Tutorial de Python

Release 2.1

Guido van Rossum
Fred L. Drake, Jr., editor
Rodrigo D. A. Senra, tradutor

3 de setembro de 2004

PythonLabs
E-mail: python-docs@python.org
E-mail Tradutor: rodsenra@gpr.com.br

Resumo

Python é uma linguagem de programação poderosa e de fácil aprendizado. Ela 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, em adição a 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 de Python e sua extensa biblioteca padrão estão disponíveis na forma de código fonte ou binário para a maioria das plataformas a partir do sítio, 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 de Python é fácilmente 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 *Python Library Reference*. O *Python Reference Manual* oferece uma definição formal da linguagem. Para se escrever extensões em C ou C++, leia *Extending and Embedding the Python Interpreter*. Existem também diversos livros abordando Python em 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 par aprender mais sobre os diversos módulos de biblioteca descritos em *Python Library Reference*.

A tradução deste tutorial foi patrocinada pela GPr Sistemas Ltda, e versões deste documento em formato eletrônico estão disponívies no site da empresa (http://www.gpr.com.br).

SUMÁRIO

CAPÍTULO

ONE

Abrindo o Apetite

Se alguma vez você já escreveu um extenso script de shell, provavelmente se sentiu assim: você adoraria adicionar mais uma característica, mas já está tão lento, e tão grande, e tão complicado; ou a nova característica implica numa chamada de sistema ou função só acessível a partir do C . . . Tipicamente o problema em questão não é sério o suficiente para motivar a re-escrita do script em C; talvez o problema exija cadeias de caracteres de comprimento variável ou tipos (como listas ordenadas de nomes de arquivos) que são fácilmente manipuláveis na shell, porém demandam muito esforço de implementação em C, ou talvez você nem esteja suficientemente familiarizado com C.

Outra situação: suponha que você tenha que trabalhar com diversas bibiotecas em C, e o típico ciclo escreve/compila/testa/re-compila seja muito lento. Você precisa desenvolver software de uma forma mais ágil. Ou, suponha que você escreveu um programa que precise fazer uso de uma linguagem de extensão, e você não quer projetar a linguagem, implementar e depurar um interpretador para ela, para só então estabelecer o vínculo com sua aplicação original.

Nestes casos, Python possivelmente é exatamente do que você está precisando. Python é simples 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 alto nível*, ela possui tipos nativos de alto nível: dicionários e vetores (*arrays*) flexíveis que lhe custariam dias para obter uma implementação eficiente em C. Devido ao suporte nativo a tipos genéricos, Python é aplicável a um domínio de problemas muito mais vasto do que *Awk* ou até mesmo *Perl*, ainda assim Python é tão fácil de usar quanto essas linguagens sob diversos aspectos.

Python permite que você organize seu programa em módulos que podem ser reutilizados em outros programas escritos 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. Há também módulos nativos que implementam manipulação de arquivos, chamadas do sistema, sockets, e até mesmo acesso a bibiotecas de construção de interfaces gráficas, como Tk.

Python é uma linguagem interpretada, que pode fazer com que você economize um tempo considerável, uma vez que não há necessidade de compilação e vinculação(*linking*) durante o desenvolvimento. 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 ou C++, por diversas razões:

- os tipos de alto-nível permitem que você expresse operações complexas em uma único comando (*statement*);
- a definição de bloco é feita por identação 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 disponívies em formato binário (como uma bibloteca 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 repulsivos répteis. Fazer referências à citações do show na documentação não é só permitido, como também é encorajado!

1.1 Para Onde Ir a Partir Daqui

Agora que você está entusiasmado com Python, vai desejar examiná-la com maior detalhe. Partindo do princípio que a melhor maneira de aprender uma linguagem é usá-la, você está convidado agora a fazê-lo.

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

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

CAPÍTULO

TWO

Utilizando o Interpretador Python

2.1 Disparando o Interpretador

O interpretador é frequentemente instalado como '/usr/local/bin/python' nas máquinas em que é disponibilizado; adicionando '/usr/local/bin' ao caminho de busca (*search path*) da shell de seu UNIX torna-se possível iniciá-lo digitando :

```
python
```

na shell. 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 outra alternativa popular para instalação.)

Digitando um caracter EOF() (`Control-D` on UNIX, `Control-Z`no DOS ou Windows) diretamente no prompt força o interpretador a sair com status de saída zero. Se isso não funcionar, você pode sair do interpretador através da digitação do seguinte: '`import sys; sys.exit()`'.

As características de edição de linha não são muito sofisticadas. Sobre UNIX, seja lá quem for que tenha instalado o interpretador talvez tenha habilitado o suporte a biblioteca readline da GNU, 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 houver um beep, você possui edição de linha de comando; veja o Apêndice A para uma introdução as teclas especiais. Se nada acontecer, ou se `^P` aparecer na tela, a opção de edição não está disponível; você apenas será capaz de usar o *backspace* para remover caracteres da linha corrente.

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

Uma terceira forma de disparar o interpretador é '**python -c** *command* `[arg] ...`', que executa o(s) comando(s) especificados na posição *command*, analogamente a opção de shell **-c**. Considerando que comandos Python possuem frequentemente espaços em branco (ou outros caracteres que sejam especiais para a shell) é aconselhável que o comando especificado em *command* esteja dentro de aspas duplas.

Observe que há uma diferença entre '`python file`' e '`python <file`'. No útlimo caso, chamadas do tipo `input()` e `raw_input()` serão satisfeitas a partir de *file*. Uma vez que *file* já foi inteiramente lido antes de que o script Python entrasse em execução, o programa encontrará o fim de arquivo (*EOF*) imediatamente. Já no primeiro caso, que é o mais frequente (provavelmente o que você quer), a entrada de dados será fornecida pelo dispositivo vinculado a entrada padrão do interpretador.

Quando um arquivo de script é utilizado, às vezes é útil ser capaz de executá-lo para logo em seguida entrar em modo interativo. Este efeito pode ser obtido pela adição do parâmetro **-i** antes do nome do script. (Observe que isto não irá

funcionar se o script for lido a partir da entrada padrão, pelas mesmas razões explicadas no parágrafo anterior).

2.1.1 Passagem de Argumentos

O nome do script e subsequentes parâmetros da linha de comando da shell são acessíveis ao próprio script através da variável `sys.argv`, que é uma lista de strings. Essa lista tem sempre ao menos um elemento; quando nenhum script ou parâmetro forem passados para o interpretador, `sys.argv[0]` será uma lista vazia. Quando o nome do script for `'-'` (significando entrada padrão), o conteúdo de `sys.argv[0]` será `'-'`. Quando for utilizado **-c** *command*, `sys.argv[0]` conterá `'-c'`. Opções especificadas após **-c** *command* não serão consumidas pelo interpretador mas armazenadas em `sys.argv`.

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 requisita por um próximo comando através do *prompt primário*, tipicamente três sinais de maior-que (‘`>>> `’); para linhas de continuação do comando corrente, o *prompt secundário* default são três pontos (‘`... `’). O interpretador imprime uma mensagem de boas vindas, informando seu número de versão e uma nota legal de copyright antes de oferecer o primeiro prompt, ex.:

```
python
Python 1.5.2b2 (#1, Feb 28 1999, 00:02:06)  [GCC 2.8.1] on sunos5
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>>
```

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 fora dele!"
...
Cuidado para não cair fora dele!
```

2.2 O Interpretador e seu Ambiente

2.2.1 Tratamento de Erros

Quando ocorre um erro, o interpretador imprime uma mensagem de erro e a situação da pilha (daqui em diante *stack trace*). No modo interativo, retorna-se ao prompt primário; quando a entrada vem de um arquivo, o interpretador aborta sua execução com status de erro diferente de zero após imprimir o *stack trace* (Exceções tratadas por um `except` num bloco `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.

Teclando o caracter de interrupção (tipicamente Control-C ou DEL) no prompt primário ou secundário cancela a entrada de dados corrente e retorna-se ao prompt primário. [1] Provocando uma interrupção enquanto um comando está em execução levanta a exceção `KeyboardInterrupt`, a qual pode ser tratada em um bloco `try`.

[1] Um problema com o pacote Readline da GNU evita isso

2.2.2 Scripts Executáveis em Python

Em sistemas UNIXBSD, scripts Python podem ser transformados em executáveis, como shell scripts, pela inclusão do cabeçalho

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

(Assumindo que o interpretador foi incluído do caminho de busca do usuário (PATH)) e que o script tenha a permissão de acesso habilitada para execução. O '`#!`' deve estar no início do arquivo . Observe que o caracter '#' designa comentários em Python.

2.2.3 O Arquivo de Inicialização para Modo Interativo

Quando você for utilizar Python interativamente, pode ser útil adicionar uma série de comandos a serem executados por default antes de cada sessão de utilização do interpretador. Isto pode ser obtido pela configuração da variável de ambiente PYTHONSTARTUP para indicar o nome do arquivo script que contém o script de inicialização. Essa característica assemelha-se aos arquivos '.profile' de shells UNIX.

O 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 (caso contrário se comportaria como uma sessão interativa). O script de inicialização é executado no mesmo contexto (doravante *namespace*) 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` através deste arquivo.

Se for necessário ler um script adicional de inicialização a partir do diretório corrente, você pode programar isso a partir do script de inicialização global, ex.: '`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)
```

Uma Introdução Informal a Python

Nos exemplos seguintes, pode-se distinguir a entrada da saída pela presença ou ausência dos prompts (‘`>>> `’ and ‘`... `’): para repetir o exemplo, você deve digitar tudo após o prompt, quando o mesmo aparece; linhas que não começarem com o prompt são na verdade as saídas geradas pelo interpretador.

Observe que existe um segundo prompt indicando a linha de continuação de um comando com múltiplas linhas, o qual pode ser encerrado pela digitação de um linha em branco.

Muitos dos exemplos neste manual, até mesmo aqueles digitados interativamente, incluem comentários. Comentários em Python são delimitados pelo caracter ‘#’, e se extendem até o final da linha. Um comentário pode aparecer no início da linha, depois de um espaço em branco ou código, mas nunca dentro de uma string (literal). O delimitar de comentário dentro de uma string é interpretado como o próprio caracter.

Alguns exemplos:

```
# primeiro comentário
SPAM = 1                 # e esse é o segundo comentário
                         # ... e ainda um terceiro !
STRING = "# Este não é um comentário."
```

3.1 Utilizando Python Como Uma Calculadora

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

3.1.1 Números

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

```
>>> 2+2
4
>>> # Isso é um comentário
... 2+2
4
>>> 2+2  # e um comentário na mesma linha de um comando
4
>>> (50-5*6)/4
```

```
5
>>> # Divisão inteira retorna com aredondamento para base
... 7/3
2
>>> 7/-3
-3
```

Como em C, o sinal de igual (‘=’) é utilizado para atribuição de um valor a uma variável. O valor da atribuição não é escrito:

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

Um valor pode ser atribuído a diversas variáveis simultaneamente:

```
>>> x = y = z = 0  # Zero x, y e z
>>> x
0
>>> y
0
>>> z
0
```

Há total suporte para ponto-flutuante; operadores com operandos de diferentes tipos convertem o inteiro para ponto-flutuante:

```
>>> 4 * 2.5 / 3.3
3.0303030303
>>> 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*`+`*imag*`j)`’, 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 números ponto-flutuante, a parte real e a parte imaginária. Para extrair as partes de um número *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 ponto-flutuante e inteiro (`float()`, `int()` e `long()`) não funcionam para números complexos — não existe maneira correta de converter um número complexo para um número real. Utilize `abs(z)` para obter sua magnitude (como ponto-flutuante) ou `z.real` para obter sua parte real.

```
>>> a=1.5+0.5j
>>> float(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: can't convert complex to float; use e.g. abs(z)
>>> a.real
1.5
>>> abs(a)
1.58113883008
```

No modo interativo, a última expressão a ser impressa é atribuída a variável `_`. Isso significa que ao utilizar Python como uma calculadora, é muitas vezes mais fácil prosseguir com os cálculos da seguinte forma:

```
>>> tax = 17.5 / 100
>>> price = 3.50
>>> price * tax
0.61249999999999993
>>> price + _
4.1124999999999998
>>> round(_, 2)
4.1100000000000003
```

Essa variável especial deve ser tratada somente para leitura pelo usuário. Nunca lhe atribua explicitamente um valor – do contrário, estaria se criando uma outra variável (homônima) independente, que mascararia o comportamento mágico da variável especial.

3.1.2 Strings

Além de números, Python também pode manipular strings, que podem ser expressas de diversas formas. Elas podem ser delimitadas pos 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.'
```

Strings que contém mais de uma linha podem ser construídas de diversas maneiras. Terminadores de linha podem ser embutidos na string com barras invertidas, ex.:

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

que produziria o seguinte resultado:

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

Ou, strings podem ser delimitadas por pares de aspas tríplices: `"` ou `'''`. Neste caso não é necessário embutir terminadores de linha, pois o texto da string será tratado verbatim.

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

produz a seguinte saída:

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

O interpretador imprime o resultado de operações sobre strings da mesma forma que as strings são formatadas na digitação: dentro de aspas, e com caracteres especiais embutidos em *escape sequences*, para mostar seu valor com precisão. A string será delimitada por aspas duplas se ela contém um único caracter de aspas simples e nenhum de aspas duplas, caso contrário a string será delimitada por aspas simples. (O comando `print`, descrito posteriormente, pode ser utilizado para escrever strings sem aspas ou *escape sequences*.)

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

```
>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> '<' + word*5 + '>'
'<HelpAHelpAHelpAHelpAHelpA>'
```

Duas strings literais justapostas são automaticamente concatenadas; a primeira linha do exemplo anterior poderia ter sido escrita como '`word = 'Help''A'`'; isso funciona somente com strings literais, não com expressões arbitrárias:

```
>>> import string
>>> 'str' 'ing'                   #  <-  This is ok
'string'
>>> string.strip('str') + 'ing'   #  <-  This is ok
'string'
>>> string.strip('str') 'ing'     #  <-  This is invalid
  File "<stdin>", line 1
    string.strip('str') 'ing'
                        ^
SyntaxError: invalid syntax
```

Strings podem ser indexadas; como em C, o primeiro índice da string é o 0. Não existe um tipo separado para caracteres; um caracter é simplesmente uma string unitária. Assim como na linguagem Icon, substrings podem ser especificadas através da notação *slice* (N.d.T: fatiar): dois índices separados por dois pontos.

```
>>> word[4]
'A'
>>> word[0:2]
'He'
>>> word[2:4]
```

```
'lp'
```

Diferentemente de C, strings não podem ser alteradas em Python. Atribuir para uma posição (índice) dentro de uma string resultará em erro:

```
>>> word[0] = 'x'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
>>> word[:1] = 'Splat'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support slice assignment
```

Entretanto, criar uma nova string com o conteúdo combinado é fácil e eficiente:

```
>>> 'x' + word[1:]
'xelpA'
>>> 'Splat' + word[4]
'SplatA'
```

Índices de slice possuem valores default úteis; o primeiro índice se omitido possui valor default 0, o segundo possui valor default igual ao tamanho da string.

```
>>> word[:2]    # Os dois primeiros caracteres
'He'
>>> word[2:]    # Todos menos os dois primeiros caracteres
'lpA'
```

Aqui está um invariante interessante relacionado a operações de slice: `s[:i] + s[i:]` equals `s`.

```
>>> word[:2] + word[2:]
'HelpA'
>>> word[:3] + word[3:]
'HelpA'
```

Índices de slice degenerados são tratados "graciosamente" (N.d.T: este termo indica robustez no tratamento de erros): um índice muito maior que o comprimento é trocado pelo comprimento, um limitante superior que seja menor que o limitante inferior produz uma string vazia como resultado.

```
>>> word[1:100]
'elpA'
>>> word[10:]
''
>>> word[2:1]
''
```

Índices podem ser números negativos, para iniciar a contagem a partir da direita ao invés da esquerda. Por exemplo:

```
>>> word[-1]     # O útlimo caracter
'A'
>>> word[-2]     # O penúltimo caracter
'p'
>>> word[-2:]    # Os dois últimos caracteres
'pA'
>>> word[:-2]    # Todos menos os dois últimos caracteres
'Hel'
```

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

```
>>> word[-0]     # ( -0 ==  0)
'H'
```

Intervalos fora dos limites da string são truncados, mas não tente isso em indexações com um único índice (que não seja um slice):

```
>>> word[-100:]
'HelpA'
>>> word[-10]    # error
Traceback (most recent call last):
  File "<stdin>", line 1
IndexError: string index out of range
```

A melhor maneira de lembrar como slices funcionam é pensar nos índices como ponteiros para os espaços entre caracteres, onde a beirada esquerda do primeiro caracter é 0. Logo a beirada direita do último caracter de uma string de comprimento *n* tem índice *n*, por exemplo:

```
 +---+---+---+---+---+
 | H | e | l | p | A |
 +---+---+---+---+---+
 0   1   2   3   4   5
-5  -4  -3  -2  -1
```

A primeira fileira de números indica a posição dos índices 0..5 na string; a segunda fileira indica a posição dos respectivos índices negativos. Um slice de *i* até *j* consiste em todos os caracteres entre as beiradas *i* e *j*, respectivamente.

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

A função interna (N.d.T: interna == *built-in*) `len()` devolve o comprimento de uma string:

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

3.1.3 Strings Unicode

A partir de Python 2.0 um novo tipo foi introduzido: o objeto Unicode. Ele pode ser usado para armazenar e manipular dados Unicode (veja http://www.unicode.org) e se integra bem aos demais objetos strings pré-existentes de forma a realizar auto-conversões quando necessário.

Unicode tem a vantagem de prover um único número ordinal para cada caracter usado em textos modernos ou antigos. Previamente, havia somente 256 números ordinais. Logo, mapeamentos entre conjuntos de caracteres e os 256 números ordinais precisavam ser indexados por códigos de página. Isso levou a uma enorme confusão especialmente no âmbito da internacionalização (tipicamente escrito como '`i18n`' – '`i`' + 18 caracteres + '`n`') de software. Unicode resolve esses problemas ao definir um único código de página 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 pequeno '`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 Python *Unicode-Escape*.

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

```
u’Hello World !’
```

O código de escape `\u0020` indica a inserção do caracter Unicode com valor ordinal 0x0020 (o espaço em branco) na posição determinada.

Os outros caracteres são interpretados através de seus respectivos valores ordinais diretamente para os valores ordinais em Unicode. Se você possui strings literais na codificação padrão Latin-1 que é utilizada na maioria do oeste europeu, você achará conveniente que os 256 caracteres inferiores do Unicode coincidem com os 256 caracteres inferiores do Latin-1.

Para experts, existe ainda um modo cru (N.d.T: sem processamento de caracteres escape) da mesma forma que existe para strings normais. Basta prefixar a string com ’ur’ para utilizar a codificação Python *Raw-Unicode-Escape*. Só será aplicado a conversão `\uXXXX` 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 cru (N.d.T: *raw*) é muito útil para evitar excesso de barras invertidas em expressões regulares.

Além dessas codificações padrão, Python oferece um outro conjunto de maneiras de se criar strings Unicode sobre uma codificação padrão.

A função interna `unicode()` provê acesso a todos os Unicode codecs registrados (COders and DECoders).

Alguns dos mais conhecidos codecs são : *Latin-1*, *ASCII*, *UTF-8*, and *UTF-16*. Os dois últimos são codificações de tamanho variável para armazenar cada caracter Unicode em um ou mais bytes. 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 impressa, escrita em arquivo ou convertida por `str()`, a 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 ?
UnicodeError: ASCII encoding error: ordinal not in range(128)
```

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

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

Também pode ser utilizada a função `unicode()` para efetuar a converção de um string em outra codificação. Neste caso, o primeiro parâmetro é a string a ser convertida e o segundo o nome da codificação almejada. O valor de retorno da função é a string na nova codificação.

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

3.1.4 Listas

Python possui diversas estruturas de dados nativas, utilizadas para agrupar outros valores. A mais versátil delas é a lista (*list*), que pode ser escrita como uma lista de valores separados por vírgula e entre colchetes. Mais importante, os valores contidos na lista não precisam ser do mesmo tipo.

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

Da mesma forma que índices de string, índices de lista começam do 0, listas também podem ser concatenadas e sofrer o operador de *slice*.

```
>>> 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] + ['Boe!']
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam', 'eggs', 100, 'Boe!']
```

Diferentemente de strings, que são imutáveis, é possível mudar elementos individuais da lista:

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

Atribuição à fatias (*slices*) é possível, e isso pode até alterar o tamanho da lista:

```
>>> # Replace some items:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # Remove some:
... a[0:2] = []
>>> a
[123, 1234]
>>> # Insert some:
... a[1:1] = ['bletch', 'xyzzy']
>>> a
[123, 'bletch', 'xyzzy', 1234]
>>> a[:0] = a     # Insert (a copy of) itself at the beginning
>>> a
[123, 'bletch', 'xyzzy', 1234, 123, 'bletch', 'xyzzy', 1234]
```

A função interna `len()` também se aplica a lista:

```
>>> len(a)
```

```
8
```

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')     # See section 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 do objeto.

3.2 Primeiros Passos em Direção à Programação

Naturalmente, nós podemos utilizar Python para tarefas mais complicadas do que somar dois números. A título de exemplificação, nós podemos escrever o início da sequência de *Fibonacci* assim:

```
>>> # Serie de Fibonacci :
... # A soma de dois elementos define o proximo
... 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` simultaneamente recebem 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 inteiro não nulo é considerado verdadeiro (valor *true*), zero tem valor *false*. A condição pode ser ainda uma lista ou string, na verdade qualquer sequência; qualquer coisa com comprimento não nulo tem valor *true* e sequências vazias tem valor *false*. O teste utilizado no exemplo é uma simples comparação. Os operadores padrão para comparação são os mesmos de C: `<` (menor que), `>` (maior que), `==` (igual), `<=` (menor ou igual), `>=` (maior ou igual) and `!=` (diferente).
- O corpo do laço é identado: identação em Python é a maneira de agrupar comandos. Python (ainda!) não possui facilidades automáticas de edição de linha. Na prática você irá preparar scripts Python complexos em um

editor de texto; a maioria dos editores de texto possui facilidades de identação automática. Quando comandos compostos forem alimentados ao interpretador interativamente, devem ser encerrados 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 possuir a mesma identação.

- O comando `print` escreve o valor da expressão dada. Ele difere de apenas escrever a expressão no interpretador (como foi feito no exemplo da calculadora) ao aceitar múltiplas expressões e strings. Strings são impressas sem aspas, um espaço é inserido entre itens de forma a formatar o resultado assim:

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

Mais Ferramentas de Controle de Fluxo

Além do `while` recém apresentado, Python possui as estruturas usuais de controle de fluxo conhecidas em outras linguagens, com algumas variações:

4.1 Construção `if`

Provavelmente uma das mais conhecidas construções é o `if`. Por exemplo:

```
>>> x = int(raw_input("Por favor entre com um numero: "))
>>> if x < 0:
...      x = 0
...      print 'Negativo mudou para zero'
... elif x == 0:
...      print 'Zero'
... elif x == 1:
...      print 'Um'
... else:
...      print 'Mais'
...
```

Pode haver zero ou mais seções `elif`. A seção `else` é opcional. A palavra-chave '`elif`' é uma abreviação para 'else if', e é útil para evitar identação excessiva. Uma sequência `if` ... `elif` ... `elif` ... substitui as construções `switch` e `case` encontradas em outras linguagens.

4.2 Construção `for`

A construção `for` em Python difere um pouco do que se está acostumado em C ou Pascal. Ao invés de se iterar sobre progressões aritiméticas (como em Pascal), ou fornecer ao usuário a habilidade de definir tanto o passo da iteração quanto a condição de parada (como em C), o `for` de Python itera sobre os itens de uma sequência (ex.: uma lista ou uma string), na ordem em que aparecem na sequência. Por exemplo :

```
>>> # Medindo 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 só pode acontecer se a sequência for mutável, isto é, uma lista). Se você precisar modificar a lista sobre a qual se está iterando, por exemplo, para duplicar itens selecionados, você deve iterar sobre uma cópia da lista ao invés da própria. A notação *slice* é particularmente conveniente para isso:

```
>>> for x in a[:]: # faz 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 interna `range()` é a resposta. Ela gera listas contendo progressões aritiméticas, ex.:

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

O ponto de parada fornecido nunca é gerado 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 progresão (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
```

4.4 Cláusulas `break`, `continue` e `else` em Laços

O `break` , como no C, quebra o laço mais interno de um `for` ou `while`.

O `continue` , também emprestado do C, continua o próximo passo 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 encerrado 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, 'equals', x, '*', n/x
...            break
...     else:
...          print n, 'is a prime number'
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

4.5 Construção `pass`

A construção `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 1:
...       pass # Busy-wait para interrupção de teclado
...
```

4.6 Definindo Funções

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

```
>>> def fib(n):    # escreve  a serie de Fibonacci ate n
...     "Print a Fibonacci series up to n"
...     a, b = 0, 1
...     while b < n:
...         print b,
...         a, b = b, a+b
...
>>> # Agora invoca a funçao que acabamos de definir
... fib(2000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

A palavra-reservada `def` serve para definir uma função. Ela deve ser seguida do nome da função, da lista formal de parâmetros entre parênteses e dois pontos.

O corpo da função deve começar na linha seguinte e deve ser identado. Opcionalmente, a primeira linha do corpo da função pode ser uma string literal, cujo propósito é documentar a função. Se presente, essa string chama-se *docstring*.

Existem ferramentas que utilizam docstrings para produzir automaticamente documentação impressa, on-line, ou ainda permitir que o usuário navegue interativamente pelo código. É uma boa prárica incluir sempre docstrings em suas funções, portanto, tente fazer disto um hábito.

A execução da função gera uma nova tabela de símbolos utilizada 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

símbolos interna (*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 chamada, portanto, argumentos são passados por valor (onde valor é sempre uma referência para objeto, não o valor do objeto)[1] Quando uma função chama outra, uma nova tabela de símbolos é criada para tal chamada.

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

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

Você pode afirmar que `fib` não é uma função, mas um procedimento. Em Python, assim como em C, procedimentos são apenas funções que não retornam valores. Na verdade, falando tecnicamente, procedimentos retornam um valor, ainda que meio chato. Esse valor é chamado `None` (é um nome interno). A escrita do valor None é supressa pelo interpretador se ele estiver sozinho. Você pode verificar isso se quiser.

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

É muito simples escrever uma função que retorna a lista da série de Fibonacci, ao invés de imprimi-la:

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

Este exemplo, como sempre, demonstra algumas características novas:

- A palavra-chave `return` termina a função retornando um valor. Se `return` não for seguido de nada, então retorna o valor `None`. Se a função chegar ao fim sem o uso explícito do `return`, então também será retornado o valor `None`.
- O trecho `result.append(b)` chama um método do objeto lista `result`. Um método é uma função que pertence a um objeto e é chamada através de `obj.methodname`, onde `obj` é um objeto qualquer, e `methodname` é o nome de um método que foi definido pelo tipo do objeto. Tipos diferentes definem métodos diferentes. Sobretudo, métodos de diferentes tipos podem ser homônimos sem ambiguidade (é possível definir seus próprios tipos de objetos e métodos, utilizando *classes*, como será discutido mais tarde neste tutorial).

[1] De fato, passagem por referência para objeto seria uma melhor descrição, pois quando um objeto mutável for passado, o *chamador* irá perceber as alterações feitas pelo *chamado* (ex: inserção de itens em uma lista).

O método `append()` mostrado no exemplo é definido para todos objetos do tipo lista. Este método permite a adição de novos elementos à lista. Neste exemplo, ele é equivalente a '`result = result + [b]`', só que `append()` ainda é mais eficiente.

4.7 Mais sobre Definição de Funções

Ainda 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 argumentos. Isso cria uma função que pode ser invocada com um número menor de argumentos do que quando foi definida, ex:

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

Essa função pode ser chamada de duas formas: `ask_ok('Do you really want to quit?')` ou como `ask_ok('OK to overwrite the file?', 2)`.

Os valores default são avaliados durante a definição da função, e no escopo em que a função foi definida:

```
i = 5
def f(arg = i): print arg
i = 6
f()
```

irá imprimir `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. 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)
```

Isso irá imprimir:

```
[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 Parâmetros na Forma Chave-Valor

Funções também podem ser chamadas passando argumentos no formato chave-valor como '*keyword* = *value*'. Por exemplo:

```
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, "!"
```

poderia ser chamada em qualquer uma das seguintes maneiras:

```
parrot(1000)
parrot(action = 'VOOOOOM', voltage = 1000000)
parrot('a thousand', state = 'pushing up the daisies')
parrot('a million', 'bereft of life', 'jump')
```

porém, existem maneiras inválidas:

```
parrot()                     # parâmetro exigido faltando
parrot(voltage=5.0, 'dead')  # parâmetro não-chave-valor depois de parâmetro chave-valor
parrot(110, voltage=220)     # valor duplicado para mesmo parâmetro
parrot(actor='John Cleese')  # parâmetro desconhecido
```

Em geral, uma lista de argumentos tem que apresentar todos argumentos posicionais antes de qualquer um dos seus argumentos chave-valor, onde as chaves têm que ser escolhidas a partir dos nomes formais dos argumentos. Não é importante se um dado argumento já possuia valor default ou não. Nenhum argumento deve receber um valor mais do que uma única vez. Nomes de parâmetros formais correspondendo a argumentos posicionais não podem ser usados na forma chave-valor em uma mesma chamada. O próximo exemplo ilustra essa limitação.

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: keyword parameter redefined
```

Quando o último parâmetro formal for ***name*, ele armazenará todos os parâmetros efetivamente passados para a função e que não correspondiam a parâmetros formais. Isto pode ser combinado com o parâmetro formal **name* (descrito na próxima sub-seção) que recebe a lista contendo todos argumentos posicionais que não correspondiam a parâmetros formais. O importante é que (**name* deve ser declarado antes de ***name*.) Siga o exemplo:

```
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
```

```
    for kw in keywords.keys(): print kw, ':', keywords[kw]
```

Poderia ser chamado assim:

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

4.7.3 Listas Arbitrárias de Argumentos

Finalmente, a opção menos frequentemente usada é chamar a função com um número arbitrário de argumentos. Esses argumentos serão encapsulados em uma sequência (tupla). Antes do número variável de argumentos, zero ou mais argumentos normais podem estar presentes.

```
def fprintf(file, format, *args):
    file.write(format % args)
```

4.7.4 Funções Lambda

Dada a demanda do público, algumas características encontradas em linguagens de programação funcionais (como Lisp) foram adicionadas a Python. Com a palavra-chave `lambda`, funções curtas e anônimas podem ser criadas.

Aqui está uma função que devolve a soma de seus dois argumentos: '`lambda a, b: a+b`'. Funções Lambda podem ser utilizadas em qualquer lugar que exigiria uma função tradicional. Sintaticamente, funções Lambda estão restritas a uma única expressão. Semanticamente, ela são apenas açúcar sintático para a definição de funções normais. Assim como definições de funções aninhadas, funções lambda não podem referenciar variáveis de um escopo mais externo, o que pode ser contornado pelo parcimonioso uso de argumentos com valores default, ex:

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

4.7.5 Strings de Documentação

Há uma convenção sobre o conteúdo e formato de strings de documentação.

A primeira linha deve ser sempre curta, representando um consiso sumário do propósito do objeto. Por brevidade, não deve explicitamente se referir ao nome ou tipo do objeto, uma vez que estas informações estão disponívies 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 múltiplas linhas na string de documentação, a segunda linha deve estar em branco, visulamente separando o sumário 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 toca na identação de comentários multi-linha. Portanto, ferramentas que processem strings de documentação precisam lidar com isso (se desejado). Existe uma convenção para isso. A primeira linha não nula após a linha de sumário determina a identação para o resto da string de documentação. A partir daí, espaços em branco podem ser removidos de todas as linhas da string.

Aqui está um exemplo de uma docstring multi-linha:

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

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

Estruturas de Dados

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

5.1 Mais sobre Listas

O tipo *list* possui mais métodos. Aqui estão todos os métodos disponívies em um objeto lista.

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

`extend(L)` Extende a lista adicionando no fim todos os elementos da lista passada como parâmetro; equivalente a `a[len(a):] = L`.

`insert(i, x)` Insere um item em uma posição especificada. O primeiro argumento é o índice do elemento anterior ao que está para ser inserido, assim `a.insert(0, x)` insere no início da lista, e `a.insert(len(a), x)` é equivalente a `a.append(x)`.

`remove(x)` Remove o primeiro item da lista cujo valor é `x`. É gerado um erro se este valor não existir.

`pop([i])` Remove o item na posição dada e o retorna . Se nenhum item for especificado, `a.pop()` remove e retorna o último item na lista.

`index(x)` Retorna o índice do primeiro item cujo valor é igual ao argumento fornecido em `x`, gerando erro se este valor não existe.

`count(x)` Retorna o número de vezes que o valor `x` aparece na lista.

`sort()` Ordena os itens da lista sem gerar uma nova lista.

`reverse()` Inverte a ordem dos elementos na lista sem gerar uma nova lista.

Um exemplo que utiliza a maioria dos métodos:

```
>>> a = [66.6, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.6), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.6, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
```

```
>>> a
[66.6, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.6]
>>> a.sort()
>>> a
[-1, 1, 66.6, 333, 333, 1234.5]
```

5.1.1 Usando Listas como Pilhas

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

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

5.1.2 Usando Listas como Filas

Você pode também utilizar uma lista como uma fila, onde o primeiro item adicionado é o primeiro a ser recuperado (política "primeiro a entrar, primeiro a sair"). Para adicionar um elemento ao fim da fila utiliza `append()`. Para recuperar um elemento do início da fila use `pop()` com `0` no índice. Por exemplo:

```
>>> queue = ["Eric", "John", "Michael"]
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")          # Graham arrives
>>> queue.pop(0)
'Eric'
>>> queue.pop(0)
'John'
>>> queue
['Michael', 'Terry', 'Graham']
```

5.1.3 Ferramentas para Programação Funcional

Existem três funções internas que são muito úteis sobre listas: `filter()`, `map()`, e `reduce()`.

'`filter(`*function*`,` *sequence*`)`' retorna uma sequência (do mesmo tipo se possível) consistindo dos itens pertencentes a sequência para os quais *function*`(`*item*`)` é verdadeiro. Por exemplo, para computar números primos:

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

'`map(`*function*`,` *sequence*`)`' aplica *function*`(`*item*`)` para cada item da sequência e retorna a lista de valores retornados a cada aplicação. Por exemplo, para computar quadrados:

```
>>> def cube(x): return x*x*x
...
>>> map(cube, 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 possuir tantos parâmetros formais quantas sequências forem alimentadas para '`map`'. Se `None` for passado no lugar da função, então será aplicada uma função que apenas devolve os argumentos recebidos. Dessa forma, '`map(None,` *list1*`,` *list2*`)`' é uma forma conveniente de concatenar listas em uma única. Por exemplo:

```
>>> seq = range(8)
>>> def square(x): return x*x
...
>>> map(None, seq, map(square, seq))
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36), (7, 49)]
```

'`reduce(`*func*`,` *sequence*`)`' retorna um único valor construído a partir da sucessiva aplicação da função binária *func* a todos os elementos da lista fornecida, dois de cada vez. Por exemplo, para computar a soma dos 10 primeiros números inteiros:

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

Se apenas houver um único elemento na sequência fornecida como parâmetro, então seu valor será retornado. Se a sequência for vazia uma exceção será levantada.

Um terceiro argumento pode ser passado para indicar o valor inicial. Neste caso, redução de sequências vazias retornará o valor inicial. Se a sequência não for vazia, a redução se iniciará a partir do valor inicial.

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

5.1.4 Abrangência de Lista (*List Comprehensions*)

Abrangência de listas (ou *list comprehensions*) permitem a criação de listas de forma concisa sem apelar para o uso de `map()`, `filter()` e/ou `lambda`. A definição resultante tende a ser mais clara do que o uso das construções funcionais citadas anteriormente.

Cada abrangência de lista consiste numa expressão seguida da cláusula `for`, e então zero ou mais cláusulas `for` ou `if`. O resultado será uma lista proveniente da avaliação da expressão no contexto das cláusulas `for` e `if` subsequentes. Se a expressão gerar uma tupla, a mesma deve ser inserida entre parênteses.

```
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
>>> [{x: x**2} for x in vec]
[{2: 4}, {4: 16}, {6: 36}]
>>> [[x,x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
>>> [x, x**2 for x in vec] # erro - parenteses requerido para tuplas
  File "<stdin>", line 1
    [x, x**2 for x in vec]
               ^
SyntaxError: invalid syntax
>>> [(x, x**2) for x in vec]
[(2, 4), (4, 16), (6, 36)]
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>> [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
```

5.2 O comando `del`

Existe uma maneira de remover um item de uma lista a partir de seu índice, ao invés de seu valor: o comando `del`. Ele também pode ser utilizado para remover fatias (*slices*) da lista. Por exemplo:

```
>>> a
[-1, 1, 66.6, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.6, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.6, 1234.5]
```

`del` também pode ser utilizado para apagar variáveis:

```
>>> del a
```

Referenciar a variável `a` posteriormente a sua remoção constitui erro (pelo menos até que seja feita uma nova atribuição para ela). Nós encontraremos outros usos para o comando `del` mais tarde.

5.3 Tuplas e Sequências

Nós vimos que listas e strings possuem muitas propriedades em comum como indexação e operações de *slicing*. Elas são dois dos exemplos possíveis de sequências. Como Python é uma linguagem em evolução, outros tipos de sequências podem ser adicionados. Existe ainda um outro tipo de sequência já presente na linguagem: a tupla (*tuple*).

Uma tupla consiste em uma sequência imutável de valores separados por vírgulas.

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuplas podem ser aninhadas:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

Como você pode ver no trecho acima, tuplas são sempre envolvidas por parênteses. Na criação, tuplas podem ser envolvidas ou não por parênteses, desde que o contexto não exija os parênteses (como no caso da tupla pertencer a uma expressão maior).

Tuplas podem ter os mais diversos usos: pares ordenados, registros de empregados em 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). 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 tem certos truques para acomodar estes casos. Tuplas vazias são construídas por uma par de parênteses vazios. E uma tupla unitária é construída por um único valor e uma vírgula entre parênteses (sem a vírgula a tupla não será gerada!). Feio, mas efetivo:

```
>>> empty = ()
>>> singleton = 'hello',    # <-- observe a vírgula extra
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

O comando `t = 12345, 54321, 'hello!'` é um exemplo de empacotamento em tupla (*tuple packing*): os valores `12345`, `54321` e `'hello!'` 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*), e requer que a lista de variáveis do lado esquerdo corresponda ao comprimento da sequência à direita. Sendo assim, a atribuição múltipla é um caso de empacotamento e desempacotamento de tupla.

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

5.4 Dicionários

Outra estrutura de dados interna de Python, e muito útil, é o dicionário. Dicionários são também chamados de "memória associativa", ou "vetor associativo" . Diferentemente 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 são mutáveis.

O melhor modelo mental de um dicionário é um conjunto não ordenado de pares chave-valor, onde as chaves são únicas em uma dada instância do dicionário.

Dicionários são delimitados por : `{}`. Uma lista de pares *chave:valor* separada por vírgulas dentro desse delimitadores define a constituição inicial do dicionário. Dessa forma também será impresso o conteúdo de um dicionário em uma seção de depuração.

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 dada uma chave inexistente será gerado um erro.

O método `keys()` do dicionário retorna a lista de todas as chaves presentes no dicionário, em ordem aleatória (se desejar ordená-las basta aplicar o método `sort()` na lista devolvida). Para verificar a existência de uma chave, utilize o método `has_key()` do dicionário.

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']
>>> tel.has_key('guido')
1
```

5.5 Mais sobre Condições

As condições de controle utilizadas no `while` e `if` acima podem conter outros operadores além de 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 objetos; o que só é significativo no contexto de objetos mutáveis, como listas. Todos operadores de comparação possuem a mesma precedência, que é menor do que a prioridade dos operadores numéricos.

Comparações podem ser encadeadas: `a < b == c` testa se `a` é menor que `b` e ainda por cima se `b` é igual a `c`.

Comparações podem ser combinadas através de operadores booleanos `and` e `or`, e negados 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` and `or` são também operadores *atalhos* : seus argumentos são avaliados da esquerda para a direita, e a avaliação pára quando o resultado se torna conhecido. Ex., se `A` e `C` são verdadeiros mas `B` é falso, então `A and B and C` não avaliará a expressão `C`. Em geral, o valor de retorno de um operador atalho, quando usado sobre valores genéricos e não como booleanos, 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, diferentemente 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 `==`.

5.6 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 *léxicografica* : 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 léxicografica. 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 dita menor (operador ¡). A comparação lexicográfica 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)
```

Observe que é permitido comparar objetos de diferentes tipos. O resultado é determinístico, porém, arbitrário: os tipos são ordenados pelos seus nomes. Então, uma lista é sempre menor do que uma string, uma string é sempre menor do que uma tupla, etc. Tipos numéricos mistos são comparados de acordo com seus valores numéricos, logo 0 é igual a 0.0, etc.[1]

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

Módulos

Se você sair do interpretador Python e entrar novamente, todas as definições de funções e variáveis serão perdidas. Logo, se você desejar escrever um programa que dure é melhor preparar o código em um editor de textos. Quando estiver pronto, dispare o interpretador sobre o arquivo-fonte gerado. Isto se chama gerar um *script*.

A medida que seus programas crescem, pode ser desejável dividi-los em vários arquivos para facilitar a manutenção. Talvez você até queira reutilizar uma função sem copiar sua definição a cada novo programa.

Para permitir isto, Python possui uma maneira de depositar definições em um arquivo e posteriormente reutilizá-las em um script ou seção interativa do interpretador. Esse arquivo é denominado módulo. Definições de um módulo podem ser importadas por outros módulos ou no módulo principal.

Um módulo é um arquivo contendo definições e comandos Python. O nome do arquivo recebe o sufixo '.py'. Dentro de um módulo, seu nome (uma string) está disponível na variável global `__name__`. Por exemplo, use seu editor de textos favorito para criar um arquivo chamado 'fibo.py' no diretório corrente com o seguinte conteúdo:

```
# Módulo Sequências de Fibonacci

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

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

Agora inicie o interpretador e importe o módulo da seguinte forma:

```
>>> import fibo
```

Isso não incorpora as funções definidas em `fibo` diretamente na tabela de símbolos corrente, apenas coloca o nome do módulo lá. Através do 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 você pretende utilizar uma função frequentemente, é possível atribuir a ela 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 como definições. Os comandos servem ao propósito de inicializar o módulo, sendo executados apenas na primeira vez em que o mesmo é importado.[1]

Cada módulo possui sua própria tabela de símbolos, que é usada como tabela de símbolos global por todas as funções definidas no próprio módulo. Portanto, o autor do módulo pode utilizar variáveis globais no módulo sem se preocupar com colisão de nomes acidental com as variáveis globais de usuário.

Por outro lado, se você sabe o que está fazendo, é possível o acesso as variáveis globais do módulo através da mesma notação. O que permite o acesso às funções do módulo: `modname.itemname`.

Módulos podem ser importados por outros módulos. É costume, porém não obrigatório, colocar todos os comandos de importação (`import`) no início do módulo (ou script, se preferir).

Existe uma variante do comando `import` statement que importa nomes de um dado módulo diretamente para a tabela do módulo importador. Os nomes do módulo importado são adicionados a tabela de símbolos global 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 introduz o nome do módulo importado na tabela de símbolos local, mas sim o nome da função diretamente.

Existe ainda uma variante que permite importar diretamente todos os nomes definidos em um dado módulo.

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

Isso importa todos os nomes exceto aqueles iniciados por um sublinhado (_).

6.1.1 O Caminho de Busca dos Módulos

Quando um módulo denominado `spam` é importado, o interpretador busca por um arquivo chamado ‘spam.py’ no diretório corrente, depois na lista de diretórios especificados pela variável de ambiente PYTHONPATH. Esta última possui a mesma sintaxe da variável de ambiente PATH, isto é, uma lista de caminhos. Quando PYTHONPATH não existir, ou o arquivo não for achado nesta lista, a busca continua num caminho que depende da instalação. No caso do UNIX esse caminho é quase sempre ‘.:/usr/local/lib/python’.

De fato, módulos são buscados pela lista de caminhos especificados na variável `sys.path` inicializada com os caminhos citados acima, o que permite aos programas Python manipularem o processo de busca de módulos se desejado. Veja a seção Módulos Padrão para obter mais detalhes.

[1] Na verdade, definições de funções são também “comandos” que são “executados”. A execução desses comandos é colocar o nome da função na tabela de símbolos global do módulo.

6.1.2 Arquivos Python “Compilados”

Um fator que agiliza a carga de programas curtos que utilizam muitos módulos padrão é a existência de um arquivo com extensão ‘.pyc’ no mesmo diretório do fonte ‘.py’. O arquivo ‘.pyc’ contém uma versão “byte-compilada” do fonte ‘.py’. A data de modificação de ‘.py’ é armazenada dentro do ‘.pyc’, e verificada automaticamente antes da utilização do último. Se não conferir, o arquivo ‘.pyc’ existente é re-compilado a partir do ‘.py’ mais atual.

Normalmente, não é preciso fazer nada para gerar o arquivo ‘.pyc’. Sempre que um módulo ‘.py’ é compilado com sucesso, é feita uma tentativa de se escrever sua versão compilada para o ‘.pyc’. Não há geração de erro se essa tentativa falha. Se por qualquer razão o arquivo compilado não é inteiramente escrito em disco, o ‘.pyc’ resultante será reconhecido como inválido e, portanto, ignorado. O conteúdo do ‘.pyc’ é independente de plataforma, assim um diretório de módulos Python pode ser compartilhado por diferentes arquiteturas.

Algumas dicas dos experts:

- Quando o interpretador Python é invocado com a diretiva **-O**, código otimizado é gerado e armazenado em arquivos ‘.pyo’. O otimizador corrente não faz muita coisa, ele apenas remove construções `assert` e instruções `SET_LINENO`. Quando o **-O** é utilizado, *todo* bytecode é otimizado. Arquivos `.pyc` são ignorados e arquivos `.py` são compilados em bytecode otimizado.
- Passando dois flags **-O** ao interpretador (**-OO**) irá forçar o compilador de bytecode a efetuar otimizações arriscadas que poderiam em casos raros acarretar o mal funcionamento de programas. Presentemente, apenas strings `__doc__` são removidas do bytecode, proporcionando arquivos ‘.pyo’ mais compactos. Uma vez que alguns programas podem supor a existência das docstrings, é melhor você só se utilizar disto se tiver segurança de que não acarretará nenhum efeito colateral negativo.
- Um programa não executa mais rápido quando é lido de um arquivo ‘.pyc’ ou de um ‘.pyo’ em comparação a quando é lido de um ‘.py’. A única diferença é que nos dois primeiros casos o tempo de carga do programa é menor.
- Quando um script é executado diretamente a partir de seu nome da linha de comando, não são geradas as formas compiladas deste script em arquivos ‘.pyo’ ou ‘.pyc’. Portanto, o tempo de carga de um script pode ser melhorado se transportarmos a maioria de seu código para um módulo e utilizarmos outro script apenas para o disparo. É possível disparar o interpretador diretamente sobre arquivos compilados.
- Na presença das formas compiladas (‘.pyc’ e‘.pyo’) de um script, não há necessidade da presença da forma textual(‘.py’). Isto é útil na hora de se distribuir bibliotecas Python dificultando práticas de engenharia reversa.
- O módulo `compileall` pode criar arquivos ‘.pyc’ (ou ‘.pyo’ quando é usado **-O**) para todos os módulos em um dado diretório.

6.2 Módulos Padrão

Python possui um 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, seja por eficiência ou para permitir o acesso a chamadas de sistema. O conjunto presente destes módulos é configurável, por exemplo, o módulo `amoeba` só está disponível em sistemas que suportam as primitivas do Amoeba. Existe um módulo que requer especial atenção: `sys`, que é embutido em qualquer interpretador Python. As variáveis `sys.ps1` e `sys.ps2` definem as strings utilizadas como prompt primário e secundário:

```
>>> import sys
>>> sys.ps1
'>>> '
```

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

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

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

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

6.3 A Função `dir()`

A função interna `dir()` é utilizada para se descobrir que nomes são definidos por um módulo. Ela retorna uma lista ordenada de strings:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__name__', 'argv', 'builtin_module_names', 'copyright', 'exit',
'maxint', 'modules', 'path', 'ps1', 'ps2', 'setprofile', 'settrace',
'stderr', 'stdin', 'stdout', 'version']
```

Sem nenhum argumento, `dir()` lista os nomes correntemente definidos:

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

Observe que ela lista nomes dos mais diversos tipos: variáveis, módulos, funções, etc.

`dir()` não lista nomes de funções ou variáveis internas. Se você desejar conhecê-los, eles estão definidos no módulo padrão `__builtin__`:

```
>>> import __builtin__
>>> dir(__builtin__)
['AccessError', 'AttributeError', 'ConflictError', 'EOFError', 'IOError',
'ImportError', 'IndexError', 'KeyError', 'KeyboardInterrupt',
'MemoryError', 'NameError', 'None', 'OverflowError', 'RuntimeError',
'SyntaxError', 'SystemError', 'SystemExit', 'TypeError', 'ValueError',
'ZeroDivisionError', '__name__', 'abs', 'apply', 'chr', 'cmp', 'coerce',
'compile', 'dir', 'divmod', 'eval', 'execfile', 'filter', 'float',
'getattr', 'hasattr', 'hash', 'hex', 'id', 'input', 'int', 'len', 'long',
'map', 'max', 'min', 'oct', 'open', 'ord', 'pow', 'range', 'raw_input',
'reduce', 'reload', 'repr', 'round', 'setattr', 'str', 'type', 'xrange']
```

6.4 Pacotes

Pacotes são uma maneira de estruturar espaços de nomes para módulos utilizando a sintaxe de "separação por ponto". Como exemplo, o módulo `A.B` designa um sub-módulo chamado '`B`' num pacote denominado '`A`'. O uso de pacotes permite aos autores de grupos de módulos (como NumPy ou PIL) não terem que se preocupar com colisão entre os nomes de seus módulos e os nomes de módulos de outros autores.

Suponha que você deseje 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, ex. '.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 (ex. mixagem, eco, equalização, efeito stereo artificial). Logo, possivelmente você também estará escrevendo uma interminável coleção de módulos para aplicar estas operações.

Aqui está uma possível estrutura para o seu pacote (expressa em termos de um sistema hierárquico de arquivos):

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

Os arquivos '__init__.py' são necessários para que Python trate os diretórios como um conjunto de módulos. Isso foi feito para evitar diretórios com nomes comuns, como '`string`', de inadvertidamente esconder módulos válidos que ocorram a posteriori 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 gerar a variável `__all__`, que será descrita depois.

Usuários do pacote podem importar módulos individuais, por exemplo:

```
import Sound.Effects.echo
```

Assim se carrega um sub-módulo `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 alternativa para a importação é:

```
from Sound.Effects import echo
```

Assim se carrega o módulo sem necessidade de prefixação na hora do uso. Logo, pode ser utilizado como se segue:

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

Também é possível importar diretamente uma única variável ou função, como em:

```
from Sound.Effects.echo import echofilter
```

Novamente, há carga do sub-módulo `echo`, mas a função `echofilter()` está acessível diretamente sem prefixação:

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

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

Em oposição, na construção `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 entidade 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 todos sub-módulos presentes no pacote fossem importados. Infelizmente, essa operação não funciona muito bem nas plataformas Mac ou Windows, onde não existe distinção entre maiúsculas ou minúsculas nos sistema de arquivos. Nestas plataformas não há como saber como importar o arquivo 'ECHO.PY', deveria ser com o nome `echo`, `Echo` ou `ECHO` (por exemplo, o Windows 95 tem o irritante hábito de colocar a primeira letra em maiúscula). A restrição de nomes de arquivo em DOS com o formato 8+3 adiciona um outro problema na hora de se utilizar arquivos com nomes longos.

A única solução é o autor do pacote fornecer um índice explícito do pacote. O comando de importação utiliza a seguinte convenção: se o arquivo '__init__.py' do pacote define a lista chamada `__all__`, então esta lista indica os nomes dos módulos a serem importados quando o comando `from` *package* `import *` é encontrado.

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 do `from` *package* `import *`. Por exemplo, o arquivo 'Sounds/Effects/__init__.py' poderia conter apenas:

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

Isso significaria que `from Sound.Effects import *` iria importar apenas os três sub-módulos especificados no pacote `Sound`.

Se `__all__` não estiver definido, o comando `from Sound.Effects import *` não importará todos os sub-módulos do pacote `Sound.Effects` no espaço de nomes corrente. Há apenas garantia que o pacote `Sound.Effects` foi importado (possivelmente executando seu código de inicialização '__init__.py') juntamente com os nomes definidos no pacote. Isso inclui todo nome definido em '__init__.py' bem como em qualquer sub-módulo importado a partir deste. Por exemplo:

```
import Sound.Effects.echo
import Sound.Effects.surround
from Sound.Effects import *
```

Neste exemplo, os módulos *echo* e *surround* são importados no espaço de nomes corrente, pois estão definidos no pacote `Sound.Effects`. O que também funciona quando `__all__` estiver definida.

Em geral, a prática de importar tudo de um dado módulo é desaconselhada, principalmente por prejudicar a legibilidade do código. Contudo, é recomendada em sessões interativas para evitar excesso de digitação.

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

6.4.2 Referências em Um Mesmo Pacote

Os sub-módulos 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 corrente (o pacote do qual o módulo corrente é sub-módulo), então o comando `import` procura por um módulo de mesmo nome no escopo global.

Quando pacotes são estruturados em sub-pacotes (como em `Sound`), não existe atalho para referenciar sub-módulos de pacotes irmãos - o nome completo do pacote deve ser utilizado. Por exemplo, se o módulo `Sound.Filters.vocoder` precisa utilizar o módulo `echo` no pacote `Sound.Effects` , é preciso importá-lo como `from Sound.Effects import echo`.

Entrada e Saída

Existem diversas maneiras de se apresentar a saída de um programa. Dados podem ser impressos em forma imediatamente legível, ou escritos em um arquivo para uso futuro. Este capítulo vai discutir algumas das possibilidades.

7.1 Refinando a Formatação de Saída

Até agora nós encontramos duas maneiras de escrever valores: através de *expressões* e pelo comando `print` (uma terceira maneira é utilizar o método `write()` de objetos de arquivo; a saída padrão pode ser referenciada como `sys.stdout`). Veja o documento *Library Reference* para mais informações sobre este tópico.

Frequentemente você desejará mais controle sobre a formatação de saída do que simplesmente imprimindo valores separados por espaços. Existem duas formas. A primeira é você mesmo manipular a string através de recortes (*slicing*) e concatenação. O módulo padrão `string` contém algumas rotinas úteis a esta finalidade. A segunda maneira é utilizar o operador %.

O operador % interpreta seu argumento à esquerda como uma string de formatação de um `sprintf()` aplicada ao argumento à direita do operador. O resultado é uma string formatada.

Permanece a questão: como converter valores para strings? Por sorte, Python possui uma maneira de converter qualquer valor para uma string: basta submetê-lo a função `repr()`, ou simplesmente escrevê-lo entre aspas reversas (‘‘). Alguns exemplos:

```
>>> x = 10 * 3.14
>>> y = 200*200
>>> s = 'The value of x is ' + `x` + ', and y is ' + `y` + '...'
>>> print s
The value of x is 31.4, and y is 40000...
>>> # Aspas reversas tambem funcionam em outros tipos alem de numeros:
... p = [x, y]
>>> ps = repr(p)
>>> ps
'[31.400000000000002, 40000]'
>>> # Converter uma string adiciona barras invertidas e aspas
... hello = 'hello, world\n'
>>> hellos = `hello`
>>> print hellos
'hello, world\n'
>>> # O argumento de aspas reversas pode ser uma tupla
... `x, y, ('spam', 'eggs')`
"(31.400000000000002, 40000, ('spam', 'eggs'))"
```

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

```
>>> import string
>>> for x in range(1, 11):
...     print string.rjust(‘x‘, 2), string.rjust(‘x*x‘, 3),
...     # Observe a vírgula final na linha anterior
...     print string.rjust(‘x*x*x‘, 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 ’%2d %3d %4d’ % (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 os espaços adicionados entre as colunas se devem a forma de funcionamento do comando `print` : ele sempre adiciona espaço entre seus argumentos.

Este exemplo demonstra a função `string.rjust()`, que justifica uma string à direita gerando espaços adicionais à esquerda. Existem funções análogas `string.ljust()` e `string.center()`. Essas funções apenas retornam a string formatada. Se a entrada extrapolar o comprimento exigido a string original é devolvida sem modificação. A razão para isso é não apresentar um valor potencialmente corrompido por truncamento (se for desejado truncar o valor pode-se utilizar operações de recorte como em ‘`string.ljust(x, n)[0:n]`’).

Existe ainda a função `string.zfill()` que preenche uma string numérica com zeros à esquerda. Ela entende sinais positivos e negativos.

```
>>> import string
>>> string.zfill(’12’, 5)
’00012’
>>> string.zfill(’-3.14’, 7)
’-003.14’
>>> string.zfill(’3.14159265359’, 5)
’3.14159265359’
```

Um exemplo de uso do operador `%`:

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

Se há mais do que um formato, então o argumento à direita deve ser uma tupla com os valores de formatação. Exemplo:

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

A maioria dos formatos funciona da mesma maneira que em C, e exigem que você passe o tipo apropriado. Entretanto, em caso de erro ocorre uma exceção e não uma falha do sistema operacional.

O formato `%s` é mais relaxado: se o argumento correspondente não for um objeto string, então ele é convertido para string pela função interna `str()`. Há suporte para o modificador `*` determinar o comprimento ou precisão num argumento inteiro em separado. Os formatadores (em C) `%n` e `%p` também são suportados.

Se você possuir uma string de formatação muito longa, seria bom referenciar as variáveis de formatação *por nome*, ao invés de *por posição*. Isso pode ser obtido passando um dicionário como argumento à direita e prefixando campos na string de formatação com `%(name)format`. Veja o exermplo:

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

Isso é particularmente útil em combinação com a nova função interna `vars()`, que retorna um dicionário contendo todas as variáveis locais.

7.2 Leitura e Escrita de Arquivos

A função `open()` retorna um objeto de arquivo, e é frequentemente usada com dois argumentos: '`open(`*filename*`, `*mode*`)`'.

```
>>> 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 assumir valor `'r'` quando o arquivo será só de leitura, `'w'` quando for só de escrita (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 e no Macintosh, `'b'` adicionado a string de modo indica que o arquivo será aberto no formato 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 levemente alterados em leituras e escritas. Essa mudança por-trás-do-pano é útil em arquivos texto ASCII, mas irá corromper um arquivo binário como no caso de JPEGs ou executáveis. Seja muito cuidadoso ao manipular arquivos binários (no caso do Macintosh a semântica exata do modo texto depende da biblioteca C subjacente sendo utilizada).

7.2.1 Métodos de Objetos de Arquivos

A título de simplificação, o resto dos exemplos nesta seção irá assumir que o objeto de arquivo chamado `f` já foi criado.

Para ler o conteúdo de um arquivo chame `f.read(`*size*`)`, que lê um punhado de dados retornando-os como string. O argumento numérico *size* é opcional. Quando *size* for omitido ou negativo, todo o conteúdo do arquivo será lido e

retornado. É problema seu se o conteúdo do arquivo é o dobro da memória disponível na máquina. Caso contrário, no máximo *size* bytes serão lidos e retornados. Se o fim do arquivo for atingido, `f.read()` irá retornar uma string vazia ().

```
>>> f.read()
'Esse é todo o conteúdo do arquivo.\n'
>>> f.read()
''
```

`f.readline()` lê uma única linha do arquivo. O caracter de retorno de linha (`\n`) é deixado ao final da string, só sendo omitido na última linha do arquivo se ele já não estiver presente lá. Isso elimina a ambiguidade no valor de retorno. Se `f.readline()` retornar uma string vazia, então o arquivo acabou. Linhas em branco são representadas por `'\n'`: uma string contendo unicamente o terminador de linha.

```
>>> f.readline()
'Essa é a primeira linha do arquivo.\n'
>>> f.readline()
'Segunda linha do arquivo\n'
>>> f.readline()
''
```

`f.readlines()` retorna 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, esta operação é utilizada para ler arquivos muito grandes sem ter que ler todo o arquivo para a memória de uma só vez. Apenas linhas completas serão retornadas.

```
>>> f.readlines()
['Essa é a primeira linha do arquivo.\n', 'Segunda linha do arquivo\n']
```

`f.write(`*string*`)` escreve o conteúdo da *string* para o arquivo, retornando `None`.

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

`f.tell()` retorna um inteiro que indica a posição corrente de leitura ou escrita no arquivo, medida em bytes desde o início do arquivo. Para mudar a posição utilize '`f.seek(`*offset, from_what*`)`'. A nova posição é computada pela soma do *offset* a um ponto de referência, que por sua vez é definido pelo argumento *from_what*. O argumento *from_what* pode assumir o valor 0 para indicar o início do arquivo, 1 para indicar a posição corrente e 2 para indicar o fim do arquivo. Este parâmetro pode ser omitido, quando é assumido o valor default 0.

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

Quando acabar de utilizar o arquivo, chame `f.close()` para fechá-lo e liberar recursos. Qualquer tentativa de acesso ao arquivo depois dele ter sido fechado implicará em falha.

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
```

```
ValueError: I/O operation on closed file
```

Objetos de arquivo podem ter métodos adicionais, tais como `isatty()` e `truncate()` que são usados com menos frequência, consulte a *Library Reference* para obter maiores 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 trabalha com strings. Portanto, pode ser utilizada a função `string.atoi()`, que recebe uma string `'123'` e a converte para o respectivo valor inteiro. Entretanto, quando estruturas de dados mais complexas (listas, dicionários, instâncias de classe,etc) estão envolvidas, o processo se torna mais complicado.

Para que não seja necessário que usuários estejam constantemente escrevendo e depurando código que torna estruturas de dados persistentes, Python oferece o módulo padrão `pickle`. Este módulo permite que praticamente qualquer objeto Python (até mesmo código!) seja convertido para uma representação string. Este processo é denominado *pickling*. E *unpickling* é o processo reverso de reconstruir o objeto a partir de sua representação string. Enquanto estiver representado como uma string, o objeto pode ser armazenado em arquivo, transferido pela rede, etc.

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 desfazer, se `f` for agora um objeto de arquivo pronto 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 *Library Reference* para obter informações detalhadas.

Utilizar o módulo `pickle` é a forma padrão de tornar objetos Python persistentes, permitindo a reutilização dos mesmos entre diferentes programas, ou pelo mesmo programa em diferentes sessões de utilização. A representação string dos dados é tecnicamente chamada *objeto persistente*, como já foi visto. 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 sejam compatíveis com esse processo.

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

O parser repete a linha inválida e apresenta uma pequena 'flecha' apontando para o ponto da linha em que o erro foi encontrado. O erro é detectado pelo token que precede a flecha. No exemplo, o erro foi detectado na palavra-reservada `print`, uma vez que o dois-pontos (`':'`) está faltando. Nome de arquivo e número de linha são impressos 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
ZeroDivisionError: integer division or modulo
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1
NameError: spam
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1
TypeError: illegal argument type for built-in operation
```

A última linha da mensagem de erro indica o que aconteceu. Exceções surgem com diferentes tipos, e o tipo é impresso como parte da mensagem . Os tipos no exemplo são: `ZeroDivisionError`, `NameError` e `TypeError`. A string impressa 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.

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

A parte anterior da mensagem de erro apresenta o contexto onde ocorreu a exceção. Essa informação é denominada *stack backtrace* (N.d.T: rastreamento da pilha para trás). Em geral, contém uma lista de linhas do código fonte, sem apresentar, no entanto, valores lidos da entrada padrão.

O documento *Python Library Reference* 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 1:
...     try:
...         x = int(raw_input("Entre com um número: "))
...         break
...     except ValueError:
...         print "Opa! Esse número não é válido. Tente de novo..."
...
```

A construção `try` funciona da seguinte maneira:

- Primeiramente, a cláusula *try* (o conjunto de comandos entre as palavras-reservadas `try` e `except`) é executado.
- Se não for gerada exceção, a cláusula *except* é ignorada e termina a execução da construção `try`.
- Se uma execução ocorre durante a execução da cláusula `try`, os comandos remanescentes na cláusula são ignorados. Se o tipo da exceção ocorrida tiver sido previsto junto àlguma palavra-reservada `except`, então essa cláusula será executada. Ao fim da cláusula também termina a execução do `try` como um todo.
- Se a exceção ocorrida não foi prevista em nenhum tratador `except` da construção `try` em que ocorreu, então ela é entregue a uma construção `try` mais externa. Se não existir nenhum tratador previsto para tal exceção (chamada *unhandled exception*), a execução encerra com uma mensagem de erro.

A construção `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 as exceções levantadas no interior da cláusula `try`, e não que tenha ocorrido no inteiror de outro tratador num mesmo `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, servindo como uma máscara genérica. Utilize esse recurso com extrema cautela, uma vez que isso pode esconder errors do programador e do usuário. Também pode ser utilizado para imprimir uma mensagem de erro, ou re-levantar (*re-raise*) a exceção de forma que um “chamador” também possa tratá-la.

```
import string, sys

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

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

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

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

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. Para exceções possuidoras de argumento, a cláusula `else` pode especificar uma variável depois da lista de nomes de exceção para receber o argumento. Veja:

```
>>> try:
...     spam()
... except NameError, x:
...     print 'name', x, 'undefined'
...
name spam undefined
```

Se uma exceção possui argumento, ele é impresso ao final do detalhamento de expressões não tratadas (*unhandled exceptions*).

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

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

8.4 Levantando Exceções

A palavra-reservada `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
NameError: HiThere
```

O primeiro argumento de `raise` é o nome da exceção a ser levantada. O segundo argumento, opcional, é o argumento da exceção.

8.5 Exceções Definidas pelo Usuário

Programas podem definir novos tipos de exceções, através da criação de uma nova classe. Por exemplo:

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

Muitos módulos padrão se utilizam disto para reportar erros que ocorrem no interior das funções que definem.

Mais informações sobre esse mecanismo serão descritas no capítulo 9, “Classes.”

8.6 Definindo Ações de Limpeza

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

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

A cláusula *finally* é executada sempre, ocorrendo ou não uma exceção. Quando ocorre a exceção, é como se a exceção fosse sempre levantada após a execução do código na cláusula *finally*. Mesmo que haja um `break` ou `return` dentro do `try`, ainda assim o `finally` será executado.

A construção `try` pode ser seguida da cláusula `finally` ou de um conjunto de cláusulas `except`, mas nunca ambas. Essas opções são mutuamente exclusivas.

Classes

O mecanismo de classes em Python foi adicionado à linguagem de forma a minimizar a sobrecarga sintática e semântica. É uma mistura de mecanismos equivalentes encontrados em C++ e Modula-3. Assim como é válido para módulos, classes em Python não impõe barreiras entre o usuário e a definição. Contudo, dependem da "cordialidade" do usuário para não quebrar a definição. Todavia, as características mais importantes de classes foram asseguradas: o mecanismo de herança permite múltiplas classes base, uma classe derivada pode sobrescrever quaisquer métodos de uma classe ancestral, um método pode invocar outro método homônimo de uma classe ancestral. E mais, objetos podem armazenar uma quantidade arbitrária de dados privados.

Na terminologia de C++, todos os membros de uma classe (incluindo dados) são *public*, e todos as funções membro são *virtual*. Não existem construtores ou destrutores especiais.

Como em Modula-3, não existem atalhos para referenciar membros do objeto de dentro dos seus métodos. Um método (função de classe) é declarado com um primeiro argumento explícito representando o objeto (instância da classe) , que é fornecido implicitamente pela invocação.

Como em Smalltalk (e em Java), classes são objetos. Mas em Python, todos os tipos de dados são objetos. Isso fornece uma semântica para importação e renomeamento. Ainda como em C++ ou Modula-3, tipos pré-definidos não podem ser utilizados como classes base para extensões de usuário por herança. Como em C++, mas diferentemente de Modula-3, a maioria dos operadores (aritiméticos, indexação,etc) podem ser redefinidos para instâncias de classe.

9.1 Uma Palavra Sobre Terminologia

Na falta de uma universalmente aceita terminologia de classes, eu irei ocasionalmente fazer uso de termos comuns a Smalltalk ou C++ (eu usaria termos de Modula-3 já que sua semântica é muito próxima a de Python, mas tenho a impressão de que um número menor de usuários estará familiarizado com esta linguagem).

Eu também preciso alertar para uma armadilha terminológica para leitores familiarizados com orientação a objetos: a palavra "objeto" em Python não necessariamente implica em uma instância de classe. Como em C++ e Modula-3 e, diferentemente de Smalltalk, nem todos os tipos em Python são classes: tipos básicos como inteiros, listas e arquivos não são. Contudo, *todos* tipos de Python compartilham uma semântica comum que é melhor descrita pela palavra objeto.

Objetos tem individualidade, e podem estar vinculados a múltiplos nomes (em diferentes escopos). Essa facilidade é chamada *aliasing* em outras linguagens. À primeira vista não é muito apreciada, e pode ser seguramente ignorada ao lidar com tipos imutáveis (números, strings, tuplas). Entretanto, *aliasing* tem um efeito intencional sobre a semântica de código em Python envolvendo objetos mutáveis como listas, dicionários, e a maioria das entidades externas a um programa como arquivos, janelas, etc. Alias (N.d.T: sinônimos) funcionam de certa forma como ponteiros, em benefício do programamdor. Por exemplo, passagem de objetos como parâmetro é barato, pois só o ponteiro é passado na implementação. E se uma função modifica um objeto passado como argumento, o chamador vai ver a mudança – o que torna obsoleta a necessidade de um duplo mecanismo de passagem de parâmetros como em Pascal.

9.2 Escopos e Espaços de Nomes em Python

Antes de introduzir classes, é preciso falar das regras de escopo em Python. Definições de classe empregam alguns truques com espaços de nomes. Portanto, é preciso entender bem de escopos e espaços de nomes antes. Esse conhecimento é muito útil para o programador avançado em Python.

Iniciando com algumas definições.

Um espaço de nomes é um mapeamento entre nomes e objetos. Presentemente, são implementados como dicionários, 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 exceções), nomes globais em módulos e nomes locais em uma função. De uma certa forma, os atributos de um objeto também formam um espaço de nomes. O que há de importante para saber é 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, eu utilizo a palavra *atributo* para qualquer nome depois de um ponto. Na expressão `z.real`, por exemplo, `real` é um atributo do objeto `z`. Estritamente falando, referências para nomes em módulos são atributos: na expressão `modname.funcname`, `modname` é um objeto módulo e `funcname` é seu atributo. 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 de leitura ou não. Atributos de módulo são passíveis de atribuição, você pode escrever '`modname.the_answer = 42`', e remoção pelo comando `del` ('`del modname.the_answer`').

Espaços de nomes são criados em momentos diferentes e possuem diferentes longevidades. O espaço de nomes que contém os nomes pré-definidos é criado quando o interpretador inicializa e nunca é removido. O espaço de nomes global é criado quando uma definição de módulo é lida, e normalmente duram até a saída do interpretador.

Os comandos executados pela invocação do interpertador, ou pela leitura de um script, ou interativamente são parte do módulo chamado `__main__`, e portanto possuem seu próprio espaço de nomes (os nomes pré-definidos possuem seu próprio espaço de nomes no módulo chamado `__builtin__`).

O espaço de nomes local para uma função é criado quando a função é chamada, e removido quando a função retorna ou levanta uma exceção que não é tratada na própria função. Naturalmente, chamadas recursivas de uma função possuem seus próprios espaços de nomes.

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

Ainda que escopos sejam determinados estaticamente, eles são usados dinamicamente. A qualquer momento durante a execução, existem exatamente três escopos em uso (três escopos diretamente acessíveis): o escopo interno (que é procurado primeiro) contendo nomes locais, o escopo intermediário (com os nomes globais do módulo) e o escopo externo (procurado por último) contendo os nomes pré-definidos.

Normalmente, o escopo local referencia os nomes locais da função corrente. Fora de funções, o escopo local referencia os nomes do escopo global (espaço de nomes do módulo). Definições de classes adicionam um espaço de nomes ao escopo local.

É importante perceber que escopos são determinados textualmente. O escopo global de uma função definida em um módulo é o espaço de nomes deste módulo, sem importar de onde ou de que alias (N.d.T. sinônimo) a função é invocada. Por outro lado, a efetiva busca de nomes é dinâmica, ocorrendo durante a execução. A evolução da linguagem está caminhando para uma resolução de nomes estática, em tempo de compilação, que não dependa de resolução dinâmica de nomes. (de fato, variáveis locais já são resolvidas estaticamente.)

Um detalhe especial é que atribuições são sempre vinculadas ao escopo interno. Atribuições não copiam dados, simplesmente vinculam objetos a nomes. O mesmo é verdade para remoções. O comando '`del x`' remove o vínculo

[1] Exceto por uma coisa. Objetos Módulo possuem um atributo de leitura escondido chamado `__dict__` que retorna o dicionário utilizado para implementar o espaço de nomes do módulo. O nome `__dict__` é um atributo, porém não é um nome global. Obviamente, utilizar isto violaria a abstração de espaço de nomes, portanto seu uso deve ser restrito. Um exemplo válido é o caso de depuradores *post-mortem*.

de `x` do espaço de nomes referenciado pelo escopo local. De fato, todas operações que introduzem novos nomes usam o escopo local. Em particular, comandos `import` e definições de função vinculam o módulo ou a 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 semântica nova.

9.3.1 Sintaxe de Definição de Classe

A forma mais simples de definir uma classe é:

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

Definições de classes, como definições de funções (comandos `def`) devem ser executados antes que tenham qualquer efeito. Você pode colocar uma definição de classe após um teste condicional `if` ou dentro de uma função.

Na prática, os comandos dentro de uma classe serão definições de funções, mas existem outros comandos que são permitidos. Definições de função dentro da classe possuem um lista peculiar de argumentos determinada pela convenção de chamada a métodos.

Quando se fornece uma definição de classe, um novo espaço de nomes é criado. Todas atribuições de variáveis são vinculadas a este escopo local. Em particular, definições de função também são armazenadas neste escopo.

Quando termina o processamento de uma definição de classe (normalmente, sem erros), um objeto de classe é criado. Este objeto encapsula o conteúdo do espaço de nomes criado pela definição da classe. O escopo local ativo antes da definição da classe é restaurado, e o objeto classe é vinculado a este escopo com o nome dado a classe.

9.3.2 Objetos de Classe

Objetos de 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.name`. Atributos válidos são todos os nomes presentes no espaço de nomes da classe quando o objeto classe foi criado. Portanto, se a definição da classe era:

```
class MyClass:
    "Um exemplo simples de classe"
    i = 12345
    def f(x):
        return 'hello world'
```

então `MyClass.i` e `MyClass.f` são referências a atributos válidos, retornando um inteiro e um objeto método, respectivamente. Atributos de classe podem receber atribuições, logo você pode mudar o valor de `MyClass.i` por atribuição. `__doc__` é também um atributo válido, que retorna a docstring pertencente a classe: `"Um exemplo simples de classe"`.

Instanciação de Classe também utiliza uma notação de função. Apenas finja que o objeto classe não possui parâmetros e retorna uma nova instância da classe. Por exemplo:

```
x = MyClass()
```

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

A operação de instanciação ("calling" um objeto de classe) cria um objeto vazio. Muitas classes preferem criar um novo objeto em um estado inicial pré-determinado. Para tanto, existe um método especial que pode ser definido pela classe, `__init__()`, veja:

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

Quando uma classe define um método `__init__()`, o processo de instânciação automáticamente invoca `__init__()` sobre a recém-criada instância de classe. Neste exemplo, uma nova intância já inicializada pode ser obtida por:

```
x = MyClass()
```

Naturalmente, o método `__init__()` pode ter argumentos para aumentar sua flexibilidade. Neste caso, os argumentos passados para a instanciação de classe serão delegados para o método `__init__()`. Como ilustrado em:

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

9.3.3 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.

O primeiro eu chamo de *atributos-de-dados*, que correspondem a "variáveis de instância" em Smalltalk, e a "membros" em C++. Atributos-de-dados não 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 de `MyClass` criada acima, o próximo trecho de código irá imprimir o valor `16`, sem deixar rastro (por causa do `del`):

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

O segundo tipo de referência a atributo entendido por instâncias são *métodos*. Um método é uma função que "pertence a" uma instância (em Python, o termo método não é aplicado exclusivamente a intâncias de classe: outros tipos de objetos também podem ter métodos; por exemplo, objetos listas possuem os métodos: append, insert, remove, sort, etc).

Nomes de métodos válidos de uma instância dependem de sua classe. Por definição, todos atributos de uma classe que são funções equivalem a um método presente nas instâncias. No nosso exemplo, `x.f` é uma referência de método válida já que `MyClass.f` é uma função, enquanto `x.i` não é, já que `MyClass.i` não é função. Entretanto, `x.f` não é o mesmo que `MyClass.f` — o primeiro é um método de objeto , o segundo é um objeto função.

9.3.4 Métodos de Objeto

Normalmente, um método é chamado imediatamente, isto é:

```
x.f()
```

No nosso exemplo o resultado será a string `'hello world'`. No entanto, não é obrigatótio chamar o método imediatamente: como `x.f` é também um objeto (tipo método), ele pode ser armazenado e invocado a posteriori. Por exemplo:

```
xf = x.f
while 1:
    print xf()
```

continuará imprimindo '`hello world`' até o final dos tempos.

O que ocorre precisamente quando um método é chamado ? Você deve ter notado que `x.f()` foi chamado sem nenhum parâmetro, porém a definição da função `f` especificava um argumento. O que aconteceu com o argumento ? Certamente Python levantaria uma exceção se o argumento estivesse faltando...

Talvez você já tenha advinhado a resposta: o que há de especial em métodos é que o objeto (a qual o método pertence) é passado como o primeiro argumento da função. No nosso exemplo, a chamada `x.f()` é exatamente equivalente a `MyClass.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 argumentos.

Se você ainda não entendeu como métodos funcionam, talvez uma olhadela na implementação sirva para clarear as coisas. Quando um atributo de instância é referenciado e não é um atributo de dado, a sua classe é procurada. Se o nome indica um atributo de classe válido que seja um objeto função, um objeto método é criado pela composição da instância alvo e do objeto função. Quando o método é chamado com uma lista de argumentos, ele é desempacotado, uma nova lista de argumentos é criada a partir da instância original e da lista original de argumentos do método. Finalmente, a função é chamada com a nova lista de argumentos.

9.4 Observações Aleatórias

[Talvez se devesse colocar o que está por vir com mais cuidado...]

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 utilizar algum tipo de convenção que minimize conflitos, como colocar nomes de métodos com inicial maiúscula, prefixar atributos de dados com uma string única (quem sabe "_"), ou simplesmente utilizar substantivos para atributos e verbos para métodos.

Atributos de dados podem ser referenciados por métodos da própria instância, bem como por qualquer outro usuário 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 Python, escrita em C, pode esconder completamente detalhes de um objeto ou regular seu acesso se necessário. Isto pode ser utilizado por extensões a Python escritas em C.

Clientes devem utilizar atributos de dados com cuidado, pois podem bagunçar invariantes mantidos pelos métodos ao esbarrar nos seus atributos. Portanto, clientes podem adicionar à vontade atributos de dados para 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.

Não existe atalho para referenciar atributos de dados (ou métodos) de dentro de um método. Isso na verdade aumenta a legibilidade dos métodos: não há como confundir uma variável local com uma instância global ao dar uma olhadela em um método desconhecido.

Por convenção, o primeiro argumento de qualquer método é frequentemente chamado `self`. Isso não é nada mais do que uma convenção, `self` não possui nenhum significado especial em Python. Observe que ao seguir a convenção seu código se torna legível por uma grande comunidade de desenvolvedores Python e potencialmente poderá se beneficiar de ferramentas feitas por outrém que se baseie na convenção.

Qualquer função que é também atributo de classe define um método nas 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:
    f = f1
    def g(self):
        return 'hello world'
    h = g
```

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

Métodos podem chamar outros métodos utilizando o argumento `self` :

```
class Bag:
    def __init__(self):
        self.data = []
    def add(self, x):
        self.data.append(x)
    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

Métodos podem referenciar nomes globais da mesma forma que funções ordinárias. O escopo global associado a um método é o módulo contendo sua definição de classe (a classe propriamente dita nunca é usada como escopo global!). Ainda que seja raro encontrar a necessidade de utilizar 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 um conjunto de razões pelas quais um método desejaria referenciar sua própria classe.

9.5 Herança

Obviamente, uma característica de linguagem não seria digna do nome “classe” se não suportasse herança. A sintaxe para uma classe derivada se parece com:

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

O nome `BaseClassName` deve estar definido em um escopo contendo a definição da classe derivada. No lugar do nome da classe base também são aceitas expressões. O que é muito útil quando a classe base é definida em outro

módulo, ex:

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

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 existe nada de especial sobre instanciação de classes derivadas. `DerivedClassName()` cria uma nova instância da classe. Referências a métodos são resolvidas da seguinte forma: o atributo correspondente é procurado através da cadeia de derivação, e referências a métodos são válidas desde que produzam um objeto do tipo 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 efetivamente `virtual`.

Um método que sobrescreva outro em uma classe derivada 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 a classe base '`BaseClassName.methodname(self, arguments)`'. O que pode ser útil para os usuários da classe também. Note que isso só funciona se a classe base for definida ou importada diretamente no escopo global.

9.5.1 Herança Múltipla

Python também suporta uma forma limitada de herança múltipla. Uma definição de classe que herda de várias classes bases é:

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

A única regra que precisa ser explicada é a semântica de resolução para as referências a atributos de classe. É feita uma busca em profundidade da esquerda para a direita. Logo, se um atributo não é encontrado em `DerivedClassName`, 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`. Enquanto a busca em profundidade não diferencia o acesso a atributos diretos ou herdados de `Base1`.

É sabido que o uso indiscriminado de herança múltipla é o pesadelo da manutenção, sobretudo pela confiança de Python na adoção de uma convenção de nomenclatura para evitar conflitos. Um problema bem conhecido com herança múltipla é quando há uma classe derivada de outras que por sua vez possuem um ancestral em comum. Ainda que seja perfeitamente claro o que acontecerá (a instância possuirá uma única cópia dos atributos de dados do ancestral comum), não está claro se a semântica é útil.

9.6 Variáveis Privadas

Existe um suporte limitado a identificadores privados em classes. Qualquer identificador no formato `__spam` (no mínimo dois caracteres '_' no prefixo e no máximo u m como sufixo) é realmente substituído por `_classname__spam`, onde `classname` é o nome da classe corrente. Essa construção independe da posição sintática do identificador, e pode ser usada para tornar privadas: instâncias, variáveis de classe e métodos. Pode haver truncamento se o nome combinado extrapolar 255 caracteres. Fora de classes, ou quando o nome da classe só tem '_', não se aplica esta construção.

Este procedimento visa oferecer a classes uma maneira fácil de definir variáveis de instância e métodos "privados", sem ter que se preocupar com outras variáveis de instância definidas em classes derivadas ou definidas fora da classe. Apesar da regra de nomenclatura ter sido projetada para evitar "acidentes", ainda é possível o acesso e a manipulação de entidades privadas. O que é útil no caso de depuradores, talvez a única razão pela qual essa característica ainda não tenha sido suprimida (detalhe: derivar uma classe com o mesmo nome da classe base torna possível o uso de seus membros privados! Isso pode ser corrigido em versões futuras).

Observe que código passado para `exec`, `eval()` ou `evalfile()` não considera o nome da classe que o invocou como sendo a classe corrente. O modificador `global` funciona de maneira semelhante, quando se está restrito ao código que foi byte-compilado em conjunto. A mesma restrição se aplica aos comandos `getattr()`, `setattr()` e `delattr()`, e a manipulação direta do dicionário `__dict__`.

Aqui está um exemplo de uma classe que implementa seus próprios métodos `__getattr__()` e `__setattr__()`, armazenando todos os atributos em variáveis privadas. Da forma como foi codificado este exemplo, ele funcionará até em versões de Python em que esta característica ainda não havia sido implementada.

```
class VirtualAttributes:
    __vdict = None
    __vdict_name = locals().keys()[0]

    def __init__(self):
        self.__dict__[self.__vdict_name] = {}

    def __getattr__(self, name):
        return self.__vdict[name]

    def __setattr__(self, name, value):
        self.__vdict[name] = value
```

9.7 Particularidades

Às vezes, é útil ter um tipo semelhante ao "record" de Pascal ou ao "struct" de C. Uma definição de classe vazia atende esse propósito:

```
class Employee:
    pass

john = Employee() # Cria um registro vazio de Empregado

# Preenche os campos do registro
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

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

ao invés do arquivo propriamente dito.

Métodos de instância são objetos, e podem possuir atributos também.

9.7.1 Exceções Podem Ser Classes

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

Existem duas novas semânticas válidas para o comando `raise`:

```
raise Class, instance

raise instance
```

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

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

Uma cláusula de exceção pode listar tanto classes como strings. Uma classe em uma cláusula de exceção é compatível com a exceção se é a mesma classe prevista na cláusula ou ancestral dela (não o contrário, se na cláusula estiver uma classe derivada não haverá casamento com exceções base levantadas). No exemplo a seguir será impresso B, C, D nessa ordem: order:

```
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 (B no início), seria impresso B, B, B – sempre a primeira cláusula válida é ativada.

Quando uma mensagem de erro é impressa para uma exceção não tratada que for uma classe, o nome da classe é impresso, e depois a instância da exceção é convertida para string pela função `str()` e é também impressa (após um espaço e uma vírgula).

E agora?

Espera-se que a leitura deste manual tenha aguçado seu interesse em utilizar Python. Agora o que você deve fazer ?

Você deve ler, ou pelo menos correr os olhos pela *Library Reference* que provê material completo de referência sobre tipos, funções e módulos que podem poupar um bocado de tempo na hora de escrever programas em Python. Na distribuição padrão de Python, há módulos para leitura de caixas postais, recuperar documentos via HTTP, gerar números aleatórios, escrever programas CGI, comprimir dados, e muito mais. O material de referência pode lhe dar uma boa idéia do que já está disponível.

O principal Web site de Python é http://www.python.org/. Ele contém código, documentação e links para outras páginas relacionadas pelo mundo afora. Este site é espelhado em diversos cantos do mundo, como na Europa, Japão e Austrália. Um site espelho pode ser mais rápido de ser acessado dependendo da sua posição geográfica. Outro site importante é http://starship.python.net/, que contém páginas pessoais de gente envolvida com Python, muito software está disponível para download neste site.

Para questões relacionadas a Python e reportagem problemas, você pode enviar uma mensagem para o newsgroup comp.lang.python, ou para a lista de e-mail python-list@python.org. Ambos estão vinculados e tanto faz mandar através de um ou através de outro que o resultado é o mesmo. Existem em média 120 mensagens por dia,

perguntando e respondendo questões, sugerindo novas funcionalidades, e anunciando novos módulos. Antes de submeter, esteja certo que o que você procura não consta na Frequently Asked Questions (também conhecida por FAQ), em http://www.python.org/doc/FAQ.html, ou procure no diretório 'Misc/' da distribuição (código fonte). Os arquivos da lista de discussão estão disponíveis em http://www.python.org/pipermail/. A FAQ responde muitas das questões reincidentes na lista, talvez lá esteja a solução para o seu problema.

Edição de Entrada Interativa e Substituição por Histórico

Algumas versões do interpretador Python suportam facilidades de edição e substituição semelhantes as encontradas na Korn shell ou na GNU Bash shell. 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 disponívies 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.1 Edição de Linha

Se for suportado, edição de linha está ativa sempre que o interpretador imprimir um dos prompts (primário ou secundário). A linha corrente pode ser editada usando comandos típicos do Emacs. Os mais imoportantes 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 caracter 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` é o undo e tem efeito cumulativo.

A.2 Substituição de Histórico

Funciona da seguinte maneira: todas linhas não vazias são armazenadas em um buffer (histórico). `C-P` volta uma posição no histórico, `C-N` avança uma posição. Pressionando `Return` a linha corrente é alimentada para o interpretador. `C-R` inicia uma busca para trás no histórico, e `C-S` um busca para frente.

A.3 Vinculação de Teclas

A vinculação de teclas e outros parâmetros da biblioteca Readline podem ser personalizados por configurações colocadas no arquivo ‘~/.inputrc’. Vinculação de teclas tem o formato:

```
key-name: function-name
```

ou

```
"string": function-name
```

e opções podem ser especificadas com:

```
set option-name value
```

Por exemplo:

```
# Qume prefere editar estilo 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 vinculação default para `Tab` em Python é inserir um caracter `Tab` ao invés de completar o nome de um arquivo (default no Readline). Isto pode ser reconfigurado de volta através:

```
Tab: complete
```

no '~/.inputrc'. Todavia, isto torna mais difícil digitar comandos identados em linhas de continuação.

Preenchimento automático de nomes de variáveis e módulos estão opcionalmente disponívies. Para habilitá-los no modo interativo, adicione o seguinte ao seu arquivo de inicialização: [1]

```
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. Um determinado nome é procurado 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 extensões. Isso pode até implicar na execução de código definido por aplicação quando um objeto que define o método `__getattr__()` for parte da expressão.

A.4 Comentário

Essa facilidade representa um enorme passo em comparação com versões anteriores do interpretador. Todavia, ainda há características desejáveis deixadas de fora. Seria interessante se a indentação apropriada fosse sugerida em linhas de continuação, pois o parser sabe se um token de identação é necessário. O mecanismo de preenchimento 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.

[1] Python executará o conteúdo do arquivo identificado pela variável de ambiente PYTHONSTARTUP quando se dispara o interpretador interativamente.