Mergulhando no Python

Dive Into Python

8 October 2003

Este livro está disponível em http://diveintopython.org/. Caso esteja lendo em algum outro lugar sua versão pode estar desatualizada .

Table of Contents

Chapter 1. Instalando o Python

1.1. Qual Python é o certo para você?

Bem-vindo ao Python. Vamos mergulhar.

A primeira coisa que você precisa fazer com Python é instalar. Ou não? Se você está usando uma conta hospedada em um provedor, Python pode já estar instalado. A maioria das distribuições Linux instala Python por padrão. O Mac OS X 10.2 ou posterior já vem com uma versão linha de comando de Python, embora provavelmente você queira escolher uma versão com uma interface gráfica no estilo Mac.

O Windows não vem com Python. Mas não se preocupe! Há diversas formas fáceis de entrar no mundo Python usando Windows.

Como você pode ver, Python roda em muitos sistemas operacionais. A lista completa inclui Windows, o Mac OS, o Mac OS X e todos os vários sistemas gratuitos compatível com UNIX como o próprio Linux. Há também versões que rodam no Sun Solaris, AS/400, Amiga, OS/2, BeOS e muitas outras plataformas que você provavelmente nunca ouviu falar.

E o que é melhor, programas Python escritos em uma plataforma podem, com um pouco de cuidado, rodar em *qualquer* plataforma suportada. Por exemplo, eu regularmente desenvolvo programas Python no Windows e mais tarde os executo no Linux.

Voltando à pergunta que abriu esta seção: “qual Python é o certo para você?” A resposta é “aquele que roda no computador que você tem”

1.2. Python em Windows

No Windows, você tem diversas escolhas para instalar o Python.

A ActiveState tem um instalador Python para Windows que inclui a versão completa do Python, um IDE com um editor de código e extensões Windows para o Python que permitem acesso a serviços específicos do Windows, suas APIs e o registro.

O ActivePython pode ser baixado gratuitamente, mas não é open source. Foi com ele que eu aprendi a usar Python, e recomendo que você use a não ser que tenha algum motivo em particular para o evitar. (Um desses motivos pode ser o fato da ActiveState demorar alguns meses para atualizar o ActivePython com a última versão de Python disponível. Se você realmente precisa da última versão do Python e o ActivePython ainda está desatualizado, pule para a opção 2.)

Procedure 1.1. Opção 1: Instalando o ActivePython

1. Baixe o ActivePython em http://www.activestate.com/Products/ActivePython/.
2. Se você usa Windows 95, Windows 98 ou Windows ME, deve instalar o Windows Installer 2.0 antes de continuar.
3. Dê um clique duplo no arquivo `ActivePython-2.2.2-224-win32-ix86.msi`.
4. Siga as instruções na tela.
5. Se seu espaço em disco for limitado, é possível fazer uma instalação personalizada ("custom") e deixar de instalar a documentação, mas isso não é recomendado a não ser que você não realmente não possa gastar 14 megabytes a mais.
6. Após o término da instalação, feche o instalador e abra Iniciar->Programas->ActiveState ActivePython 2.2->PythonWin IDE.

Example 1.1. IDE ActivePython

```
PythonWin 2.2.2 (#37, Nov 26 2002, 10:24:37) [MSC 32 bit (Intel)] on win32.
Portions Copyright 1994-2001 Mark Hammond (mhammond@skippinet.com.au) -
see 'Help/About PythonWin' for further copyright information.
>>>
```

A segunda opção é usar o instalador Python “oficial”, distribuído pelos próprios desenvolvedores do Python. Esse instalador pode ser baixado gratuitamente, tem código fonte aberto e está sempre atualizado.

Procedure 1.2. Opção 2: Instalando o Python do Python.org

1. Baixe o `instalador Windows do Python` em http://www.python.org/ftp/python/2.3.2/.
2. Execute o arquivo `Python-2.3.2.exe`.
3. Siga as instruções na tela.
4. Se seu espaço em disco for limitado, é possível deselecionar o arquivo HTMLHelp, os scripts (`Tools/`), e/ou o kit de testes (`Lib/test/`).
5. Se você não tem direitos administrativos, pode selecionar Advanced Options ... e selecionar Non-Admin Install. A única diferença é a forma como as entradas de registro e os atalhos de menus são criados.
6. Após o término da instalação, feche o instalador e abra Iniciar->Programas->Python 2.3->IDLE (Python GUI).

Example 1.2. IDLE (GUI Python)

```
Python 2.3.2 (#49, Oct  2 2003, 20:02:00) [MSC v.1200 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.

    **********************************************************************
    Personal firewall software may warn about the connection IDLE
    makes to its subprocess using this computer's internal loopback
    interface.  This connection is not visible on any external
    interface and no data is sent to or received from the Internet.
    **********************************************************************

IDLE 1.0
>>>
```

1.3. Python no Mac OS X

No Mac OS X, você tem duas opções: instalar ou não instalar. Você provavelmente quer instalar.

O Mac OS X 10.2 ou posterior vem com uma versão linha de comando do Python pré-instalada. Se você usa confortavelmente a linha de comando, pode usar essa versão durante o primeiro terço desse livro. A versão pré-instalada não vem com um parser XML, então quando chegar ao capítulo sobre XML será necessário instalar a versão completa.

Procedure 1.3. Executando a versão pré-instalada do Python no Mac OS X

1. Abra a pasta `/Applications`.
2. Abra a pasta `Utilities`.
3. Dê um clique duplo no `Terminal` para abrir uma janela de terminal e chegar à linha de comando.
4. Digite **`python`** na linha de comando.

Example 1.3. Usando a versão pré-instalada de Python no Mac OS X

```
Welcome to Darwin!
[localhost:~] you% python
Python 2.2 (#1, 07/14/02, 23:25:09)
[GCC Apple cpp-precomp 6.14] on darwin
Type "help", "copyright", "credits", or "license" for more information.
>>> [aperte Ctrl+D para voltar ao prompt de comando]
[localhost:~] you%
```

Isso funciona, mas provavelmente você vai preferir instalar a última versão, que também vem com um interpretador gráfico interativo.

Procedure 1.4. Instalando no Mac OS X

1. Baixe a imagem de disco `MacPython-OSX` em http://homepages.cwi.nl/~jack/macpython/download.html.
2. Se o browser não o fizer automaticamente, dê um clique duplo no arquivo `MacPython-OSX-2.3-1.dmg` para montar a imagem de disco no desktop.
3. Dê um clique duplo no instalador, `MacPython-OSX.pkg`.
4. O instalador pedirá o nome de usuário e a senha do administrador.
5. Siga as instruções na tela.
6. Após a instalação, feche o instalador e abra a pasta `/Applications`.
7. Abra a pasta `MacPython-2.3`
8. Dê um clique duplo em `PythonIDE` para executar o Python.

O IDE MacPython deve mostrar uma tela de abertura e abrir um interpretador interativo. Caso o interpretador não apareça, selecione Window->Python Interactive (**Cmd-0**).

Example 1.4. O IDE MacPython no Mac OS X

```
Python 2.3 (#2, Jul 30 2003, 11:45:28)
```

```
[GCC 3.1 20020420 (prerelease)]
Type "copyright", "credits" or "license" for more information.
MacPython IDE 1.0.1
>>>
```

Note que mesmo instalando a versão mais nova, a versão pré-instalada continua presente. Se estiver rodando scripts a partir da linha de comando, você precisa tomar cuidado com que versão de Python está usando.

Example 1.5. Duas versões de Python

```
[localhost:~] you% python
Python 2.2 (#1, 07/14/02, 23:25:09)
[GCC Apple cpp-precomp 6.14] on darwin
Type "help", "copyright", "credits", or "license" for more information.
>>> [aperte Ctrl+D para voltar ao prompt de comando]
[localhost:~] you% /usr/local/bin/python
Python 2.3 (#2, Jul 30 2003, 11:45:28)
[GCC 3.1 20020420 (prerelease)] on darwin
Type "help", "copyright", "credits", or "license" for more information.
>>> [aperte Ctrl+D para voltar ao prompt de comando]
[localhost:~] you%
```

1.4. Python no Mac OS 9

O Mac OS 9 não vem com qualquer versão de Python, mas a instalação é simples e só há uma opção.

Procedure 1.5. Instalando no Mac OS 9

1. Baixe o arquivo `MacPython23full.bin` em http://homepages.cwi.nl/~jack/macpython/download.html.
2. Se seu browser não abrir o arquivo automaticamente, dê um clique duplo no arquivo `MacPython23full.bin` para descomprimir o arquivo com o Stuffit Expander.
3. Dê um clique duplo no instalador, `MacPython23full`.
4. Siga as instruções na tela.
5. Após o término da instalação, feche o instalador e abra a pasta `/Application`.
6. Abra a pasta `MacPython-OS9 2.3`.
7. Dê um clique duplo em `Python IDE` para executar o Python.

O IDE MacPython IDE deve mostrar uma tela de abertura e um interpretador interativo. Caso o shell interativo não apareça, selecione Window->Python Interactive (**Cmd-0**).

Example 1.6. O IDE MacPython no Mac OS 9

```
Python 2.3 (#2, Jul 30 2003, 11:45:28)
[GCC 3.1 20020420 (prerelease)]
Type "copyright", "credits" or "license" for more information.
MacPython IDE 1.0.1
>>>
```

1.5. Python no RedHat Linux

A instalação em sistemas operacionais compatíveis com UNIX (como o Linux) é fácil se você usar um pacote binário. Pacotes binários pré-compilados estão disponíveis para as distribuições Linux mais populares, ou você pode compilar a partir do código fonte.

Para instalar no RedHat Linux, você deve baixar o RPM em http://www.python.org/ftp/python/2.3.2/rpms/ e instalar usando o comando **rpm**.

Example 1.7. Instalando no RedHat Linux 9

```
localhost:~$ su -
Password: [digite sua senha de root]
[root@localhost root]# wget http://python.org/ftp/python/2.3/rpms/redhat-
9/python2.3-2.3-5pydotorg.i386.rpm
Resolving python.org... done.
Connecting to python.org[194.109.137.226]:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 7,495,111 [application/octet-stream]
...
[root@localhost root]# rpm -Uvh python2.3-2.3-5pydotorg.i386.rpm
Preparing...                ########################################### [100%]
   1:python2.3              ########################################### [100%]
[root@localhost root]# python          ❶
Python 2.2.2 (#1, Feb 24 2003, 19:13:11)
[GCC 3.2.2 20030222 (Red Hat Linux 3.2.2-4)] on linux2
Type "help", "copyright", "credits", or "license" for more information.
>>> [aperte Ctrl+D para sair]
[root@localhost root]# python2.3       ❷
Python 2.3 (#1, Sep 12 2003, 10:53:56)
[GCC 3.2.2 20030222 (Red Hat Linux 3.2.2-5)] on linux2
Type "help", "copyright", "credits", or "license" for more information.
>>> [aperte Ctrl+D para sair]
[root@localhost root]# which python2.3 ❸
/usr/bin/python2.3
```

❶ Ops! Apenas digitando **`python`** executamos a versão velha do Python, que veio instalada por padrão. Não é isso que queremos.

❷ A versão nova se chama **`python2.3`**. Você provavelmente deve mudar o caminho na primeira linha dos scripts de exemplo para que aponte para a versão mais nova.

❸ Este é o caminho completo da versão mais nova do Python que acabamos de instalar. Use isso na linha que começa com `#!` no começo de seus scripts para garantir que os scripts serão executados com a versão mais recente do Python, e lembre-se sempre de executar **`python2.3`** para entrar no interpretador interativo.

1.6. Python no Debian GNU/Linux

Se você é sortudo o suficiente para estar rodando o Debian GNU/Linux, a instalação é feita com o comando **apt**.

Example 1.8. Instalando no Debian GNU/Linux

```
localhost:~$ su -
Password: [digite sua senha de root]
localhost:~# apt-get install python
Reading Package Lists... Done
Building Dependency Tree... Done
The following extra packages will be installed:
  python2.3
Suggested packages:
  python-tk python2.3-doc
The following NEW packages will be installed:
  python python2.3
0 upgraded, 2 newly installed, 0 to remove and 3 not upgraded.
Need to get 0B/2880kB of archives.
After unpacking 9351kB of additional disk space will be used.
Do you want to continue? [Y/n] Y
Selecting previously deselected package python2.3.
(Reading database ... 22848 files and directories currently installed.)
Unpacking python2.3 (from .../python2.3_2.3.1-1_i386.deb) ...
Selecting previously deselected package python.
Unpacking python (from .../python_2.3.1-1_all.deb) ...
Setting up python (2.3.1-1) ...
Setting up python2.3 (2.3.1-1) ...
Compiling python modules in /usr/lib/python2.3 ...
Compiling optimized python modules in /usr/lib/python2.3 ...
localhost:~# exit
logout
localhost:~$ python
Python 2.3.1 (#2, Sep 24 2003, 11:39:14)
[GCC 3.3.2 20030908 (Debian prerelease)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> [aperte Ctrl+D para sair]
```

1.7. Instalando a partir do código fonte

Caso você prefira compilar seu Python, baixe o código fonte em http://www.python.org/ftp/python/2.3.2/ e faça o ritual de **`configure`**, **`make`**, **`make install`**.

Example 1.9. Instalando a partir do código fonte

```
localhost:~$ su -
Password: [digite sua senha de root]
localhost:~# wget http://www.python.org/ftp/python/2.3/Python-2.3.tgz
Resolving www.python.org... done.
Connecting to www.python.org[194.109.137.226]:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 8,436,880 [application/x-tar]
...
localhost:~# tar xfz Python-2.3.tgz
localhost:~# cd Python-2.3
localhost:~/Python-2.3# ./configure
checking MACHDEP... linux2
checking EXTRAPLATDIR...
checking for --without-gcc... no
...
localhost:~/Python-2.3# make
gcc -pthread -c -fno-strict-aliasing -DNDEBUG -g -O3 -Wall -Wstrict-prototypes
-I. -I./Include  -DPy_BUILD_CORE -o Modules/python.o Modules/python.c
```

```
gcc -pthread -c -fno-strict-aliasing -DNDEBUG -g -O3 -Wall -Wstrict-prototypes
-I. -I./Include  -DPy_BUILD_CORE -o Parser/acceler.o Parser/acceler.c
gcc -pthread -c -fno-strict-aliasing -DNDEBUG -g -O3 -Wall -Wstrict-prototypes
-I. -I./Include  -DPy_BUILD_CORE -o Parser/grammar1.o Parser/grammar1.c
...
localhost:~/Python-2.3# make install
/usr/bin/install -c python /usr/local/bin/python2.3
...
localhost:~/Python-2.3# exit
logout
localhost:~$ which python
/usr/local/bin/python
localhost:~$ python
Python 2.3.1 (#2, Sep 24 2003, 11:39:14)
[GCC 3.3.2 20030908 (Debian prerelease)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> [aperte Ctrl+D para voltar ao prompt de comando]
localhost:~$
```

1.8. O interpretador interativo

Agora que temos o Python instalado, o que é esse tal de interpretador interativo?

Funciona assim: o Python tem vida dupla. Ele pode ser usado como um interpretador de scripts que você pode executar linha de comando ou dando um duplo clique neles como se fossem aplicações. E também é possível usá-lo como um interpretador interativo para avaliar expressões e comandos. Isso é útil para depurar, desenvolver rapidamente e executar testes. Há até pessoas que usam o interpretador interativo como uma calculadora!

Abra o interpretador interativo do Python da maneira como for necessário em sua plataforma, e vamos mergulhar.

Example 1.10. Primeiros passos no interpretador interativo

```
>>> 1 + 1                ❶
2
>>> print 'hello world'  ❷
hello world
>>> x = 1                ❸
>>> y = 2
>>> x + y
3
```

❶ O interpretador interativo do Python pode avaliar expressões Python arbitrárias, inclusive expressões aritméticas básicas.

❷ O interpretador interativo pode executar comandos Python arbitrários, inclusive o comando **print**.

❸ Você pode atribuir valores a variáveis, e os valores serão lembrados por quanto tempo o interpretador esteja aberto (e nada além disso).

1.9. Resumo

Você deve agora ter uma versão de Python que funciona.

Dependendo de sua plataforma, você pode ter mais de uma. Caso seja seu caso, é necessário tomar cuidado com os caminhos dos interpretadores. Caso a simples execução do comando **python** não abra a versão do Python que você queira usar, pode ser necessário digitar o caminho completo para a versão completa.

Fora isso, parabéns, e bem-vindo ao Python.

Chapter 2. Conhecendo o Python

2.1. Mergulhando

Aqui temos um programa Python completo.

Provavelmente ele não faz o menor sentido para você. Não se preocupe ainda. Vamos apenas analisá-lo, linha a linha. Mas leia o programa e veja se consegue entender algo.

Example 2.1. `odbchelper.py`

If you have not already done so, you can download this and other examples used in this book.

```
def buildConnectionString(params):
    """Build a connection string from a dictionary of parameters.

    Returns string."""
    return ";".join(["%s=%s" % (k, v) for k, v in params.items()])

if __name__ == "__main__":
    myParams = {"server":"mpilgrim", \
                "database":"master", \
                "uid":"sa", \
                "pwd":"secret" \
                }
    print buildConnectionString(myParams)
```

Agora rode esse programa e veja o que acontece.

 No ActivePython em Windows, o módulo pode ser executado com File->Run... (**Ctrl-R**). A saída é exibida na janela interativa.

 No IDE do Mac OS X, o módulo pode ser executado em Python->Run window... (**Cmd-R**), mas há uma opção importante que você deve verificar antes. Abra o módulo no IDE, abra o menu de opções do módulo clicando no triângulo preto no canto superior direito da janela, e verifique que a opção "Run as __main__" esteja marcada. Essa configuração é guardada com o módulo, portanto só precisa ser feita uma vez por módulo.

 Em sistemas compatíveis com UNIX (incluindo Mac OS X), é possivel executar um módulo da linha de comando: **`python odbchelper.py`**

Example 2.2. Saída de `odbchelper.py`

```
server=mpilgrim;uid=sa;database=master;pwd=secret
```

2.2. Declarando funções

O Python tem funções como a maioria das linguagens, mas não tem arquivos de cabeçalho separados como C++ ou seções `interface` e `implementation` como Pascal. Quando precisar de uma função, apenas declare e saia programando.

Example 2.3. Declarando a função `buildConnectionString`

```
def buildConnectionString(params):
```

Várias coisas devem ser notadas. Em primeiro lugar, a palavra-chave `def` inicia uma declaração de função, seguida pelo seu nome, seguido pelos argumentos entre parênteses. Múltiplos argumentos (não é o caso aqui) podem ser separados por vírgulas.

A função não define um tipo de dado de retorno. Funções Python não especificam o tipo de seu valor de retorno; nem sequer se especifica se de fato existe um valor de retorno. Caso a função execute um comando `return`, retornará um valor, e caso contrário retornará `None`, o valor nulo padrão do Python.

 Em Visual Basic, funções (que retornam valor) começam com `function`, e subrotinas (que não retornam um valor) começam com `sub`. Não há subrotinas em Python, apenas funções. Todas as funções retornam um valor (mesmo que seja `None`), e todas as funções são declarads com `def`.

O argumento, `params`, não especifica um tipo de dado. Em Python, variáveis nunca são explicitamente tipadas. Python descobre qual o tipo da variável e toma conta disso internamente.

 Em Java, C++ e outras linguagens de tipagem estática é necessário especificar o tipo do valor de retorno e de cada argumento de função. Em Python, o tipo de dados nunca é explicitamente declarado. O Python toma conta do tipo internamente baseado no valor que você atribuir.

Adendo. Um leitor erudito enviou essa explicação de como Python se compara com outras linguagens:

linguagens estaticamente tipadas
: Linguagens em que os tipos são definidos em tempo de compilação. A maior parte das linguagens estaticamente tipadas força isso obrigando que todas as variáveis sejam declaradas

com tipos definidos antes que possam ser usadas. Java e C são linguagens estaticamente tipadas.

linguagens dinamicamente tipadas
: Linguagens em que os tipos são descobertos em tempo de execução; o oposto das linguagens estaticamente tipadas. VBScript e Python são dinamicamente tipadas, porque descobrem o tipo da variável quando você atribui um valor à mesma.

linguagens fortemente tipadas
: Linguagens em que os tipos são sempre estritamente seguidos. Java e Python são fortemente tipadas. Se você tem um inteiro, não pode tratar como se fosse uma string sem conversão explícita (mais sobre esse assunto mais adiante nesse capítulo).

linguagens fracamente tipadas
: Linguagens em que o tipo é ignorado; o oposto das linguagens fortemente tipadas. VBScript é fracamente tipado. Em VBScript, é possivel concatenar a string `'12'` como o inteiro `3` e ter como resultado a string `'123'`, que também pode ser usada como se fosse o inteiro `123`, tudo isso sem conversão explícita.

Portanto Python é tanto *dinamicamente tipada* (porque não requer declarações de tipo) e *fortemente tipada* (porque a partir do momento em que a variável tem tipo, esse tipo faz diferença).

2.3. Documentando funções

É possivel documentar uma função do Python usando uma `doc string`.

Example 2.4. Definindo a `doc string` de `buildConnectionString`

```
def buildConnectionString(params):
    """Build a connection string from a dictionary of parameters.

    Returns string."""
```

Três aspas duplas limitam uma string multi-linhas. Tudo entre os aspas iniciais e finais faz parte da string, inclusive quebras de linha e outros tipos de aspas. É possível usar strings multi-linhas em qualquer lugar, mas a maior parte das vezes será na definição de uma `doc string`.


Aspas triplas são uma maneira útil de definir uma string com aspas simples e duplas, como `qq/.../` em Perl.

Tudo entre as triplas aspas é a `doc string` da função, a documentação do que a função faz. Uma `doc string`, caso exista, deve ser a primeira coisa definida em uma função (*i.e.* a primeira coisa depois do dois-pontos). Não é tecnicamente obrigatório usar uma `doc string`, mas é uma boa idéia. Eu sei que você deve ter ouvido isso todas as aulas de programação que você já teve, mas Python tem um incentivo a mais: a `doc string` está disponível em tempo de execução como um atributo da função.


Muitos IDEs Python usam a `doc string` para prover documentação sensível ao contexto, de forma que quando você digita o nome da função a `doc string` aparece como uma dica flutuante. Isso é muito útil, mas é apenas tão bom quanto as `doc string`s que você escreve.

Leitura recomendada

- PEP 257 define convenções de `doc string`.

- *Python Style Guide* discute como escrever uma boa `doc string`.
- *Python Tutorial* discute convenções de espaçamento em `doc strings`.

2.4. Tudo é objeto

Caso você não tenha prestado atenção, eu disse que funções Python têm atributos, e que estes atributos estão disponíveis em tempo de execução.

Uma função, como tudo em Python, é objeto.

Example 2.5. Acessando a `doc string` da função `buildConnectionString`

```
>>> import odbchelper                                    ❶
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa",
"pwd":"secret"}
>>> print odbchelper.buildConnectionString(params)       ❷
server=mpilgrim;uid=sa;database=master;pwd=secret
>>> print odbchelper.buildConnectionString.__doc__      ❸
Build a connection string from a dictionary

Returns string.
```

❶ A primeira linha importa o programa `odbchelper` como um módulo. Uma vez que se importe o módulo, é possivel fazer referência a qualquer das funções, classes ou atributos públicos. Módulos podem podem fazer isso para fazer uso de recursos de outros módulos, e você pode fazer isso no IDE também. Esse conceito é importante, e retornaremos a ele mais tarde.

❷ Quando você quiser usar funções definidas em módulos importados, deve incluir o nome do módulo. Você não pode dizer apenas `buildConnectionString`, é obrigatório usar `odbchelper.buildConnectionString`. Isto é parecido com classes em Java.

❸ Ao invés de chamar a função como poderia ser esperado, pedimos um dos atributos da função, chamado `__doc__`.

`import` em Python é como `require` em Perl. Uma vez que você `import` um módulo Python, pode acessar seu conteúdo com *`módulo.função`* assim como em Perl você importa com `require` e acessa com *`módulo::função`*.

Antes de prosseguir, quero mencionar brevemente o caminho de busca de bibliotecas. O Python procura em diversos lugares o módulo a ser importado. Mais especificamente, ele procura em todos os diretórios definidos em `sys.path`. O conteúdo dessa variável é uma lista, e você pode facilmente visualizar ou modificar seu conteúdo com os métodos de manipulação de lista, que veremos mais tarde nesse capítulo.

Example 2.6. Caminhos de busca de módulos

```
>>> import sys                                           ❶
>>> sys.path                                             ❷
['', '/usr/local/lib/python2.2', '/usr/local/lib/python2.2/plat-linux2',
'/usr/local/lib/python2.2/lib-dynload', '/usr/local/lib/python2.2/site-packages',
'/usr/local/lib/python2.2/site-packages/PIL', '/usr/local/lib/python2.2/site-
packages/piddle']
>>> sys                                                  ❸
<module 'sys' (built-in)>
>>> sys.path.append('/meu/novo/diretório')               ❹
```

❶ Importar o módulo `sys` faz todas as suas funções e atributos ficarem disponíveis para uso.

❷ `sys.path` é uma lista de nomes de diretório que define o caminho de busca atual. (O seu `sys.path` vai ser diferente, dependendo do sistema operacional, da versão de Python e de onde foi instalado.) O Python procura em todos esses diretórios, na ordem, por arquivos `.py` com o mesmo nome que o módulo que você está tentando importar.

❸ Na verdade, eu menti: a verdade é um pouco mais complicada. Nem todos os módulos são arquivos `.py`. Alguns, como o próprio módulo `sys`, são módulos "embutidos" ("built-in"): eles fazem parte do próprio Python. Módulos embutidos funcionam como módulos normais, mas o código Python deles não está disponível, porque não foram escritos em Python (o módulo `sys` é escrito em C).

❹ Você pode adicionar um novo diretório ao caminho de procura em tempo de execução adicionando o nome do diretório ao `sys.path`. O Python automaticamente fará a procura no novo diretório quando você importar um módulo. O efeito dura por tanto tempo quanto o Python esteja procurando. (Falaremos sobre o `append` e outros métodos de lista mais tarde nesse capítulo.)

Tudo em Python é objeto, e quase tudo tem atributos e métodos.[1] Todas as funções têm um atributo interno `__doc__`, que retorna a `doc string` definida no código fonte da função. O módulo `sys` é um objeto que (entre outras coisas) um atributo chamado `path`, e assim por diante.

Isto é tão importante que vou repetir mais uma vez caso você não tenha entendido ainda: *tudo em Python é objeto*. Strings são objetos. Listas são objetos. Funções são objetos. Até módulos são objetos.

Leitura recomendada

- *Python Reference Manual* explica exatamente o que significa dizer que tudo em Python é objeto porque algumas pessoas são pedantes e adoram discutos extensivamente esse tipo de coisa.
- eff-bot sumariza objetos Python.

Footnotes

[1] Diferentes linguagens definem “objeto” de diferentes formas. Em algumas, significa que *todos* os objetos *precisam* ter atributos e métodos. Em outras, significa que todos os objetos são herdáveis FIXME. Em Python a definição é mais relaxada: alguns objetos não têm atributos nem métodos (mais sobre isso nesse capítulo), e nem todos os objetos são herdáveis (mais sobre isso no capítulo 3). Mas tudo é um objeto no sentido de que pode ser usado como valor de uma variável ou passado como argumento para uma função (mais sobre isso no capítulo 2).

2.5. Indentando[2] código

Funções Python não têm marcações explícitas de começo e fim como `begin` e `end` ou chaves. O único delimitador é o sinal de dois-pontos (“`:`”) e a própria indentação do código.

Example 2.7. Indentando a função `buildConnectionString`

```
def buildConnectionString(params):
    """Build a connection string from a dictionary of parameters.

    Returns string."""
    return ";".join(["%s=%s" % (k, v) for k, v in params.items()])
```

Blocos de código (funções, comandos `if`, laços `for`, etc.) são definidos pelo uso de indentação. Indentar o código inicia um bloco, remover a indentação termina. Não há sinais específicos ou palavras-chave. Isso significa que espaço em branco é significativo e deve ser consistente. Nesse exemplo, o código da função (incluindo a `doc string`) está indentado com 4 espaços. Não é necessário que seja 4, apenas que seja consistente. A primeira linha não indentada após isso está fora da função.

Após alguns protestos iniciais e várias analogias cínicas com Fortran, você fará as pazes com esse recurso e começará a ver os benefícios. Um grande benefício é que todos os programas Python são parecidos, uma vez que indentação é um requerimento da linguagem e não uma questão de estilo. Isso torna mais fácil a leitura de código escrito por terceiros.


Python usa quebras de linha para separar comandos e dois-pontos e indentação para separar códigos de bloco. C++ e Java usam ponto-e-vírgula para separar comandos e chaves para separar blocos de código.

Leitura recomendada

- *Python Reference Manual* discute questões de indentação entre plataformas diferentes e mostra vários erros de indentação.
- *Python Style Guide* discute estilo de indentação.

Footnotes

[2] Isso é um anglicismo mas os programadores falam assim (N. do T.)

2.6. Testando módulos

Módulos Python são objetos e têm diversos atributos úteis. Você pode usar isso para testar seus módulos facilmente à medida que os escreve.

Example 2.8. O truque do `if __name__`

```
if __name__ == "__main__":
```

Primeiro algumas observações antes de atacar o núcleo da questão. Em primeiro lugar, parênteses não são obrigatórios em torno da expressão `if`. Além disso, o comando `if` termina com dois-pontos, e é seguido por código indentado.


Como C, Python usa == para comparação e = para atribuição. Ao contrário de C, Python não aceita atribuição em-linha, o que torna impossível fazer uma atribuição de valor quando você queria fazer uma comparação.

Então por que esse `if` em particular é um truque? Módulos são objetos, e todos os módulos têm um atributo interno chamado `__name__`. O valor desse atributo depende da forma como você está usando o módulo. Se você usar `import`, o `__name__` será o nome de arquivo do módulo, sem o caminho e sem a extensão. Mas também é possível executar um módulo diretamente como um programa interativo. Nesse caso, `__name__` terá o valor especial `__main__`.

Example 2.9. O __name__ de um módulo importado

```
>>> import odbchelper
>>> odbchelper.__name__
'odbchelper'
```

Sabendo isso, você pode criar um conjunto de testes para seu módulo dentro do próprio módulo, protegido por esse `if`. Quando você executar o módulo diretamente, `__name__` será `__main__`, e os testes serão executados. Quando importar o módulo, `__name__` será alguma outra coisa e os testes serão ignorados. Com isso é mais fácil desenvolver e depurar novos módulos antes de integrá-los a um programa maior.


No MacPython há um passo adicional para fazer o truque do `if __name__` funcionar. Abra o menu de opções do módulo clicando o triângulo preto no canto superior direito da janela, e certifique-se de que Run as __main__ esteja selecionado.

Leitura recomendada

- *Python Reference Manual* discute os detalhes de baixo nível da importação de módulos .

2.7. Apresentando dicionários

Uma breve digressão é necessária, porque você precisa aprender sobre dicionários, tuplas e listas (tudo isso!). Caso você seja um programador Perl, pode apenas dar uma olhada nos dicionários e listas, mas deve prestar atenção nas tuplas.

Um dos tipos básicos do Python é o dicionário, que define relações um-para-um entre chaves e valores.


Um dicionário em Python é como um hash (ou array associativo) em Perl. Em Perl, variáveis que armazenam hashes sempre iniciam com um caracter `%`; em Python, variáveis podem ter qualquer nome, e o tipo do dado é mantido internamente.


Um dicionário em Python é como uma instância da classe `Hashtable` em Java.


Um dicionário em Python é como a instância de um objeto `Scripting.Dictionary` em Visual Basic.

Example 2.10. Definindo um dicionário

```
>>> d = {"server":"mpilgrim", "database":"master"}
>>> d
{'server': 'mpilgrim', 'database': 'master'}
>>> d["server"]
'mpilgrim'
>>> d["database"]
'master'
>>> d["mpilgrim"]
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
KeyError: mpilgrim
```

❶ Primeiramente, criamos um dicionário com dois elementos e o atribuímos à variável `d`. Cada elemento é um par chave-valor, e o conjunto dos elementos é limitado por chaves.

❷ `'server'` é uma chave, e seu valor associado, referenciado com `d["server"]`, é `'mpilgrim'`.

❸ `'database'` é uma chave, e seu valor associado, referenciado com `d["database"]`, é `'master'`.

❹ É possível achar os valores a partir da chave, mas não as chaves a partir do valor. Portanto `d["server"]` é `'mpilgrim'`, mas `d["mpilgrim"]` levanta uma exceção porque `'mpilgrim'` não é uma chave.

Example 2.11. Modificando um dicionário

```
>>> d
{'server': 'mpilgrim', 'database': 'master'}
>>> d["database"] = "pubs" ❶
>>> d
{'server': 'mpilgrim', 'database': 'pubs'}
>>> d["uid"] = "sa"        ❷
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'pubs'}
```

❶ Não é possível ter chaves duplicadas em um dicionário. Atribuir um valor a uma chave vai apagar o valor antigo.

❷ É possível adicionar pares chave-valor a qualquer momento. A sintaxe é idêntica à de modificação de valores existentes. (Sim, isso vai incomodar você algum dia quando você achar que está adicionando valores mas está na verdade apenas modificando o mesmo valor porque uma chave não está mudando da forma como você espera.)

Note que o novo elemento (chave `'uid'`, valor `'sa'`) parece estar no meio. Na metade, é apenas uma coincidência os elementos terem aparecido em ordem no primeiro elemento; aqui é apenas uma coincidência que pareça estar fora de ordem.

Dicionários não têm o conceito de ordenamento entre os elementos. É incorreto dizer que os elementos estão “fora de ordem”; eles estão simplesmente desordenados. Essa distinção é importante e vai incomodar você quando você quiser acessar um dicionário em uma ordem específica, reproduzível (como por ordem alfabética de chave). Há maneiras de fazer isso, mas não dentro do objeto de dicionário.

Example 2.12. Misturando tipos em um dicionário

```
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'pubs'}
>>> d["retrycount"] = 3 ❶
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'master', 'retrycount': 3}
>>> d[42] = "douglas"   ❷
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'master', 42: 'douglas',
'retrycount': 3}
```

❶ Dicionários não são apenas para strings. Valores de dicionários podem ter qualquer tipo, inclusive strings, inteiros, objetos e até outros dicionários. E dentro de um dicionário os valores não precisam ser de um tipo uniforme. É possível usar os tipos que forem necessários, à vontade.

❷ Chaves de dicionários são mais restritivas, mas podem ser inteiros, strings e alguns outros tipos (mais sobre isso adiante). Os tipos de chaves também podem ser misturados.

Example 2.13. Removendo itens de um dicionário

```
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'master', 42: 'douglas',
'retrycount': 3}
>>> del d[42] ❶
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'master', 'retrycount': 3}
>>> d.clear() ❷
>>> d
{}
```

❶ `del` remove itens individuais do dicionário por chave.

❷ `clear` remove todos os itens de um dicionário. Note que um par de chaves vazio significa um dicionário sem itens.

Example 2.14. Strings são sensíveis a caso

```
>>> d = {}
>>> d["key"] = "value"
>>> d["key"] = "other value" ❶
>>> d
{'key': 'other value'}
>>> d["Key"] = "third value" ❷
>>> d
{'Key': 'third value', 'key': 'other value'}
```

❶ Atribuir um valor a uma chave existente de dicionário simplesmente substitui o valor antigo com o novo.

❷ Isso não é atribuir um valor novo a uma chave existente porque strings no Python são sensíveis a caso, então `'key'` não é o mesmo que `'Key'`. Isso cria um novo par de chave/valor no dicionário. Pode ser parecido para você, mas para o Python é algo totalmente diferente.

Leitura recomendada

- *How to Think Like a Computer Scientist* ensina como usar dicionários e como usar dicionários para modelar matrizes esparsas.
- Python Knowledge Base Tem muitos exemplos de uso de dicionários.
- Python Cookbook discute como ordenar valores de um dicionário por chave.
- *Python Library Reference* sumariza todos os métodos de dicionários.

2.8. Apresentando listas

Listas são o tipo mais produtivo do Python. Se sua única experiência com listas for arrays[3] em Visual Basic ou (coitado de você) o datastore em Powerbuilder, prepare-se psicologicamente para aprender sobre as listas do Python.


Uma lista em Python é como um array em Perl. Em Perl, variáveis que armazenam arrays sempre começam c variáveis podem ter qualquer nome, uma vez que o Python cuida do tipo internamente.


Uma lista em Python faz muito mais que um array em Java (embora possa ser usada da mesma forma se é tud
melhor seria com a classe `Vector`, que pode armazenar objetos arbitrários e crescer automaticamente à med

Example 2.15. Definindo uma lista

```
>>> li = ["a", "b", "mpilgrim", "z", "example"] ❶
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li[0]                                       ❷
'a'
>>> li[4]                                       ❸
'example'
```

❶ Primeiramente definimos uma lista com 5 elementos. Note que eles mantêm a ordem original. Isso não é obra d
ordenado de elementos colocados entre colchetes.

❷ Uma lista pode ser usada como um array com índice iniciado em zero. O primeiro elemento de uma lista não-va

❸ O último elemento dessa lista de cinco elementos é `li[4]`, uma vez que listas têm índice começado em zero.

Example 2.16. Índices negativos de listas

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li[-1] ❶
'example'
>>> li[-3] ❷
'mpilgrim'
```

❶ Um índice negativo acessa elementos a partir do fim da lista, contando para trás. O último elemento de uma list

❷ Se índices negativos estão criando confusão, pense dessa forma: `li[-n] == li[len(li) - n]`. Portant
`- 3] == li[2]`.

Example 2.17. Fatiando uma lista

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li[1:3]  ❶
['b', 'mpilgrim']
>>> li[1:-1] ❷
['b', 'mpilgrim', 'z']
>>> li[0:3]  ❸
['a', 'b', 'mpilgrim']
```

❶ Aqui tiramos um subconjunto da lista, chamado de "fatia", especificando dois índices. O valor de retorno é uma
elementos da lista, em ordem, a partir do primeiro índice da fatia (nesse caso `li[1]`) até o segundo índice, exc

❷ O fatiamento também funciona se um ou ambos os índices forem negativos. Você pode pensar da seguinte form
direita, o primeiro índice especifica o primeiro elemento que você quer, e o segundo índice especifica o primeir
valor de retorno é tudo que estiver no meio.

❸ Listas são baseadas em zero, portando `li[0:3]` retorna os três primeiros elementos da lista, iniciando em `li[`

Example 2.18. Atalho no fatiamento

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li[:3] ❶
```

```
['a', 'b', 'mpilgrim']
>>> li[3:] ❷ ❸
['z', 'example']
>>> li[:]  ❹
['a', 'b', 'mpilgrim', 'z', 'example']
```

❶ Se o primeiro índice da fatia for 0, é possível deixar ele de fora. `li[:3]` é o mesmo que `li[0:3]` do exempl

❷ De maneira semelhante, se o segundo índice da fatia for o último ítem da lista, é possível deixar esse índice for `li[3:5]` porque a lista tem 5 elementos.

❸ Perceba a simetria aqui. Nessa lista de 5 elementos, `li[:3]` retorna os três primeiros elementos, e `li[3:]` re `[:n]` sempre retorna `n` primeiros elementos, e `li[n:]` sempre retorna o restante, não importa o tamanho da l

❹ Se ambos os índices da fatia são deixados de fora, todos os os elementos da lista são inclusos. Mas o resultado n sim uma outra lista que contém todos os elementos iguais. `li[:]` é um atalho para fazer uma cópia completa d

Example 2.19. Adicionando elementos à lista

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li.append("new")               ❶
>>> li
['a', 'b', 'mpilgrim', 'z', 'example', 'new']
>>> li.insert(2, "new")            ❷
>>> li
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new']
>>> li.extend(["two", "elements"]) ❸
>>> li
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new', 'two', 'elements']
```

❶ `append` adiciona um único elemento ao fim da lista.

❷ `insert` insere um elemento em uma lista. O argumento numérico é o índice do primeiro elemento a ser empu atual. Note que elementos da lista não precisam ser únicos; há agora dois elementos distintos com o valor `'new`

❸ `extend` concatena listas. Note que não se chama `extend` com múltiplos argumentos; o único argumento é un elementos.

Example 2.20. Buscando em uma lista

```
>>> li
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new', 'two', 'elements']
>>> li.index("example") ❶
5
>>> li.index("new")     ❷
2
>>> li.index("c")       ❸
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: list.index(x): x not in list
>>> "c" in li           ❹
0
```

❶ `index` encontra a primeira ocorrência de um valor na lista e retorna o índice.

❷ `index` encontra a *primeira* ocorrência de valor na lista. Nesse caso, `'new'` ocorre duas vezes na lista, em `li` apenas o primeiro índice, `2`.

❸ Se o valor não for encontrado na lista, o Python levanta uma exceção. Isso é diferente da maioria das outras ling
inválido. Embora possa parecer incômodo, isso é a Coisa Certa, pois significa que seu programa vai ser interror
invés de mais tarde quando você tentar usar o índice inválido.

❹ Para testar se um valor está na lista, use `in`, que retorna `1` se o valor for encontrado ou `0` caso contrário.

Antes da versão 2.2.1, Python não tinha um tipo de dados booleano. Para compensar essa falta, Python aceita
contexto booleano (o condicional de um comando `if` por exemplo), de acordo com as seguintes regras: `0` é f
verdadeiros. Uma string vazia (`""`) é falsa, todas as demais strings são verdadeiras. Uma lista vazia (`[]`) é fa
verdadeiras. Uma tupla vazia (`()`) é falsa, todas as outras tuplas são verdadeiras. Um dicionário vazio (`{}`) é
são verdadeiro. Essas regras ainda funcionam assim do Python 2.2.1 em diante, mas agora também é possíve
com os nomes de `True` e `False`. Note que ambos os valores começam com maiúsculas: Python é sensível

Example 2.21. Removendo elementos de uma lista

```
>>> li
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new', 'two', 'elements']
>>> li.remove("z")    ❶
>>> li
['a', 'b', 'new', 'mpilgrim', 'example', 'new', 'two', 'elements']
>>> li.remove("new")  ❷
>>> li
['a', 'b', 'mpilgrim', 'example', 'new', 'two', 'elements']
>>> li.remove("c")    ❸
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: list.remove(x): x not in list
>>> li.pop()          ❹
'elements'
>>> li
['a', 'b', 'mpilgrim', 'example', 'new', 'two']
```

❶ `remove` remove a primeira ocorrência de um valor em uma lista.

❷ `remove` remove *apenas* a primeira ocorrência do valor. Nesse caso, `'new'` aparecia duas vezes na lista, mas
apenas a primeira ocorrência.

❸ Se o valor não for encontrado na lista, o Python levanta uma exceção. Isso é simétrico com o comportamento do

❹ `pop` é uma coisa interessante. Ele faz duas coisas: remove o último elemento da lista e retorna o valor. Note qu
retorna um valor mas não muda a lista, e diferente de `li.remove(`*`valor`*`)`, que muda a lista mas não retorn

Example 2.22. Operadores de lista

```
>>> li = ['a', 'b', 'mpilgrim']
>>> li = li + ['example', 'new']  ❶
>>> li
['a', 'b', 'mpilgrim', 'example', 'new']
>>> li += ['two']                 ❷
>>> li
['a', 'b', 'mpilgrim', 'example', 'new', 'two']
>>> li = [1, 2] * 3               ❸
>>> li
[1, 2, 1, 2, 1, 2]
```

❶ Listas podem ser concatenadas com o operador +. *`lista = lista + outralista`* produz o mesmo resu
(*`outralista`*). Mas o operador + retorna uma nova lista (concatenada) como valor, enquanto `extend` aper
Isto significa que `extend` é mais rápido, especialmente para listas grandes.

❷ Python suporta o operador +=. `li += ['two']` é equivalente a `li.extend(['two'])`. O operador += e pode ser sobrecarregado para funcionar com classes definidas pelo usuário. (Mais sobre classes no capítulo 3.
❸ O operador `*` funciona em listas como um repetidor. `li = [1, 2] * 3` é equivalente a `li = [1, 2]` - concatena as tres listas em uma. FIXME (* cria referencias a mesma lista)

Leitura recomendada

- *How to Think Like a Computer Scientist* fala sobre listas e ensina coisas importantes sobre o uso de listas como argumento de funçÕes.
- *Python Tutorial* mostra como usar listas como pilhas e filas.
- Python Knowledge Base responde perguntas comuns sobre listas e tem bastantes exemplos de código usando listas.
- *Python Library Reference* sumariza todos os métodos de listas.

Footnotes

[3] Traduzido, o termo array seria na maior parte dos casos um *vetor*. O termo array é mais usual. (N. do T.)

2.9. Apresentando tuplas

Uma tupla é uma lista imutável. Tuplas não podem ser alteradas depois de criadas.

Example 2.23. Definindo uma tupla

```
>>> t = ("a", "b", "mpilgrim", "z", "example") ❶
>>> t
('a', 'b', 'mpilgrim', 'z', 'example')
>>> t[0]                                       ❷
'a'
>>> t[-1]                                      ❸
'example'
>>> t[1:3]                                     ❹
('b', 'mpilgrim')
```

❶ Uma tupla é definida da mesma forma que uma lista, exceto que o conjunto de elementos deve estar entre parênteses, não colchetes.
❷ Os elementos de uma tupla têm uma ordem definida, igual a listas. Índices de tuplas também são baseados em zero. Por isso, o primeiro elemento de uma tupla não-vazia é sempre `t[0]`.
❸ Índices negativos são contados a partir do fim da tupla, como nas listas.
❹ Fatiar tuplas também funciona. Note que quando uma lista é fatiada, o resultado é uma lista, e quando uma tupla é fatiada o resultado é uma nova tupla.

Example 2.24. Tuplas não têm métodos

```
>>> t
('a', 'b', 'mpilgrim', 'z', 'example')
>>> t.append("new")    ❶
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'append'
```

```
>>> t.remove("z")      ❷
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'remove'
>>> t.index("example") ❸
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'index'
>>> "z" in t           ❹
1
```

❶ Não é possível adicionar elementos a uma tupla. Tuplas não têm métodos `append` ou `extend`.

❷ Não é possível remover elementos de uma tupla. Não existem os métodoso `remove` ou `pop`.

❸ Não é possive achar elementos na lista com o método `index` pois ele não existe..

❹ É possível, porém, usar `in` para ver se um elemento existe na tupla.

Então para que servem as tuplas?

- Tuplas são mais rápidas que listas. Se você está definindo um conjunto constante de valores e tudo que você vai fazer é iterar por eles, use uma tupla ao invés de uma lista.
- Seu código fica mais seguro caso você proteja dados que não precisam ser mudados contra gravação. Usar uma tupla é como usar um comando `assert` implícito dizendo que o dado é apenas para leitura, e que é bom pensar melhor antes de mudar isso.
- Lembra-se quando eu disse que chaves de dicionários podem ser inteiros, strings e "outros tipos"? Tuplas são um desses tipos. Tuplas podem ser usadas como chaves de dicionários, mas listas não.[4]
- Tuplas são usadas na formatação de strings, como veremos brevemente.

Tuplas podem ser convertidas em listas e vice-versa. A função interna `tuple` toma uma lista como parâmetro e retorna uma tupla com os mesmos elementos, e a funçao `list` toma uma tupla e retorna uma lista. Uma maneira de pensar isso é que `tuple` congela uma lista, `list` descongela uma tupla.

Leitura recomendada

- *How to Think Like a Computer Scientist* ensina sobre tuplas e mostra como concatenar tuplas.
- Python Knowledge Base mostra como ordenar uma tupla.
- *Python Tutorial* mostra como definir uma tupla com um elemento.

Footnotes

[4] Na verdade é mais complicado que isso. Chaves de dicionários devem ser imutáveis. Tuplas são imutáveis, mas se você tiver tuplas de listas o tipo é mutável e não é seguro usá-las como chaves. Apenas tuplas de strings, números ou outras tuplas seguras podem ser usadas como chaves de dicionários.

2.10. Declarando variáveis

Agora que você acha que sabe tudo sobre dicionários, tuplas e listas, vamos voltar ao nosso programa-exemplo, `odbchelper.py`.

Python tem variáveis locais e globais como a maioria das outras linguagens, mas não há declarações

explícitas de variáveis. Variáveis passam a existir quando um valor lhes é atribuído e deixam de existir quando saem de escopo. .

Example 2.25. Definindo a variável `myParams`

```
if __name__ == "__main__":
    myParams = {"server":"mpilgrim", \
                "database":"master", \
                "uid":"sa", \
                "pwd":"secret" \
                }
```

Várias coisas interessantes aqui. Em primeiro lugar, note a indentação. Um comando `if` é um código de bloco e precisa ser indentado como uma função.

Em segundo lugar, a atribuição de variáveis é um comando espalhado por várias linhas, com uma barra invertida ("\") servindo como marcador de continuação.


Quando um comando é separado em várias linhas com o sinal de continuação, ("\"), as linhas continuadas podem ser indentadas de qualquer maneira; as regras usuais de indentação do Python não se aplicam. Mas se seu IDE Python auto-indenta a linha continuada, aceite o padrão a não ser que haja algum bom motivo contra.


Estritamente falando, expressões entre parênteses, colchetes ou chaves (como na definição de um dicionário) podem ser separadas em várias linhas com ou sem o caracter de continuação ("\"). Eu gosto de incluir a barra invertida mesmo quando não é obrigatória porque eu acho que deixa o código mais fácil de ler, mas é uma questão de etilo.

Em terceiro lugar, você nunca declarou a variável `myParams`, você apenas atribuiu um valor a ela. Funciona da mesma forma que VBScript sem a opção `option explicit`. Felizmente, ao contrário de VBScript, Python não permite uma referência a uma variável que nunca teve um valor atribuído. A tentativa de fazer isso levanta uma exceção.

Example 2.26. Referência a variável não atribuída

```
>>> x
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
NameError: There is no variable named 'x'
>>> x = 1
>>> x
1
```

Voce vai agradecer ao Python por isso.

Leitura recomendada

- *Python Reference Manual* mostra exemplos de quando você pode deixar de usar o caracter continuador e quando você é obrigado a usar.

2.11. Atribuindo múltiplos valores de uma vez

Um dos atalhos mais interessantes em Python é usar seqüências para atribuir múltiplos valores de uma

só vez.

Example 2.27. Atribuindo múltiplos valores de uma vez

```
>>> v = ('a', 'b', 'e')
>>> (x, y, z) = v ❶
>>> x
'a'
>>> y
'b'
>>> z
'e'
```

❶ `v` é uma tupla de três elementos, e `(x, y, z)` é uma tupla de três variáveis. Atribuir um ao outro atribui os valores de `v` a cada uma das variáveis, em ordem.

Isso tem várias utilidades. Eu freqüentemente tenho que atribuir nomes a uma seqüencia de valores. Em C, é possível usar `enum` e listar manualmente cada constante e seu valor associado, o que é particularmente entendiante quando os valores são consecutivos. Em Python, é possível usar a função embutida `range` com atribuições múltiplas para rapidamente atribuir valores consecutivos

Example 2.28. Assigning consecutive values

```
>>> range(7)                                                                    ❶
[0, 1, 2, 3, 4, 5, 6]
>>> (MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY) = range(7) ❷
>>> MONDAY                                                                      ❸
0
>>> TUESDAY
1
>>> SUNDAY
6
```

❶ A função embutida `range` retorna uma lista de inteiros. Em sua forma mais simples, ela recebe um limite superior como parâmetro e retorna uma lista iniciada em `0` com todos os inteiros até o valor do parâmetro, exclusive. (Se quiser, é possível passar mais parâmetros para especificar um valor inicial diferente de `0` e um incrementador diferente de `1`. Voce pode usar `print range.__doc__` para ver detalhes.)

❷ `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY` e `SUNDAY` são variáveis que estamos definindo. (Esse exemplo vem do módulo `calendar`, um módulo interessante que imprime calendários da mesma forma que o comando UNIX `cal`. O módulo `calendar` define constantes inteiras para os dias da semana.)

❸ Agora cada variável tem seu valor: `MONDAY` vale `0`, `TUESDAY` vale `1`, etc.

É possível usar atribuições de múltiplos valores para construir funções que retornam múltiplos valores, simplesmente retornando uma tupla com todos os valores. Quem chama a função pode tratá-la como uma tupla, ou atribuir valores a variáveis individuais. Várias bibliotecas Python padrão fazem isso, inclusive o módulo `os`, que discutiremos no capítulo 3.

Leitura recomendada

- *How to Think Like a Computer Scientist* mostra como usar atribuições de múltiplos valores para trocar os valores de duas variáveis.

2.12. Formatando strings

Python formata valores em strings. Embora isso possa incluir expressões bastantes complicadas, o uso mais básico é inserir valores em uma string com usando `%s`.

 Formatação de strings em Python usa a mesma sintaxe que a função `sprintf` em C.

Example 2.29. Apresentando formatação de strings

```
>>> k = "uid"
>>> v = "sa"
>>> "%s=%s" % (k, v) ❶
'uid=sa'
```

❶ A expressão toda retorna uma string. O primeiro `%s` é substituído pelo valor de `k`; o segundo `%s` é substituído pelo valor de `v`. Todos os demais caracteres da string (nesse caso, o sinal de igual) ficam como estão.

Note que `(k, v)` é uma tupla. Eu disse que elas serviam pra algo.

Você pode estar achando que isso é trabalho demais para fazer uma simples concatenação de strings, e você está certo, exceto que formatação de strings não é apenas concatenação. Na verdade, não é nem apenas formatação, é também uma forma de coerção de tipo.

Example 2.30. Formatação de strings versus concatenação

```
>>> uid = "sa"
>>> pwd = "secret"
>>> print pwd + " is not a good password for " + uid      ❶
secret is not a good password for sa
>>> print "%s is not a good password for %s" % (pwd, uid) ❷
secret is not a good password for sa
>>> userCount = 6
>>> print "Users connected: %d" % (userCount, )           ❸ ❹
Users connected: 6
>>> print "Users connected: " + userCount                 ❺
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
TypeError: cannot add type "int" to string
```

❶ + é o operador de concatenação de strings.

❷ Nesse caso trivial, a formatação de strings tem o mesmo resultado que uma concatenação.

❸ `(userCount, )` é uma tupla com um elemento. Sim, a sintaxe é estranha, mas há uma boa razão para isso: é, sem ambigüidade, uma tupla. Na verdade, você sempre pode incluir uma vírgula após o último elemento quando define uma lista, tupla ou dicionário, mas a vírgula é obrigatória em uma tupla com apenas um elemento. Se a vírgula não fosse obrigatória, o Python não saberia se `(userCount)` é uma tupla com um elemento ou apenas o valor de `userCount`.

❹ Formatação de strings funciona com inteiros especificando `%d` ao invés de `%s`.

❺ Tentar concatenar uma string com uma não-string levanta uma exceção. Ao contrário da formatação de strings, a concatenação só funciona quando tudo já for string.

Leitura recomendada

- *Python Library Reference* sumariza todos os caracteres de formatação de strings.
- *Effective AWK Programming* discute todos os caracteres de formato e técnicas avançadas de formatação de strings como a especificação de tamanho, precisão e complemento com zeros..

2.13. Mapeando listas

Um dos recursos mais poderosos de Python é a compreensão de listas ("list comprehension"), que é uma forma compacta de mapear uma lista com outra sem aplicar uma função a cada elemento das lista.

Example 2.31. Apresentando compreensão de listas

```
>>> li = [1, 9, 8, 4]
>>> [elem*2 for elem in li]      ❶
[2, 18, 16, 8]
>>> li                           ❷
[1, 9, 8, 4]
>>> li = [elem*2 for elem in li] ❸
>>> li
[2, 18, 16, 8]
```

❶ Para entender o que isso significa, leia da direita para a esquerda. `li` é a lista que você está mapeando. Python itera pelos elementos de `li` um a um, temorariamente atribuindo o valor de outro elemento à variável `elem`. Python então aplica a expressão `elem*2` e adiciona o resultado à lista retornada.

❷ Note que compreensões de lista não mudam a lista original.

❸ É seguro atribuir o resultado da compreensão de lista à variável que você está mapeando. Não há race conditions FIXME ou qualquer esquisitisse com que se preocupar; a nova lista é construída em memória, e quando a compreensão estiver completa, atribui o resultado à variável.

Example 2.32. Compreensões de lista em `buildConnectionString`

```
["%s=%s" % (k, v) for k, v in params.items()]
```

Primeiramente, note que estamos chamando a função `items` do dicionário `params`. Essa função retorna uma lista de tuplas de todos os dados no dicionário..

Example 2.33. `keys`, `values` e `items`

```
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa",
"pwd":"secret"}
>>> params.keys()    ❶
['server', 'uid', 'database', 'pwd']
>>> params.values() ❷
['mpilgrim', 'sa', 'master', 'secret']
>>> params.items()  ❸
[('server', 'mpilgrim'), ('uid', 'sa'), ('database', 'master'), ('pwd',
'secret')]
```

❶ O método `keys` de um dicionário retorna uma lista de todas as chaves. A lista não está na ordem em que o dicionário for definido (lembre-se, elementos em um dicionário não têm ordem), mas é uma lista.

❷ O método `values` retorna uma lista de todos os valores. A lista está na mesma ordem que a lista retornada por `keys`, portanto `params.values()[n] == params[params.keys()[n]]` para todos os valores de `n`.

❸ O método `items` retorna uma tupla no formato `(chave, valor)`. A lista contém todos os dados no dicionário.

Agora vamos ver o que `buildConnectionString` faz. Ela toma uma lista, `params.items()`, e a mapeia a uma nova lista aplicanfo formatação de string a cada elemento. A nova lista tem o mesmo número de elementos que `params.items()`, mas cada elemento na nova lista será uma string que contém tanto uma chave quanto seu valor associado no dicionário `params`.

Example 2.34. Compreensão de listas em `buildConnectionString`, passo a passo

```
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa",
"pwd":"secret"}
>>> params.items()
[('server', 'mpilgrim'), ('uid', 'sa'), ('database', 'master'), ('pwd',
'secret')]
>>> [k for k, v in params.items()]                    ❶
['server', 'uid', 'database', 'pwd']
>>> [v for k, v in params.items()]                    ❷
['mpilgrim', 'sa', 'master', 'secret']
>>> ["%s=%s" % (k, v) for k, v in params.items()]     ❸
['server=mpilgrim', 'uid=sa', 'database=master', 'pwd=secret']
```

❶ Note que estamos usando duas variáveis para iterar pela lista `params.items()`. Esse é mais um caso do uso de atribuição múltipla. O primeiro elemento de `params.items()` é `('server', 'mpilgrim')`, então na primeira iteração da compreensão, `k` receberá `'server'` e `v` receberá `'mpilgrim'`. Nesse caso estamos ignorando o valor de `v` e incluindo apenas o valor de `k` na lista retornada, então essa compreensão acaba sendo equivalente a `params.keys()`. (Você não precisaria usar uma compreensão como essa em código de produção. Esse é um exemplo simples para aprendermos o que está acontecendo aqui.)

❷ Aqui fazemos a mesma coisa, ignorando o valor de `k`, portanto essa compreensão de lista acaba sendo equivalente a `params.values()`.

❸ Combinando os exemplos anteriores com alguma formatação de string, conseguimos uma lista de strings que inclui tanto a chave e quanto o valor de cada elemento do dicionário. Isso se parece muito com a saída do programa; tudo que resta é juntar os elementos dessa lista em uma única string.

Leitura recomendada

- *Python Tutorial* discute outra forma de mapear listas usando a função interna `map`.
- *Python Tutorial* mostra fazer compreensões de listas dentro de compreensões de listas.

2.14. Unindo listas e separando strings

Você tem uma lista de pares chave-valor no formato *`chave=valor`* e você quer uni-los em uma única string. Para juntar qualquer lista de strings, use o método `join` do objeto string.

Example 2.35. Unindo a lista em `buildConnectionString`

```
    return ";".join(["%s=%s" % (k, v) for k, v in params.items()])
```

Uma nota interessante antes de continuarmos. Eu repito o tempo todo que funções são objetos, strings são objetos, tudo é um objeto. Você pode ter entendido que as *variáveis* que contêm strings são objetos. Mas examine cuidadosamente o próximo exemplo e você verá que a própria string `";"` é um objeto, e você estará chamando seu método `join`.

Continuando, o método `join` une os elementos de uma lista em uma única string, com cada elemento separado por ponto-e-vírgula. O delimitador não precisa ser um ponto-e-vírgula, e nem sequer precisa ser apenas um caracter. Qualquer string serve.

`join` funciona apenas com listas de strings, e não faz nenhum outro tipo de coerção. Tentar unir com `join` uma lista que tem um ou mais elementos não-string levanta uma exceção.

Example 2.36. Saída de `odbchelper.py`

```
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa",
"pwd":"secret"}
>>> ["%s=%s" % (k, v) for k, v in params.items()]
['server=mpilgrim', 'uid=sa', 'database=master', 'pwd=secret']
>>> ";".join(["%s=%s" % (k, v) for k, v in params.items()])
'server=mpilgrim;uid=sa;database=master;pwd=secret'
```

Essa string é então retornada da função `odbchelper` e impressa pelo bloco que a chamou, que por sua vez imprime a saída que deixou você maravilhado no início desse capítulo.

Nota histórica. Quando aprendi Python, eu esperava que `join` fosse um método de uma lista e que tomaria o delimitador como argumento. Muitas pessoas acham isso, mas há uma história que justifica o método `join`. Antes do Python 1.6, strings não tinham esses métodos todos. Havia um módulo `string` separado que continha todas as funções de strings, e todas essas funções tomavam uma string como primeiro argumento. Essas funções foram consideradas importantes o suficiente para serem colcadas dentro das próprias strings, o que fazia sentido para funções como `lower`, `upper` e `split`. Mas muitos fãs tradicionais do Python se opuseram ao método `join`, argumentando que ele deveria ser um método das listas, ou pelo menos que não deveria sair do módulo `string` (que ainda tem muita coisa útil). Eu uso o novo método `join` exclusivamente, mas você pode ver código escrito das duas formas, e caso essa forma o incomode, use a funçao antiga `string.join`.

Você deve estar se pergutando se há uma forma análoga de dividir uma string em uma lista. E claro que há, e se chama `split`.

Example 2.37. Dividindo uma string

```
>>> li = ['server=mpilgrim', 'uid=sa', 'database=master', 'pwd=secret']
>>> s = ";".join(li)
>>> s
'server=mpilgrim;uid=sa;database=master;pwd=secret'
```

```
>>> s.split(";")    ❶
['server=mpilgrim', 'uid=sa', 'database=master', 'pwd=secret']
>>> s.split(";", 1) ❷
['server=mpilgrim', 'uid=sa;database=master;pwd=secret']
```

❶ `split` reverte `join`, pois divide uma string em uma lista de vários elementos. Note que o delimitador (“`;`”) foi removido totalmente, e não faz parte dos elementos da lista retornada.

❷ `split` aceita um segundo argumento opcional, a quantidade máxima de elementos da lista. (“"Oh, argumentos opcionais...” Você vai aprender como fazer isso em suas próprias funções no próximo capítulo.)

 *`qualquerstring`*`.split(`*`delimitador`*`, 1)` é uma técnica útil quando você quer procurar uma string em outra e então trabalhar com tudo antes do texto encontrado (que fica no primeiro elemento da lista retornada) e tudo depois do texto encontrado (que fica depois do segundo elemento). FIXME rever estilo.

Leitura recomendada

- Python Knowledge Base responde perguntas comuns sobre strings e tem muitos exemplos de código usando strings.
- *Python Library Reference* sumariza todos os métodos de strings.
- *Python Library Reference* documenta o módulo `string`.
- *The Whole Python FAQ* explica porque `join` é um método de string e não um método de lista.

2.15. Resumo

O programa `odbchelper.py` e sua saída fazem total sentido.

Example 2.38. `odbchelper.py`

```
def buildConnectionString(params):
    """Build a connection string from a dictionary of parameters.

    Returns string."""
    return ";".join(["%s=%s" % (k, v) for k, v in params.items()])

if __name__ == "__main__":
    myParams = {"server":"mpilgrim", \
                "database":"master", \
                "uid":"sa", \
                "pwd":"secret" \
                }
    print buildConnectionString(myParams)
```

Example 2.39. Saída de `odbchelper.py`

```
server=mpilgrim;uid=sa;database=master;pwd=secret
```

Antes de mergulhar no próximo capítulo, certifique-se de que você faz confortavelmente as seguintes tarefas:

- Usar o IDE Python para testar expressões interativamente

- Escrever módulos de forma que também possam ser usados como programas auto-suficientes, pelo menos para fins de teste
- Importar módulos e chamar suas funções
- Declarar funções usando `doc_strings`, variáveis locais e indentação correta
- Definir dicionários, tuplas e listas
- Acessar atributos e métodos de qualquer objeto, inclusive strings, listas, dicionários, funções e módulos
- Concatenar valores com formatação de strings
- Mapear listas em outras listas usando compreensões de listas
- Dividir strings em listas e unir listas em strings

Chapter 3. O poder da introspecção

3.1. Mergulhando

O presente capítulo cobre um dos pontos fortes do Python: introspecção. Como você deve saber, tudo em Python é um objeto. A introspecção é a capacidade de fazer código que examina e manipula módulos e funções em memória como objetos. A partir daqui, definiremos funções sem nomes, chamaremos funções com argumentos fora de ordem e faremos referência funções cujos nomes não conhecemos previamente.

Aqui temos um programa Python completo e funcional. Você deve entender boa parte dele. As linhas numeradas ilustram conceitos cobertos em *Conhecendo o Python*. Não se preocupe se o resto parecer intimidador; você vai aprender tudo isso no decorrer desse capítulo.

Example 3.1. `apihelper.py`

If you have not already done so, you can download this and other examples used in this book.

```
def help(object, spacing=10, collapse=1): ❶ ❷ ❸
    """Print methods and doc strings.

    Takes module, class, list, dictionary, or string."""
    methodList = [method for method in dir(object) if callable(getattr(object,
method))]
    processFunc = collapse and (lambda s: " ".join(s.split())) or (lambda s: s)
    print "\n".join(["%s %s" %
                     (method.ljust(spacing),
                      processFunc(str(getattr(object, method).__doc__)))
                     for method in methodList])

if __name__ == "__main__":                    ❹ ❺
```

```
    print help.__doc__
```

❶ Este módulo tem apenas uma função, `help`. De acordo com sua declaração, ela aceita três parâmetros: `objectv`, `spacing` e `collapse`. Os dois últimos são na verdade opcionais, como veremos em breve.

❷ A função `help` tem uma `doc string` multi-linhas que sucintamente descreve seu propósito. Note que nenhum valor de retorno é mencionado: esta função será usada apenas pelos seus efeitos, não por seu valor de retorno.

❸ O código dentro da função está indentado.

❹ O truque `if __name__` permite que o programa faça algo útil quando executado sozinho, sem impedir seu uso como um módulo por outros programas. Nesse caso, o programa simplesmente imprime a `doc string` da função `help`.

❺ Comandos `if` usam == para comparação, e parênteses não são obrigatórios.

A função `help` foi projetada para ser usada por você, programador, enquanto trabaliha no Python IDE. Ela aceita qualquer objeto que tem funções ou métodos (como um módulo, que tem funções, ou uma lista, que tem métodos) e imprime as funções e as `doc strings`.

Example 3.2. Exemplo de uso de `apihelper.py`

```
>>> from apihelper import help
>>> li = []
>>> help(li)
append     L.append(object) -- append object to end
count      L.count(value) -> integer -- return number of occurrences of value
extend     L.extend(list) -- extend list by appending list elements
index      L.index(value) -> integer -- return index of first occurrence of value
insert     L.insert(index, object) -- insert object before index
pop        L.pop([index]) -> item -- remove and return item at index (default
last)
remove     L.remove(value) -- remove first occurrence of value
reverse    L.reverse() -- reverse *IN PLACE*
sort       L.sort([cmpfunc]) -- sort *IN PLACE*; if given, cmpfunc(x, y) -> -1,
0, 1
```

Por padrão a saída é formatada para ser legível. `doc strings` multi-linhas são transformadas em uma linha longa. Esse comportamento pode ser alterado especificando `0` como valor do argumento *`collapse`*. Se os nomes de função forem mais longos que 10 caracteres, você pode especificar um valor maior para o parâmetro *`spacing`* para deixar a saída mais fácil de ler.

Example 3.3. Uso avançado de `apihelper.py`

```
>>> import odbchelper
>>> help(odbchelper)
buildConnectionString Build a connection string from a dictionary Returns string.
>>> help(odbchelper, 30)
buildConnectionString          Build a connection string from a dictionary
Returns string.
>>> help(odbchelper, 30, 0)
buildConnectionString          Build a connection string from a dictionary

    Returns string.
```

3.2. Argumentos opcionais e nomeados

Python suporta valores padrão para argumentos de funções. Se a função for chamada sem o argumento, o valor padrão é usado. Além disso, argumentos podem ser especificados em qualquer ordem usando argumentos nomeados. Procedimentos embutidos em Transact/SQL podem fazer isso também. Se você conhecer SQL Server pode passar rapidamente por essa parte.

Example 3.4. `help`, uma função com dois argumentos opcionais

```
def help(object, spacing=10, collapse=1):
```

`spacing` e `collapse` são opcionais, uma vez que seus valores padrão estão definidos. `object` é obrigatório, pois nenhum valor padrão foi definido. Se `help` for chamado com apenas um argumento, `spacing` será `10` e `collapse` será `1`. Se `help` for chamado com dois argumentos, `collapse` será por padrão `1`.

Suponha que você queira especificar o valor de `collapse` mas queira aceitar o valor padrão de `spacing`. Na maioria das outras linguagens isso não é possível pois seria obrigatório passar três parâmetros. Mas em Python argumentos podem ser especificados pelo nome, em qualquer ordem.

Example 3.5. Chamadas válidas de `help`

```
help(odbchelper)                    ❶
help(odbchelper, 12)                ❷
help(odbchelper, collapse=0)        ❸
help(spacing=15, object=odbchelper) ❹
```

❶ Com apenas um argumento, `spacing` fica com o valor padrão `10` e `collapse` fica com o valor padrão `1`.

❷ Com dois argumentos, `collapse` fica com o valor padrão `1`.

❸ Aqui você está nomeando o argumento `collapse` explicitamente e especificando seu valor. `spacing` continua com o valor padrão `10`.

❹ Mesmo argumentos obrigatórios (como `object`, que não tem um valor padrão) podem ser nomeados, e os argumentos nomeados podem aparecer em qualquer ordem.

Isso parece particularmente estranho até você perceber que os argumentos são simplesmente um dicionário. O método “normal” de chamar funções sem nomes de argumentos é apenas um atalho em que o Python associa os valores com os argumentos na ordem que você especificou na declaração. E na maior parte do tempo, as funções serão chamadas do jeito “normal”, e a flexibilidade adicional estará lá quando você precisar.


A única coisa que você ŕecisa fazer para chamar uma função é especificar um valor (de qualquer forma) para cada argumento obrigatório. A maneira e a ordem você que escolhe.

Leitura adicional

- *Python Tutorial* discute quando exatamente e como argumentos padrão são avaliados, o que importa quando o valor padrão for uma lista ou uma expressão com efeitos colaterais ("side effects").

3.3. `type`, `str`, `dir`, e outras funções internas

Python tem um pequeno conjunto de funções extremamente úteis. Todas as outras funções são particionadas em módulos. Isso foi uma decisão de projeto consciente, de forma a manter o núcleo linguagem livre dos excessos de outras linguagens de script (cof cof, Visual Basic).

A função `type` retorna o tipo de um objeto arbitrário. Os tipos possíveis são listados no módulo `types`. Esse recurso é útil em funções auxiliares que podem manipular diversos tipos de dados.

Example 3.6. Introduzindo `type`

```
>>> type(1)                    ❶
<type 'int'>
>>> li = []
>>> type(li)                   ❷
<type 'list'>
>>> import odbchelper
>>> type(odbchelper)           ❸
<type 'module'>
>>> import types               ❹
>>> type(odbchelper) == types.ModuleType
1
```

❶ `type` recebe qualquer coisa como valor e retorna o tipo. E eu realmente quis dizer qualquer coisa: inteiros, stri funções, classes, módulos, até mesmo tipos.

❷ `type` recebe uma variável e retorna seu tipo.

❸ `type` também funciona com módulos.

❹ É possível usar as constantes no módulo `types` para comparar tipos de objetos. É isso que a função `help` faz,

`str` pega dados e retorna como string. Todos os tipos podem ser transformados em string.

Example 3.7. Introduzindo `str`

```
>>> str(1)                     ❶
'1'
>>> horsemen = ['war', 'pestilence', 'famine']
>>> horsemen.append('Powerbuilder')
>>> str(horsemen)              ❷
"['war', 'pestilence', 'famine', 'Powerbuilder']"
>>> str(odbchelper)            ❸
"<module 'odbchelper' from 'c:\\docbook\\dip\\py\\odbchelper.py'>"
>>> str(None)                  ❹
'None'
```

❶ Para tipos de dados simples como inteiros, seria normal esperar que `str` funcionaria, pois virtualmente todas as converter inteiros em strings.

❷ `str`, porém, funciona com qualquer tipo. Aqui funciona com uma lista que construimos em pedaços.

❸ `str` também funciona com módulos. Note que a representação como string de um módulo inclui o caminho do resultado será diferente.

❹ Um comportamento sutil mas importante de `str` é o fato de funcionar com `None`, o valor nulo de Python. O re usar isso a nosso favor na função `help` como veremos em breve.

No coração de nossa função `help` está a poderosa função `dir`. `dir` retorna uma lista dos atributos e métodos de qualquer objeto: módulos, funções, listas, strings, dicionários... virtualmente tudo.

Example 3.8. Introduzindo `dir`

```
>>> li = []
>>> dir(li)                    ❶
['append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse',
'sort']
>>> d = {}
>>> dir(d)                     ❷
['clear', 'copy', 'get', 'has_key', 'items', 'keys', 'setdefault', 'update',
'values']
>>> import odbchelper
>>> dir(odbchelper)            ❸
['__builtins__', '__doc__', '__file__', '__name__', 'buildConnectionString']
```

❶ `li` é uma lista, portanto `dir(li)` retorna uma lista de todos os métodos da lista. Note que a lista retornada co strings, não os próprios métodos.

❷ `d` é um dicionário, portanto `dir(d)` retorna uma lista de todos os nomes dos métodos de dicionários. Pelo mer familiar.

❸ É aqui que as coisas ficam interessantes. `odbchelper` é um módulo, portanto `dir(odbchelper)` retorna definido nesse módulo, inclusive atributos internos como `__name__` e `__doc__`, e quaisquer outros atributos caso, `odbchelper` tem apenas um método definido pelo usuário, a função `buildConnectionString` q *Python*.

Finalmente, a função `callable` recebe como parâmetro um objeto e retorna `1` se o objeto puder ser chamado ou `0` caso contrário. Objetos chamáveis incluem funções, métodos de classes e até mesmo as próprias classes. (Mais sobre classes no capítulo 3.)

Example 3.9. Introduzindo `callable`

```
>>> import string
>>> string.punctuation                ❶
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
>>> string.join                       ❷
<function join at 00C55A7C>
>>> callable(string.punctuation)      ❸
0
>>> callable(string.join)             ❹
1
>>> print string.join.__doc__         ❺
join(list [,sep]) -> string

    Return a string composed of the words in list, with
    intervening occurrences of sep.  The default separator is a
    single space.

    (joinfields and join are synonymous)
```

❶ As funções no módulo `string` são obsoletas (embora muita gente ainda use a função `join`), mas o módulo c `string.punctuation`, que contém todos os caracteres de pontuação.

❷ `string.join` é uma função que une uma lista de strings.

❸ `string.punctuation` não é chamável FIXME, e sim uma string. (Uma string tem objetos chamáveis, mas

❹ `string.join` é chamável; é uma função que aceita dois argumentos.

❺ Qualquer objeto chamável pode ter uma `doc string`. Usando a função `callable` em cada um dos atributo com quais atributos nos importamos (métodos, funções, classes) e quais queremos ignorar (contantes, *etc.*) sem previamente.

`type`, `str`, `dir` e todas as outras funções internas do Python estão agrupadas no módulo

`__builtin__`. (O nome tem dois sublinhados no começo e dois no fim.) Se ajudar, pense que Python automaticamente executa `from __builtin__ import *` ao iniciar, o que importa todas as funções desse módulo no escopo atual para que possam ser usadas diretamente.

A vantagem de pensar dessa forma é que é possível acessar todas as funções e atributos internos examinando o módulo `__builtin__`. E adivinhe só, nós temos uma função para fazer isso: ela se chama `help`. Tente fazer isso e examine a lista agora; nós mergulharemos em algumas das funções mais importantes mais tarde. (algumas das classes de erro internas, como `AttributeError`, devem ser familiares.)

Example 3.10. Funções e atributos internos

```
>>> from apihelper import help
>>> import __builtin__
>>> help(__builtin__, 20)
ArithmeticError      Base class for arithmetic errors.
AssertionError       Assertion failed.
AttributeError       Attribute not found.
EOFError             Read beyond end of file.
EnvironmentError     Base class for I/O related errors.
Exception            Common base class for all exceptions.
FloatingPointError   Floating point operation failed.
IOError              I/O operation failed.

[...corte...]
```


Python vem com manuais de referência excelentes, que você deve usar largamente para conhecer todos os mó
enquanto em outras linguagens você ficaria voltando o tempo todo aos manuais (ou man pages, ou, Deus te li
usá-los, Python é na maior parte auto-documentado.

Leitura recomendada

- *Python Library Reference* documenta todas as funções internas e todas as exceções internas.

3.4. Extraindo referências a objetos com `getattr`

Você já sabe que funções são objetos. O que você não sabe ainda é que é possível fazer referência a uma função sem saber seu nome até o tempo de execução, usando a função `getattr`.

Example 3.11. Apresentando `getattr`

```
>>> li = ["Larry", "Curly"]
>>> li.pop                          ❶
<built-in method pop of list object at 010DF884>
>>> getattr(li, "pop")              ❷
<built-in method pop of list object at 010DF884>
>>> getattr(li, "append")("Moe")    ❸
>>> li
["Larry", "Curly", "Moe"]
>>> getattr({}, "clear")            ❹
<built-in method clear of dictionary object at 00F113D4>
>>> getattr((), "pop")              ❺
Traceback (innermost last):
```

```
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'pop'
```

❶ Aqui temos uma referência ao método `pop` da lista. Note que não estamos chamando o método `pop`, que seria feito com `li.pop()`. É uma referência ao próprio método.

❷ Isso também retorna uma referência ao método `pop`, mas dessa vez, o nome do método é especificado como argumento da função `getattr`. `getattr` é uma função interna muito útil que retorna qualquer atributo de qualquer objeto. Nesse caso, o objeto é uma lista, e o atributo é o método `pop`

❸ Caso você não tenha percebido como isso é útil, tente isso: o valor de retorno de `getattr` *é* o método, que você pode então chamar como se tivesse usado `li.append("Moe")` diretamente. Mas você não chamou a função diretamente: você especificou o nome da função como uma string.

❹ `getattr` também funciona com dicionários.

❺ Em tese, `getattr` deveria funcionar com tuplas, exceto que tuplas não têm métodos, portanto `getattr` levanta uma exceção não importa que nome de atributo você use.

`getattr` não é apenas para tipos embutidos. Também funciona com módulos.

Example 3.12. `getattr` in `apihelper.py`

```
>>> import odbchelper
>>> odbchelper.buildConnectionString                    ❶
<function buildConnectionString at 00D18DD4>
>>> getattr(odbchelper, "buildConnectionString")        ❷
<function buildConnectionString at 00D18DD4>
>>> object = odbchelper
>>> method = "buildConnectionString"
>>> getattr(object, method)                             ❸
<function buildConnectionString at 00D18DD4>
>>> type(getattr(object, method))                       ❹
<type 'function'>
>>> import types
>>> type(getattr(object, method)) == types.FunctionType
1
>>> callable(getattr(object, method))                   ❺
1
```

❶ Isso retorna uma referência à função `buildConnectionString` do módulo `odbchelper` que estudamos em *Conhecendo o Python*. (O endereço hexadecimal é específico da minha máquina. O seu será diferente.)

❷ Usando `getattr`, podemos pegar uma referência à mesma função. Geralmente, `getattr(objeto, "atributo")` é equivalente a *`objeto.atributo`*. Se *`objeto`* for um módulo, *`atributo`* pode ser qualquer coisa definida no módulo: uma função, classe ou variável global.

❸ E é isso que usamos na função `help`. `object` é passado para a função como um argumento, e `method` é uma string que é o nome de um método ou de uma função.

❹ Nesse caso, `method` é o nome de uma função, o que podemos provar verificando seu tipo.

❺ Como `method` é uma função, é chamável.

3.5. Filtrando listas

Como você sabe, Python tem recursos poderosos para mapear listas em outras listas usando compreensões. Isso pode ser combinado com um mecanismo de filtragem, em que alguns elementos da lista são mapeados e outros são ignorados.

Example 3.13. Sintaxe para filtragem de listas

```
[expressão-de-mapeamento for elemento in lista-de-origem if expressão-de-
filtragem]
```

Isso é uma extensão das compreensões de lista que você conhece e confia. As primeiras duas partes são iguais; a última parte, iniciada no `if`, é a expressão de filtragem. Uma expressão de filtragem pode ser qualquer expressão que avalie em verdadeiro ou falso (quem em Python pode ser quase qualquer coisa). Qualquer elemento para o qual a expressão de filtragem é verdadeira será incluído no mapeamento. Todos os demais serão ignorados, não sendo passados para a expressão de mapeamento e ficando fora da lista de saída.

Example 3.14. Introduzindo filtragem de listas

```
>>> li = ["a", "mpilgrim", "foo", "b", "c", "b", "d", "d"]
>>> [elem for elem in li if len(elem) > 1]       ❶
['mpilgrim', 'foo']
>>> [elem for elem in li if elem != "b"]         ❷
['a', 'mpilgrim', 'foo', 'c', 'd', 'd']
>>> [elem for elem in li if li.count(elem) == 1] ❸
['a', 'mpilgrim', 'foo', 'c']
```

❶ A expressão de mapeamento aqui é simples (apenas retorna o valor de cada elemento), portanto se concentre na expressão de filtragem. À medida em que o Python itera pelos elementos da lista, passa cada elemento por uma expressão de filtragem. Se a expressão de filtragem for verdadeira, o elemento é mapeado e o resultado da expressão de mapeamento é incluído na lista retornada. Aqui estamos filtrando fora todas as strings de um caracter, de forma que só sobram uma lista com as strings mais longas.

❷ Aqui estamos filtrando fora um valor específico, `b`. Note que isso filtra todas as ocorrências de `b`, uma vez que cada vez que ele aparece, a expressão se torna verdadeira.

❸ `count` é um método de lista que retorna o número de vezes que um valor ocorre em uma lista. Você pode pensar que esse filtro eliminaria as duplicatas da lista, retornando uma lista contendo apenas uma cópia de cada valor da lista original. Mas não é o caso, porque valores que aparecem duas vezes na lista original (nesse caso, `b` e `d`) são excluídos completamente. Há formas de eliminar valores duplicados em uma lista, mas não com filtragem.

Example 3.15. Filtrando uma lista em `apihelper.py`

```
    methodList = [method for method in dir(object) if callable(getattr(object,
method))]
```

Isso parece complicado, e é complicado, mas a estrutura básica é a mesma. A expressão de filtragem toda retorna uma lista, que é atribuída à variável `methodList`. A primeira metade da expressão é o mapeamento de lista. A expressão de mapeamento é uma expressão identidade: retorna o valor de cada elemento. `dir(object)` retorna uma lista dos métodos e atributos de `object`; essa é a lista

que estamos mapeando. Portanto a única parte nova é a expressão de filtragem após o `if`.

A expressão de filtragem parece assustadora, mas não é. Você já aprendeu sobre `callable`, `getattr` e `in`. Como vimos na seção anterior, a expressão `getattr(object, method)` retorna um objeto função se `object` for um módulo e `method` o nome de uma função do módulo.

Portanto essa expressão aceita um objeto (chamado `object`). Então pega uma lista dos nomes dos atributos do objeto, métodos, funções e algumas coisas mais. Então filtra a lista para remover tudo que não nos importa. Fazemos isso tomando o nome de cada atributo/método/função e pegando uma referência ao objeto real equivalente com o uso de `getattr`. Então verificamos se o objeto é chamável, o que inclui quaisquer métodos e funções, sejam embutidos (como o método `pop` em uma lista) ou definidos pelo usuário (como a função `buildConnectionString` do módulo `odbchelper`). Não nos importamos com outros atributos, como `__name__`, que faz parte de todos os módulos.

Leitura recomendada

- *Python Tutorial* discute outra forma de filtrar listas usando a função embutida `filter`.

3.6. A natureza peculiar de `and` e `or`

Em Python, `and` e `or` realizam lógica booleana da maneira como você esperaria, mas não retornam valores booleanos: eles retornam um dos valores que estão comparando.

Example 3.16. Apresentando `and`

```
>>> 'a' and 'b'          ❶
'b'
>>> '' and 'b'           ❷
''
>>> 'a' and 'b' and 'c'  ❸
'c'
```

❶ Quando usamos `and`, os valores são avaliados em um contexo booleano da esquerda para a direita. `0`, `''`, `[]`, `()`, `{}` e `None` são falsos em um contexo booleano; tudo mais é verdadeiro.[5] Se todos os valores forem verdadeiros em um contexto booleano, `and` retorna o último valor. Nesse caso, `and` avalia `'a'`, que é verdadeiro, então `'b'`, que é verdadeiro, e retorna `'b'`.

❷ Se qualquer valor for falso em um contexto booleano, `and` retorna o primeiro valor falso. Nesse caso `''` é o primeiro valor falso..

❸ Todos os valores são verdadeiros, portanto `and` retorna o último valor, `'c'`.

Example 3.17. Apresentando `or`

```
>>> 'a' or 'b'           ❶
'a'
>>> '' or 'b'            ❷
'b'
>>> '' or [] or {}       ❸
{}
>>> def sidefx():
...     print "in sidefx()"
```

```
...     return 1
>>> 'a' or sidefx()     ❹
'a'
```

❶ Quando usamos `or`, valores são avaliados em um contexto booleando da esquerda para a direita, como com `and`. Se algum valor for verdadeiro, `or` retorna esse valor imediatamente. Nesse caso, `'a'` é o primeiro valor verdadeiro.

❷ `or` avalia `''`, que é falso, então `'b'`, que é verdadeiro, e retorna `'b'`.

❸ Se todos os valores forem falsos, `or` retorna o último valor. `or` avalia `''`, que é falso, então `[]`, que é falso, então `{}`, wque é falso, e retorna `{}`.

❹ Note que `or` avalia valores apenas até achar algum que seja verdadeiro em um contexto booleano, e ignora o resto. Essa distinção é importante se alguns valores têm efeitos colaterais. Aqui, a função `sidefx` nunca é chamada, porque `or` avalia `'a'`, que é verdadeiro, e retorna `'a'` imediatamente.

Se você for um programador C, certamente conhece a expressão *`bool ? a : b`*, que avalia em `a` se *`bool`* for verdadeiro, e `b` caso contrário. Graças à forma como `and` e `or` funcionam em Python, você pode fazer a mesma coisa.

Example 3.18. Apresentando o truque `and-or`

```
>>> a = "first"
>>> b = "second"
>>> 1 and a or b ❶
'first'
>>> 0 and a or b ❷
'second'
```

❶ This syntax looks similar to the *`bool ? a : b`* expression in C. The entire expression is evaluated from left to right, so the `and` is evaluated first. `1 and 'first'` evalutes to `'first'`, then `'first' or 'second'` evalutes to `'first'`.

❷ `0 and 'first'` evalutes to `0`, then `0 or 'second'` evaluates to `'second'`.

However, since this Python expression is simply boolean logic, and not a special construct of the language, there is one very, very, very important difference between this `and-or` trick in Python and the *`bool ? a : b`* syntax in C. If the value of `a` is false, the expression will not work as you would expect it to. (Can you tell I was bitten by this? More than once?)

Example 3.19. When the `and-or` trick fails

```
>>> a = ""
>>> b = "second"
>>> 1 and a or b ❶
'second'
```

❶ Since `a` is an empty string, which Python considers false in a boolean context, `1 and ''` evalutes to `''`, then `'' or 'second'` evalutes to `'second'`. Oops! That's not what we wanted.


The `and-or` trick, *`bool and a or b`*, will not work like the C expression *`bool ? a : b`* when `a` is false in a boolean context.

The real trick behind the `and-or` trick, then, is to make sure that the value of `a` is never false. One common way of doing this is to turn `a` into `[a]` and `b` into `[b]`, then taking the first element of the returned list, which will be either `a` or `b`.

Example 3.20. Using the `and-or` trick safely

```
>>> a = ""
>>> b = "second"
>>> (1 and [a] or [b])[0] ❶
''
```

❶ Since `[a]` is a non-empty list, it is never false. Even if `a` is `0` or `''` or some other false value, the list `[a]` is true because it has one element.

By now, this trick may seem like more trouble than it's worth. You could, after all, accomplish the same thing with an `if` statement, so why go through all this fuss? Well, in many cases, you are choosing between two constant values, so you can use the simpler syntax and not worry, because you know that the `a` value will always be true. And even if you have to use the more complicated safe form, there are good reasons to do so; there are some cases in Python where `if` statements are not allowed, like `lambda` functions.

Further reading

- Python Cookbook discusses alternatives to the `and-or` trick.

Footnotes

[5] bem, quase tudo. Por padrão, instâncias de classes são verdadeiras em um contexto booleano, mas é possível definir métodos especiais em sua classe para fazer com que avaliem falso. Vamos aprender tudo sobre classes e métodos especiais no capítulo 3.