DOCUMENTAÇÃO OFICIAL

Python é uma linguagem de programação poderosa e de fácil aprendizado. Possui estruturas de dados de alto nível eficientes, bem como adota uma abordagem simples e efetiva para a programação orientada a objetos. Sua sintaxe elegante e tipagem dinâmica, além de sua natureza interpretada, tornam Python ideal para scripting e para o desenvolvimento rápido de aplicações em diversas áreas e na maioria das plataformas.

O interpretador Python e sua extensa biblioteca padrão estão disponíveis na forma de código fonte ou binário para a maioria das plataformas a partir do site, http://www.python.org/, e podem ser distribuídos livremente. No mesmo sítio estão disponíveis distribuições e referências para diversos módulos, programas, ferramentas e documentação adicional, contribuídos por terceiros.

O interpretador Python é facilmente extensível incorporando novas funções e tipos de dados implementados em C ou C++ (ou qualquer outra linguagem acessível a partir de C). Python também se adequa como linguagem de extensão para customizar aplicações. Este tutorial introduz o leitor informalmente aos conceitos básicos e aspectos do sistema e linguagem Python. É aconselhável ter um interpretador Python disponível para se poder “por a mão na massa”, porém todos os exemplos são auto-contidos, assim o tutorial também pode ser lido sem que haja a necessidade de se estar on-line.

Para uma descrição dos módulos e objetos padrão, veja o documento *The Python Standard Library*. O *The Python Language Reference*oferece uma definição formal da linguagem. Para se escrever extensões em C ou C++, leia *Extending and Embedding the Python Interprete*r e *Python/C API Reference Manual*. Existem também diversos livros abordando Python em maior profundidade.

Este tutorial não almeja ser abrangente ou abordar todos os aspectos, nem mesmo todos os mais frequentes. Ao invés disso, ele introduz muitas das características dignas de nota em Python, e fornecerá a você uma boa idéia sobre o estilo e o sabor da linguagem. Após a leitura, você deve ser capaz de ler e escrever programas e módulos em Python, e estará pronto para aprender mais sobre os diversos módulos de biblioteca descritos em *The Python Standard Library*.

1. Abrindo o apetite

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

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

Python é a linguagem para você.

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

Python é fácil de usar, sem deixar de ser uma linguagem de programação de verdade, oferecendo muito mais estruturação e suporte para programas extensos do que shell scripts 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: dicionários e vetores (arrays) flexíveis. Devido ao suporte nativo a uma variedade de tipos de dados, Python é aplicável a um domínio de problemas muito mais vasto do que Awk ou até mesmo Perl, ainda assim muitas tarefas são pelo menos tão fáceis em Python quanto nessas linguagens.

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

Python é uma linguagem interpretada, por isso você pode economizar um tempo considerável durante o desenvolvimento, uma vez que não há necessidade de compilação e vinculação (*linking*). O interpretador pode ser usado interativamente, o

que torna fácil experimentar diversas características da linguagem, escrever programas “descartáveis”, ou testar funções em um desenvolvimento bottom-up. É também uma útil calculadora de mesa.

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

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

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

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

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

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

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

2. Usando o interpretador Python

2.1. Disparando o interpretador

O interpretador é frequentemente instalado como `/usr/local/bin/python` nas máquinas onde está disponível; adicionando `/usr/local/bin` ao caminho de busca (search path) da shell de seu UNIX torna-se possível iniciá-lo digitando:

```
python
```

no console. Considerando que a escolha do diretório de instalação é uma opção de instalação, outras localizações são possíveis; verifique com seu guru local de Python ou com o administrador do sistema. (Ex.: `/usr/local/python` é uma alternativa popular para instalação.)

Em computadores com Windows, Python é instalado geralmente em `C:\Python27`, apesar de você poder mudar isso enquanto está executando o instalador. Para adicionar esse diretório ao path, você pode digitar o seguinte comando no console:

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

Digitando um caractere EOF (end-of-file; fim de arquivo: `Control-D` em Unix, `Control-Z` em 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: `quit()`.

As características de edição de linha não são muito sofisticadas. Sobre UNIX, quem instalou o interpretador talvez tenha habilitado o suporte à biblioteca GNU readline, que adiciona facilidades mais elaboradas de edição e histórico de comandos. Teclar `Control-P` no primeiro prompt oferecido pelo Python é, provavelmente, a maneira mais rápida de verificar se a edição de linha de comando é suportada. Se escutar um *beep*, você tem edição de linha de comando; veja o Apêndice *Edição interativa de entrada e substituição por histórico* para uma introdução às teclas especiais. Se nada acontecer, ou se ^P aparecer na tela, a opção de edição não está disponível; você apenas poderá usar o backspace para remover caracteres da linha atual.

O interpretador trabalha de forma semelhante a uma shell de UNIX: quando disparado com a saída padrão conectada a um console de terminal (dispositivo tty), ele lê e executa comandos interativamente; quando disparado com um nome de arquivo como parâmetro ou com redirecionamento da entrada padrão para ler um arquivo, o interpretador irá ler e executar o *script* contido em tal arquivo.

Uma segunda forma de rodar o interpretador é `python -c *comando* [arg] ...`, que executa um ou mais comandos especificados na posição*comando*, analogamente à opção de shell *-c*. Considerando que comandos Python frequentemente têm espaços

em branco (ou outros caracteres que são especiais para a shell) é aconselhável que o *comando* esteja dentro de aspas duplas.

Alguns módulos Python são também úteis como scripts. Estes podem ser chamados usando `python -m *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, as 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.

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`, que é uma lista de strings. Essa lista tem sempre ao menos um elemento; quando nenhum script ou argumento forem passados 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 (tty), diz-se que o interpretador está em modo interativo. Nesse modo ele solicita um próximo comando através do *prompt primário*, tipicamente três sinais de maior (>>>); para linhas de continuação do comando atual, o *prompt secundário* padrão é formado por três pontos (...). O interpretador exibe uma mensagem de boas vindas, informando seu número de versão e um aviso de copyright antes de exibir o primeiro prompt:

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

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

```
>>>
```

```
>>> o_mundo_eh_plano = 1
>>> if o_mundo_eh_plano:
...     print "Cuidado para não cair dele!"
...
Cuidado para não cair dele!
```

2.2. O interpretador e seu ambiente

2.2.1. Tratamento de erros

Quando ocorre um erro, o interpretador exibe uma mensagem de erro como um *stack trace* (a situação da pilha de execução). No modo interativo, ele retorna ao prompt primário; quando a entrada vem de um arquivo, o interpretador aborta a execução com status de erro diferente de zero após exibir o stack trace (Exceções tratadas por um **`except`** em um comando **`try`** não são consideradas erros neste contexto). Alguns erros são incondicionalmente fatais e causam a saída com status diferente de zero; isto se aplica a inconsistências internas e alguns casos de exaustão de memória. Todas as mensagens de erro são escritas na saída de erros padrão (*standard error*), enquanto a saída dos demais comandos é direcionada para a saída padrão.

Teclar o caractere de interrupção (tipicamente Control-C ou DEL) no prompt primário ou secundário cancela a entrada de dados corrente e volta ao prompt primário. [1] Provocar a interrupção enquanto um comando está em execução levanta a exceção **`KeyboardInterrupt`**, a qual pode ser tratada em um comando **`try`**.

2.2.2. Scripts executáveis em Python

Em sistemas UNIX, scripts Python podem ser transformados em executáveis, como shell scripts, pela inclusão desta linha no início do arquivo:

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

(assumindo que o interpretador foi incluído no **`PATH`** do usuário e que o script tenha a permissão de acesso habilitada para execução). Os caracteres `#!` devem ser os dois primeiros do arquivo. Em algumas plataformas esta linha inicial deve ser finalizada no estilo UNIX com (`'\\n'`), e não com a marca de fim de linha do Windows (`'\\r\\n'`). Observe que o caractere `'#'` inicia uma linha de comentário em Python.

Para atribuir modo executável ou permissão de execução ao seu script Python, utilize o comando **chmod** do shell do UNIX:

```
$ chmod +x meuscript.py
```

Em sistemas Windows, não há noção de um “modo executável”. O instalador de Python associa automaticamente arquivos `.py` arquivos a`python.exe` para que um clique duplo sobre um arquivo Python o execute como um script. A extensão pode também ser `.pyw`; nesse caso, a janela de console que normalmente aparece é suprimida.

2.2.3. Codificação em arquivos de código-fonte

É possível usar codificação diferente de ASCII em arquivos de código Python. A melhor maneira de fazê-lo é através de um comentário adicional logo após a linha `#!`:

```
# -*- coding: codificacao -*-
```

Com essa declaração, todos os caracteres no código-fonte serão tratados de acordo com a codificação especificada, e será possível escrever strings Unicode diretamente, usando aquela codificação. A lista de codificações possíveis pode ser encontrada na Referência da Biblioteca Python, na seção **codecs**.

Por exemplo, para escrever strings Unicode incluindo o símbolo monetário do Euro, a codificação ISO-8859-15 pode ser usada; nela símbolo do Euro tem o valor ordinal 164. Este script exibe o valor 8364 (código Unicode correspondente ao símbolo do Euro) e termina:

```
# -*- coding: iso-8859-15 -*-

currency = u"€"
print ord(currency)
```

Se o seu editor é capaz de salvar arquivos UTF-8 com *byte order mark* (conhecido como BOM), você pode usar isto ao invés da declaração de codificação. O IDLE é capaz de fazer isto se você habilitar `Options/General/Default Source Encoding/UTF-8`. Note que a assinatura BOM não é reconhecida por versões antigas (Python 2.2 e anteriores), nem pelo sistema operacional, invalidando a declaração `#!` (usada somente em sistemas UNIX).

Usando UTF-8 (seja através da assinatura ou de uma declaração de codificação), caracteres da maioria das línguas do mundo podem ser usados simultaneamente em strings e comentários. Não é possível usar caracteres não-ASCII em identificadores. Para exibir todos esses caracteres adequadamente, seu editor deve reconhecer que o arquivo é UTF-8, e deve usar uma fonte que tenha todos os caracteres usados no arquivo.

2.2.4. O arquivo de inicialização para o modo interativo

Quando usamos Python interativamente, pode ser útil executar uma série de comandos ao iniciar cada sessão do interpretador. Isso pode ser feito configurando a variável de ambiente **PYTHONSTARTUP** para indicar o nome de arquivo script que contém um script de inicialização. Essa característica assemelha-se aos arquivos `.profile` de shells UNIX.

Este arquivo só é processado em sessões interativas, nunca quando Python lê comandos de um script especificado como parâmetro, nem tampouco quando `/dev/tty` é especificado como a fonte de leitura de comandos (que de outra forma se comporta como uma sessão interativa). O script de inicialização é executado no mesmo namespace (espaço nominal ou contexto léxico) em que os comandos da sessão interativa serão executados, sendo assim, os objetos definidos e módulos importados podem ser utilizados sem qualificação durante a sessão interativa. É possível também redefinir os prompts `sys.ps1` e `sys.ps2` neste arquivo.

Se for necessário ler um script adicional de inicialização a partir do diretório atual, você pode programar isso a partir do script de inicialização global, por exemplo `if os.path.isfile('.pythonrc.py'): execfile('.pythonrc.py')`. Se você deseja utilizar o script de inicialização em outro script, você deve fazê-lo explicitamente da seguinte forma:

```
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
    execfile(filename)
```

2.2.5. Os módulos de customização

Python fornece dois hooks (ganchos) para que você possa personalizá-lo: **`sitecustomize`** e **`usercustomize`**. Para ver como funciona, você precisa primeiro encontrar o local de seu diretório site-packages de usuário. Inicie o Python e execute este código:

```
>>>
```

```
>>> import site
>>> site.getusersitepackages()
'/home/user/.local/lib/python2.7/site-packages'
```

Agora você pode criar um arquivo chamado `usercustomize.py` nesse diretório e colocar o que quiser nele. Isso afetará toda invocação de Python, a menos ele seja iniciado com a opção *-S* para desativar esta importação automática.

`sitecustomize` funciona da mesma forma, mas normalmente é criado por um administrador do sistema no diretório site-packages global, e é importado antes de **`usercustomize`**. Consulte a documentação do **`site`** para mais detalhes.

3. Uma introdução informal a Python

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

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

Alguns exemplos:

```
# este é o primeiro comentário
SPAM = 1                 # e este é o segundo comentário
                         # ... e agora um terceiro!
STRING = "# Isto não é um comentário."
```

3.1. Usando Python como uma calculadora

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

3.1.1. Números

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

```
>>>
```

```
>>> 2+2
4
>>> # Isto é um comentário
... 2+2
4
>>> 2+2  # em um comentário na mesma linha do código
4
>>> (50-5*6)/4
5
>>> # A divisão entre inteiros arredonda para baixo:
```

```
... 7/3
2
>>> 7/-3
-3
```

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

Um valor pode ser atribuído a diversas variáveis simultaneamente:

```
>>>
>>> x = y = z = 0  # Zerar x, y, z
>>> x
0
>>> y
0
>>> z
0
```

Variáveis precisam ser “definidas” (atribuídas um valor) antes que possam ser usadas, se não acontece um erro:

```
>>>
>>> # tentar acessar variável não definida
... n
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

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

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

Números complexos também são suportados; números imaginários são escritos com o sufixo `j` ou `J`. Números complexos com parte real não nula são escritos como `(real+imagJ)`, ou podem ser criados pela chamada de função `complex(real, imag)`.

```
>>>
```

```
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0,1)
(-1+0j)
>>> 3+1j*3
(3+3j)
>>> (3+1j)*3
(9+3j)
>>> (1+2j)/(1+1j)
(1.5+0.5j)
```

Números complexos são sempre representados por dois floats, a parte real e a parte imaginária. Para extrair as partes de um número complexo *z*, utilize `z.real` e `z.imag`.

```
>>>
```

```
>>> a=1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
```

As funções de conversão para float e inteiro (**`float()`**, **`int()`** e **`long()`**) não funcionam para números complexos — não existe apenas uma maneira de converter um número complexo para um número real. Use `abs(z)` para obter sua magnitude (como um float) ou `z.real` para obter sua parte real.

```
>>>
```

```
>>> a=3.0+4.0j
>>> float(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: can't convert complex to float; use abs(z)
>>> a.real
3.0
>>> a.imag
4.0
>>> abs(a)  # sqrt(a.real**2 + a.imag**2)
5.0
```

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
>>> preco = 100.50
>>> preco * taxa
12.5625
>>> preco + _
113.0625
>>> round(_, 2)
```

```
113.06
```

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

3.1.2. Strings

Além de números, Python também pode manipular strings (sequências de caracteres), que podem ser expressas de diversas formas. Elas podem ser delimitadas por aspas simples ou duplas:

```
>>>
>>> 'spam eggs'
'spam eggs'
>>> 'doesn\'t'
"doesn't"
>>> "doesn't"
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

O interpretador exibe o resultado de operações com strings da mesma forma como elas são digitadas na entrada: dentro de aspas, e com aspas, caracteres acentuados e outros caracteres especiais representados por sequências de escape com barras invertidas (como `'\t'`,`'\xc3\xa9'` etc.), para mostrar o valor preciso. A string é delimitada entre aspas simples, exceto quando ela contém uma aspa simples e nenhuma aspa dupla. O comando **`print`** produz uma saída mais legível para tais strings com caracteres especiais.

Strings que contém mais de uma linha podem ser construídas de diversas maneiras. Linhas de continuação podem ser usadas, com uma barra invertida colocada na última posição para indicar que a próxima linha física é a continuação de uma linha lógica:

```
oi = "Eis uma string longa contendo\n\
diversas linhas de texto assim como se faria em C.\n\
     Observe que os espaços em branco no inicio da linha são\
 significativos."

print oi
```

Observe que quebras de linha ainda precisam ser embutidos na string usando `\n` — a quebra de linha física após a última barra de escape é anulada. Este exemplo exibiria o seguinte resultado:

```
Eis uma string longa contendo
diversas linhas de texto assim como se faria em C.
     Observe que os espaços em branco no inicio da linha são significativos.
```

Ou, strings podem ser delimitadas por pares de aspas triplas combinando: `"""` ou `'''`. Neste caso não é necessário escapar o final das linhas físicas com `\`, mas as quebras de linha serão incluídas na string:

```
print """
Uso: treco [OPCOES]
     -h                        Exibir esta mensagem de uso
     -H hostname               Host a conectar
"""
```

produz a seguinte saída:

```
Uso: treco [OPCOES]
     -h                        Exibir esta mensagem de uso
     -H hostname               Host a conectar
```

Se fazemos uma string *raw* (N.d.T: “crua” ou sem processamento de caracteres escape) com o prefixo `r`, as sequências `\n` não são convertidas em quebras de linha. Tanto as barras invertidas quanto a quebra de linha física no código-fonte são incluídos na string como dados. Portanto, o exemplo:

```
oi = r"Eis uma string longa contendo\n\
diversas linhas de texto assim como se faria em C."

print oi
```

Exibe:

```
Eis uma string longa contendo\n\
diversas linhas de texto assim como se faria em C.
```

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

```
>>>
>>> palavra = 'Ajuda' + 'Z'
>>> palavra
'AjudaZ'
>>> '<' + palavra*5 + '>'
'<AjudaZAjudaZAjudaZAjudaZAjudaZ>'
```

Duas strings literais adjacentes são automaticamente concatenadas; a primeira linha do exemplo anterior poderia ter sido escrita como `palavra= 'Ajuda' 'Z'`; isso funciona somente com strings literais, não com expressões que produzem strings:

```
>>>
```

```
>>> 'str' 'ing'              #  <-  Isto funciona
'string'
>>> 'str'.strip() + 'ing'    #  <-  Isto funciona
'string'
>>> 'str'.strip() 'ing'      #  <-  Isto é inválido
  File "<stdin>", line 1, in ?
    'str'.strip() 'ing'
                  ^
SyntaxError: invalid syntax
```

Strings podem ser indexadas; como em C, o primeiro caractere da string tem índice 0 (zero). Não existe um tipo específico para caracteres; um caractere é simplesmente uma string de tamanho 1. Assim como na linguagem Icon, substrings podem ser especificadas através da notação de *slice* (fatiamento ou intervalo): dois índices separados por dois pontos.

```
>>>
```

```
>>> palavra[4]
'a'
>>> palavra[0:2]
'Aj'
>>> palavra[2:4]
'ud'
```

Índices de fatias têm defaults úteis; a omissão do primeiro índice equivale a zero, a omissão do segundo índice equivale ao tamanho da string sendo fatiada.:

```
>>>
```

```
>>> palavra[:2]     # Os dois primeiros caracteres
'Aj'
>>> palavra[2:]     # Tudo menos os dois primeiros caracteres
'udaZ'
```

Diferentemente de C, strings em Python não podem ser alteradas. Tentar atribuir valor a uma posição (índice ou fatia) dentro de uma string resulta em erro:

```
>>>
```

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

Entretanto, criar uma nova string combinando conteúdos é fácil e eficiente:

```
>>>
```

```
>>> 'x' + palavra[1:]
```

```
'xjudaZ'
>>> 'Splat' + palavra[5]
'SplatZ'
```

Eis uma invariante interessante das operações de fatiamento: `s[:i] + s[i:]` é igual a `s`.

```
>>>
>>> palavra[:2] + palavra[2:]
'AjudaZ'
>>> palavra[:3] + palavra[3:]
'AjudaZ'
```

Intervalos fora de limites são tratados “graciosamente” (N.d.T: o termo original “gracefully” indica robustez no tratamento de erros): um índice maior que o comprimento é trocado pelo comprimento, um limite superior menor que o limite inferior produz uma string vazia.

```
>>>
>>> palavra[1:100]
'judaZ'
>>> palavra[10:]
''
>>> palavra[2:1]
''
```

Índices podem ser números negativos, para iniciar a contagem pela direita. Por exemplo:

```
>>>
>>> palavra[-1]    # O último caractere
'Z'
>>> palavra[-2]    # O penúltimo caractere
'a'
>>> palavra[-2:]   # Os dois últimos caracteres
'aZ'
>>> palavra[:-2]   # Tudo menos os dois últimos caracteres
'Ajud'
```

Observe que -0 é o mesmo que 0, logo neste caso não se conta a partir da direita!

```
>>>
>>> palavra[-0]
'A'
```

Intervalos fora dos limites da string são truncados, mas não tente isso com índices simples (que não sejam fatias):

```
>>>
```

```
>>> palavra[-100:]
'AjudaZ'
>>> palavra[-100]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>>
```

Uma maneira de lembrar como slices 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:

```
 0   1   2   3   4   5   6
 +---+---+---+---+---+---+
 | A | j | u | d | a | z |
 +---+---+---+---+---+---+
-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, comprimento de `palavra[1:3]` é 2.

A função built-in (embutida) **`len()`** devolve o comprimento de uma string:

```
>>>
>>> s = 'anticonstitucionalissimamente'
>>> len(s)
29
```

See also

Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange

Strings, e as strings Unicode descritas na próxima seção, são exemplos de *sequências* e implementam as operações comuns associadas com esses objetos.

String Methods

Tanto strings comuns quanto Unicode oferecem um grande número de métodos para busca e transformações básicas.

String Formatting

Informações sobre formatação de strings com **`str.format()`** são descritas nesta seção.

String Formatting Operations

As operações de formatação de strings antigas, que acontecem quando strings simples e Unicode aparecem à direita do operador `%`são descritas dom mais detalhes nesta seção.

3.1.3. Strings Unicode

A partir de Python 2.0 um novo tipo para armazenar textos foi introduzido: o tipo `unicode`. Ele pode ser usado para armazenar e manipular dados no padrão Unicode (veja http://www.unicode.org/) e se integra bem aos objetos string pré-existentes, realizando conversões automáticas quando necessário.

Unicode tem a vantagem de associar um único número ordinal a cada caractere, para todas as formas de escrita usadas em textos modernos ou antigos. Anteriormente, havia somente 256 números ordinais possíveis para identificar caracteres. Cada texto era tipicamente limitado a uma "code page" (uma tabela de códigos) que associava ordinais aos caracteres. Isso levou a muita confusão especialmente no âmbito da internacionalização de software (comumente escrito como `i18n` porque `internationalization` é `'i'` + 18 letras + `'n'`). Unicode resolve esses problemas ao definir uma única tabela de códigos para todos os conjuntos de caracteres.

Criar strings Unicode em Python é tão simples quanto criar strings normais:

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

O `u` antes das aspas indica a criação de uma string Unicode. Se você desejar incluir caracteres especiais na string, você pode fazê-lo através da codificação *Unicode-Escape* de Python. O exemplo a seguir mostra como:

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

O código de escape `\u0020` insere um caractere Unicode com valor ordinal 0x0020 (o espaço em branco) naquela posição.

Os outros caracteres são interpretados usando seus valores ordinais como valores ordinais em Unicode. Se você possui strings literais na codificação padrão Latin-1 que é usada na maioria dos países ocidentais, achará conveniente que os 256 caracteres inferiores do Unicode coincidem com os 256 caracteres do Latin-1.

Para os experts, existe ainda um modo *raw* da mesma forma que existe para strings normais. Basta prefixar a string com 'ur' para usar a codificação *Raw-Unicode-Escape*.

A conversão `\uXXXX` descrita acima será aplicada somente se houver um número ímpar de barras invertidas antes do escape 'u'.

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

O modo raw (cru) é muito útil para evitar o excesso de barras invertidas, por exemplo, em expressões regulares.

Além dessas codificações padrão, Python oferece todo um conjunto de maneiras de se criar strings Unicode a partir de alguma codificação conhecida.

A função embutida **`unicode()`** dá acesso a todos os codecs Unicode registrados (COders e DECoders). Alguns dos codecs mais conhecidos são: *Latin-1*, *ASCII*, *UTF-8*, e *UTF-16*. Os dois últimos são codificações de tamanho variável para armazenar cada caractere Unicode em um ou mais bytes. (N.d.T: no Brasil, é muito útil o codec *cp1252*, variante estendida do *Latin-1* usada na maioria das versões do MS Windows distribuídas no país, contendo caracteres comuns em textos, como aspas assimétricas "x" e 'y', travessão –, bullet • etc.).

A codificação default é ASCII, que trata normalmente caracteres no intervalo de 0 a 127 mas rejeita qualquer outro com um erro. Quando uma string Unicode é exibida, escrita em arquivo ou convertida por **`str()`**, esta codificação padrão é utilizada.:

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

Para converter uma string Unicode em uma string de 8-bits usando uma codificação específica, basta invocar o método **`encode()`** de objetos Unicode passando como parâmetro o nome da codificação destino. É preferível escrever nomes de codificação em letras minúsculas.

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

Se você tem um texto em uma codificação específica, e deseja produzir uma string Unicode a partir dele, pode usar a função `unicode()`, passando o nome da codificação de origem como segundo argumento.

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

3.1.4. Listas

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

```
>>>
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
```

Da mesma forma que índices de string, índices de lista começam em 0, listas também podem ser concatenadas, fatiadas e multiplicadas:

```
>>>
>>> a[0]
'spam'
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
['eggs', 100]
>>> a[:2] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]
>>> 3*a[:3] + ['Boo!']
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam', 'eggs', 100, 'Boo!']
```

Todas as operações de fatiamento devolvem uma nova lista contendo os elementos solicitados. Isto significa que o fatiamento a seguir retorna uma cópia rasa (*shallow copy*) da lista:

```
>>>
>>> a[:]
['spam', 'eggs', 100, 1234]
```

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

```
>>>
```

```
>>> a
['spam', 'eggs', 100, 1234]
>>> a[2] = a[2] + 23
>>> a
['spam', 'eggs', 123, 1234]
```

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

```
>>>
>>> # Substituir alguns itens:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # Remover alguns:
... a[0:2] = []
>>> a
[123, 1234]
>>> # Inserir alguns:
... a[1:1] = ['bletch', 'xyzzy']
>>> a
[123, 'bletch', 'xyzzy', 1234]
>>> # Inserir uma cópia da própria lista no início
>>> a[:0] = a
>>> a
[123, 'bletch', 'xyzzy', 1234, 123, 'bletch', 'xyzzy', 1234]
>>> # Limpar a lista: substituir todos os items por uma lista vazia
>>> a[:] = []
>>> a
[]
```

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

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

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

```
>>>
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
>>> p[1][0]
2
>>> p[1].append('xtra')      # Veja a seção 5.1
>>> p
[1, [2, 3, 'xtra'], 4]
>>> q
[2, 3, 'xtra']
```

Observe que no último exemplo, `p[1]` e `q` na verdade se referem ao mesmo objeto! Mais tarde retornaremos a *semântica dos objetos*.

3.2. Primeiros passos rumo à programação

Naturalmente, podemos utilizar 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 o próximo
... a, b = 0, 1
>>> while b < 10:
...     print b
...     a, b = b, a+b
...
1
1
2
3
5
8
```

Este exemplo introduz diversas características ainda não mencionadas.

- A primeira linha contém uma atribuição múltipla: as variáveis `a` e `b` recebem simultaneamente os novos valores 0 e 1. Na última linha há outro exemplo de atribuição múltipla demonstrando que expressões do lado direito são sempre avaliadas primeiro, antes da atribuição. As expressões do lado direito são avaliadas da esquerda para a direita.
- O laço **`while`** executa enquanto a condição (aqui: `b < 10`) permanecer verdadeira. Em Python, como em C, qualquer valor não-zero é considerado verdadeiro, zero é considerado falso. A condição pode ser ainda uma lista ou string, na verdade qualquer sequência; qualquer coisa com comprimento maior que zero tem valor verdadeiro e sequências vazias são falsas. O teste utilizado no exemplo é uma comparação simples. Os operadores padrão para 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 parser 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
- O comando **`print`** escreve o valor da expressão ou expressões fornecidas. É diferente de apenas escrever a expressão no interpretador (como fizemos nos

exemplos da calculadora) pela forma como lida com múltiplas expressões e strings. Strings são exibidas sem aspas, e um espaço é inserido entre os itens para formatar o resultado assim:

```
>>>
```

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

Uma vírgula ao final evita a quebra de linha:

```
>>>
```

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

Note que o interpretador insere uma quebra de linha antes de imprimir o próximo prompt se a última linha não foi completada.

4. Mais ferramentas de controle de fluxo

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

4.1. Comando `if`

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

```
>>>
```

```
>>> x = int(raw_input("Favor digitar um inteiro: "))
Favor digitar um inteiro: 42
>>> if x < 0:
...     x = 0
...     print 'Negativo alterado para zero'
... elif x == 0:
...     print 'Zero'
... elif x == 1:
...     print 'Unidade'
... else:
...     print 'Mais'
...
Mais
```

Pode haver zero ou mais seções **elif**, e a seção **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 construções switch ou case existentes em outras linguagens.

4.2. Comando `for`

O comando **for** em Python difere um tanto do que você talvez esteja acostumado em C ou Pascal. Ao invés de se iterar sobre progressões aritméticas (como em Pascal), ou dar ao usuário o poder de definir tanto o passo da iteração quanto a condição de parada (como em C), o comando **for** de Python itera sobre os itens de qualquer sequência (como uma lista ou uma string), na ordem em que eles aparecem na sequência. Por exemplo:

```
>>>
>>> # Medir o tamanho de algumas strings:
>>> a = ['gato', 'janela', 'defenestrar']
>>> for x in a:
...     print x, len(x)
...
gato 4
janela 6
defenestrar 11
>>>
```

Não é seguro modificar a sequência sobre a qual se baseia o laço de iteração (isto pode acontecer se a sequência for mutável, isto é, uma lista). Se você precisar modificar a lista sobre a qual está iterando (por exemplo, para duplicar itens selecionados), você deve iterar sobre uma cópia da lista ao invés da própria. A notação de fatiamento é bastante conveniente para isso:

```
>>>
>>> for x in a[:]: # fazer uma cópia da lista inteira
...     if len(x) > 6: a.insert(0, x)
...
>>> a
['defenestrar', 'gato', 'janela', 'defenestrar']
```

4.3. A função `range()`

Se você precisar iterar sobre sequências numéricas, a função embutida **range()** é a resposta. Ela gera listas contendo progressões aritméticas, por exemplo:

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

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 em outro número, ou alterar a razão da progressão (inclusive com passo negativo):

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

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

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

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

4.4. Comandos `break` e `continue`, e cláusulas `else` em laços

O comando **break**, como em C, interrompe o laço **for** ou **while** mais interno.

O comando **continue**, também emprestado de C, avança para a próxima iteração do laço mais interno.

Laços podem ter uma cláusula else, que é executada sempre que o laço se encerra por exaustão da lista (no caso do **for**) ou quando a condição se torna falsa (no caso do **while**), mas nunca quando o laço é interrompido por um **break**. Isto é exemplificado no próximo exemplo que procura números primos:

```
>>>
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print n, '=', x, '*', n/x
...             break
...     else:
...         # laço terminou sem encontrar um fator
...         print n, 'é um número primo'
...
2 é um número primo
```

```
3 é um número primo
4 = 2 * 2
5 é um número primo
6 = 2 * 3
7 é um número primo
8 = 2 * 4
9 = 3 * 3
```

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

4.5. Comando `pass`

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

```
>>>
>>> while True:
...     pass  # esperar interrupção via teclado (Ctrl+C)
...
```

Isto é usado muitas vezes para se definir classes mínimas:

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

Outra situação em que **pass** pode ser usado é para reservar o lugar de uma função ou de um bloco condicional, quando você está trabalhando em código novo, o que lhe possibilita continuar a raciocinar em um nível mais abstrato. O comando **pass** é ignorado silenciosamente:

```
>>>
>>> def initlog(*args):
...     pass   # Lembrar de implementar isto!
...
```

4.6. Definindo Funções

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

```
>>>
>>> def fib(n):    # escrever série de Fibonacci até n
...     """Exibe série de Fibonacci até n"""
...     a, b = 0, 1
...     while a < n:
...         print a,
...         a, b = b, a+b
```

```
...
>>> # Agora invocamos a funçao 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. Os comandos que formam o corpo da função começam na linha seguinte e devem ser indentados.

Opcionalmente, a primeira linha lógica do corpo da função pode ser uma string literal, 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 disto um hábito.

A *execução* de uma função gera uma nova tabela de símbolos, usada para as variáveis locais da função. Mais precisamente, toda atribuição a variável dentro da função armazena o valor na tabela de símbolos local. Referências a variáveis são buscadas primeiramente na tabela local, então na tabela de símbolos global e finalmente na tabela de nomes embutidos (built-in). Portanto, não se pode atribuir diretamente um valor a uma variável global dentro de uma função (a menos que se utilize a declaração **`global`** antes), ainda que variáveis globais possam ser referenciadas livremente.

Os parâmetros reais (argumentos) de uma chamada de função são introduzidos na tabela de símbolos local da função no momento da invocação, portanto, argumentos são passados por valor (onde o *valor* é sempre uma referência para objeto, não o valor do objeto). [1] Quando uma função invoca outra, uma nova tabela de símbolos é criada para tal chamada.

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

```
>>>
```

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

Conhecendo outras linguagens, você pode questionar que `fib` não é uma função, mas um procedimento, pois ela não devolve um valor. Na verdade, mesmo funções que não

usam o comando **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 o comando **print**:

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

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

```
>>>
>>> def fib2(n): # devolve a série de Fibonacci até n
...     """Devolve uma lista a com série de Fibonacci até n."""
...     resultado = []
...     a, b = 0, 1
...     while a < n:
...         resultado.append(a)    # veja mais adiante
...         a, b = b, a+b
...     return resultado
...
>>> f100 = fib2(100)    # executar
>>> f100                # exibir o resultado
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Este exemplo, como sempre, demonstra algumas características novas:

- O comando **return** termina a função devolvendo um valor. Se não houver uma expressão após o **return**, o valor **None** é devolvido. Se a função chegar ao fim sem o uso explícito do **return**, então também será devolvido o valor **None**.
- O trecho `resultado.append(a)` invoca um *método* do objeto lista `resultado`. Um método é uma função que "pertence" a um objeto e é chamada através de `obj.nome_do_metodo` onde `obj` é um objeto qualquer (pode ser uma expressão), e `nome_do_metodo` 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; ele adiciona um novo elemento ao final da lista. Neste exemplo, ele equivale a `resultado = resultado + [a]`, só que mais eficiente.

4.7. Mais sobre definição de funções

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

4.7.1. Parâmetros com valores default

A mais útil das três é especificar um valor default para um ou mais parâmetros formais. Isso cria uma função que pode ser invocada com um número menor de argumentos do que ela pode receber. Por exemplo:

```
def confirmar(pergunta, tentativas=4, reclamacao='Sim ou não, por favor!'):
    while True:
        ok = raw_input(pergunta).lower()
        if ok in ('s', 'si', 'sim'):
            return True
        if ok in ('n', 'no', 'não', 'nananinanão'):
            return False
        tentativas = tentativas - 1
        if tentativas == 0:
            raise IOError('usuario nao quer cooperar')
        print reclamacao
```

Essa função pode ser invocada de várias formas:

- fornecendo apenas o argumento obrigatório: `confirmar('Deseja mesmo encerrar?')`
- fornecendo um dos argumentos opcionais: `confirmar('Sobrescrever o arquivo?', 2)`
- ou fornecendo todos os argumentos: `confirmar('Sobrescrever o arquivo?', 2, 'Escolha apenas s ou n')`

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

Os valores default são avaliados no momento a 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 default são avaliados apenas uma vez. Isso faz diferença quando o valor default é um objeto mutável como uma lista ou dicionário (N.d.T. dicionários são como arrays associativos ou HashMaps em outras linguagens; ver *Dicionários*).

Por exemplo, a função a seguir acumula os argumentos passados em chamadas subsequentes:

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

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

Esse código vai exibir:

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

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

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

4.7.2. Argumentos nomeados

Funções também podem ser chamadas passando *keyword arguments* (argumentos nomeados) no formato `chave=valor`. Por exemplo, a seguinte função:

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

aceita um argumento obrigatório (`voltage`) e três argumentos opcionais (`state`, `action`, e `type`). Esta função pode ser invocada de todas estas formas:

```
parrot(1000)                                    # 1 arg. posicional
parrot(voltage=1000)                            # 1 arg. nomeado
parrot(voltage=1000000, action='VOOOOOM')       # 2 arg. nomeados
parrot(action='VOOOOOM', voltage=1000000)       # 2 arg. nomeados
```

```
parrot('a million', 'bereft of life', 'jump')  # 3 arg. posicionais
# 1 arg. positional e 1 arg. nomeado
parrot('a thousand', state='pushing up the daisies')
```

mas todas as invocações a seguir seriam inválidas:

```
parrot()                     # argumento obrigatório faltando
parrot(voltage=5.0, 'dead')  # argumento posicional depois do nomeado
parrot(110, voltage=220)     # valur duplicado para o mesmo argument
parrot(actor='John Cleese')  # argumento nomeado desconhecido
```

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

```
>>>
```

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

Quando o último parâmetro formal usar a sintaxe `**nome`, ele receberá um dicionário (ver *Dicionários* ou *Mapping Types — dict* [online]) com todos os parâmetros nomeados passados para a função, exceto aqueles que corresponderam a parâmetros formais definidos antes. Isto pode ser combinado com o parâmetro formal `*nome` (descrito na próxima subseção) que recebe uma tupla (N.d.T. uma sequência de itens, semelhante a uma lista imutável; ver *Tuplas e sequências*) contendo todos argumentos posicionais que não correspondem à lista da parâmetros formais. (`*nome` deve ser declarado antes de `**nome`.) Por exemplo, se definimos uma função como esta:

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

Ela pode ser invocada assim:

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

e, naturalmente, produziria:

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

Note que criamos uma lista de chaves `keys` ordenando o resultado do método `keys()` do dicionário `keywords` antes de exibir seu conteúdo; se isso não fosse feito, os argumentos seriam exibidos em uma ordem não especificada.

4.7.3. Listas arbitrárias de argumentos

Finalmente, a opção menos usada possibilita que função seja invocada 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 escrever_multiplos_items(arquivo, separador, *args):
    arquivo.write(separador.join(args))
```

4.7.4. Desempacotando listas de argumentos

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

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

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

```
>>>
```

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

4.7.5. Construções lambda

Atendendo a pedidos, algumas características encontradas em linguagens de programação funcionais como Lisp foram adicionadas a Python. Com a palavra reservada **`lambda`**, pequenas funções anônimas podem ser criadas. Eis uma função que devolve a soma de seus dois argumentos: `lambda a, b: a+b`.

Construções lambda podem ser empregadas em qualquer lugar que exigiria uma função. Sintaticamente, estão restritas a uma única expressão. Semanticamente, são apenas açúcar sintático para a definição de funções normais. Assim como definições de funções aninhadas, construções lambda podem referenciar variáveis do escopo onde são definidas (N.d.T isso significa que Python implementa *closures*, recurso encontrado em Lisp, JavaScript, Ruby etc.):

```
>>>
```

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

4.7.6. Strings de documentação

A comunidade Python está convencionando o conteúdo e o formato de strings de documentação (*docstrings*).

A primeira linha deve ser um resumo curto e 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 parser do Python não remove a indentação de comentários multi-linha. Portanto, ferramentas que processem strings de documentação precisam lidar com isso, quando desejável. Existe uma convenção para isso. A primeira linha não vazia após a linha de sumário determina a indentação para o resto da string de documentação. (Não podemos usar a primeira linha para isso porque ela em geral está adjacente às aspas que iniciam a string, portanto sua indentação real não fica aparente.) Espaços em branco ou tabs “equivalentes” a esta indentação são então removidos do início das demais linhas da string. Linhas indentação menor não devem ocorrer, mas se ocorrerem, todos os espaços à sua esquerda são removidos. Para determinar a indentação, normalmente considera-se que um caractere tab equivale a 8 espaços.

Eis um exemplo de uma docstring multi-linha:

```
>>>
>>> def minha_funcao():
...     """Não faz nada, mas é documentada.
...
...     Realmente ela não faz nada.
...     """
...     pass
...
>>> print minha_funcao.__doc__
Não faz nada, mas é documentada.

    Realmente ela não faz nada.
```

4.8. Intermezzo: estilo de codificação

Agora que você está prestes a escrever peças mais longas e complexas em Python, é uma 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, o **PEP 8** tornou-se o guia de estilo adotado pela maioria dos projetos; ele promove um estilo de codificação muito legível e visualmente agradável. Todo desenvolvedor Python deve lê-lo em algum momento; aqui estão os pontos mais importantes selecionados para você:

- Use 4 espaços de recuo, e nenhum tab.

  4 espaços são um bom meio termo entre indentação estreita (permite maior profundidade de aninhamento) e indentação larga (mais fácil de ler). Tabs trazem complicações; é melhor 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 as funções e classes, e grandes blocos de código dentro de funções.
- Quando possível, coloque comentários em uma linha própria.
- Escreva docstrings.
- 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 modo consistente; a convenção é usar `CamelCase` (literalmente, *CaixaCamelo*) para classes e `caixa_baixa_com_underscores` para funções e métodos. Sempre use `self` como nome do primeiro parâmetro formal dos métodos de instância (veja *Primeiro contato com classes* para saber mais sobre classes e métodos).
- Não use codificações exóticas se o seu código é feito para ser usado em um contexto internacional. ASCII puro funciona bem em qualquer caso. (N.d.T. para programadores de língua portuguesa, UTF-8 é atualmente a melhor opção, e já se tornou o default em Python 3 conforme o **PEP 3120**).

Notas

[1] Na verdade, *passagem por referência para objeto* (*call by object reference*) seria uma descrição melhor do que *passagem por valor*(*call-by-value*), pois, se um objeto mutável for passado, o invocador (*caller*) verá as alterações feitas pelo invocado (*callee*), como por exemplo a inserção de itens em uma lista.

5. Estruturas de dados

Este capítulo descreve alguns pontos já abordados, porém com mais detalhes, e adiciona outros pontos.

5.1. Mais sobre listas

O tipo `list` possui mais métodos. Aqui estão todos os métodos disponíveis em um objeto lista:

`list.`**`append`**(*x*)
Adiciona um item ao fim da lista; equivale a `a[len(a):] = [x]`.

`list.`**`extend`**(*L*)
Prolonga a lista, adicionando no fim todos os elementos da lista `L` passada como argumento; equivalente a `a[len(a):] = L`.

`list.`**`insert`**(*i*, *x*)

Insere um item em uma posição especificada. O primeiro argumento é o índice do elemento antes do qual será feita a inserção, assim`a.insert(0, x)` insere no início da lista, e `a.insert(len(a), x)` 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 dada e o devolve. Se nenhum índice for especificado, `a.pop()` remove e devolve o último item na lista. (Os colchetes ao redor do *i* indicam que o parâmetro é opcional, não que você deva digitá-los daquela maneira. Você verá essa notação com frequência na Referência da Biblioteca Python.)

`list.`**`index`**(*x*)

Devolve o índice do primeiro item cujo valor é igual a *x*, gerando `ValueError` se este valor não existe

`list.`**`count`**(*x*)

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

`list.`**`sort`**()

Ordena os itens na própria lista *in place*.

`list.`**`reverse`**()

Inverte a ordem dos elementos na lista *in place* (sem gerar uma nova lista).

Um exemplo que utiliza a maioria dos métodos::

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

```
[-1, 1, 66.25, 333, 333, 1234.5]
```

(N.d.T. Note que os métodos que alteram a lista, inclusive `sort` e `reverse`, devolvem **`None`** para lembrar o programador de que modificam a própria lista, e não criam uma nova. O único método que altera a lista e devolve um valor é o `pop`)

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()
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(["Eric", "John", "Michael"])
>>> fila.append("Terry")     # Terry chega
>>> fila.append("Graham")    # Graham chega
>>> fila.popleft()           # O primeiro a chegar parte
```

```
'Eric'
>>> fila.popleft()          # O segundo a chegar parte
'John'
>>> fila                    # O resto da fila, em ordem de chegada
deque(['Michael', 'Terry', 'Graham'])
```

(N.d.T. neste exemplo são usados nomes de membros do grupo *Monty Python*)

5.1.3. Ferramentas de programação funcional

Existem três funções embutidas que são muito úteis para processar listas: **filter()**, **map()**, e **reduce()**.

filter(funcao, sequencia) devolve uma nova sequência formada pelos itens do segundo argumento para os quais funcao(item) é verdadeiro. Se a sequencia de entrada for string ou tupla, a saída será do mesmo tipo; caso contrário, o resultado será sempre uma lista. Por exemplo, para computar uma sequência de números não divisíveis por 2 ou 3:

```
>>>
>>> def f(x): return x % 2 != 0 and x % 3 != 0
...
>>> filter(f, range(2, 25))
[5, 7, 11, 13, 17, 19, 23]
```

map(funcao, sequencia) aplica funcao(item) a cada item da sequência e devolve uma lista formada pelo resultado de cada aplicação. Por exemplo, para computar cubos:

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

Mais de uma sequência pode ser passada; a função a ser aplicada deve aceitar tantos argumentos quantas sequências forem passadas, e é invocada com o item correspondente de cada sequência (ou **None**, se alguma sequência for menor que outra). Por exemplo:

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

Se **None** for passado no lugar da função, então será aplicada a função identidade (apenas devolve o argumento recebido). Se várias sequências forem passadas, a lista resultante terá tuplas formadas pelos elementos correspondentes de cada sequência.

Isso se parece com a função **zip()**, exceto que **map()** devolve uma lista com o comprimento da sequência mais longa que foi passada, preenchendo as lacunas com **None** quando necessário, e **zip()** devolve uma lista com o comprimento da mais curta. Confira:

```
>>>
>>> map(None, range(5))
[0, 1, 2, 3, 4]
>>> map(None, range(5), range(3))
[(0, 0), (1, 1), (2, 2), (3, None), (4, None)]
>>> zip(range(5), range(3))
[(0, 0), (1, 1), (2, 2)]
>>>
```

A função reduce(funcao, sequencia) devolve um único valor construído a partir da sucessiva aplicação da função binária (N.d.T. que recebe dois argumentos) a todos os elementos da lista fornecida, começando pelos dois primeiros itens, depois aplicando a função ao primeiro resultado obtido e ao próximo item, e assim por diante. Por exemplo, para computar a soma dos inteiros de 1 a 10:

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

Se houver um único elemento na sequência fornecida, seu valor será devolvido. Se a sequência estiver vazia, uma exceção será levantada.

Um terceiro argumento pode ser passado para definir o valor inicial. Neste caso, redução de uma sequência vazia devolve o valor inicial. Do contrário, a redução se inicia aplicando a função ao valor inicial e ao primeiro elemento da sequência, e continuando a partir daí.

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

Não use a função somatória deste exemplo; somar sequências de números é uma necessidade comum, e para isso Python tem a função embutida **sum()**, que faz exatamente isto, e também aceita um valor inicial (opcional).

5.1.4. List comprehensions ou abrangências de listas

Uma *list comprehension* é uma maneira concisa de construir uma lista preenchida. (N.d.T. literalmente, *abrangência de lista* mas no Brasil o termo em inglês é muito usado; também se usa a abreviação *listcomp*)

Um uso comum é construir uma nova lista onde cada elemento é o resultado de alguma expressão aplicada a cada membro de outra sequência ou iterável, ou para construir uma subsequência cujos elementos satisfazem uma certa condição.

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]
```

Podemos obter o mesmo resultado desta forma:

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

Isso equivale a `quadrados = map(lambda x: x**2, range(10))`, mas é mais conciso e legível.

Uma abrangência de lista é formada por 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 lista resultante da avaliação da expressão no contexto das cláusulas **`for`** e **`if`**.

Por exemplo, esta listcomp combina os elementos de duas listas quando eles são 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)]
```

Isto equivale a:

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

```
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Note como a ordem dos `for` e `if` é a mesma nos dois exemplos acima.

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

```
>>>
>>> vec = [-4, -2, 0, 2, 4]
>>> # criar uma lista com os valores dobrados
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filtrar a lista para excluir números negativos
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # aplicar uma função a todos os elementos
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # invocar um método em cada elemento
>>> frutas = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [arma.strip() for arma in frutas]
['banana', 'loganberry', 'passion fruit']
>>> # criar uma lista de duplas, ou tuplas de 2, como (numero, 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 emtre parênteses, do contrário ocorre um erro
>>> [x, x**2 for x in range(6)]
  File "<stdin>", line 1
    [x, x**2 for x in range(6)]
               ^
SyntaxError: invalid syntax
>>> # achatar uma lista usando uma listcomp 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]
```

A abrangência de lista é mais flexível do que `map()` e pode conter expressões complexas e funções aninhadas, sem necessidade do uso de `lambda`:

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

5.1.4.1. Listcomps aninhadas

A expressão inicial de uma listcomp pode ser uma expressão arbitrária, inclusive outra listcomp.

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 abrangência de listas abaixo transpõe as linhas e colunas:

```
>>>
```

```
>>> [[linha[i] for linha in matriz] for i in range(len(matriz[0]))]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

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

```
>>>
```

```
>>> transposta = []
>>> for i in range(len(matriz[0])):
...     transposta.append([linha[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(len(matriz[0])):
...     # as próximas 3 linhas implementam a listcomp 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 expressões complexas. A função **`zip()`** resolve muito bem este caso de uso:

```
>>>
```

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

5.2. O comando `del`

Existe uma maneira de remover um item de uma lista conhecendo seu índice, ao invés de seu valor: o comando **del**. Ele difere do método **list.pop()**, que devolve o item removido. O comando **del** também pode ser utilizado para remover fatias (slices) da lista, ou mesmo limpar a lista toda (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]
>>> 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
>>> a
Traceback (most recent call last):
  ...
NameError: name 'a' is not defined
```

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 o comando **del** mais tarde.

5.3. Tuplas e sequências

Vimos que listas e strings têm muitas propriedades em comum, como indexação e operações de fatiamento (*slicing*). Elas são dois exemplos de *sequências* (veja *Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange*). 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 (*tuple*).

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

```
>>>
>>> t = 12345, 54321, 'bom dia!'
>>> t[0]
12345
>>> t
```

```
(12345, 54321, 'bom dia!')
>>> # Tuplas podem ser aninhadas:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'bom dia!'), (1, 2, 3, 4, 5))
```

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 a uma expressão maior).

Tuplas podem ser usadas de diversas formas: pares ordenados `(x, y)`, registros de funcionário extraídos uma base de dados, etc. Tuplas, assim como strings, são imutáveis: não é possível atribuir valores a itens individuais de uma tupla (você pode simular o mesmo efeito através de operações de fatiamento e concatenação; N.d.T. mas neste caso nunca estará modificando tuplas, apenas criando novas). Também é possível criar tuplas contendo objetos mutáveis, como listas.

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 uma par de parênteses vazios; uma tupla unitária é construída por um único valor e uma vírgula entre parênteses (não basta colocar um único valor entre parênteses). Feio, mas funciona:

```
>>>
>>> vazia = ()
>>> upla = 'hello',    # <-- note a vírgula no final
>>> len(vazia)
0
>>> len(upla)
1
>>> upla
('hello',)
```

O comando `t = 12345, 54321, 'hello!'` é um exemplo de *empacotamento de tupla* (*tuple packing*): os valores `12345`, `54321` e `'bom dia!'` são empacotados juntos em uma tupla. A operação inversa também é possível:

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

Isto é chamado de desempacotamento de sequência (*sequence unpacking*), funciona para qualquer tipo de sequência do lado direito. Para funcionar, é necessário que a lista de variáveis do lado esquerdo tenha o mesmo comprimento da sequência à direita. Sendo assim, a atribuição múltipla é um caso de empacotamento de tupla e desempacotamento de sequência:

```
>>>
```

```
>>> a, b = b, a  # troca os valores de a e b
```

Existe uma certa assimetria aqui: empacotamento de múltiplos valores sempre cria tuplas, mas o desempacotamento funciona para qualquer sequência.

5.4. Sets (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 sets 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.

Uma pequena demonstração:

```
>>>
>>> cesta = ['uva', 'laranja', 'uva', 'abacaxi', 'laranja', 'banana']
>>> frutas = set(cesta)   # criar um conjunto sem duplicatas
>>> frutas
set(['abacaxi', 'uva', 'laranja', 'banana'])
>>> 'laranja' in frutas   # testar se um elemento existe é muito rápido
True
>>> 'capim' in frutas
False
>>>
>>> # Demonstrar operaçes de conjunto em letras únicas de duas palavras
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                 # letras unicas em a
set(['a', 'r', 'b', 'c', 'd'])
>>> a - b                             # letras em a mas não em b
set(['r', 'd', 'b'])
>>> a | b                             # letras em a ou em b
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
>>> a & b                             # letras tanto em a como em b
set(['a', 'c'])
>>> a ^ b                             # letras em a ou b mas não em ambos
set(['r', 'd', 'b', 'm', 'z', 'l'])
```

N.d.T. A sintaxe de sets do Python 3.1 foi portada para o Python 2.7, tornando possível escrever `{10, 20, 30}` para definir `set([10, 20, 30])`. O conjunto vazio tem que ser escrito como `set()` ou `set([])`, pois `{}` sempre representou um dicionário vazio, como veremos a seguir. Também existe uma sintaxe para *set comprehensions*, que nos permite escrever `{x*10 for x in [1, 2, 3]}` para construir `{10, 20, 30}`.

5.5. Dicionários

Outra estrutura de dados muito útil embutida em Python é o *dicionário*, cujo tipo é `dict` (ver *Mapping Types — dict*). 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 *in place* 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.

O método **`keys()`** do dicionário devolve a lista de todas as chaves presentes no dicionário, em ordem arbitrária (se desejar ordená-las basta aplicar o a função **`sorted()`** à lista devolvida). 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
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
```

O construtor **`dict()`** produz dicionários diretamente a partir de uma lista de chaves-valores, armazenadas como duplas (tuplas de 2 elementos). Quando os pares formam

um padrão, uma list comprehensions pode especificar a lista de chaves-valores de forma mais compacta.

```
>>>
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
>>> dict([(x, x**2) for x in (2, 4, 6)])     # use uma list comprehension
{2: 4, 4: 16, 6: 36}
```

N.d.T. A partir do Python 2.7 também existem *dict comprehensions* (abrangências de dicionário). Com esta sintaxe o último dict acima pode ser construído assim `{x: x**2 for x in (2, 4, 6)}`.

Mais adiante no tutorial aprenderemos sobre *expressões geradoras*, que são ainda mais adequados para fornecer os pares de chave-valor para o construtor **`dict()`**.

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

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

N.d.T. Naturalmente, por limitações da sintaxe, essa sugestão só vale se as chaves forem strings ASCII, sem acentos, conforme a regras para formação de identificadores do Python.

5.6. Técnicas de iteração

Ao percorrer um dicionário em um laço, a variável de iteração receberá uma chave de cada vez:

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

Quando conveniente, a chave e o valor correspondente podem ser obtidos simultaneamente com o método **`iteritems()`**.

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

```
robin the brave
```

Ao percorrer uma sequência qualquer, o índice da posição atual e o valor correspondente podem ser obtidos simultaneamente usando a função **`enumerate()`**:

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

Para percorrer duas ou mais sequências simultaneamente com o laço, os itens podem ser agrupados com a função **`zip()`**.

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

N.d.T. O exemplo acima reproduz um diálogo do filme *Monty Python - Em Busca do Cálice Sagrado*. Este trecho não pode ser traduzido, exceto por um decreto real.

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

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

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

```
>>>
>>> cesta = ['uva', 'laranja', 'uva', 'abacaxi', 'laranja', 'banana']
>>> for fruta in sorted(cesta):
...     print fruta
...
abacaxi
```

```
banana
laranja
laranja
uva
uva
>>> for fruta in sorted(set(cesta)):  # sem duplicações
...     print fruta
...
abacaxi
banana
laranja
uva
>>>
```

5.7. Mais sobre condições

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

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

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

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

Os operadores booleanos `and` e `or` são operadores *short-circuit*: seus argumentos são avaliados da esquerda para a direita, e a avaliação para 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 atalho 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 em Python, diferente de C, atribuição não pode ocorrer dentro de uma expressão. Programadores C podem resmungar, mas isso evita toda uma classe de problemas frequentemente encontrados em programas C: digitar = numa expressão quando a intenção era ==.

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 ASCII 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)
```

É permitido comparar objetos de diferentes tipos. O resultado é determinístico, porém, arbitrário: os tipos são ordenados pelos seus nomes. Então, uma `list` é sempre menor do que uma `str`, uma `str` é sempre menor do que uma `tuple`, etc. [1] Tipos numéricos misturados são comparados de acordo com seus valores numéricos, logo 0 é igual a 0.0, etc.

Notas

[1] As regras para comparação de objetos de tipos diferentes não são definitivas; elas podem variar em futuras versões da linguagem.

6. Módulos

Se você sair do interpretador do Python e entrar novamente, as definições (funções e variáveis) que você havia feito estarão perdidas. Portanto, se você quer escrever um programa um pouco mais longo, você se sairá melhor usando um editor de texto para criar e salvar o programa em um arquivo, usando depois esse arquivo como entrada para a execução do interpretador. Isso é conhecido como gerar um *script*. A medida que seus programas crescem, pode ser desejável dividi-los em vários arquivos para facilitar a manutenção. Você também pode querer reutilizar uma função sem copiar sua definição a cada novo programa.

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

Um módulo é um arquivo Python contendo definições e instruções. O nome do arquivo é o módulo com o sufixo `.py` adicionado. Dentro de um módulo, o nome do módulo (como uma string) está disponível na 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:

```
# coding: utf-8
# Módulo números de Fibonacci

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

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

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

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

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

```
>>>
```

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

Se pretende usar uma função frequentemente, pode associá-la a um nome local:

```
>>>
```

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

6.1. Mais sobre módulos

Um módulo pode conter tanto comandos quanto definições de funções e classes. Esses comandos servem para inicializar o módulo. Eles são executados somente na *primeira* vez que o módulo é importado em algum lugar. [1]

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

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

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

```
>>>
```

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

Isso não coloca o nome do módulo de onde foram feitas as importações para a tabela de símbolos local (assim, no exemplo `fibo` não está definido), mas somente o nome das funções `fib` e `fib2`.

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

```
>>>
```

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

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.

Note

Por razões de eficiência, cada módulo é importado somente uma vez por sessão do interpretador. Portanto, se você alterar seus módulos, você deve reiniciar o interpretador – ou, se é somente um módulo que você quer testar interativamente, use **`reload()`**, ex.`reload(nome_do_modulo)`.

6.1.1. Executando módulos como scripts

Quando você executa um módulo Python assim:

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

o código no módulo será executado, da mesma forma como você estivesse apenas importado, mas com a variável global `__name__` com o valor`"__main__"`. Isso significa que você pode acrescentar este código no fim do seu módulo:

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

para permitir que o arquivo seja usado tanto como um script quanto como um módulo que pode ser importado, porque o código que lê o argumento da linha de comando só será acionado se o módulo foi executado como o arquivo “principal”:

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

Se o módulo é importado, o bloco dentro do `if __name__`... não é executado:

```
>>>
```

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

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

6.1.2. O caminho de busca dos módulos

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

- o diretório que contém o script importador (ou o diretório atual).
- a variável de ambiente **`PYTHONPATH`** (uma lista de nomes de diretórios, com a mesma sintaxe da variável de ambiente **`PATH`**).
- um caminho default que depende da instalação do Python.

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 padrão* para mais informações.

6.1.3. Arquivos Python “compilados”

Para acelerar a inicialização de programas curtos que usam muitos módulos da biblioteca padrão, sempre que existe um arquivo chamado `spam.pyc` no mesmo diretório de `spam.py`, o interpretador assume que aquele arquivo contém uma versão “byte-compilada” de **spam**. O horário de modificação da versão de `spam.py` a partir da qual `spam.pyc` foi gerado é é armazenada no arquivo compilado, e o `.pyc` não é utilizado se o horário não confere.

Normalmente, não é preciso fazer nada para gerar o arquivo `spam.pyc`. Sempre que `spam.py` é compilado com sucesso, o interpretador tenta salvar a versão compilada em `spam.pyc`. Não há geração de um erro se essa tentativa falhar; se por alguma razão o arquivo compilado não for inteiramente gravado, o arquivo `spam.pyc` resultante será reconhecido como inválido e, portanto, ignorado. O conteúdo do arquivo `spam.pyc` é independente de plataforma, assim um diretório de módulos Python pode ser compartilhado por máquinas de diferentes arquiteturas.

Algumas dicas para os experts:

- Quando o interpretador Python é invocado com a opção *-O*, é gerado um código otimizado, armazenado em arquivos `.pyo`. O otimizador atual não faz muita coisa; ele apenas remove instruções **`assert`**. Quando *-O* é

utilizada, *todo bytecode* é otimizado; arquivos `.pyc` são ignorados e os arquivos `.py` são compilados para bytecode otimizado.

- Passar duas opções *-O* para o interpretador Python (*-OO*) fará com que o compilador realize otimizações mais arriscadas, que em alguns casos raros podem acarretar o mal funcionamento de programas. Atualmente apenas strings `__doc__` são removidas do bytecode, resultando em arquivos `.pyo` mais compactos. Uma vez que alguns programas podem contar com a existência dessas docstrings, use essa opção somente se você souber o que está fazendo.
- Um programa não executa mais rápido quando é lido de um arquivo `.pyc` ou `.pyo` em comparação a quando é lido de um arquivo `.py`. A única diferença é que nos dois primeiros casos o tempo de inicialização do programa é menor.
- Quando um script é executado diretamente a partir o seu nome da linha de comando, não são geradas as formas compiladas deste script em formato `.pyc` ou `.pyo`. Portanto, o tempo de carga de um script pode ser melhorado se transferirmos a maior parte de seu código para um módulo e utilizarmos o script menor apenas para inicialização. Também é possível fornecer um arquivo `.pyc` ou `.pyo`diretamente para execução do interpretador, passando seu nome na linha de comando.
- Na presença das formas compiladas (`spam.pyc` e `spam.pyo`) de um script, não há necessidade do código fonte (`spam.py`). Isto é útil na para se distribuir bibliotecas Python de uma forma que dificulta moderadamente a engenharia reversa.
- O módulo **`compileall`** pode criar arquivos `.pyc` (ou `.pyo` quando *-O* é usada) para todos os módulos em um dado diretório.

6.2. Módulos padrão

Python possui uma biblioteca padrão de módulos, descrita em um documento em separado, a Python Library Reference (doravante “Library Reference”). 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 subjacente. Por exemplo, o módulo **`winreg`** só está disponível em sistemas Windows. Existe um módulo que requer especial atenção: **`sys`**, que é embutido em qualquer interpretador Python. As variáveis `sys.ps1`e `sys.ps2` definem as strings utilizadas como prompt primário e secundário:

```
>>>
```

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

```
Eca!
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 default interno se a variável não estiver definida. Você pode modificar `sys.path` 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 se descobrir quais nomes são definidos por um módulo. Ela devolve uma lista ordenada de strings:

```
>>>
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__stderr__',
 '__stdin__', '__stdout__', '_getframe', 'api_version', 'argv',
 'builtin_module_names', 'byteorder', 'callstats', 'copyright',
 'displayhook', 'exc_clear', 'exc_info', 'exc_type', 'excepthook',
 'exec_prefix', 'executable', 'exit', 'getdefaultencoding', 'getdlopenflags',
 'getrecursionlimit', 'getrefcount', 'hexversion', 'maxint', 'maxunicode',
 'meta_path', 'modules', 'path', 'path_hooks', 'path_importer_cache',
 'platform', 'prefix', 'ps1', 'ps2', 'setcheckinterval', 'setdlopenflags',
 'setprofile', 'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout',
 '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__', '__doc__', '__file__', '__name__', 'a', 'fib', 'fibo',
'sys']
```

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

dir() não lista nomes de funções ou variáveis embutidas. Se quiser conhecê-las, seus nomes estão definidos no módulo padrão **__builtin__**:

```
>>>
```

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

6.4. Pacotes

Pacotes são uma maneira de estruturar espaços de nomes para módulos Python utilizando a sintaxe de “nomes pontuados” (dotted names). Como exemplo, o nome **`A.B`** designa um submódulo chamado `B` em um pacote denominado `A`. O uso de pacotes permite que os autores de pacotes com muitos módulos, como NumPy ou PIL (Python Imaging Library) não se preocupem com colisão entre os nomes de seus módulos e os nomes de módulos de outros autores.

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 principal
      __init__.py               Inicializar o pacote sound
      formats/                  Subpacote para conversão de formatos
              __init__.py
```

```
              wavread.py
              wavwrite.py
              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 Python trate os diretórios como pacotes; isso foi feito para evitar que diretórios com nomes comuns, como `string`, inadvertidamente ocultassem módulos válidos que ocorram depois no caminho de busca. No caso mais simples,`__init__.py` pode ser um arquivo vazio. Porém, ele pode conter código de inicialização para o pacote ou definir a variável `__all__`, que será descrita depois.

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

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 submódulos presentes no pacote, e os importassem. Isso pode demorar muito e a importação de submódulos pode ocasionar efeitos colaterais que somente deveriam ocorrer quando o submódulo é explicitamente importado.

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

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

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

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

do pacote que tenham sido carregados explicitamente por comandos `import` anteriores. Considere o código abaixo:

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

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

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

Lembre-se que não há nada de errado em utilizar `from pacote import submodulo_especifico`! De fato, essa é a notação recomendada a menos que o módulo efetuando a importação precise utilizar submódulos homônimos de diferentes pacotes.

6.4.2. Referências em um mesmo pacote

Os submódulos frequentemente precisam referenciar uns aos outros. Por exemplo, o módulo **`surround`** talvez precise utilizar o módulo **`echo`**. De fato, tais referências são tão comuns que o comando `import` primeiro busca módulos dentro do pacote antes de utilizar o caminho de busca padrão. Portanto, o módulo **`surround`** pode usar simplesmente `import echo` ou `from echo import echofilter`. Se o módulo importado não for encontrado no pacote atual (o pacote do qual o módulo atual é submódulo), então o comando `import` procura por um módulo de mesmo nome fora do pacote (nos locais definidos em `sys.path`).

Quando pacotes são estruturados em subpacotes (como no pacote **`sound`** do exemplo), pode ser usar a sintaxe de um `import` absoluto 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 importecho`.

A partir do Python 2.5, em adição à importação relativa implícita descrita acima, você pode usar importação relativa explícita na forma `fromimport`. Essas importações relativas explícitas usam prefixos com pontos indicar os pacotes atuais e seus pais envolvidos na importação. A partir do módulo **`surround`** por exemplo, pode-se usar:

```
from . import echo
```

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

Note que tanto a importação relativa explícita quanto a implícita baseiam-se no nome do módulo atual. Uma vez que o nome do módulo principal é sempre "__main__", módulos que serão o módulo principal de uma aplicação Python devem sempre usar importações absolutas.

6.4.3. Pacotes em múltiplos diretórios

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

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

Notas

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

7. Entrada e saída

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

7.1. Refinando a formatação de saída

Até agora vimos duas maneiras de exibir valores no console interativo: escrevendo expressões e usando o comando **print**. Em programas, apenas o **print** gera saída. (Uma outra maneira é utilizar o método **write()** de objetos arquivo; a saída padrão pode ser referenciada como sys.stdout. Veja a Referência da Biblioteca Python para mais informações sobre isto.)

Frequentemente é desejável mais controle sobre a formatação de saída do que simplesmente exibir valores separados por espaços. Existem duas formas de formatar a saída. A primeira é manipular strings através de fatiamento (slicing) e concatenação. Os tipos string têm métodos úteis para criar strings com tamanhos determinados, usando caracteres de preenchimento; eles serão apresentados a seguir. A segunda forma é usar o método **str.format()**.

O módulo **string** tem uma classe **string.Template** que oferece uma outra maneira de inserir valores em strings.

Permanece a questão: como converter valores para strings? Felizmente, Python possui duas maneiras de converter qualquer valor para uma string: as funções **repr()** e **str()**.

A função **str()** serve para produzir representações de valores que sejam legíveis para as pessoas, enquanto **repr()** é para gerar representações que o interpretador Python consegue ler (caso não exista uma forma de representar o valor, a representação devolvida por**repr()** produz um **SyntaxError** [N.d.T. **repr()** procura gerar representações fiéis; quando isso é inviável, é melhor encontrar um erro do que obter um objeto diferente do original]). Para objetos que não têm uma representação adequada para consumo humano, **str()** devolve o mesmo valor que **repr()**. Muitos valores, tal como inteiros e estruturas como listas e dicionários, têm a mesma representação usando ambas funções. Strings e números de ponto flutuante, em particular, têm duas representações distintas.

Alguns exemplos:

```
>>>
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
>>> str(1.0/7.0)
'0.142857142857'
>>> repr(1.0/7.0)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
>>> print s
The value of x is 32.5, and y is 40000...
>>> # O repr() de uma string acresenta aspas e contrabarras:
... hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print hellos
'hello, world\n'
>>> # O argumento de repr() pode ser qualquer objeto Python:
... repr((x, y, ('spam', 'eggs')))
"(32.5, 40000, ('spam', 'eggs'))"
```

A seguir, duas maneiras de se escrever uma tabela de quadrados e cubos::

```
>>>
>>> for x in range(1, 11):
...     print repr(x).rjust(2), repr(x*x).rjust(3),
...     # Note a vírgula final 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

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

(Note que um espaço foi inserido entre as colunas no primeiro exemplo. É assim que o comando print **print** funciona: ele sempre insere espaço entre seus argumentos.)

Esse exemplo demonstra o método **str.rjust()** de objetos string, que alinha uma string à direita juntando espaços adicionais à esquerda. Existem métodos análogas **str.ljust()** e **str.center()**. Esses métodos não exibem nada na tela, apenas devolvem uma nova string formatada. Se a entrada extrapolar o comprimento especificado, a string original é devolvida sem modificação; isso pode estragar o alinhamento das colunas, mas é melhor do que a alternativa, que seria apresentar um valor mentiroso. (Se for realmente desejável truncar o valor, pode-se usar fatiamento, por exemplo: x.ljust(n)[:n].)

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

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

Um uso básico do método **str.format()** é assim:

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

As chaves e seus conteúdos (chamados de campos de formatação) são substituídos pelos objetos passados para o método **str.format()**, respeitando a ordem dos argumentos.

Um número na primeira posição dentro das chaves identifica o argumento pela sua posição na chamada do método (N.d.T. esse número era obrigatório na versão 2.6 do Python; tornou-se opcional na versão 2.7):

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

Se argumentos nomeados são passados para o método **str.format()**, seus valores pode ser identificados pelo nome do argumento:

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

As marcações '!s' e '!r' podem ser usadas para forçar a conversão de valores aplicando respectivamente as funções **str()** e **repr()**:

```
>>>
>>> import math
>>> print 'O valor de PI é aproximadamente {}.'.format(math.pi)
O valor de PI é aproximadamente 3.14159265359.
>>> print 'O valor de PI é aproximadamente {!r}.'.format(math.pi)
O valor de PI é aproximadamente 3.141592653589793.
```

Após o identificador do campo, uma especificação de formato opcional pode ser colocada depois de : (dois pontos). O exemplo abaixo arredonda Pi até a terceira casa após o ponto decimal.

```
>>>
>>> import math
>>> print 'O valor de PI é aproximadamente {0:.3f}.'.format(math.pi)
O valor de PI é aproximadamente 3.142.
```

Colocar um inteiro *n* logo após o : fará o campo ocupar uma largura mínima de *n* caracteres. Isto é útil para organizar tabelas.

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

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

```
>>>
>>> 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 como argumentos nomeados, usando 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.

Para uma visão completa da formatação de strings com **str.format()**, veja a seção *Format String Syntax* na Referência da Biblioteca Python.

7.1.1. Formatação de strings à moda antiga: operador %

O operador % também pode ser usado para formatação de strings. Ele interpreta o operando da esquerda de forma semelhante à função **sprintf()** da linguagem C, aplicando a formatação ao operando da direita, e devolvendo a string resultante. Por exemplo:

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

Como o método **str.format()** é bem novo (apareceu no Python 2.6), muito código Python ainda usa o operador %. Porém, como esta formatação antiga será um dia removida da linguagem, **str.format()** deve ser usado.

Mais informações podem ser encontradas na seção *String Formatting Operations* da Referência da Biblioteca Python.

7.2. Leitura e escrita de arquivos

A função **open()** devolve um objeto arquivo, e é frequentemente usada com dois argumentos: open(nome_do_arquivo, modo).

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

O primeiro argumento é uma string contendo o nome do arquivo. O segundo argumento é outra string contendo alguns caracteres que descrevem o modo como o arquivo será usado. O parâmetro mode 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 parâmetro mode é opcional, em caso de omissão será assumido'r'.

No Windows, 'b' adicionado a string de modo indica que o arquivo será aberto em modo binário. Sendo assim, existem os modos compostos : 'rb', 'wb', e 'r+b'. O Windows faz distinção entre arquivos texto e binários: os caracteres terminadores de linha em arquivos texto são alterados ao ler e escrever. Essa mudança automática é útil em arquivos de texto ASCII, mas corrompe arquivos binários como .JPEG ou .EXE. Seja cuidadoso e use sempre o modo binário ao manipular tais arquivos. No Unix, não faz diferença colocar um 'b' no modo, então você pode usar isto sempre que quiser lidar com arquivos binários de forma independente da plataforma.

N.d.T. Para ler arquivos de texto contendo acentuação e outros caracteres não-ASCII, a melhor prática desde o Python 2.6 é usar a função**io.open()**, do módulo **io**, em vez da função embutida **open()**. O motivo é que **io.open()** permite especificar a codificação logo ao abrir um arquivo em modo texto para leitura ou escrita. Desta forma, a leitura do arquivo texto sempre devolverá objetos unicode, independente da codificação interna do arquivo no disco. E ao escrever em um arquivo aberto via **io.open()**, basta enviar strings unicode, pois a conversão para o encoding do arquivo será feita automaticamente. Note que ao usar **io.open()** sempre faz diferença especificar se o arquivo é binário ou texto, em todos os sistemas operacionais: o modo texto é o default, mas se quiser ser explícito coloque a letra t no parâmetro mode (ex. rt, wt etc.); use use a letra b (ex. rb, wb etc.) para especificar modo binário. Somente em modo texto os métodos de gravação e leitura aceitam e devolvem strings unicode. Em modo binário, o

método write aceita strings de bytes, e os métodos de leitura devolvem strings de bytes também.

7.2.1. Métodos de objetos arquivo

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

Para ler o conteúdo de um arquivo, invoque `f.read(size)`, que lê um punhado de dados devolvendo-os como uma string de bytes `str`. O argumento numérico *size* é opcional. Quando *size* 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 *size* bytes serão lidos e devolvidos. Se o fim do arquivo for atingido, `f.read()` devolve uma string vazia (`""`).

```
>>>
```

```
>>> f.read()
'Texto completo do arquivo.\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, só não ocorrendo na última linha do arquivo, se ela não termina com uma quebra de linha. Isso elimina a ambiguidade do valor devolvido; se `f.readline()` devolver uma string vazia, então é certo que arquivo acabou. Linhas em branco são representadas por um `'\n'` – uma string contendo apenas o terminador de linha.

```
>>>
```

```
>>> f.readline()
'Primeira linha do arquivo.\n'
>>> f.readline()
'Segunda linha do arquivo.\n'
>>> f.readline()
''
```

O método `f.readlines()` devolve uma lista contendo todas as linhas do arquivo. Se for fornecido o parâmetro opcional *sizehint*, será lida a quantidade especificada de bytes e mais o suficiente para completar uma linha. Frequentemente, isso é usado para ler arquivos muito grandes por linhas, sem ter que ler todo o arquivo para a memória de uma só vez. Apenas linhas completas serão devolvidas.

```
>>>
```

```
>>> f.readlines()
['Primeira linha do arquivo.\n', 'Segunda linha do arquivo.\n']
```

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,

Primeira linha do arquivo.
Segunda linha do arquivo.
```

Essa alternativa é mais simples, mas não oferece tanto controle. Como as duas maneiras gerenciam o buffer do arquivo de modo diferente, elas não devem ser misturadas.

O método `f.write(string)` escreve o conteúdo da string de bytes para o arquivo, devolvendo **`None`**.

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

N.d.T. Neste exemplo, a quantidade de bytes que será escrita no arquivo vai depender do encoding usado no console do Python. Por exemplo, no encoding UTF-8, a string acima tem 18 bytes, incluindo a quebra de linha, porque são necessários dois bytes para representar o caractere `'é'`. Mas no encoding CP1252 (comum em Windows no Brasil), a mesma string tem 17 bytes. O método `f.write` apenas escreve bytes; o que eles representam você decide.

Ao escrever algo que não seja uma string de bytes, é necessário converter antes:

```
>>>
>>> valor = ('a resposta', 42)
>>> s = str(valor)
>>> f.write(s)
```

N.d.T. Em particular, se você abriu um arquivo `f` com a função embutida **`open()`**, e deseja escrever uma string Unicode `x` usando `f.write`, deverá usar o método **`unicode.encode()`** explicitamente para converter `x` do tipo `unicode` para uma string de bytes `str`, deste modo:`f.write(x.encode('utf-8'))`. Por outro lado, se abriu um arquivo `f2` com **`io.open()`**, pode usar `f2.write(x)` diretamente, pois a conversão de `x` – de `unicode` para o encoding do arquivo – será feita automaticamente.

O método `f.tell()` devolve um inteiro `long` que indica a posição atual de leitura ou escrita no arquivo, medida em bytes desde o início do arquivo. Para mudar a posição utilize `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; o valor default é 0.

```
>>>
>>> f = open('/tmp/workfile', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5)      # Vai para o sexto byte do arquivo
>>> f.read(1)
'5'
>>> f.seek(-3, 2) # Vai para terceiro byte antes do fim
>>> f.read(1)
'd'
```

Quando acabar de utilizar o arquivo, invoque `f.close()` para fechá-lo e liberar recursos do sistema (buffers, descritores de arquivo etc.). Qualquer tentativa de acesso ao arquivo depois dele ter sido fechado resultará em falha.

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

É uma boa prática usar o comando **`with`** ao lidar com objetos arquivo. Isto tem a vantagem de garantir que o arquivo seja fechado quando a execução sair do bloco dentro do **`with`**, mesmo que uma exceção tenha sido levantada. É também muito mais sucinto do que escrever os blocos **`try-finally`** necessários para garantir que isso aconteça.

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

Objetos arquivo têm métodos adicionais, como **`isatty()`** e **`truncate()`** que são usados com menos frequência; consulte a Referência da Biblioteca Python para mais informações.

7.2.2. O módulo `pickle`

Strings podem ser facilmente escritas e lidas de um arquivo. Números exigem um pouco mais de esforço, uma vez que o método **`read()`** só devolve strings, obrigando o uso de uma função como **`int()`** para produzir o número 123 a partir da string `'123'`. Entretanto, quando estruturas de dados mais complexas (listas, dicionários, instâncias de classe,etc) estão envolvidas, o processo se torna bem mais complicado.

Para não obrigar os usuários a escrever e depurar constantemente código para salvar estruturas de dados, Python oferece o módulo padrão**pickle**. Este é um módulo incrível que permite converter praticamente qualquer objeto Python (até mesmo certas formas de código!) para uma string de bytes. Este processo é denominado pickling (N.d.T. literalmente, “colocar em conserva”, como picles de pepinos em conserva). E unpickling é o processo reverso: reconstruir o objeto a partir de sua representação como string de bytes. Enquanto estiver representado como uma string, o objeto pode ser facilmente armazenado em um arquivo ou banco de dados, ou transferido pela rede para uma outra máquina.

Se você possui um objeto qualquer `x`, e um objeto arquivo `f` que foi aberto para escrita, a maneira mais simples de utilizar este módulo é:

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

Para reconstruir o objeto `x`, sendo que `f` agora é um arquivo aberto para leitura:

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

(Existem outras variações desse processo, úteis quando se precisa aplicar sobre muitos objetos ou o destino da representação string não é um arquivo; consulte a documentação do módulo **pickle** na Referência da Biblioteca Python.)

O módulo **pickle** é a forma padrão da fazer objetos Python que possam ser compartilhados entre diferentes programas Python, ou pelo mesmo programa em diferentes sessões de execução; o termo técnico para isso é *objeto persistente*. Justamente porque o módulo **pickle** é amplamente utilizado, vários autores que escrevem extensões para Python tomam o cuidado de garantir que novos tipos de dados, como matrizes numéricas, sejam compatíveis com esse processo.

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, in ?
    while True print 'Olá mundo'
                   ^
SyntaxError: invalid syntax
```

O parser repete a linha inválida e apresenta uma pequena 'seta' apontando para o ponto da linha em que o erro foi encontrado. O erro é causado (ou ao menos detectado) pelo token que *precede* a seta: no exemplo, o erro foi detectado na palavra reservada `print`, uma vez que o 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 e acabam resultando em mensagens de erro:

```
>>>
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

A última linha da mensagem de erro indica o que aconteceu. Exceções surgem com diferentes tipos, e o tipo é exibido como parte da mensagem: os tipos no exemplo

são **ZeroDivisionError**, **NameError** e **TypeError**. A string exibida como sendo o tipo da exceção é o nome interno da exceção 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.

Built-in Exceptions lista as exceções pré-definidas e seus significados.

8.3. Tratamento de exceções

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

```
>>>
```

```
>>> while True:
...     try:
...         x = int(raw_input("Por favor, informe um número: "))
...         break
...     except ValueError:
...         print "Oops!  Não foi 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 execução durante a execução da cláusula try, as instruções remanescentes na cláusula são ignoradas. Se o tipo da exceção ocorrida tiver sido previsto em algum **except**, então essa cláusula será executada. Depois disso, a execução continua na próxima instrução após o conjunto *try/except*.
- Se a exceção levantada não foi prevista em nenhuma cláusula **except** da cláusula **try** em que ocorreu, então ela é entregue a uma instrução **try** mais externa. Se não existir nenhum tratador previsto para tal exceção, será

uma *exceção não tratada* e a execução do programa termina com uma mensagem de erro.

A instrução **try** pode ter mais de uma cláusula **except** para especificar múltiplos tratadores para diferentes exceções. No máximo um único tratador será ativado. Tratadores só são sensíveis às exceções levantadas no interior da cláusula try, e não às que tenham ocorrido no interior de outro tratador numa mesma instrução **try**. Um tratador pode ser sensível a múltiplas exceções, desde que as especifique em uma tupla:

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

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

```
import sys

try:
    f = open('meuarquivo.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as (errno, strerror):
    print "I/O error({0}): {1}".format(errno, strerror)
except ValueError:
    print "Não foi possível converter o dado para inteiro."
except:
    print "Erro inesperado:", sys.exc_info()[0]
    raise
```

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

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print 'não foi possível abrir', arg
    else:
        print arg, 'tem', len(f.readlines()), 'linhas'
        f.close()
```

Esse recurso é melhor do que simplesmente adicionar o código da cláusula **else** ao corpo da cláusula **try**, pois mantém as exceções levantadas no **else** num escopo diferente de tratamento das exceções levantadas na cláusula **try**, evitando que

acidentalmente seja tratada uma exceção que não foi levantada pelo código protegido pela construção **try** ... **except**.

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

A cláusula except pode especificar uma variável depois do nome (ou da tupla de nomes) da exceção. A variável é associada à instância de exceção capturada, com os argumentos armazenados em `instancia.args`. Por conveniência, a instância define o método **__str__()** para que os argumentos possam ser exibidos diretamente sem necessidade de acessar `.args`.

Pode-se também instanciar uma exceção antes de levantá-la e adicionar qualquer atributo a ela, conforme desejado.

```
>>>
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print type(inst) # a instância da exceção
...     print inst.args  # argumentos armazenados em .args
...     print inst       # __str__ permite exibir args diretamente
...     x, y = inst      # __getitem__ permite desempacotar args diretamente
...     print 'x =', x
...     print 'y =', y
...
<type 'exceptions.Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

Se uma exceção possui argumento, ele é exibido ao final (‘detalhe’) da mensagem de exceções não tratadas.

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

```
>>>
>>> def isso_falha():
...     x = 1/0
...
>>> try:
...     isso_falha()
... except ZeroDivisionError as detalhe:
...     print 'Tratando erros em tempo de execução:', detalhe
...
Tratando erros em tempo de execução: integer division or modulo by zero
```

8.4. Levantando exceções

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

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

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 **Exception**)

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

```
>>>
>>> try:
...     raise NameError('HiThere')
... except NameError:
...     print 'Uma exceção voou!'
...     raise
...
Uma exceção voou!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
NameError: HiThere
```

8.5. 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. Por exemplo:

```
>>>
>>> class MeuErro(Exception):
...     def __init__(self, valor):
...         self.valor = valor
...     def __str__(self):
...         return repr(self.valor)
...
>>> try:
...     raise MeuErro(2*2)
... except MeuErro as e:
...     print 'Minha exceção ocorreu, valor:', e.valor
...
Minha exceção ocorreu, valor: 4
>>> raise MeuErro('oops!')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
```

```
__main__.MeuErro: 'oops!'
```

Neste exemplo, o método padrão **__init__()** da classe **Exception** foi redefinido. O novo comportamento simplesmente cria o atributo *valor*. Isso substitui o comportamento padrão de criar o atributo *args*.

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

```
class Error(Exception):
    """Classe base para exceções dessa módulo"""
    pass

class InputError(Error):
    """Exceções levantadas por erros na entrada

    Atributos:
        expr -- expressão da entrada onde o erro ocorreu
        msg  -- explicação do erro
    """

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

class TransitionError(Error):
    """Levantada quando uma operação tenta fazer uma transição de estado não
    permitida.

    Atributos:
        anterior -- estado do início da transição
        proximo -- novo estado
        msg  -- explicação do porquê a transação específica não é permitida
    """

    def __init__(self, anterior, proximo, msg):
        self.anterior = anterior
        self.proximo = proximo
        self.msg = msg
```

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

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

8.6. Definindo ações de limpeza

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

```
>>>
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print 'Adeus, mundo!'
...
Adeus, mundo!
KeyboardInterrupt
```

Uma *cláusula finally* é sempre executada, ocorrendo ou não uma exceção. Quando ocorre uma exceção na cláusula **try** e ela não é tratada por uma cláusula **except** (ou quando ocorre em cláusulas **except** ou **else**), ela é re-levantada depois que a cláusula **finally** é executada. A cláusula **finally** é executada "na saída" quando qualquer outra cláusula da instrução **try** é finalizada, mesmo que seja por meio de qualquer uma das instruções **break**, **continue** ou **return**. Um exemplo mais completo:

```
>>>
>>> def divide(x, y):
...     try:
...         resultado = x / y
...     except ZeroDivisionError:
...         print "divisão por zero!"
...     else:
...         print "resultado é", resultado
...     finally:
...         print "executando a cláusula finally"
...
>>> divide(2, 1)
resultado é 2
executando a cláusula finally
>>> divide(2, 0)
divisão por zero!
executando a cláusula finally
>>> divide("2", "1")
executando a cláusula finally
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Como você pode ver, a cláusula **finally** é executado 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.

8.7. Ações de limpeza predefinidas

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

```
for linha in open("meuarquivo.txt"):
    print linha
```

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 a:
    for linha in a:
        print linha
```

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

9. Classes

Em comparação com outras linguagens, 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 a 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, e todos as funções membro são virtuais. Como em Modula-3, não existem atalhos para referenciar membros do objeto de dentro dos seus métodos. Um método (função definida em uma classe) é declarado com um primeiro argumento explícito representando o objeto (instância da classe), que é fornecido implicitamente pela invocação. 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 de usuário por herança. Como em C++, mas diferentemente de Modula-3, a maioria dos operadores (aritméticos, indexação,etc) podem ser redefinidos para 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 é mais próxima a 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 (inclusive em diferentes escopos) podem estar vinculados a um mesmo objeto. Isso é chamado de*aliasing* em outras linguagens. (N.d.T. *aliasing* é, literalmente, “apelidamento”: um mesmo objeto pode ter vários apelidos.) À primeira vista, esta característica não é muito apreciada, e pode ser seguramente ignorada ao lidar com tipos imutáveis (números, strings, tuplas). Entretanto, aliasing pode ter um efeito inesperado sobre a semântica de código Python envolvendo objetos mutáveis como listas, dicionários e a maioria dos outros tipos. Isso pode ser usado em benefício do programa, porque os *aliases* (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.

N.d.T. Na terminologia de C++ e Java, o que o parágrafo acima denomina “apelidos” são identificadores de referências (variáveis de referência), e os ponteiros são as próprias referências. Se uma variável `a` está associada a um objeto qualquer, informalmente dizemos que a variável “contém” o objeto, mas na realidade o objeto existe independente da variável, e o conteúdo da variável é apenas uma referência (um ponteiro) para o objeto. O *aliasing* ocorre quando existem diversas variáveis, digamos `a`, `b` e `c`, apontando para o mesmo objeto.

9.2. Escopos e *namespaces*

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

Vamos começar com algumas definições.

Um *namespace* (ou 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 embutidas); nomes globais em um módulo; e nomes locais na invocação de uma função. De uma certa forma, os atributos de um objeto também formam um espaço de nomes. O mais importante é saber que não existe nenhuma relação entre nomes em espaços 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 `nomemod.nomefunc`, `nomemod` é um objeto módulo e `nomefunc` é um de seus atributos. Neste caso, existe um mapeamento direto entre os os atributos de um módulo e os nomes globais definidos no módulo: eles compartilham o mesmo espaço de nomes! [1]

Atributos podem ser somente para leitura ou para leitura e escrita. No segundo caso, é possível atribuir um novo valor ao atributo. (N.d.T. Também é possível criar novos atributos.) Atributos de módulos são passíveis de atribuição: você pode escrever `nomemod.a_reposta = 42`. Atributos que aceitam escrita também podem ser apagados através do comando **`del`**. Por exemplo, `del nomemod.a_reposta` remove o atributo **`a_resposta`** do objeto referenciado por `nomemod`.

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 `__builtin__`.).

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

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

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

- o escopo mais interno (que é acessado primeiro) contendo nomes locais;
- os escopos das funções que envolvem a função atual, que são acessados a partir do escopo mias próximo, contém 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 valores vão diretamente para o escopo intermediário que contém os nomes globais do módulo. Caso contrário, todas as variáveis encontradas fora do escopo mais interno são apenas para leitura (a tentativa de atribuir valores a essas variáveis irá simplesmente criar uma *nova* variável local, no escopo interno, não alterando nada na variável de nome idêntico fora dele).

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" (N.d.T. quando um módulo é carregado ele é compilado em memória), portanto não conte com a resolução dinâmica de nomes! (De fato, variáveis locais já são resolvidas estaticamente.)

Uma peculiaridade de Python é que atribuições ocorrem sempre no escopo mais interno, exceto quando o comando `global` é usado. Atribuições não copiam dados, apenas associam nomes a objetos. O mesmo vale para remoções: o comando `del x` remove o vínculo de `x` do espaço de nomes do escopo local. De fato, todas as operações que introduzem novos nomes usam o escopo local. Em particular, instruções`import` e definições de funções associam o nome módulo ou da função ao escopo local. (A palavra reservada `global` pode ser usada para indicar que certas variáveis residem no escopo global ao invés do local.)

9.3. Primeiro contato com classes

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

9.3.1. Sintaxe de definição de classe

A forma mais simples de definir uma classe é:

```
class NomeDaClasse:
    <instrução-1>
    .
    .
    .
    <instrução-N>
```

Definições de classes, assim como definições de funções (instruções `def`), precisam ser executados antes que tenham qualquer efeito. (Por exemplo, você pode colocar uma definição de classe dentro de teste condicional `if` ou dentro de uma função.)

Na prática, as instruções dentro da definição de uma classe em geral 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 lista peculiar de parâmetros formais determinada pela convenção de chamada a métodos — isso também será explicado mais tarde.

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

Quando o processamento de uma definição de classe é completado (normalmente, sem erros), um *objeto classe* é criado. Este objeto encapsula o conteúdo do espaço de

nomes criado pela definição da classe; aprenderemos mais sobre objetos classe na próxima seção. O escopo local que estava vigente antes da definição da classe é reativado, e o objeto classe é vinculado ao identificador da classe nesse escopo (no exemplo acima, **`NomeDaClasse`** é o identificador da classe).

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`. Atributos válidos são todos os nomes presentes dentro do namespace da classe quando o objeto classe foi criado. Portanto, se a definição da classe foi assim:

```
class MinhaClasse:
    """Um exemplo simples de classe"""
    i = 12345
    def f(self):
        return 'olá, mundo'
```

então `MinhaClasse.i` e `MinhaClasse.f` são referências válidas, que acessam, respectivamente, um inteiro e um objeto função. É possível mudar os valores dos atributos da classe, ou mesmo criar novos atributos, fazendo uma atribuição simples assim: `MinhaClasse.i = 10`. O nome `__doc__` identifica outro atributo válido da classe, referenciando a *docstring* associada à ela: `"Um exemplo simples de classe"`.

Para *instanciar* uma classe, usa-se a 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. Continuando o exemplo 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.dados = []
```

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 Complexo:
...     def __init__(self, parte_real, parte_imag):
...         self.r = parte_real
...         self.i = parte_imag
...
>>> x = Complexo(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

9.3.3. Instâncias

Agora, o que podemos fazer com instâncias? As únicas operações reconhecidas por instâncias são referências a atributos. Existem dois tipos de nomes de atributos válidos: atributos de dados (*data attributes*) e métodos.

Atributos de dados correspondem a “variáveis de instância” em Smalltalk, e a “data members” 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 na instância (por causa do uso de **`del`**):

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

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

Nomes de métodos válidos de uma instância dependem de sua classe. Por definição, cada atributo de uma classe que é uma função corresponde a um método das instâncias. Em nosso exemplo, `x.f` é uma referência de método válida já que `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 (*method object*), e a `MinhaClasse.f` acessa um objeto função.

9.3.4. Objetos método

Normalmente, um método é invocado imediatamente após ser acessado:

```
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 (um objeto método), ele pode atribuído a uma variável invocado depois. Por exemplo:

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

Esse código 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 parâmetro, porém a definição da função **`f()`** especificava um parâmetro. O que aconteceu com esse parâmetro? Certamente Python levanta uma exceção quando uma função que declara um parâmetro é invocada sem nenhum argumento — mesmo que o argumento não seja usado no corpo da função...

Talvez você já tenha adivinhado a resposta: o que os métodos têm de especial é que eles passam o objeto (ao qual o método está vinculado) como primeiro argumento da função definida na classe. No nosso exemplo, a chamada `x.f()` equivale exatamente `MinhaClasse.f(x)`. Em geral, chamar um método com uma lista de *n* argumentos é equivalente a chamar a função na classe correspondente passando a instância como o primeiro argumento antes dos demais *n* argumentos.

Se você ainda não entendeu como métodos funcionam, talvez uma olhada na implementação de Python sirva para clarear as coisas. Quando um atributo de instância é referenciado e não é um atributo de dado, a busca continua na classe. Se o nome indica um atributo de classe válido que é um objeto função, um objeto método é criado pela composição da instância alvo e do objeto função. Quando o método é invocado com uma lista de argumentos, uma nova lista de argumentos é criada inserindo a instância na posição 0 da lista. Finalmente, o objeto função — empacotado dentro do objeto método — é invocado com a nova lista de argumentos.

9.4. Observações aleatórias

Atributos de dados sobrescrevem atributos métodos homônimos. Para evitar conflitos de nome acidentais, que podem gerar bugs de difícil rastreio em programas extensos, é sábio adotar algum tipo de convenção que minimize a chance de conflitos. Convenções comuns incluem: definir nomes de métodos com inicial maiúscula, prefixar atributos de dados com uma string única (quem sabe "_" [*underscore* ou sublinhado]), ou usar sempre verbos para nomear métodos e substantivos para atributos de dados.

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 convenção. (Por outro lado, a implementação de Python, escrita em C, pode esconder completamente detalhes de um objeto ou controlar seu acesso, 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 à vontade atributos de dados a uma instância sem afetar a validade dos métodos, desde que seja evitado o conflito de nomes. Novamente, uma convenção de nomenclatura poupa muita dor de cabeça.

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.

Não existe atalho para referenciar atributos de dados (ou outros métodos!) de dentro de um método: sempre é preciso fazer referência explícita ao `self`. para acessar qualquer atributo da instância. Em minha opinião isso aumenta a legibilidade dos métodos: não há como confundir uma variável local com um atributo da instância quando lemos rapidamente um método desconhecido.

Qualquer objeto função que é atributo de uma classe, define um método para as instâncias desta 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:
    def g(self):
```

```
        return 'olá mundo'
    h = g

C.f = f1
```

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 equivalente a `g`. No entanto, essa prática serve apenas para confundir o leitor do programa.

Métodos podem chamar outros métodos como atributos do argumento `self`:

```
class Saco:
    def __init__(self):
        self.data = []
    def adicionar(self, x):
        self.data.append(x)
    def adicionar2vezez(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 a definição de sua 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.

Todo valor em Python é um objeto, e portanto tem uma *classe* (também conhecida como seu tipo, ou *type*). A classe de um objeto pode ser referenciada como `objeto.__class__`.

9.5. Herança

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

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

O identificador **`NomeClasseBase`** deve estar definido no escopo que contém a definição da classe derivada. No lugar do nome da classe base, também são aceitas

outras expressões. Isso é muito útil, por exemplo, quando a classe base é definida em outro módulo:

```
class NomeClasseDerivada(nomemod.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 desde se essa procura produza 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 invocava 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.)

Em uma classe derivada, um método que sobrescreva outro pode desejar na verdade estender, ao invés de substituir, o método sobrescrito de mesmo nome na classe base. A maneira mais simples de implementar esse comportamento é chamar diretamente o método na classe base, passando explicitamente a instância como primeiro argumento:

```
NomeClasseBase.nomemetodo(self, argumento1, argumento2)
```

Às vezes essa forma de invocação pode ser útil até mesmo em código que apenas usa a classe, sem estendê-la. (Note que para esse exemplo funcionar, `NomeClasseBase` precisa estar definida ou importada diretamente no escopo global do módulo.)

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`**.
  Entretanto,`issubclass(unicode, str)` é **`False`** porque **`unicode`** não é uma

subclasse **str** (essas duas classes derivam da mesma classe base:**basestring**).

9.5.1. Herança múltipla

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

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

A única regra que precisa ser explicada é a semântica de resolução para as referências a atributos herdados. Em classes no estilo antigo (old-style classes [2]), a busca é feita em profundidade e da esquerda para a direita. Logo, se um atributo não é encontrado em **NomeClasseDerivada**, ele é procurado em **Base1**, e recursivamente nas classes bases de **Base1**, e apenas se não for encontrado lá a busca prosseguirá em **Base2**, e assim sucessivamente. (Para algumas pessoas a busca em largura — procurar antes em **Base2** e **Base3** do que nos ancestrais de **Base1** — parece mais natural. Entretanto, seria preciso conhecer toda a hierarquia de **Base1** para evitar um conflito com um atributo de **Base2**. Na prática, a busca em profundidade não diferencia entre atributos diretos ou herdados de **Base1**.)

Em *new-style classes*), a ordem de resolução de métodos muda dinamicamente para suportar invocações cooperativas via **super()**. Esta abordagem é conhecida em certas outras linguagens que têm herança múltipla como *call-next-method* (invocar próximo método) e é mais poderoso que o mecanismo de invocação via super encontrado em linguagens de herança simples.

A ordenação dinâmica é necessária nas classes new-style, porque todos os casos de herança múltipla apresentam uma ou mais estruturas de diamante (um losango no grafo de herança, onde pelo menos uma das superclasses pode ser acessada através de vários caminhos a partir de uma classe derivada). Por exemplo, todas as classes new-style herdam de **object**, portanto, qualquer caso de herança múltipla envolvendo apenas classes new-style fornece mais de um caminho para chegar a **object**. Para evitar que uma classe base seja acessada mais de uma vez, o algoritmo dinâmico lineariza a ordem de pesquisa de uma maneira que:

- preserva a ordem da esquerda para a direita especificada em cada classe;
- acessa cada classe base apenas uma vez;
- é monotônica (significa que uma classe pode ser derivada sem que isso afete a ordem de precedência de suas classes base).

Juntas, essas características tornam possível criar classes confiáveis e extensíveis usando herança múltipla.

9.6. Variáveis privadas

Variáveis instância “privadas”, que não podem ser acessados 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 ele uma função, um método ou um atributo de dados). Tais nomes devem ser considerados um detalhe de implementação e sujeito a alteração sem aviso prévio.

Uma vez que existe um caso de uso válido para a definição de atributos privados em classes (especificamente para evitar conflitos com nomes definidos em subclasses), existe um suporte limitado a identificadores privados em classes, chamado *name mangling* (literalmente: desfiguração de nomes). Qualquer identificador no formato `__spam` (no mínimo dois underscores `_` no prefixo e no máximo um sufixo) é substituído por `_nomeclasse__spam`, onde `nomeclasse` é o nome da classe corrente (exceto quando o nome da classe é prefixado com um ou mais underscores _; nesse caso eles são omitidos). Essa desfiguração independe da posição sintática do identificador, desde que ele apareça dentro da definição de uma classe.

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

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable) # referencia ao nome privado

    def update(self, iterable): # parte da API, pode ser sobrescrito
        for item in iterable:
            self.items_list.append(item)

    __update = update   # nome privado do método update

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # altera a assinatura de update()
        # mas não quebra o __init__() original
        for item in zip(keys, values):
            self.items_list.append(item)
```

Note que as regras de desfiguração de nomes foram projetadas para evitar acidentes; ainda é possível acessar e alterar intencionalmente variáveis protegidas por esse mecanismo. De fato isso pode ser útil em certas circunstâncias, por exemplo, durante uma sessão com o **pdb**, o depurador interativo do Python.

Código passado para `exec`, `eval()` ou `execfile()` não considera o nome da classe que invocou como sendo a classe corrente; isso é semelhante ao funcionamento da declaração **`global`**, cujo efeito se aplica somente ao código que é byte-compilado junto. A mesma restrição se aplica as funções `getattr()`, `setattr()` e `delattr()`, e quando acessamos diretamente o `__dict__` da classe: lá as chaves já estão desfiguradas.

9.7. Miscelânea

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

```
class Empregado:
    pass

joao = Empregado() # Criar um registro de empregado vazio

# Preencher campos do registrp
joao.nome = u'João da Silva'
joao.depto = u'laboratório de informática'
joao.salario = 1000
```

Um trecho de código Python que espera um tipo abstrato de dado em particular, pode receber, ao invés disso, um objeto que emula os métodos que aquele tipo suporta. Por exemplo, se você tem uma função que formata dados obtidos de um objeto arquivo, pode passar como argumento para essa função uma instância de uma classe que implemente os métodos **`read()`** e **`readline()`** que obtém os dados lendo um buffer ao invés de ler um arquivo real. (N.d.T. isso é um exemplo de “duck typing” [3].)

Objetos método têm seus próprios atributos: `m.im_self` é uma referência à instância vinculada ao método **`m()`**, e `m.im_func` é o objeto função (atributo da classe) que corresponde ao método.

9.8. Exceções também são classes

Exceções definidas pelo usuário são identificadas por classes. Através deste mecanismo é possível criar hierarquias extensíveis de exceções.

Há duas novas formas semanticamente válidas para o comando **`raise`**:

```
raise Classe, instancia

raise instancia
```

Na primeira forma, `instancia` deve ser uma instância de **`Classe`** ou de uma classe derivada dela. A segunda forma é um atalho para:

```
raise instancia.__class__, instancia
```

Em uma cláusula **except**, uma classe é compatível com a exceção levantada se é a mesma classe ou uma classe ancestral dela (mas não o contrário: uma cláusula **except** que menciona uma classe derivada daquela que foi levantada não vai capturar tal exceção). No exemplo a seguir será exibido B, C e D nessa ordem:

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

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

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

No caso de uma exceção não tratada, quando a mensagem de erro é gerada, o nome da classe da exceção é exibido, seguido de `': '` (dois pontos e um espaço), e finalmente aparece a instância da exceção convertida para string através da função embutida **str()**.

9.9. 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 {'one':1, 'two':2}:
    print chave
for car in "123":
    print car
for linha in open("myfile.txt"):
    print linha
```

Esse estilo de acesso é limpo, conciso e conveniente. O uso de iteradores promove uma unificação ao longo de toda a linguagem. Nos bastidores, o comando **for** aplica a função embutida **iter()** à coleção. Essa função devolve um iterador que define o

método **next()**, que acessa os elementos da coleção em sequência, um por vez. Quando acabam os elementos, **next()** levanta uma exceção **StopIteration**, indicando que o laço **for** deve encerrar. Este exemplo mostra como tudo funciona:

```
>>>
```

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

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

```
class Inversor:
    """Iterador para percorrer uma sequencia de trás para frente."""
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def __iter__(self):
        return self
    def next(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

```
>>>
```

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

9.10. Geradores

Funções geradoras (*generator*) são uma maneira fácil e poderosa de criar um iterador. Uma função geradora é escrita como uma função normal, mas usa o comando **`yield`** para produzir resultados. (N.d.T. Quando invocada, a função geradora produz um objeto gerador.) Cada vez que **`next()`** é invocado, o gerador continua a partir de onde parou (ele mantem na memória seus dados internos e a próxima instrução a ser executada). Um exemplo mostra como geradores podem ser muito fáceis de criar:

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

```
>>>
```

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

N.d.T. Veja como a função geradora produz um objeto gerador, que implementa o protocolo de iterador:

```
>>>
```

```
>>> gerador = inversor('golf')
>>> gerador
<generator object inversor at 0xb7797a2c>
>>> gerador.next()
'f'
>>> gerador.next()
'l'
>>> gerador.next()
'o'
>>> gerador.next()
'g'
>>> gerador.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Qualquer coisa 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 de **`next()`**. Isto torna a função mais fácil de escrever e muito mais clara do que uma implementação usando variáveis de instância como `self.index` e`self.data`.

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

9.11. Expressões geradoras

Alguns geradores simples podem ser escritos sucintamente como expressões usando uma sintaxe similar a de abrangência de listas (*list comprehensions*), mas com parênteses ao invés de colchetes. Essas expressões são destinadas a situações em que o gerador é usado imediatamente como argumento para função. Uma expressão geradora é mais compacta, porém menos versátil do que uma função geradora, e tende a usar muito menos memória do que a abrangência de lista equivalente.

Examplos:

```
>>>
>>> sum(i*i for i in range(10))           # soma de quadrados
285

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

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

>>> palavras_unicas = set(palavra for linha in pagina
...                               for palavra in linha.split())

>>> melhor_aluno = max((aluno.media, aluno.nome) for aluno in formados)

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

Notas

[1]	Exceto por um detalhe. Objetos módulo têm um atributo secreto apenas para leitura chamado **`__dict__`** que é uma referência ao dicionário usado para implementar o namespace do módulo; o nome **`__dict__`** é um atributo mas não um nome global. Obviamente, acessar esse atributo viola a abstração da implementação de namespaces, e é algo que só deve ser feito por ferramentas especiais, como depuradores “post-mortem”.
[2]	N.d.T. Os termos *new-style class* e “old-style class” referem-se a duas implementações de classes que convivem desde o Python 2.2. A implementação mais antiga, “old-style classes” foi preservada até o Python 2.7 para manter a compatibilidade com bibliotecas e scripts antigos, mas deixou de existir a partir do Python 3.0. As “new-style classes” suportam o mecanismo de descritores, usado

para implementar propriedades (*properties*). Recomenda-se que todo código Python novo use apenas “new-style classes”.

Desde o Python 2.2, a forma de declarar uma classe determina se ela usa a implementação nova ou antiga. Qualquer classe derivada direta ou indiretamente de `object` é uma classe “new-style”. Objetos classe novos são do tipo `type` e objetos classe antigos são do tipo `classobj`. Veja este exemplo:

```
>>>
>>> class Nova(object):
...     pass
...
>>> type(Nova)
<type 'type'>
>>> class Velha:
...     pass
...
>>> type(Velha)
<type 'classobj'>
```

Note que a definição acima é recursiva. Em particular, uma classe que herda de uma classe antiga e de uma nova é uma classe “new-style”, pois através da classe nova ela é uma subclasse indireta de `object`. Não é uma boa prática misturar os dois estilos de classes, mas eis um exemplo para ilustrar esse ponto:

```
>>>
>>> class Mista(Velha, Nova):
...     pass
...
>>> type(Mista)
<type 'type'>
```

Para saber mais sobre as diferenças, veja New Class vs Classic Class no wiki do python.org ou artigo original de Guido van Rossum,Unifying types and classes in Python 2.2.

[3] N.d.T. Esse parágrafo descreve uma aplicação do conceito de “duck typing” (literalmente, “tipagem pato”), cuja ideia central é que os atributos e comportamentos de um objeto são mais importantes que seu tipo: “Quando vejo um pássaro que anda com um pato, nada como um pato, e grasna como um pato, chamo esse pássaro de pato.” (James Whitcomb Riley). Segundo a Wikipedia (em inglês), a metáfora dos atributos de um pato no contexto de programação orientada a objetos foi usada pela primeira vez por Alex Martelli no grupo*comp.lang.python* em 26/jul/2000. O assunto da mensagem era polymorphism.

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()      # Devolve o diretório de trabalho atual
'C:\\Python26'
>>> os.chdir('/server/accesslogs')   # Altera o diretório de trabalho atual
>>> os.system('mkdir today')   # Executa o comando mkdir no shell do sistema
0
```

Tome cuidado para 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 pra lidar com módulos grandes como **os**:

```
>>>
>>> import os
>>> dir(os)
<devolve uma lista com todas as funções do módulo>
>>> help(os)
<devolve uma extensa página de manual 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('data.db', 'archive.db')
>>> shutil.move('/build/executables', 'installdir')
```

10.2. Caracteres curinga

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

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

10.3. Argumentos de linha de comando

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

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

O módulo **getopt** processa os argumentos passados em *sys.argv* usando as convenções da função Unix **getopt**. Um processamento mais poderoso e flexível é fornecido pelo módulo **argparse**.

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: iniciando novo arquivo de log\n')
Aviso: iniciando novo arquivo de log
```

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

10.5. Reconhecimento de padrões em strings

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

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

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

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

10.6. Matemática

O módulo **math** oferece acesso as funções da biblioteca C para matemática de ponto flutuante:

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

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

```
>>>
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(xrange(100), 10)   # sampling without replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()      # float aleatório entre 0 e 1 exclusive
0.17970987693706186
>>> random.randrange(6)  # inteiro aleatório escolhido entre range(6)
4
```

10.7. Acesso à internet

Há diversos módulos para acesso e processamento de protocolos da internet. Dois dos mais simples são **urllib2** para efetuar download de dados a partir de urls e **smtplib** para enviar mensagens de correio eletrônico:

```
>>>
>>> import urllib2
>>> for line in urllib2.urlopen('http://tycho.usno.navy.mil/cgi-
bin/timer.pl'):
...     if 'EST' in line or 'EDT' in line:  # procurar pela hora do leste
...         print line

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

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

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

10.8. Data e Hora

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

```
>>>
```

```
>>> # é fácil construir e formatar datas
>>> 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 implementam operações aritméticas
>>> 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**, **zipfile** e **tarfile**.

```
>>>
```

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

10.10. Medição de desempenho

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

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

```
>>>
```

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

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

10.11. Controle de qualidade

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

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

```
def media(valores):
    """Calcula a média aritmética de uma lista de números.

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

import doctest
doctest.testmod()   # Automaticamente valida os testes embutidos
```

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)
        self.assertRaises(ZeroDivisionError, average, [])
        self.assertRaises(TypeError, average, 20, 30, 70)

unittest.main() # Chamando da linha de comando, executa 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 `xmlrpclib` e `SimpleXMLRPCServer` tornam a implementação de chamadas remotas (remote procedure calls) uma tarefa quase trivial. Apesar dos nomes dos módulos, nenhum conhecimento 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 `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.
- Os pacotes `xml.dom` e `xml.sax` oferecem uma implementação robusta deste popular formato de intercâmbio de dados. De modo similar, o módulo `csv` permite ler e escrever diretamente num formato comum de bancos de dados. Juntos esses módulos e pacotes simplificam muito a troca de dados entre aplicações em Python e outras ferramentas.
- Internacionalização está disponível através de diversos módulos, como `gettext`, `locale`, e o pacote `codecs`.

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 **repr** oferece uma versão modificada da função **repr()** para abreviar a exibição de coleções grandes ou profundamente aninhadas:

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

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

```
>>>
>>> import pprint
>>> t = [[[['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() funciona como o método fill() exceto pelo fato
... de devolver uma lista de strings ao invés de uma única string com
... quebras de linha para separar as linhas."""
...
>>> print textwrap.fill(doc, width=40)
O método wrap() funciona como o método
fill() exceto pelo fato de devolver uma
lista de strings ao invés de uma única
string com quebras de linha para separar
as linhas.
```

O módulo **locale** acessa uma base de dados de formatos específicos a determinada cultura. O argumento grouping da função **format()** oferece uma forma direta de formatar números com separadores de grupo:

```
>>>
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'pt_BR.utf8')
'pt_BR.utf8'
>>> conv = locale.localeconv()          # pega um dicionário das convenções
>>> x = 1234567.8
>>> locale.format("%d", x, grouping=True)
'1.234.567'
>>> locale.format_string("%s%.*f", (conv['currency_symbol'],
...                      conv['frac_digits'], x), grouping=True)
'R$1.234.567,80'
```

11.2. Usando templates

O 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('Os ${lugar}nos enviaram $$10 para $causa.')
>>> t.substitute(lugar='Curitiba', causa='as obras de saneamento')
'Os Curitibanos enviaram $10 para as obras de saneamento.'
```

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('Encontre o $item e volte para $lugar.')
>>> d = dict(item='cálice')
>>> print t.substitute(d)
Traceback (most recent call last):
  . . .
KeyError: 'lugar'
>>> print t.safe_substitute(d)
Encontre o cálice e volte para $lugar
```

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
>>> fotos = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class RenomeiaLote(Template):
...     delimiter = '%'
>>> fmt = raw_input('Estilo para o nome (%d-data %n-numseq %f-formato):  ')
Estilo para o nome (%d-data %n-numseq %f-formato):  Ashley_%n%f

>>> t = RenomeiaLote(fmt)
>>> data = time.strftime('%d%b%y')
>>> for i, nome_arquivo in enumerate(fotos):
...     base, ext = os.path.splitext(nome_arquivo)
...     novo_nome = t.substitute(d=data, n=i, f=ext)
...     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

data = open('myfile.zip', 'rb').read()
start = 0
for i in range(3):          # mostra o cabeçalho dos 3 primeiros arquivos
    start += 14
    fields = struct.unpack('<IIIHH', data[start:start+16])
    crc32, comp_size, uncomp_size, filenamesize, extra_size = fields

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

```
    start += extra_size + comp_size     # avança 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, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile
    def run(self):
        f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.infile)
        f.close()
        print 'Terminei de zipar em segundo plano o arquivo: ', self.infile

background = AsyncZip('mydata.txt', 'myarchive.zip')
background.start()
print 'O programa principal continua a sua execução em primeiro plano.'

background.join()    # espera até que a tarefa em segundo plano termine
print 'O programa principal esperou até a tarefa em segundo plano terminar.'
```

O principal desafio para as aplicações que usam múltiplas threads é coordenar as threads que compartilham dados ou outros recursos. Para esta finalidade, o módulo threading oferece alguns mecanismos primitivos de sincronização, como travas (locks), eventos, variáveis de condição e semáforos.

Apesar dessas ferramentas serem poderosas, pequenos erros de projeto podem resultar em problemas difíceis de serem reproduzidos. Então, a maneira preferida de coordenar tarefas é concentrar todo o acesso a determinado recurso em uma única thread e usar o módulo **`Queue`** para alimentar aquela thread com requisições de outras threads. Aplicações usando objetos do tipo **`Queue.Queue`** para comunicação e coordenação inter-thread são mais fáceis de implementar, 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('Informação de debug')
logging.info('Mensagem informativa')
logging.warning('Aviso:arquivo de configuração %s não encontrado',
                'server.conf')
logging.error('Um erro ocorreu')
logging.critical('Erro crítico -- encerrando o programa.')
```

Isso produz a seguinte saída:

```
WARNING:root:Aviso:arquivo de configuração server.conf não encontrado
ERROR:root:Um erro ocorreu
CRITICAL:root:Erro crítico -- encerrando o programa.
```

Por padrão, mensagens informativas e de depuração são suprimidas e a saída é enviada para a saída de erros padrão (stderr). Outras opções de saída incluem envio de mensagens através de correio eletrônico, datagramas, sockets ou para um servidor HTTP. Novos filtros podem selecionar diferentes formas de envio de mensagens, baseadas na prioridade da mensagem: **DEBUG**, **INFO**, **WARNING**, **ERROR** e **CRITICAL**.

O sistema de log pode ser configurado diretamente do Python ou pode ser carregado a partir de um arquivo de configuração editável pelo usuário para logs personalizados sem a necessidade de alterar a aplicação.

11.6. Referências fracas

Python faz gerenciamento automático de memória (contagem de referências para a maioria dos objetos e *garbage collection* [coleta de lixo] para eliminar ciclos). A memória ocupada por um objeto é liberada logo depois da última referência a ele ser eliminada.

Essa abordagem funciona bem para a maioria das aplicações, mas ocasionalmente surge a necessidade de rastrear objetos apenas enquanto estão sendo usados por algum outro. Infelizmente rastreá-los cria uma referência, e isso os fazem permanentes. O módulo **weakref**oferece ferramentas para rastrear objetos sem criar uma referência. Quando o objeto não é mais necessário, ele é automaticamente removido de uma tabela de referências fracas e uma chamada (*callback*) é disparada. Aplicações típicas incluem cacheamento de objetos que são muito custosos para criar:

```
>>>
>>> import weakref, gc
>>> class A:
...     def __init__(self, value):
```

```
...             self.value = value
...     def __repr__(self):
...             return str(self.value)
...
>>> a = A(10)                   # cria uma referência
>>> d = weakref.WeakValueDictionary()
>>> d['primary'] = a            # não cria uma referência
>>> d['primary']                # pega o objeto se ele ainda estiver vivo
10
>>> del a                       # remove a única referência
>>> gc.collect()                # roda o coletor de lixo logo em seguida
0
>>> d['primary']                # A entrada foi automaticamente removida
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    d['primary']                # A entrada foi automaticamente removida
  File "C:/python26/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
KeyError: 'primary'
```

11.7. Ferramentas para trabalhar com listas

Muitas necessidades envolvendo estruturas de dados podem ser satisfeitas com o tipo embutido lista. Entretanto, algumas vezes há uma necessidade por implementações alternativas que sacrificam algumas facilidades em nome de melhor desempenho.

O módulo **array** oferece uma classe **array**, semelhante a uma lista, mas que armazena apenas dados homogêneos e de maneira mais compacta. O exemplo a seguir mostra um vetor de números armazenados como números binários de dois bytes sem sinal (código de tipo "H") ao invés dos 16 bytes usuais para cada item em uma lista de int:

```
>>>
>>> from array import array
>>> a = array('H', [4000, 10, 700, 22222])
>>> sum(a)
26932
>>> a[1:3]
array('H', [10, 700])
```

O módulo **collections** oferece um objeto **deque()** que comporta-se como uma lista mas com *appends* e *pops* pela esquerda mais rápidos, porém mais lento ao percorrer o meio da sequência. Esses objetos são adequados para implementar filas e buscas de amplitude em árvores de dados (*breadth first tree searches*):

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

nao_buscados = deque([noh_inicial])
```

```
def busca_em_amplitude(nao_buscados):
    noh = nao_buscados.popleft()
    for m in gen_moves(noh):
        if eh_objetivo(m):
                return m
        nao_buscados.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
>>> pontos = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500, 'python')]
>>> bisect.insort(pontos, (300, 'ruby'))
>>> pontos
[(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400, 'lua'), (500, 'python')]
```

O módulo **heapq** oferece funções para implementação de *heaps* baseadas em listas normais. O valor mais baixo é sempre mantido na posição zero. Isso é útil para aplicações que acessam repetidamente o menor elemento, mas não querem reordenar a lista toda a cada acesso:

```
>>>
>>> from heapq import heapify, heappop, heappush
>>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(data)                      # re-arranja a lista numa ordem heap
>>> heappush(data, -5)                 # adiciona um novo item
>>> [heappop(data) for i in range(3)]  # recupera os três menores itens
[-5, 0, 1]
```

11.8. Aritmética decimal com ponto flutuante

O módulo **decimal** oferece o tipo **Decimal** para aritmética decimal com ponto flutuante. Comparado a implementação embutida **float** que usa aritmética binária de ponto flutuante, a classe é especialmente útil para:

- aplicações financeiras que requerem representação decimal exata,
- controle sobre a precisão,
- controle sobre arredondamento para satisfazer requisitos legais,
- rastreamento de casas decimais significativas, ou
- aplicações onde o usuário espera que os resultados sejam os mesmos que os dos cálculos feitos à mão.

Por exemplo, calcular um imposto de 5% sobre uma chamada telefônica de 70 centavos devolve diferentes resultados com aritmética de ponto flutuante decimal ou binária. A diferença torna-se significativa se os resultados são arredondados para o centavo mais próximo.

```
>>>
```

```
>>> from decimal import *
>>> x = Decimal('0.70') * Decimal('1.05')
>>> x
Decimal('0.7350')
>>> x.quantize(Decimal('0.01'))  # arredonda para o centavo mais próximo
Decimal('0.74')
>>> round(.70 * 1.05, 2)         # o mesmo cálculo com float
0.73
```

O resultado de **Decimal** considera zeros à direita, automaticamente inferindo quatro casas decimais a partir de multiplicandos com duas casas decimais. O módulo **decimal** reproduz a aritmética como fazemos à mão e evita problemas que podem ocorrer quando a representação binária do ponto flutuante não consegue representar quantidades decimais com exatidão.

A representação exata permite à classe **Decimal** executar cálculos de módulo e testes de igualdade que não funcionam bem em ponto flutuante binário:

```
>>>
```

```
>>> Decimal('1.00') % Decimal('.10')
Decimal('0.00')
>>> 1.00 % 0.10
0.09999999999999995

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

O módulo **decimal** implementa a aritmética com tanta precisão quanto necessária:

```
>>>
```

```
>>> getcontext().prec = 36
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857')
```

12. E agora?

Espera-se que a leitura deste tutorial tenha aguçado seu interesse em usar Python — você deve estar sedento para aplicar Python na solução de seus problemas reais. Agora, onde você deveria ir para aprender mais?

Este tutorial é uma parte da documentação do Python. Algumas outras partes desta documentação são:

- *The Python Standard Library*: Você deveria navegar através deste manual que é um material de referência completo (embora conciso) sobre os tipos, funções e módulos incluídos na biblioteca padrão. A distribuição padrão de Python inclui *muito* código extra. Há módulos para ler caixas de e-mail Unix, recuperar documentos via HTTP, gerar números aleatórios, tratar opções de linha de comando, escrever programas CGI, comprimir dados e muitas outras tarefas. Passear pela Referência da Biblioteca Padrão te dará uma ideia do que está disponível.
- *Installing Python Modules* explica como instalar módulos externos escritos por outros usuários de Python.
- *The Python Language Reference*: Uma explicação detalhada sobre a sintaxe e a semântica do Python. É uma leitura pesada, mas é útil como um guia completo à linguagem em si.

Mais recursos sobre python:

- http://www.python.org: O site oficial do Python. Contém código, documentação a aponta para para outras páginas na web relacionadas a Python. Este site é espelhado em vários lugares ao redor do mundo como Europa, Japão e Austrália; Um servidor secundário pode ser mais rápido que o site principal dependendo da sua localização geográfica.
- http://www.python.org.br: Site mantido pela comunidade brasileira de Python.
- http://docs.python.org: Acesso rápido à documentação do Python.
- http://pypi.python.org: O índice de pacotes Python (Python package index), anteriormente apelidado de Cheese Shop [1], é um índice de módulos Python criados pelos usuários. Uma vez que você começar a publicar código, pode registrar seus pacotes aqui para que outros possam encontrá-los.
- http://aspn.activestate.com/ASPN/Python/Cookbook/: O *Python Cookbook* (livro de receitas de Python) é uma grande coleção de exemplos de código, módulos maiores e scripts úteis. Contribuições particularmente notáveis foram reunidas em um livro chamado Python Cookbook (O'Reilly & Associates, ISBN 0-596-00797-3.)

Você pode enviar suas questões e problemas relacionados a Python, em inglês, para o newsgroup *comp.lang.python* ou enviá-las para a lista de e-mails em python-list@python.org. O newsgroup e a lista de e-mails são interligadas então mensagens

enviadas para uma serão automaticamente encaminhadas para outra. São por volta de 55 mensagens diárias (com picos de muitas centenas), com perguntas (e respostas), sugestões de novas funcionalidades e anúncios de novos módulos. Antes de enviar mensagens, leia a lista de perguntas frequentes (também chamada de FAQ), ou dê uma olhada no diretório `Misc/` incluído na distribuição dos fontes do Python. Os arquivos da lista de e-mails estão disponíveis em http://mail.python.org/pipermail/. A FAQ responde a muitas questões que chegam frequentemente e pode ter a solução para o seu problema.

Existe também o grupo de discussão da comunidade brasileira de Python: python-brasil no Google Groups, (média de 17,6 mensagens por dia em 2011), com discussões de ótimo nível técnico. Fique à vontade para mandar suas perguntas, mas antes de enviá-las, dê uma lida na página AntesDePerguntar.

[1] N.d.T. “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.

13. Edição interativa de entrada e substituição por histórico

Algumas versões do interpretador Python suportam facilidades de edição e substituição semelhantes às encontradas na shell Korn ou na GNU Bash. Isso é implementado através da biblioteca GNU Readline, que suporta edição no estilo Emacs ou vi. Essa biblioteca possui sua própria documentação, que não será duplicada aqui. Porém os fundamentos são fáceis de serem explicados. As facilidades aqui descritas estão disponíveis nas versões Unix e Cygwin do interpretador.

Este capítulo *não* documenta as facilidades de edição do pacote PythonWin de Mark Hammond, ou do ambiente IDLE baseado em Tk e distribuído junto com Python. A recuperação de histórico da linha de comando do DOS ou NT e outros sabores de DOS e Windows também são bichos diferentes.

13.1. Edição de linha

Se instalada, a edição de linha está ativa sempre que o interpretador exibir um dos prompts (primário ou secundário). A linha atual pode ser editada usando comandos típicos do Emacs. Os mais importantes são: `C-A` (Control-A) move o cursor para o início da linha, `C-E` para o fim, `C-B`move uma posição para à esquerda, `C-F` para a direita. `Backspace` apaga o caractere à esquerda, `C-D` apaga o da direita. `C-K` apaga do cursor até o resto da linha à direita, `C-Y` cola a linha apagada. `C-underscore` desfaz a última alteração que você fez; e pode ser repetido com efeito cumulativo.

13.2. Substituição por histórico

Funciona da seguinte maneira: todas linhas não vazias são armazenadas em um buffer de histórico, e ao digitar uma nova linha no prompt você está editando a última linha deste buffer. `C-P` retrocede uma linha no histórico, `C-N` avança uma linha. Qualquer linha desse histórico pode ser editada; quando você faz isso, um astesrico aparece na frente do prompt. Pressionando `Enter` a linha atual é enviada para o interpretador. `C-R`inicia uma busca para trás no histórico, e `C-S` faz uma busca para frente.

13.3. Definição de atalhos

Atalhos de teclado e outros parâmetros da biblioteca Readline podem ser personalizados colocando configurações no arquivo `~/.inputrc`. A definição de atalhos tem o formato

```
nome-da-tecla: nome-da-funcao
```

ou

```
"string": nome-da-funcao
```

e opções podem ser especificadas com

```
set nome-da-opcao valor
```

Por exemplo:

```
# Prefiro o estilo de edição do vi:
set editing-mode vi

# Edição em uma única linha:
set horizontal-scroll-mode On

# Redefinição de algumas teclas:
Meta-h: backward-kill-word
"\C-u": universal-argument
"\C-x\C-r": re-read-init-file
```

Observe que a definição padrão para `Tab` em Python é inserir um caractere `Tab` ao invés de completar o nome de um arquivo (padrão no Readline). Isto pode ser reconfigurado de volta colocando:

Tab: complete

em seu `~/.inputrc`. Todavia, isto torna mais difícil digitar comandos indentados em linhas de continuação se você estiver acostumado a usar`Tab` para isso.

O preenchimento automático de nomes de variáveis e módulos estão opcionalmente disponíveis. Para habilitá-los no modo interativo, adicione o seguinte ao seu arquivo de inicialização: [2]

```
import rlcompleter, readline
readline.parse_and_bind('tab: complete')
```

Isso vincula a tecla `Tab` para o preenchimento automático de nomes de função. Assim, teclar `Tab` duas vezes dispara o preenchimento, procurando um determinado nome entre as variáveis locais e módulos disponíveis. Para expressões terminadas em ponto, como em `string.a`, a expressão será avaliada até o último `'.'` quando serão sugeridos possíveis complementos. Note que isso pode executar código da sua aplicação quando um objeto que define o método **`__getattr__()`** fizer parte da expressão.

Um arquivo de inicialização mais completo seria algo como esse exemplo. Note que ele deleta os nomes que cria quando não são mais necessários; isso é feito porque o arquivo de inicialização é executado no mesmo ambiente dos comandos interativos, e remover os nomes evita criar efeitos colaterais no ambiente interativo. Você pode achar conveniente manter alguns dos módulos importados, como **`os`**, que acaba sendo necessário na maior parte das sessões com o interpretador.

```
# Adiciona autocompletar e um arquivo de histórico de comandos ao
# interpretador interativo Python. Requer Python 2.0+ e Readline.
# O autocompletar está associado, por padrão, à tecla Esc (você pode
# alterar isso, veja a documentação do Readline)
#
# Salve o arquivo como ~/.pystartup e defina uma variável de ambiente
# apontando para ele digitando no bash:
#   $ export PYTHONSTARTUP=~/.pystartup

import atexit
import os
import readline
import rlcompleter

historyPath = os.path.expanduser("~/.pyhistory")

def save_history(historyPath=historyPath):
    import readline
    readline.write_history_file(historyPath)

if os.path.exists(historyPath):
    readline.read_history_file(historyPath)

atexit.register(save_history)
del os, atexit, readline, rlcompleter, save_history, historyPath
```

13.4. Alternativas para o interpretador interativo

Essa facilidade representa um enorme passo em comparação com versões anteriores do interpretador. Todavia, ainda há desejos não atendidos. Seria interessante se a indentação apropriada fosse sugerida em linhas de continuação, pois o parser sabe se um token de indentação é necessário. O mecanismo de autocompletar poderia utilizar a tabela de símbolos do interpretador. Também seria útil um comando para verificar (ou até mesmo sugerir) o balanceamento de parênteses, aspas, etc.

Um poderoso interpretador interativo alternativo que tem sido bastante utilizado já há algum tempo é o IPython, que possui recursos de autocompletar, exploração de objetos e avançado gerenciamento de histórico. Ele também pode ser personalizado e incorporada em outras aplicações. Outro poderoso ambiente interativo similar é o bpython.

Footnotes

[1] N.d.T: Algumas vezes, o `C-S` pode conflitar com a controle de fluxo XON/XOFF (no Konsole por exemplo). Como essa pesquisa é um característica da GNU Readline, você pode associa-lá a outra tecla. Contudo, é melhor e mais simples simplesmente desativar o XON/XOFF executando o seguinte comando: “stty -ixon” no shell. Além disso, é necessário que a opção mark-modified-lines da GNU Readline esteja ativa para que o asterisco apareça quando uma linha do histórico é alterada.

[2] Python executará o conteúdo do arquivo identificado pela variável de

ambiente **PYTHONSTARTUP** quando se inicia o interpretador no modo interativo. Para personalizar Python no modo não-interativo, veja *Os módulos de customização*.

14. Aritmética de ponto flutuante: problemas e limitações

Números de ponto flutuante são representados no hardware do computador como frações binárias (base 2). Por exemplo, a fração decimal:

```
0.125
```

tem o valor 1/10 + 2/100 + 5/1000, e da mesma maneira a fração binária:

```
0.001
```

tem o valor 0/2 + 0/4 + 1/8. Essas duas frações têm valores idênticos, a única diferença real é que a primeira está representada na forma de frações base 10, e a segunda na base 2.

Infelizmente, muitas frações decimais não podem ser representadas precisamente como frações binárias. O resultado é que, em geral, os números decimais de ponto flutuante que você digita acabam sendo armazenados de forma apenas aproximada, na forma de números binários de ponto flutuante.

O problema é mais fácil de entender primeiro em base 10. Considere a fração 1/3. Podemos representá-la aproximadamente como uma fração base 10:

```
0.3
```

ou melhor,

```
0.33
```

ou melhor,

```
0.333
```

e assim por diante. Não importa quantos dígitos você está disposto a escrever, o resultado nunca será exatamente 1/3, mas será uma aproximação de cada vez melhor de 1/3.

Da mesma forma, não importa quantos dígitos de base 2 você está disposto a usar, o valor decimal 0.1 não pode ser representado exatamente como uma fração de base 2. Em base 2, 1/10 é uma fração binária que se repete infinitamente:

```
0.0001100110011001100110011001100110011001100110011...
```

Se limitamos a representação a qualquer número finito de bits, obtemos apenas uma aproximação.

Em uma máquina típica rodando Python, há 53 bits de precisão disponível para um `float`, de modo que o valor armazenado internamente quando você digita o número decimal `0.1` é esta fração binária:

```
0.0001100110011001100110011001100110011001100110011001100110011010
```

que chega perto, mas não é exatamente igual a 1/10.

É fácil esquecer que o valor armazenado é uma aproximação da fração decimal original, devido à forma como floats são exibidos no interpretador interativo. Python exibe apenas uma aproximação decimal do verdadeiro valor decimal da aproximação binária armazenada pela máquina. Se Python exibisse o verdadeiro valor decimal da aproximação binária que representa o decimal 0.1, teria que mostrar:

```
>>>
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

Isso é bem mais dígitos do que a maioria das pessoas considera útil, então Python limita o número de dígitos, apresentando em vez disso um valor arredondado:

```
>>>
>>> 0.1
0.1
```

É importante perceber que isso é, de fato, uma ilusão: o valor na máquina não é exatamente 1/10, estamos simplesmente arredondando a exibição do verdadeiro valor na máquina. Esse fato torna-se evidente logo que você tenta fazer aritmética com estes valores:

```
>>>
>>> 0.1 + 0.2
0.30000000000000004
```

Note que esse é a própria natureza do ponto flutuante binário: não é um bug em Python, e nem é um bug em seu código. Você verá o mesmo tipo de coisa em todas as linguagens que usam as instruções de aritmética de ponto flutuante do hardware (apesar de algumas linguagens não*mostrarem* a diferença, por padrão, ou em todos os modos de saída).

Outras surpresas decorrem desse fato. Por exemplo, se tentar arredondar o valor 2.675 para duas casas decimais, obterá esse resultado:

```
>>>
```

```
>>> round(2.675, 2)
2.67
```

A documentação da função embutida `round()` diz que ela arredonda para o valor mais próximo, e em caso de empate opta pela aproximação mais distante de zero. Uma vez que a fração decimal 2.675 fica exatamente a meio caminho entre 2.67 e 2.68, poderíamos esperar que o resultado fosse (uma aproximação binária de) 2.68. Mas não é, porque quando a string decimal `2.675` é convertida em um número de ponto flutuante binário, é substituída por uma aproximação binária, cujo valor exato é:

```
2.674999999999999822364316059974953532218933105468 75
```

Uma vez que esta aproximação é ligeiramente mais próxima de 2.67 do que de 2.68, acaba sendo arredondada para baixo.

Se você estiver em uma situação onde precisa saber exatamente como esses valores intermediários são arredondados, considere usar o módulo `decimal`. Aliás, o módulo `decimal` também oferece uma boa maneira de "ver" o valor exato que é armazenado em qualquer float em Python:

```
>>>
```

```
>>> from decimal import Decimal
>>> Decimal(2.675)
Decimal('2.674999999999999822364316059974953532218933105468 75')
```

Outra consequência é que, uma vez que 0.1 não é exatamente 1/10, somar 0.1 dez vezes também não produz exatamente 1.0:

```
>>>
```

```
>>> soma = 0.0
>>> for i in range(10):
...     soma += 0.1
...
>>> soma
0.9999999999999999
```

A aritmética de ponto flutuante binário traz muitas surpresas como essas. O problema do "0.1" é explicado em detalhes precisos abaixo, na seção Erro de Representação. Para uma descrição mais completa de outras surpresas comuns, veja The Perils of Floating Point.

Apesar de que os casos patológicos existem, na maioria dos usos cotidianos de aritmética de ponto flutuante ao fim você verá o resultado esperado simplesmente arredondando a exibição dos resultados finais para o número de dígitos decimais que deseja. Para ter um bom controle sobre como um float é exibido, veja os especificadores de formato do método `str.format()` em *Format String Syntax*.

14.1. Erro de representação

Esta seção explica o exemplo do “0,1” em detalhes, e mostra como você pode realizar uma análise exata de casos semelhantes. Assumimos que você tem uma familiaridade básica com 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 Python (ou Perl, C, C++, Java, Fortran, e muitas outras) frequentemente não exibe o número decimal exato que esperamos:

```
>>>
>>> 0.1 + 0.2
0.30000000000000004
```

Por que isso acontece? 1/10 e 2/10 não são representáveis exatamente como frações binárias. Quase todas as máquinas atuais (julho de 2010) usam aritmética de ponto flutuante conforme a norma IEEE-754, e Python, em quase todas as plataformas, representa um float como um “IEEE-754 double precision float” (“float de precisão dupla IEEE-754”). Os tais “doubles 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 pode, 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* tem exatamente 53 bits (é `>= 2**52`, mas `< 2**53`), o melhor valor para *N* é 56:

```
>>>
>>> 2**52
4503599627370496
>>> 2**53
9007199254740992
>>> 2**56/10
7205759403792793
```

Ou seja, 56 é o único valor de *N* que deixa *J* com exatamente 53 bits. O melhor valor possível para *J* então é aquele quociente arredondado:

```
>>>
>>> q, r = divmod(2**56, 10)
>>> r
6
```

Uma vez que o resto é maior que a metade de 10, a melhor aproximação é obtida arredondando para cima:

```
>>>
>>> q+1
7205759403792794
```

Portanto, a melhor aproximação possível de 1/10 como um “IEEE-754 double precision” é aquele valor dividido por 2** 56, ou:

```
7205759403792794 / 72057594037927936
```

Note que, como arredondamos para cima, esse valor é de fato um pouco maior que 1/10; se não tivéssemos arredondado para cima, o quociente teria sido um pouco menor que 1/10. Mas em nenhum caso ele pode ser *exatamente* 1/10!

Por isso, o computador nunca “vê” 1/10: o que ele vê é exatamente a fração dada acima, a melhor aproximação “IEEE-754 double” possível:

```
>>>
>>> .1 * 2**56
7205759403792794.0
```

Se multiplicarmos essa fração por 10**30, podemos ver o valor (truncado) de seus 30 dígitos mais significativos:

```
>>>
>>> 7205759403792794 * 10**30 // 2**56
100000000000000005551115123125L
```

o que significa que o número exato armazenados no computador é aproximadamente igual ao o valor decimal 0.100000000000000005551115123125. Em versões de Python anteriores a 2.7 e 3.1, Python exibia esse valor arredondado para 17 dígitos significativos, produzindo ‘0.10000000000000001’. Nas versões atuais, Python exibe a fração decimal mais curta que pode ser convertida para o verdadeiro valor binário, o que resulta simplesmente em ‘0.1’.