazenados em objetos imutáveis ("objetos byte ou similar para somente leitura"); exemplos disso incluem `bytes` e a `memoryview` de um objeto `bytes`.

bytecode
O código-fonte Python é compilado para bytecode, a representação interna de um programa em Python no interpretador CPython. O bytecode também é mantido em cache em arquivos `.pyc` e `.pyo`, de forma que executar um mesmo arquivo é mais rápido na segunda vez (a recompilação dos fontes para bytecode não é necessária). Esta "linguagem intermediária" é adequada para execução em uma *máquina virtual*, que executa o código de máquina correspondente para cada bytecode. Tenha em mente que não se espera que bytecodes sejam executados entre máquinas virtuais Python diferentes, nem que se mantenham estáveis entre versões de Python.

Uma lista de instruções bytecode pode ser encontrada na documentação para o módulo dis.

chamável
Um chamável é um objeto que pode ser chamado, possivelmente com um conjunto de argumentos (veja *argumento*), com a seguinte sintaxe:

```
chamavel(argumento1, argumento2, argumentoN)
```

Uma *função*, e por extensão um *método*, é um chamável. Uma instância de uma classe que implementa o método `__call__()` também é um chamável.

função de retorno
Também conhecida como callback, é uma função sub-rotina que é passada como um argumento a ser executado em algum ponto no futuro.

classe
Um modelo para criação de objetos definidos pelo usuário. Definições de classe normalmente contém definições de métodos que operam sobre instâncias da classe.

variável de classe
Uma variável definida em uma classe e destinada a ser modificada apenas no nível da classe (ou seja, não em uma instância da classe).

variável de clausura
Uma *variável livre* referenciada de um *escopo aninhado* que é definida em um escopo externo em vez de ser resolvida em tempo de execução a partir dos espaços de nomes embutido ou globais. Pode ser explicitamente definida com a palavra reservada `nonlocal` para permitir acesso de gravação, ou implicitamente definida se a variável estiver sendo somente lida.

Por exemplo, na função `interna` no código a seguir, tanto `x` quanto `print` são *variáveis livres*, mas somente `x` é uma *variável de clausura*:

```
def externa():
    x = 0
    def interna():
        nonlocal x
        x += 1
        print(x)
    return interna
```

Devido ao atributo `codeobject.co_freevars` (que, apesar do nome, inclui apenas os nomes das variáveis de clausura em vez de listar todas as variáveis livres referenciadas), o termo mais geral *variável livre* às vezes é usado mesmo quando o significado pretendido é se referir especificamente às variáveis de clausura.

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.

contexto
Este termo tem diferentes significados dependendo de onde e como ele é usado. Alguns significados comuns:

- O estado ou ambiente temporário estabelecido por um *gerenciador de contexto* por meio de uma instrução `with`.
- A coleção de ligações de chave-valor associadas a um objeto `contextvars.Context` específico e acessadas por meio de objetos `ContextVar`. Veja também *variável de contexto*.
- Um objeto `contextvars.Context`. Veja também *contexto atual*.

protocolo de gerenciamento de contexto
Os métodos `__enter__()` e `__exit__()` chamados pela instrução `with`. Veja **PEP 343**.

gerenciador de contexto
Um objeto que implementa o *protocolo de gerenciamento de contexto* e controla o ambiente visto em uma instrução `with`. Veja **PEP 343**.

variável de contexto
Uma variável cujo valor depende de qual contexto é o *contexto atual*. Os valores são acessados por meio de objetos `contextvars.ContextVar`. Variáveis de contexto são usadas principalmente para isolar o estado entre tarefas assíncronas simultâneas.

contíguo
Um buffer é considerado contíguo exatamente se for *contíguo C* ou *contíguo Fortran*. Os buffers de dimensão zero são contíguos C e Fortran. Em vetores unidimensionais, os itens devem ser dispostos na memória próximos um do outro, em ordem crescente de índices, começando do zero. Em vetores multidimensionais contíguos C, o último índice varia mais rapidamente ao visitar itens em ordem de endereço de memória. No entanto, nos vetores contíguos do Fortran, o primeiro índice varia mais rapidamente.

corrotina
Corrotinas são uma forma mais generalizada de sub-rotinas. Sub-rotinas tem a entrada iniciada em um ponto, e a saída em outro ponto. Corrotinas podem entrar, sair, e continuar em muitos pontos diferentes. Elas podem ser implementadas com a instrução `async def`. Veja também **PEP 492**.

função de corrotina
Uma função que retorna um objeto do tipo *corrotina*. Uma função de corrotina pode ser definida com a instrução `async def`, e pode conter as palavras chaves `await`, `async for`, e `async with`. Isso foi introduzido pela **PEP 492**.

CPython
A implementação canônica da linguagem de programação Python, como disponibilizada pelo python.org. O termo "CPython" é usado quando necessário distinguir esta implementação de outras como Jython ou IronPython.

contexto atual
O *contexto* (objeto `contextvars.Context`) que é usado atualmente pelos objetos `ContextVar` para acessar (obter ou definir) os valores de *variáveis de contexto*. Cada thread tem seu próprio contexto atual. Frameworks para executar tarefas assíncronas (veja `asyncio`) associam cada tarefa a um contexto que se torna o contexto atual sempre que a tarefa inicia ou retoma a execução.

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(arg):
    ...
f = staticmethod(f)

@staticmethod
def f(arg):
    ...
```

O mesmo conceito existe para as classes, mas não é comumente utilizado. Veja a documentação de definições de função e definições de classe para obter mais informações sobre decoradores.

descritor
Qualquer objeto que define os métodos `__get__()`, `__set__()` ou `__delete__()`. Quando um atributo de classe é um descritor, seu comportamento de associação especial é acionado no acesso a um atributo. Normalmente, ao se utilizar *a.b* para se obter, definir ou excluir, um atributo dispara uma busca no objeto chamado *b* no dicionário de classe de *a*, mas se *b* for um descritor, o respectivo método descritor é chamado. Compreender descritores é a chave para um profundo entendimento de Python pois eles são a base de muitas funcionalidades incluindo funções, métodos, propriedades, métodos de classe, métodos estáticos e referências para superclasses.

Para obter mais informações sobre os métodos dos descritores, veja: descriptors ou o Guia de Descritores.

dicionário
Um vetor associativo em que chaves arbitrárias são mapeadas para valores. As chaves podem ser quaisquer objetos que possuam os métodos `__hash__()` e `__eq__()`. Isso é chamado de hash em Perl.

compreensão de dicionário
Uma maneira compacta de processar todos ou parte dos elementos de um iterável e retornar um dicionário com os resultados. `results = {n: n ** 2 for n in range(10)}` gera um dicionário contendo a chave `n` mapeada para o valor `n ** 2`. Veja comprehensions.

visão de dicionário
Os objetos retornados por `dict.keys()`, `dict.values()` e `dict.items()` são chamados de visões de dicionário. Eles fornecem uma visão dinâmica das entradas do dicionário, o que significa que quando o dicionário é alterado, a visão reflete essas alterações. Para forçar a visão de dicionário a se tornar uma lista completa use `list(dictview)`. Veja dict-views.

docstring
Abreviatura de "documentation string" (string de documentação). 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.

tipagem pato
Também conhecida como *duck-typing*, é 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 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 no Python presume 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
Uma parte da sintaxe que pode ser avaliada para algum valor. Em outras palavras, uma expressão é a acumulação de elementos de expressão como literais, nomes, atributos de acesso, operadores ou chamadas de funções, todos os quais retornam um valor. Em contraste com muitas outras linguagens, nem todas as construções de linguagem são expressões. Também existem *instruções*, as quais não podem ser usadas como expressões, como, por exemplo, `while`. Atribuições também são instruções, não expressões.

módulo de extensão
Um módulo escrito em C ou C++, usando a API C do Python para interagir tanto com código de usuário quanto do núcleo.

f-string
Literais string prefixadas com `'f'` ou `'F'` são conhecidas como "f-strings" que é uma abreviação de formatted string literals. Veja também **PEP 498**.

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 *objetos arquivo ou similares* ou *fluxos*.

Atualmente há três categorias de objetos arquivo: *arquivos binários* brutos, *arquivos binários* em buffer e *arquivos textos*. Suas interfaces estão definidas no módulo `io`. A forma canônica para criar um objeto arquivo é usando a função `open()`.

objeto arquivo ou similar
Um sinônimo do termo *objeto arquivo*.

tratador de erros e codificação do sistema de arquivos
Tratador de erros e codificação usado pelo Python para decodificar bytes do sistema operacional e codificar Unicode para o sistema operacional.

A codificação do sistema de arquivos deve garantir a decodificação bem-sucedida de todos os bytes abaixo de 128. Se a codificação do sistema de arquivos falhar em fornecer essa garantia, as funções da API podem levantar `UnicodeError`.

As funções `sys.getfilesystemencoding()` e `sys.getfilesystemencodeerrors()` podem ser usadas para obter o tratador de erros e codificação do sistema de arquivos.

O *tratador de erros e codificação do sistema de arquivos* são configurados na inicialização do Python pela função `PyConfig_Read()`: veja os membros `filesystem_encoding` e `filesystem_errors` do `PyConfig`.

Veja também *codificação da localidade*.

localizador
Um objeto que tenta encontrar o *carregador* para um módulo que está sendo importado.

Existem dois tipos de localizador: *localizadores de metacaminho* para uso com `sys.meta_path`, e *localizadores de entrada de caminho* para uso com `sys.path_hooks`.

Veja finders-and-loaders e `importlib` para muito mais detalhes.

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**.

threads livres
Um modelo de threads onde múltiplas threads podem simultaneamente executar bytecode Python no mesmo interpretador. Isso está em contraste com a *trava global do interpretador* que permite apenas uma thread por vez executar bytecode Python. Veja **PEP 703**.

variável livre
Formalmente, conforme definido no modelo de execução de linguagem, uma variável livre é qualquer variável usada em um espaço de nomes que não seja uma variável local naquele espaço de nomes. Veja *variável de clausura* para um exemplo. Pragmaticamente, devido ao nome do atributo `codeobject.co_freevars`, o termo também é usado algumas vezes como sinônimo de *variável de clausura*.

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*, *método* e a seção function.

anotação de função
Uma *anotação* de um parâmetro de função ou valor de retorno.

Anotações de função são comumente usados por *dicas de tipo*: por exemplo, essa função espera receber dois argumentos `int` e também é esperado que devolva um valor `int`:

```
def soma_dois_numeros(a: int, b: int) -> int:
   return a + b
```

A sintaxe de anotação de função é explicada na seção function.

Veja *anotação de variável* e **PEP 484**, que descrevem esta funcionalidade. Veja também annotations-howto para as melhores práticas sobre como trabalhar com anotações.

__future__
A instrução future, `from __future__ import <feature>`, direciona o compilador a compilar o módulo atual usando sintaxe ou semântica que será padrão em uma versão futura de Python. O módulo `__future__` documenta os possíveis valores de *feature*. Importando esse módulo e avaliando suas variáveis, você pode ver quando um novo recurso foi inicialmente adicionado à linguagem e quando será (ou se já é) o padrão:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

coleta de lixo
Também conhecido como *garbage collection*, é o processo de liberar a memória quando ela não é mais utilizada. Python executa a liberação da memória através da contagem de referências e um coletor de lixo cíclico que é capaz de detectar e interromper referências cíclicas. O coletor de lixo pode ser controlado usando o módulo `gc`.

gerador
Uma função que retorna um *iterador gerador*. É parecida com uma função normal, exceto pelo fato de conter expressões `yield` para produzir uma série de valores que podem ser usados em um laço "for" ou que podem ser obtidos um de cada vez com a função `next()`.

Normalmente refere-se a uma função geradora, mas pode referir-se a um *iterador gerador* em alguns contextos. Em alguns casos onde o significado desejado não está claro, usar o termo completo evita ambiguidade.

iterador gerador
Um objeto criado por uma função *geradora*.

Cada `yield` suspende temporariamente o processamento, memorizando o estado da execução (incluindo variáveis locais e instruções try pendentes). Quando o *iterador gerador* retorna, ele se recupera do último ponto onde estava (em contrapartida as funções que iniciam uma nova execução a cada vez que são invocadas).

expressão geradora
Uma *expressão* que retorna um *iterador*. Parece uma expressão normal, seguido de uma cláusula `for` definindo uma variável de laço, um intervalo, e uma cláusula `if` opcional. A expressão combinada gera valores para uma função encapsuladora:

```
>>> sum(i*i for i in range(10))         # soma dos quadrados 0, 1, 4, ... 81
285
```

função genérica
Uma função composta por várias funções implementando a mesma operação para diferentes tipos. Qual implementação deverá ser usada durante a execução é determinada pelo algoritmo de despacho.

Veja também a entrada *despacho único* no glossário, o decorador `functools.singledispatch()`, e a **PEP 443**.

tipo genérico
Um *tipo* que pode ser parametrizado; tipicamente uma classe contêiner tal como `list` ou `dict`. Usado para *dicas de tipo* e *anotações*.

Para mais detalhes, veja tipo apelido genérico, **PEP 483**, **PEP 484**, **PEP 585**, e o módulo `typing`.

GIL
Veja *trava global do interpretador*.

trava 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 embutidos 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 a GIL ao realizar tarefas computacionalmente muito intensas, como compactação ou cálculos de hash. Além disso, a GIL é sempre liberado nas operações de E/S.

A partir de Python 3.13, o GIL pode ser desabilitado usando a configuração de construção `--disable-gil`. Depois de construir Python com essa opção, o código deve ser executado com a opção `-X gil=0` ou a variável de ambiente `PYTHON_GIL=0` deve estar definida. Esse recurso provê um desempenho melhor para aplicações com múltiplas threads e torna mais fácil o uso eficiente de CPUs com múltiplos núcleos. Para mais detalhes, veja **PEP 703**.

pyc baseado em hash
Um arquivo de cache em bytecode que usa hash ao invés do tempo, no qual o arquivo de código-fonte foi modificado pela última vez, para determinar a sua validade. Veja pyc-invalidation.

hasheável
Um objeto é *hasheável* se tem um valor de hash que nunca muda durante seu ciclo de vida (precisa ter um método `__hash__()`) e pode ser comparado com outros objetos (precisa ter um método `__eq__()`). Objetos hasheáveis que são comparados como iguais devem ter o mesmo valor de hash.

A hasheabilidade faz com que um objeto possa ser usado como uma chave de dicionário e como um membro de conjunto, pois estas estruturas de dados utilizam os valores de hash internamente.

A maioria dos objetos embutidos imutáveis do Python são hasheáveis; containers mutáveis (tais como listas ou dicionários) não são; containers imutáveis (tais como tuplas e frozensets) são hasheáveis apenas se os seus

elementos são hasheáveis. Objetos que são instâncias de classes definidas pelo usuário são hasheáveis por padrão. Todos eles comparam de forma desigual (exceto entre si mesmos), e o seu valor hash é derivado a partir do seu `id()`.

IDLE
Um ambiente de desenvolvimento e aprendizado integrado para Python. idle é um editor básico e um ambiente interpretador que vem junto com a distribuição padrão do Python.

imortal
Objetos imortais são um detalhe da implementação do CPython introduzida na **PEP 683**.

Se um objeto é imortal, sua *contagem de referências* nunca é modificada e, portanto, nunca é desalocado enquanto o interpretador está em execução. Por exemplo, `True` e `None` são imortais no CPython.

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.

caminho de importação
Uma lista de localizações (ou *entradas de caminho*) que são buscadas pelo *localizador baseado no caminho* por módulos para importar. Durante a importação, esta lista de localizações usualmente vem a partir de `sys.path`, mas para subpacotes ela também pode vir do atributo `__path__` de pacotes-pai.

importação
O processo pelo qual o código Python em um módulo é disponibilizado para o código Python em outro módulo.

importador
Um objeto que localiza e carrega um módulo; Tanto um *localizador* e o objeto *carregador*.

interativo
Python tem um interpretador interativo, o que significa que você pode digitar instruções 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)`). Para saber mais sobre modo interativo, veja *Modo interativo*.

interpretado
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*.

desligamento do interpretador
Quando solicitado para desligar, o interpretador Python entra em uma fase especial, onde ele gradualmente libera todos os recursos alocados, tais como módulos e várias estruturas internas críticas. Ele também faz diversas chamadas para o *coletor de lixo*. Isto pode disparar a execução de código em destrutores definidos pelo usuário ou função de retorno de referência fraca. Código executado durante a fase de desligamento pode encontrar diversas exceções, pois os recursos que ele depende podem não funcionar mais (exemplos comuns são os módulos de bibliotecas, ou os mecanismos de avisos).

A principal razão para o interpretador desligar, é que o módulo `__main__` ou o script sendo executado terminou sua execução.

iterável
Um objeto capaz de retornar seus membros um de cada vez. Exemplos de iteráveis incluem todos os tipos de sequência (tais como `list`, `str` e `tuple`) e alguns tipos de não-sequência, como o `dict`, *objetos arquivos*, além dos objetos de quaisquer classes que você definir com um método `__iter__()` ou `__getitem__()` que implementam a semântica de *sequência* .

Iteráveis podem ser usados em um laço `for` e em vários outros lugares em que uma sequência é necessária (`zip()`, `map()`, ...). Quando um objeto iterável é passado como argumento para a função embutida `iter()`, ela retorna um iterador para o objeto. Este iterador é adequado para se varrer todo o conjunto de valores. Ao

usar iteráveis, normalmente não é necessário chamar `iter()` ou lidar com os objetos iteradores em si. A instrução `for` faz isso automaticamente para você, criando uma variável temporária para armazenar o iterador durante a execução do laço. Veja também *iterador*, *sequência*, e *gerador*.

iterador
Um objeto que representa um fluxo de dados. Repetidas chamadas ao método `__next__()` de um iterador (ou passando o objeto para a função embutida `next()`) vão retornar itens sucessivos do fluxo. Quando não houver mais dados disponíveis uma exceção `StopIteration` será levantada. Neste ponto, o objeto iterador se esgotou e quaisquer chamadas subsequentes a seu método `__next__()` vão apenas levantar a exceção `StopIteration` novamente. Iteradores precisam ter um método `__iter__()` que retorne o objeto iterador em si, de forma que todo iterador também é iterável e pode ser usado na maioria dos lugares em que um iterável é requerido. Uma notável exceção é código que tenta realizar passagens em múltiplas iterações. Um objeto contêiner (como uma `list`) produz um novo iterador a cada vez que você passá-lo para a função `iter()` ou utilizá-lo em um laço `for`. Tentar isso com o mesmo iterador apenas iria retornar o mesmo objeto iterador esgotado já utilizado na iteração anterior, como se fosse um contêiner vazio.

Mais informações podem ser encontradas em typeiter.

O CPython não aplica consistentemente o requisito de que um iterador defina `__iter__()`. E também observe que o CPython com threads livres não garante a segurança do thread das operações do iterador.

função chave
Uma função chave ou função colação é um chamável que retorna um valor usado 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.

Uma porção de ferramentas no Python aceitam funções chave para controlar como os elementos são ordenados ou agrupados. Algumas delas incluem `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.merge()`, `heapq.nsmallest()`, `heapq.nlargest()` e `itertools.groupby()`.

Há várias maneiras de se criar funções chave. Por exemplo, o método `str.lower()` pode servir como uma função chave para ordenações insensíveis à caixa. Alternativamente, uma função chave ad-hoc pode ser construída a partir de uma expressão `lambda`, como `lambda r: (r[0], r[2])`. Além disso, `operator.attrgetter()`, `operator.itemgetter()` e `operator.methodcaller()` são três construtores de função chave. Consulte o guia de Ordenação para ver exemplos de como criar e utilizar funções chave.

argumento nomeado
Veja *argumento*.

lambda
Uma função de linha anônima consistindo de uma única *expressão*, 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 muitas instruções `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 travas ou usando a abordagem EAFP.

lista
Uma *sequência* 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 $O(1)$.

compreensão de lista
Uma maneira compacta de processar todos ou parte dos elementos de uma sequência e retornar os resultados em uma lista. `result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]` gera uma lista de strings contendo números hexadecimais (0x..) no intervalo de 0 a 255. A cláusula `if` é opcional. Se omitida, todos os elementos no `range(256)` serão processados.

carregador
Um objeto que carrega um módulo. Ele deve definir os métodos `exec_module()` e `create_module()` para

implementar a interface `Loader`. Um carregador é normalmente retornado por um *localizador*. Veja também:

- finders-and-loaders
- `importlib.abc.Loader`
- **PEP 302**

codificação da localidade
No Unix, é a codificação da localidade do LC_CTYPE, que pode ser definida com `locale.setlocale(locale.LC_CTYPE, new_locale)`.

No Windows, é a página de código ANSI (ex: `"cp1252"`).

No Android e no VxWorks, o Python usa `"utf-8"` como a codificação da localidade.

`locale.getencoding()` pode ser usado para obter a codificação da localidade.

Veja também *tratador de erros e codificação do sistema de arquivos*.

método mágico
Um sinônimo informal para um *método especial*.

mapeamento
Um objeto contêiner que tem suporte a pesquisas de chave arbitrária e implementa os métodos especificados nas `collections.abc.Mapping` ou `collections.abc.MutableMapping` classes base abstratas. Exemplos incluem `dict`, `collections.defaultdict`, `collections.OrderedDict` e `collections.Counter`.

localizador de metacaminho
Um *localizador* retornado por uma busca de `sys.meta_path`. Localizadores de metacaminho são relacionados a, mas diferentes de, *localizadores de entrada de caminho*.

Veja `importlib.abc.MetaPathFinder` para os métodos que localizadores de metacaminho implementam.

metaclasse
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.

Mais informações podem ser encontradas em metaclasses.

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 *escopo aninhado*.

ordem de resolução de métodos
Ordem de resolução de métodos é a ordem em que os membros de uma classe base são buscados durante a pesquisa. Veja python_2.3_mro para detalhes do algoritmo usado pelo interpretador do Python desde a versão 2.3.

módulo
Um objeto que serve como uma unidade organizacional de código Python. Os módulos têm um espaço de nomes contendo objetos Python arbitrários. Os módulos são carregados pelo Python através do processo de *importação*.

Veja também *pacote*.

spec de módulo
Um espaço de nomes que contém as informações relacionadas à importação usadas para carregar um módulo. Uma instância de `importlib.machinery.ModuleSpec`.

Veja também module-specs.

MRO
Veja *ordem de resolução de métodos*.

mutável
Objeto mutável é aquele que pode modificar seus valor mas manter seu `id()`. Veja também *imutável*.

tupla nomeada
O termo "tupla nomeada" é aplicado a qualquer tipo ou classe que herda de tupla e cujos elementos indexáveis também são acessíveis usando atributos nomeados. O tipo ou classe pode ter outras funcionalidades também.

Diversos tipos embutidos são tuplas nomeadas, incluindo os valores retornados por `time.localtime()` e `os.stat()`. Outro exemplo é `sys.float_info`:

```
>>> sys.float_info[1]                    # acesso indexado
1024
>>> sys.float_info.max_exp               # acesso a campo nomeado
1024
>>> isinstance(sys.float_info, tuple)    # tipo de tupla
True
```

Algumas tuplas nomeadas são tipos embutidos (tal como os exemplos acima). Alternativamente, uma tupla nomeada pode ser criada a partir de uma definição de classe regular, que herde de `tuple` e que defina campos nomeados. Tal classe pode ser escrita a mão, ou ela pode ser criada herdando `typing.NamedTuple` ou com uma função fábrica `collections.namedtuple()`. As duas últimas técnicas também adicionam alguns métodos extras, que podem não ser encontrados quando foi escrita manualmente, ou em tuplas nomeadas embutidas.

espaço de nomes
O lugar em que uma variável é armazenada. Espaços de nomes são implementados como dicionários. Existem os espaços de nomes local, global e nativo, bem como espaços de nomes aninhados em objetos (em métodos). Espaços de nomes suportam modularidade ao prevenir conflitos de nomes. Por exemplo, as funções `__builtin__.open()` e `os.open()` são diferenciadas por seus espaços de nomes. Espaços de nomes também auxiliam na legibilidade e na manutenibilidade ao torar mais claro quais módulos implementam uma função. Escrever `random.seed()` ou `itertools.izip()`, por exemplo, deixa claro que estas funções são implementadas pelos módulos `random` e `itertools` respectivamente.

pacote de espaço de nomes
Um *pacote* que serve apenas como contêiner para subpacotes. Pacotes de espaços de nomes podem não ter representação física, e especificamente não são como um *pacote regular* porque eles não tem um arquivo `__init__.py`.

Pacotes de espaço de nomes permitem que vários pacotes instaláveis individualmente tenham um pacote pai comum. Caso contrário, é recomendado usar um *pacote regular*.

Para mais informações, veja **PEP 420** e reference-namespace-package.

Veja também *módulo*.

escopo aninhado
A habilidade de referir-se a uma variável em uma definição de fechamento. Por exemplo, uma função definida dentro de outra pode referenciar variáveis da função externa. Perceba que escopos aninhados por padrão funcionam apenas por referência e não por atribuição. Variáveis locais podem ler e escrever no escopo mais interno. De forma similar, variáveis globais podem ler e escrever para o espaço de nomes global. O `nonlocal` permite escrita para escopos externos.

classe estilo novo
Antigo nome para o tipo de classes agora usado para todos os objetos de classes. Em versões anteriores do Python, apenas classes estilo podiam usar recursos novos e versáteis do Python, tais como `__slots__`, descritores, propriedades, `__getattribute__()`, métodos de classe, e métodos estáticos.

objeto
Qualquer dado que tenha estado (atributos ou valores) e comportamento definidos (métodos). Também a última classe base de qualquer *classe estilo novo*.

escopo otimizado
Um escopo no qual os nomes das variáveis locais de destino são conhecidos de forma confiável pelo compilador quando o código é compilado, permitindo a otimização do acesso de leitura e gravação a esses nomes. Os espaços de nomes locais para funções, geradores, corrotinas, compreensões e expressões geradoras são otimizados desta forma. Nota: a maioria das otimizações de interpretador são aplicadas a todos os escopos, apenas aquelas que dependem de um conjunto conhecido de nomes de variáveis locais e não locais são restritas a escopos otimizados.

pacote
Um *módulo* Python é capaz de conter submódulos ou recursivamente, subpacotes. Tecnicamente, um pacote é um módulo Python com um atributo `__path__`.

Veja também *pacote regular* e *pacote de espaço de nomes*.

parâmetro
Uma entidade nomeada na definição de uma *função* (ou método) que especifica um *argumento* (ou em alguns casos, argumentos) que a função pode receber. Existem cinco tipos de parâmetros:

- *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): ...
```

- *somente-posicional*: especifica um argumento que pode ser fornecido apenas por posição. Parâmetros somente-posicionais podem ser definidos incluindo o caractere `/` na lista de parâmetros da definição da função após eles, por exemplo *somentepos1* e *somentepos2* a seguir:

```
def func(somentepos1, somentepos2, /, posicional_ou_nomeado): ...
```

- *somente-nomeado*: especifica um argumento que pode ser passado para a função somente por nome. Parâmetros somente-nomeados podem ser definidos com um simples parâmetro var-posicional ou um `*` antes deles na lista de parâmetros na definição da função, por exemplo *somente_nom1* and *somente_nom2* a seguir:

```
def func(arg, *, somente_nom1, somente_nom2): ...
```

- *var-posicional*: especifica que 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 do parâmetro, 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 padrão para alguns argumentos opcionais.

Veja também o termo *argumento* no glossário, a pergunta do FAQ sobre a diferença entre argumentos e parâmetros, a classe `inspect.Parameter`, a seção function e a **PEP 362**.

entrada de caminho
Um local único no *caminho de importação* que o *localizador baseado no caminho* consulta para encontrar módulos a serem importados.

localizador de entrada de caminho
Um *localizador* retornado por um chamável em `sys.path_hooks` (ou seja, um *gancho de entrada de caminho*) que sabe como localizar os módulos *entrada de caminho*.

Veja `importlib.abc.PathEntryFinder` para os métodos que localizadores de entrada de caminho implementam.

gancho de entrada de caminho
Um chamável na lista `sys.path_hooks` que retorna um *localizador de entrada de caminho* caso saiba como localizar módulos em uma *entrada de caminho* específica.

localizador baseado no caminho
Um dos *localizadores de metacaminho* padrão que procura por um *caminho de importação* de módulos.

objeto caminho ou similar
Um objeto representando um caminho de sistema de arquivos. Um objeto caminho ou similar é ou um objeto `str` ou `bytes` representando um caminho, ou um objeto implementando o protocolo `os.PathLike`. Um objeto que suporta o protocolo `os.PathLike` pode ser convertido para um arquivo de caminho do sistema `str` ou `bytes`, através da chamada da função `os.fspath()`; `os.fsdecode()` e `os.fsencode()` podem ser usadas para garantir um `str` ou `bytes` como resultado, respectivamente. Introduzido na **PEP 519**.

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 têm 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**.

porção
Um conjunto de arquivos em um único diretório (possivelmente armazenado em um arquivo zip) que contribuem para um pacote de espaço de nomes, conforme definido em **PEP 420**.

argumento posicional
Veja *argumento*.

API provisória
Uma API provisória é uma API que foi deliberadamente excluída das bibliotecas padrões com compatibilidade retroativa garantida. Enquanto mudanças maiores para tais interfaces não são esperadas, contanto que elas sejam marcadas como provisórias, mudanças retroativas incompatíveis (até e incluindo a remoção da interface) podem ocorrer se consideradas necessárias pelos desenvolvedores principais. Tais mudanças não serão feitas gratuitamente – elas irão ocorrer apenas se sérias falhas fundamentais forem descobertas, que foram esquecidas anteriormente a inclusão da API.

Mesmo para APIs provisórias, mudanças retroativas incompatíveis são vistas como uma "solução em último caso" - cada tentativa ainda será feita para encontrar uma resolução retroativa compatível para quaisquer problemas encontrados.

Esse processo permite que a biblioteca padrão continue a evoluir com o passar do tempo, sem se prender em erros de design problemáticos por períodos de tempo prolongados. Veja **PEP 411** para mais detalhes.

pacote provisório
Veja *API provisória*.

Python 3000
Apelido para a linha de lançamento da versão do Python 3.x (cunhada há muito tempo, quando o lançamento da versão 3 era algo em um futuro muito distante.) Esse termo possui a seguinte abreviação: "Py3k".

Pythônico
Uma ideia ou um pedaço de código que segue de perto as formas de escritas mais comuns da linguagem Python, ao invés de implementar códigos usando conceitos comuns a outras linguagens. Por exemplo, um formato comum em Python é fazer um laço 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(comida)):
    print(comida[i])
```

Ao contrário do método mais limpo, Pythônico:

```
for parte in comida:
    print(parte)
```

nome qualificado
Um nome pontilhado (quando 2 termos são ligados por um ponto) que mostra o "path" do escopo global de um módulo para uma classe, função ou método definido num determinado módulo, conforme definido pela **PEP 3155**. Para funções e classes de nível superior, o nome qualificado é o mesmo que o nome do objeto:

```
>>> class C:
...     class D:
...         def metodo(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.metodo.__qualname__
'C.D.metodo'
```

Quando usado para se referir a módulos, o *nome totalmente qualificado* significa todo o caminho pontilhado para o módulo, incluindo quaisquer pacotes pai, por exemplo: `email.mime.text`:

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

contagem de referências
O número de referências a um objeto. Quando a contagem de referências de um objeto cai para zero, ele é desalocado. Alguns objetos são *imortais* e têm contagens de referências que nunca são modificadas e, portanto, os objetos nunca são desalocados. A contagem de referências geralmente não é visível para o código Python, mas é um elemento-chave da implementação do *CPython*. Os programadores podem chamar a função `sys.getrefcount()` para retornar a contagem de referências para um objeto específico.

pacote regular
Um *pacote* tradicional, como um diretório contendo um arquivo `__init__.py`.

Veja também *pacote de espaço de nomes*.

REPL
Um acrônimo para "read–eval–print loop", outro nome para o console *interativo* do interpretador.

__slots__
Uma declaração dentro de uma classe que economiza memória pré-declarando espaço para atributos de instâncias, e eliminando dicionários de instâncias. Apesar de popular, a técnica é um tanto quanto complicada de acertar, e é melhor se for reservada para casos raros, onde existe uma grande quantidade de instâncias em uma aplicação onde a memória é crítica.

sequência
Um *iterável* com suporte para acesso eficiente a seus elementos através de índices inteiros via método especial `__getitem__()` e que define o método `__len__()` que devolve o tamanho da sequência. Alguns tipos de sequência embutidos são: `list`, `str`, `tuple`, e `bytes`. Note que `dict` também tem suporte para `__getitem__()` e `__len__()`, mas é considerado um mapeamento e não uma sequência porque a busca usa uma chave *hasheável* arbitrária em vez de inteiros.

A classe base abstrata `collections.abc.Sequence` define uma interface mais rica que vai além de apenas `__getitem__()` e `__len__()`, adicionando `count()`, `index()`, `__contains__()`, e `__reversed__()`. Tipos que implementam essa interface podem ser explicitamente registrados usando `register()`. Para mais documentação sobre métodos de sequências em geral, veja Operações comuns de sequências.

compreensão de conjunto
Uma maneira compacta de processar todos ou parte dos elementos em iterável e retornar um conjunto com os resultados. `results = {c for c in 'abracadabra' if c not in 'abc'}` gera um conjunto de strings `{'r', 'd'}`. Veja comprehensions.

despacho único
Uma forma de despacho de *função genérica* onde a implementação é escolhida com base no tipo de um único argumento.

fatia
Um objeto geralmente contendo uma parte de uma *sequência*. Uma fatia é criada usando a notação de subscrito `[]` pode conter também até dois pontos entre números, como em `variable_name[1:3:5]`. A notação de suporte (subscrito) utiliza objetos `slice` internamente.

suavemente descontinuado
Uma API suavemente descontinuada não deve ser usada em código novo, mas é seguro para código já existente usá-la. A API continua documentada e testada, mas não será aprimorada mais.

A descontinuação suave, diferentemente da descontinuação normal, não planeja remover a API e não emitirá avisos.

Veja PEP 387: Descontinuação suave.

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 *expressão* ou uma de várias construções com uma palavra reservada, tal como `if`, `while` ou `for`.

verificador de tipo estático
Uma ferramenta externa que lê o código Python e o analisa, procurando por problemas como tipos incorretos. Consulte também *dicas de tipo* e o módulo `typing`.

referência forte
Na API C do Python, uma referência forte é uma referência a um objeto que pertence ao código que contém a referência. A referência forte é obtida chamando `Py_INCREF()` quando a referência é criada e liberada com `Py_DECREF()` quando a referência é excluída.

A função `Py_NewRef()` pode ser usada para criar uma referência forte para um objeto. Normalmente, a função `Py_DECREF()` deve ser chamada na referência forte antes de sair do escopo da referência forte, para evitar o vazamento de uma referência.

Veja também *referência emprestada*.

codificador de texto
Uma string em Python é uma sequência de pontos de código Unicode (no intervalo `U+0000–U+10FFFF`). Para armazenar ou transferir uma string, ela precisa ser serializada como uma sequência de bytes.

A serialização de uma string em uma sequência de bytes é conhecida como "codificação" e a recriação da string a partir de uma sequência de bytes é conhecida como "decodificação".

Há uma variedade de diferentes serializações de texto codecs, que são coletivamente chamadas de "codificações de texto".

arquivo texto
Um *objeto arquivo* apto a ler e escrever objetos `str`. Geralmente, um arquivo texto, na verdade, acessa um fluxo de dados de bytes e captura o *codificador de texto* automaticamente. Exemplos de arquivos texto são: arquivos abertos em modo texto (`'r'` or `'w'`), `sys.stdin`, `sys.stdout`, e instâncias de `io.StringIO`.

Veja também *arquivo binário* para um objeto arquivo apto a ler e escrever *objetos byte ou similar*.

aspas triplas
Uma string que está definida com três ocorrências de aspas duplas (") ou apóstrofos ('). 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 escapadas 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.

tipo
O tipo de um objeto Python determina qual classe 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)`.

apelido de tipo
Um sinônimo para um tipo, criado através da atribuição do tipo para um identificador.

Apelidos de tipo são úteis para simplificar *dicas de tipo*. Por exemplo:

```
def remove_tons_de_cinza(
        cores: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
    pass
```

pode tornar-se mais legível desta forma:

```
Cor = tuple[int, int, int]

def remove_tons_de_cinza(cores: list[Cor]) -> list[Cor]:
    pass
```

Veja `typing` e **PEP 484**, a qual descreve esta funcionalidade.

dica de tipo
Uma *anotação* que especifica o tipo esperado para uma variável, um atributo de classe, ou um parâmetro de função ou um valor de retorno.

Dicas de tipo são opcionais e não são forçadas pelo Python, mas elas são úteis para *verificadores de tipo estático*. Eles também ajudam IDEs a completar e refatorar código.

Dicas de tipos de variáveis globais, atributos de classes, e funções, mas não de variáveis locais, podem ser acessadas usando `typing.get_type_hints()`.

Veja `typing` e **PEP 484**, a qual descreve esta funcionalidade.

novas linhas universais
Uma maneira de interpretar fluxos de textos, na qual todos estes são reconhecidos como caracteres de fim de linha: a convenção para fim de linha no Unix `'\n'`, a convenção no Windows `'\r\n'`, e a antiga convenção no Macintosh `'\r'`. Veja **PEP 278** e **PEP 3116**, bem como `bytes.splitlines()` para uso adicional.

anotação de variável
Uma *anotação* de uma variável ou um atributo de classe.

Ao fazer uma anotação de uma variável ou um atributo de classe, a atribuição é opcional:

```
class C:
    campo: 'anotação'
```

Anotações de variáveis são normalmente usadas para *dicas de tipo*: por exemplo, espera-se que esta variável receba valores do tipo `int`:

```
contagem: int = 0
```

A sintaxe de anotação de variável é explicada na seção annassign.

Veja *anotação de função*, **PEP 484** e **PEP 526**, que descrevem esta funcionalidade. Veja também annotations-howto para as melhores práticas sobre como trabalhar com anotações.

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.

Veja também `venv`.

máquina virtual
: Um computador definido inteiramente em software. A máquina virtual de Python executa o *bytecode* emitido pelo compilador de bytecode.

Zen do 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 esta documentação

A documentação do Python é gerada a partir de fontes reStructuredText usando Sphinx, um gerador de documentação criado originalmente para Python e agora mantido como um projeto independente.

O desenvolvimento da documentação e de suas ferramentas é um esforço totalmente voluntário, como 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 Python e autor de boa parte do conteúdo;
- O projeto Docutils por criar reStructuredText e o pacote Docutils;
- Fredrik Lundh, pelo seu projeto de referência alternativa em Python, do qual Sphinx pegou muitas boas ideias.

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

Python foi criado no início dos anos 1990 por Guido van Rossum no Stichting Mathematisch Centrum (CWI, veja https://www.cwi.nl) na Holanda como sucessor de uma linguagem chamada ABC. Guido continua sendo o principal autor do Python, embora inclua muitas contribuições de outros.

Em 1995, Guido continuou seu trabalho em Python na Corporation for National Research Initiatives (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 de desenvolvimento do núcleo Python mudaram-se para BeOpen.com para formar a equipe BeOpen PythonLabs. Em outubro do mesmo ano, a equipe PythonLabs mudou-se para a Digital Creations, que se tornou Zope Corporation. Em 2001, a Python Software Foundation (PSF, veja https://www.python.org/psf/) foi formada, uma organização sem fins lucrativos criada especificamente para possuir Propriedade Intelectual relacionada ao Python. A Zope Corporation era um membro patrocinador da 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.

Versão	Derivada de	Ano	Proprietário	Compatível com a GPL? (1)
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	sim (2)
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

(1) Compatível com a 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 a GPL possibilitam combinar o Python com outro software lançado sob a GPL; os outros não.

(2) De acordo com Richard Stallman, 1.6.1 não é compatível com GPL, porque sua licença tem uma cláusula de escolha de lei. De acordo com a CNRI, no entanto, o advogado de Stallman disse ao advogado da CNRI que 1.6.1 "não é incompatível" com a GPL.

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

O software e a documentação do Python são licenciados sob a Python Software Foundation License Versão 2.

A partir do Python 3.8.6, exemplos, receitas e outros códigos na documentação são licenciados duplamente sob o Licença PSF versão 2 e a *Licença BSD de Zero Cláusula*.

Alguns softwares incorporados ao Python estão sob licenças diferentes. As licenças são listadas com o código abrangido por essa licença. Veja *Licenças e Reconhecimentos para Software Incorporado* para uma lista incompleta dessas licenças.

C.2.1 PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and
   the Individual or Organization ("Licensee") accessing and otherwise using this
   software ("Python") 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 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice of
   copyright, i.e., "Copyright © 2001-2024 Python Software Foundation; All Rights
   Reserved" are retained in Python 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 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.

4. PSF is making Python 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 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON, 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.
```

(continua na próxima página)

(continuação da página anterior)

```
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, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.
```

C.2.2 ACORDO DE LICENCIAMENTO DA BEOPEN.COM PARA PYTHON 2.0

ACORDO DE LICENCIAMENTO DA BEOPEN DE FONTE ABERTA DO 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
   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
   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.
```

(continua na próxima página)

(continuação da página anterior)

```
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.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON DOCUMENTATION

```
Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH
REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT,
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 reconhecimentos para softwares de terceiros incorporados na distribuição do Python.

C.3.1 Mersenne Twister

A extensão C `_random` subjacente ao 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)
```

(continua na próxima página)

(continuação da página anterior)

```
or init_by_array(init_key, key_length).

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 Soquetes

O módulo `socket` usa as funções `getaddrinfo()` e `getnameinfo()`, que são codificadas em arquivos de origem separados do Projeto WIDE, https://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 Serviços de soquete assíncrono

Os módulos `test.support.asynchat` e `test.support.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.3.4 Gerenciamento de cookies

O módulo `http.cookies` contém o seguinte aviso:

```
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
```

(continua na próxima página)

(continuação da página anterior)

```
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.3.5 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.3.6 Funções UUencode e UUdecode

O codec `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
```

(continua na próxima página)

(continuação da página anterior)

```
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
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.7 Chamadas de procedimento remoto XML

O módulo `xmlrpc.client` contém o seguinte aviso:

```
    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.3.8 test_epoll

O módulo `test.test_epoll` contém o seguinte aviso:

```
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:
```

(continua na próxima página)

(continuação da página anterior)

```
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.3.9 kqueue de seleção

O módulo `select` contém o seguinte aviso para a interface do kqueue:

```
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.3.10 SipHash24

O arquivo `Python/pyhash.c` contém a implementação de Marek Majkowski do algoritmo SipHash24 de Dan Bernstein. Contém a seguinte nota:

```
<MIT License>
Copyright (c) 2013  Marek Majkowski <marek@popcount.org>

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.
```

(continua na próxima página)

(continuação da página anterior)

```
</MIT License>

Original location:
   https://github.com/majek/csiphash/

Solution inspired by code from:
   Samuel Neves (supercop/crypto_auth/siphash24/little)
   djb (supercop/crypto_auth/siphash24/little2)
   Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

C.3.11 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 https://web.archive.org/web/20220517033456/http://www.netlib.org/fp/dtoa.c. 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.
 *
 * 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.3.12 OpenSSL

Os módulos `hashlib`, `posix` e `ssl` 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: Para o lançamento do OpenSSL 3.0, e lançamentos posteriores derivados deste, se aplica a Apache License v2:

```
                                 Apache License
                           Version 2.0, January 2004
                        https://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by
   the copyright owner that is granting the License.
```

(continua na próxima página)

(continuação da página anterior)

```
   "Legal Entity" shall mean the union of the acting entity and all
   other entities that control, are controlled by, or are under common
   control with that entity. For the purposes of this definition,
   "control" means (i) the power, direct or indirect, to cause the
   direction or management of such entity, whether by contract or
   otherwise, or (ii) ownership of fifty percent (50%) or more of the
   outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity
   exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications,
   including but not limited to software source code, documentation
   source, and configuration files.

   "Object" form shall mean any form resulting from mechanical
   transformation or translation of a Source form, including but
   not limited to compiled object code, generated documentation,
   and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or
   Object form, made available under the License, as indicated by a
   copyright notice that is included in or attached to the work
   (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object
   form, that is based on (or derived from) the Work and for which the
   editorial revisions, annotations, elaborations, or other modifications
   represent, as a whole, an original work of authorship. For the purposes
   of this License, Derivative Works shall not include works that remain
   separable from, or merely link (or bind by name) to the interfaces of,
   the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including
   the original version of the Work and any modifications or additions
   to that Work or Derivative Works thereof, that is intentionally
   submitted to Licensor for inclusion in the Work by the copyright owner
   or by an individual or Legal Entity authorized to submit on behalf of
   the copyright owner. For the purposes of this definition, "submitted"
   means any form of electronic, verbal, or written communication sent
   to the Licensor or its representatives, including but not limited to
   communication on electronic mailing lists, source code control systems,
   and issue tracking systems that are managed by, or on behalf of, the
   Licensor for the purpose of discussing and improving the Work, but
   excluding communication that is conspicuously marked or otherwise
   designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity
   on behalf of whom a Contribution has been received by Licensor and
   subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
```

(continua na próxima página)

(continuação da página anterior)

```
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
       excluding those notices that do not pertain to any part of
       the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
       distribution, then any Derivative Works that You distribute must
       include a readable copy of the attribution notices contained
       within such NOTICE file, excluding those notices that do not
       pertain to any part of the Derivative Works, in at least one
       of the following places: within a NOTICE text file distributed
       as part of the Derivative Works; within the Source form or
       documentation, if provided along with the Derivative Works; or,
       within a display generated by the Derivative Works, if and
       wherever such third-party notices normally appear. The contents
       of the NOTICE file are for informational purposes only and
       do not modify the License. You may add Your own attribution
       notices within Derivative Works that You distribute, alongside
       or as an addendum to the NOTICE text from the Work, provided
       that such additional attribution notices cannot be construed
       as modifying the License.

   You may add Your own copyright statement to Your modifications and
   may provide additional or different license terms and conditions
   for use, reproduction, or distribution of Your modifications, or
```

(continua na próxima página)

(continuação da página anterior)

```
   for any such Derivative Works as a whole, provided Your use,
   reproduction, and distribution of the Work otherwise complies with
   the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
   this License, without any additional terms or conditions.
   Notwithstanding the above, nothing herein shall supersede or modify
   the terms of any separate license agreement you may have executed
   with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
   names, trademarks, service marks, or product names of the Licensor,
   except as required for reasonable and customary use in describing the
   origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS
```

C.3.13 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.3.14 libffi

A extensão C `_ctypes` subjacente ao módulo `ctypes` é construída usando uma cópia incluída das fontes do libffi, a menos que a construção esteja configurada com `--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.3.15 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) 1995-2011 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
```

C.3.16 cfuhash

A implementação da tabela de hash usada pelo `tracemalloc` é baseada no projeto cfuhash:

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

  * Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

  * 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.

  * Neither the name of the author nor the names of its
    contributors may 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,
```

(continua na próxima página)

(continuação da página anterior)

```
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.3.17 libmpdec

A extensão C `_decimal` subjacente ao módulo `decimal` é construída usando uma cópia incluída da biblioteca libmpdec, a menos que a construção esteja configurada com `--with-system-libmpdec`:

```
Copyright (c) 2008-2020 Stefan Krah. 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.3.18 Conjunto de testes C14N do W3C

O conjunto de testes C14N 2.0 no pacote `test` (`Lib/test/xmltestdata/c14n-20/`) foi recuperado do site do W3C em https://www.w3.org/TR/xml-c14n2-testcases/ e é distribuído sob a licença BSD de 3 cláusulas:

```
Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang),
All Rights Reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

* Redistributions of works must retain the original copyright notice,
  this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the original copyright
  notice, this list of conditions and the following disclaimer in the
  documentation and/or other materials provided with the distribution.
* Neither the name of the W3C nor the names of its contributors may be
```

(continua na próxima página)

(continuação da página anterior)

```
  used to endorse or promote products derived from this work 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.
```

C.3.19 mimalloc

Licença MIT:

```
Copyright (c) 2018-2021 Microsoft Corporation, Daan Leijen

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.3.20 asyncio

Partes do módulo `asyncio` são incorporadas do uvloop 0.16, que é distribuído sob a licença MIT:

```
Copyright (c) 2015-2021 MagicStack Inc.  http://magic.io

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.
```

(continua na próxima página)

(continuação da página anterior)

```
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.3.21 Global Unbounded Sequences (GUS)

O arquivo `Python/qsbr.c` é adaptado do esquema de recuperação de memória segura "Global Unbounded Sequences" do FreeBSD em subr_smr.c. O arquivo é distribuído sob a licença BSD de 2 cláusulas:

```
Copyright (c) 2019,2020 Jeffrey Roberson <jeff@FreeBSD.org>

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 unmodified, 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 "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 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.
```

APÊNDICE D

Direitos autorais

Veja: *História e Licença* para informações completas de licença e permissões.

Índice

Não alfabético

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

Z