Introdução

Python é uma dessas raras linguagens em que se pode alegar ser ao mesmo tempo **simples** e **poderosa**. Você vai se surpreender ao descobrir o quanto é fácil se concentrar na solução para o problema em vez da sintaxe e estrutura da linguagem que você está programando.

A introdução oficial para Python é:

> Python é uma poderosa linguagem de programação fácil de aprender. Possui eficientes estruturas de dados de alto-nível e uma simples e efetiva abordagem para programação orientada a objetos. A elegante sintaxe e a tipagem dinâmica de Python, aliada a sua natureza interpretada, faz dela uma linguagem ideal para criar *scripts* e desenvolver aplicações de modo ágil em diversas áreas e na maioria das plataformas.

Vou detalhar a maioria dessas características na próxima seção.

Nota

> Guido van Rossum, criador da linguagem Python, cujo nome foi baseado na série "Monty Python's Flying Circus" da BBC. Ele, particularmente, não gosta de cobras que para se alimentar, esmagam outros animais usando seu extenso corpo.

Características de Python

Simples

> Python é uma linguagem simples e minimalista. Ler um bom programa em Python é quase como ler em inglês, ainda que seja um inglês bem restrito. Este caráter de pseudo-código do Python é um de seus maiores pontos fortes. Ele permite que você se concentre na solução do problema e não na linguagem em si.

Fácil de Aprender

> Como você verá, é extremamente fácil iniciar-se em Python. A linguagem possui uma sintaxe extraordinariamente simples, como já mencionado.

Livre e de Código Aberto

> Python é um exemplo de FLOSS (Free/Libre and Open Source Software. Traduzindo, você pode distribuir livremente cópias deste software, ler seu código-fonte, modificá-lo, usar trechos em novos programas livres e tudo o que você quiser fazer. FLOSS é baseado no conceito de uma comunidade que compartilha conhecimento. Este é um dos motivos pelos quais Python é tão bom - ele vem sendo criado e constantemente melhorado por uma comunidade que simplesmente quer ver a Python cada vez melhor.

Linguagem de Alto Nível

> Quando você escreve programas em Python, não há necessidade de se preocupar com detalhes de baixo nível tais como manipular a memória utilizada pelo programa, etc.

Portável

> Por ser uma linguagem de código aberto, Python foi portada (ou seja, modificada para funcionar) em muitas plataformas. Todos os seus programas em Python podem rodar em qualquer uma destas plataformas sem precisar de mudanças, desde que você seja cuidadoso o suficiente para evitar usar características que dependam do sistema.
>
> Você pode usar Python em Linux, Windows, FreeBSD, Macintosh, Solaris, OS/2, Amiga, AROS, AS/$)), BeOS, OS/390, z/OS, Palm OS, QNX, VMS, Psion, Acorn RISC OS, VxWorks, PlayStation, Sharp Zaurus, Windows CE e até mesmo no PocketPC!

Interpretada

Isto requer uma pequena explicação

Um programa escrito em uma linguagem compilada como C ou C++ é convertido da linguagem de origem (C ou C++) para a linguagem que é falada por seu computador (código binário, ou seja, 0s e 1s) usando um compilador com vários flags e opções. Quando você roda o programa, o software linkeditor/carregador copia o programa do disco rígido para a memória e começa a executá-lo.

Python, por sua vez, não necessita de compilação para código binário. Você simplemente executa o programa diretamente do código-fonte. Internamente, Python converte o código-fonte em um formato intermediário chamado bytecode, o traduz para a linguagem nativa do seu computador e então o executa. Tudo isso, na verdade, torna Python muito mais fácil, uma vez que você não precisa se preocupar com a compilação do programa, certificar-se que as bibliotecas necessárias estão todas linkeditadas e carregadas etc, etc. Isto também torna seus programas em Python muito mais portáveis, já que você pode simplesmente copiar seu programa em Python em outro computador e executá-lo!

Orientada a Objetos

Python suporta tanto programação procedural (estruturada) quanto orientada a objetos. Em linguagens procedurais, o programa é construído com base em procedimentos e funções, que nada mais são que trechos reutilizáveis de programas. Em linguagens orientadas a objeto, com base em objetos que combinam dados e funcionalidade. Python tem uma maneira simples mas poderosa de implementar programação orientada a objetos, especialmente quando comparada a linguagens como C++ ou Java.

Extensível

Se você precisa que um trecho crítico de código seja executado com muita rapidez ou quer que parte de algum algoritmo não seja aberta, você pode codificá-los em C ou C++ e então usá-los a partir de seu programa Python.

Embarcável

Você pode embarcar Pyhton em seus programas C/C++ para fornecer funcionalidades de scripting aos usuários dos programas.

Bibliotecas Extensivas

A Biblioteca Padrão do Python (Python Standard Library) é de fato enorme. Ela pode ajudá-lo a fazer várias coisas envolvendo expressões regulares, geração de documentação, testes unitários, threading, acesso a bancos de dados, browsers web, CGi, FTP, e-mail, XML, XML-RPC, HTML, arquivos WAV, criptografia, GUI (Interfaces Gráficas com o Usuário), Tk e outras coisas dependentes do sistema. Lembre-se de que tudo isto está disponível onde quer que Python esteja instalado. Esta é a chamada filosofia das "Pilhas Incluídas" do Python.

Além da biblioteca padrão, existem várias outras bibliotecas de alta qualidade, tais como wxPython , Twisted, Python Imaging Library e muitas outras.

Python é, de fato, uma linguagem poderosa e empolgante. Ela possui a combinação exata de desempenho e características que fazem escrever programas em Python uma tarefa fácil e divertida.

Por que não Perl?

Se você ainda não conhece, Perl é outra linguagem de programação interpretada e de código aberto extremamente popular.

Se você alguma vez tentou escrever um programa grande em Perl, você poderia ter respondido essa pergunta sozinho! Em outras palavras, programas em Perl são fáceis quando eles são pequenos e são excelentes para pequenos "hacks" e scripts "pra fazer o serviço". No entanto, eles se tornam rapidamente

difíceis de manter à medida em que se tornam maiores. E estou falando isso com base na minha experiência no Yahoo! escrevendo programas grandes em Perl.

Quando comparados a Perl, os programas em Python são definitivamente mais simples, claros, fáceis de escrever e, consequentemente, mais legíveis e fáceis de manter. Eu admiro Perl e uso diariamente para várias tarefas, mas toda vez que escrevo um programa, começo a pensar em termos de Python, porque se tornou natural para mim. Perl passou por tantos hacks e mudanças que parece que a linguagem toda é um grande (mas ótimo) hack. Infelizmente, o futuro Perl 6 não parece trazer nenhuma melhoria quanto a isso.

A única e bastante significativa vantagem do Perl, na minha opinião, é sua enorme biblioteca CPAN (Comprehensive Perl Archive Network). Como o nome sugere, é uma imensa coleção de módulos Perl e é simplesmente inacreditável, por causa de seu tamanho e profundidade - você pode fazer praticamente qualquer coisa que seja possível de se fazer com um computador usando estes módulos. Uma das razões pelas quais Perl tem mais bibliotecas que Python é o fato de existir há muito mais tempo. No entanto, isto parece estar mudando com o crescimento do Python Package Index.

Por que não Ruby?

Se você ainda não sabia, Ruby é uma outra popular linguagem de programação de código aberto.

Se você já gosta e usa Ruby, então eu definitivamente recomendo que continue a usá-la.

Para as outras pessoas que ainda não a usaram e estão tentando escolher entre aprender Python ou aprender Ruby, então eu recomendaria Python, unicamente pela perspectiva de facilidade de aprendizado. Pessoalmente, achei difícil compreender a linguagem Ruby, mas todas as pessoas que conseguem aprovam a beleza da linguagem. Infelizmente não tive tanta sorte.

O Que Os Programadores Dizem

É interessante ler o que grandes hackers como ESR têm a dizer sobre Python:

- **Eric S. Raymond** é o autor de 'A Catedral e o Bazar' e também é o indivíduo que cunhou o termo 'Open Source'. Ele diz que Python se tornou sua linguagem de programação favorita. Este artigo foi a real inspiração para meu primeiro contato com Python.
- **Bruce Eckel** é o autor dos famosos livros 'Thinking in Java' e 'Thinking in C++'. Ele diz que nenhuma outra linguagem o deixou tão produtivo quanto Python. Diz ainda que Python é talvez a única linguagem que se concentra em tornar as coisas mais fáceis para o programador. Leia a entrevista completa para mais detalhes.
- **Peter Norvig** é o conhecido autor do Lisp e Diretor de Qualidade de Busca no google (obrigado a Guido van Rossum por avisar). Ele diz que Python sempre foi parte integrante do Google. Esta afirmação pode ser comprovada verificando a página do Google Jobs que lista conhecimento em Python como requisito para os engenheiros de software.

Sobre Python 3.0

Python 3.0 é a nova versão da linguagem que será lançada em breve. Também é conhecida como Python 3000 ou Py3k.

A motivação principal para uma nova versão de Python é remover todos os pequenos problemas e detalhes que foram se acumulando no decorrer dos anos e tornar a linguagem ainda mais limpa.

Se você já tem muito código Python 2.x, então existe um utilitário para ajudar na conversão de código 2.x para 3.x.
Mais detalhes em:

- Introdução por Guido van Rossum
- O que há de novo no Python 2.6 (recursos significativamente diferentes das versões anteriores de Python 2.x e que provavelmente serão incluídos no Python 3.0)
- O que há de novo no Python 3.0
- Cronograma de lançamento do Python 2.6 e 3.0

- Python 3000 (a lista oficial de mudanças propostas)
- Planos diversos para Python 3.0
- Novidades do Python (lista detalhada das mudanças)

Python pt-br:Instalacao

Se você já tem o Python 2.x instalado, não é preciso removê-lo para instalar o Python 3.0. Você pode manter ambos instalados ao mesmo tempo.

Para usuários Linux e BSD

Se você está usando uma distribuição Linux como Ubuntu, Fedora, OpenSUSE ou {coloque sua escolha aquí}, ou um sistema BSD tal como FreeBSD, então é provável que você já tenha o Python instalado em seu sistema.
Para testar se o Python já está instalado em seu Linux, abra um shell (como`konsole` ou `gnome-terminal`) e então entre com o comando `python -V` como mostrado abaixo:

```
$ python -V


Python 3.0b1
```

Nota

> `$` é o prompt do shell. Ele será diferente para você dependendo das configurações do seu sistema operacional, portanto eu indicarei o prompt apenas pelo símbolo `$`.

Se você vê alguma informação sobre a versão, como mostrado acima, então o Python já está instalado. Contudo, se você obter uma mensagem como esta:

```
$ python -V


bash: Python: command not found
```

Então o Python não está instalado. Isto é altamente improvável, mas é possível.
Neste caso, você tem dois meios para instalar o Python em seu sistema:

- Você pode compilar o código fonte do Python e então instalá-lo. As instruções para a compilação são informadas no website do Python.
- Instalar os pacotes binários usando um gerenciador de pacotes que vem com o seu sistema operacional, tal como `apt-get` no Ubuntu/Debian e outros Linux baseados em Debian, `yum` no Fedora, `pkg_add` no FreeBSD, etc. Você precisará de uma conexão com a internet para usar este método. Uma alternativa é baixar os binários de algum outro lugar e então copiá-los e instalá-los em seu PC. [Esta opção estará disponível após o lançamento final do Python 3.0]

Para usuários Windows

Visite o site http://www.python.org/download/releases/3.0/ e baixe a última versão, que era a versão 3.0 beta 1, quando este livro foi escrito. Ele tem apenas 12.8 MB, o que é bem compacto quando comparado com a maioria das outras linguagens ou programas.

Cuidado

> Quando for dada a opção para você desmarcar componentes opcionais, não desmaque nenhum! Alguns destes componentes podem ser úteis para você, especialmente IDLE.

Um fato interessante é que usuários Windows são os que mais baixam os arquivos do Python. É claro que isso não fornece uma visão geral, uma vez que quase todos os usuários Linux já têm o Python instalado por padrão em seus sistema.

Prompt do DOS

Se você quer ser capaz de usar o Python da linha de comando do Windows, i.e. o prompt do DOS, então você precisa configurar a variável PATH corretamente.
Pra usuários Windows 2000, XP ou 2003, clique em `Painel de Controle` ->`Sistema` -> `Avançado` -> `Variáveis de ambiente`. Clique na variável chamada**PATH** na seção 'Variáveis de Sistema', então

selecione `Editar` e adicione ;C:\Python30 no fim do que já estiver lá. É claro que você deve usar o diretório correto onde você instalou o Python.
Para versões antigas do Windows, adicione as seguintes linhas no arquivo`C:\AUTOEXEC.BAT`: `'PATH=%PATH%;C:\Python30'` (sem as aspas) e reinicie o sistema. Em Windows NT use o arquivo `AUTOEXEC.NT`.

Para usuários MAC OS X

Usuários Mac OS X já encontrarão o Python instalado em seu sistema. Abra o`Terminal.app` e rode `python -V`, e siga os conselhos da seção para usuários Linux.

Sumário

Em um sistema Linux, é provável que você já tenha o Python instalado. Caso contrário, você pode instalá-lo usando o gerenciador de pacotes que vem com sua distribuição. Em um sistema Windows, instalar o Python é tão fácil como baixar o instalador e clicar duas vezes sobre ele. Daqui para frente nós assumiremos que você já está com o Python instalado em seu sistema.
A seguir nós escreveremos nosso primeiro programa em Python.

Python pt-br:Primeiros Passos

Introdução

Nós veremos agora como rodar o tradicional programa 'Olá Mundo' em Python. Isto ensinará a você como escrever, salvar e rodar programas no Python.
Há duas formas de usar o Python para rodar um programa - usando o prompt do interpretador interativo ou usando um arquivo fonte. Nós veremos como usar os dois métodos.

Usando o Prompt do Interpretador

Inicie o interpretador na linha de comando digitando `python` prompt do shell.
Em sistemas Windows, você pode rodar o interpretador na linha de comando se a variável `PATH` estiver corretamente configurada.
Se você está usando o IDLE, clique em `Iniciar` → `Programas` → `Python 3.0` →`IDLE (Python GUI)`.

Agora digite `print('Olá Mundo')` e em seguida pressione a tecla `Enter`. Você verá as palavras `Olá Mundo` como saída.

```
$ python

Python 3.0b2 (r30b2:65106, Jul 18 2008, 18:44:17) [MSC v.1500 32 bit (Intel)] on
win32

Type "help", "copyright", "credits" or "license" for more information.

>>> print('Olá Mundo')

Olá Mundo

>>>
```

Perceba que o Python mostra a saída da linha imediatamente! O que você acabou de digitar é uma simples *instrução*. Nós usamos `print` para imprimir qualquer valor que você forneça. Aqui, nós estamos fornecendo o texto `Olá Mundo` e este é imediatamente impresso na tela.

Como Sair do Interpretador

> Para sair do prompt, pressione `ctrl-d` se você está usando o IDLE ou o shell do Linux/BSD. No caso do prompt de comando do Windows, pressione `ctrl-z`seguido da tecla `enter`.

Escolhendo um Editor

Antes de começarmos a escrever programas em arquivos fonte, nós precisamos de um editor para escrever estes arquivos. A escolha de um editor é crucial. Você deve escolher o editor como se você fosse escolher um carro novo. Um bom editor o ajudará a escrever programas para Python facilmente, fazendo sua jornada mais confortável e te ajudando a chegar em seu destino (atingindo seu objetivo) de uma forma fácil e segura.

Uma das necessidades mais básicas é o **destaque de sintaxe** que torna colorida todas as diferentes partes de seu programa, de forma que você possa *ver* seu programa e visualizar seu funcionamento.

Se você está usando o Windows, eu sugiro que você use o IDLE. O IDLE destaca a sintaxe e permite que você rode seus programas dentro dele mesmo, além de fazer outras coisas. Uma observação especial: **Não use o Notepad** - é uma má escolha pois ele não destaca a sintaxe e também não tem suporte a indentação do texto, que é muito importante no nosso caso, como veremos adiante. Bons editores tais como o IDLE (e também o VIM) automaticamente o ajudarão a fazer isso.

Se você está usando Linux/FreeBSD, então você tem diversas escolhas para um editor. Se você é um programador experiente, então você já deve estar usando`Vim` ou `Emacs`. É desnecessário dizer que estes são dois dos mais poderosos editores e que você será beneficiado por usá-los para escrever seus programas em Python. Eu pessoalmente uso `Vim` para a maioria de meus programas. Se você é um programador iniciante, você pode usar o `Kate`, que é um dos meus favoritos. No caso de você ter tempo para aprender `Vim` ou `Emacs`, então eu realmente recomendo que você aprenda a usar um deles, pois será de grande utilidade para você em sua longa jornada.

Neste livro, nós usaremos o **IDLE**, nosso IDE (*Integrated Development Environment* - Ambiente Integrado de Desenvolvimento) como editor. O IDLE é instalado por padrão com os instaladores Python para Windows e Mac OS X. Ele também está disponível para instalação em Linux e BSDs nos seus respectivos repositórios. Nós exploraremos como usar o IDLE na próxima seção. Para mais detalhes, por favor veja a documentação do IDLE.

Se você ainda deseja explorar outras escolhas para um editor, veja a ampla lista de editores para Python e faça sua escolha. Você também pode escolher um IDE para Python. Veja a ampla lista de IDEs que suportam Python para mais detalhes. Uma vez que você passe a escrever grandes programas em Python, IDEs podem ser muito úteis.

Eu repito novamente, por favor escolha um editor apropriado - ele pode fazer a elaboração de programas em Python mais fácil e divertida.

Para usuários do Vim

> Há uma boa introdução sobre como tornar o Vim um poderoso IDE para Python por John M Anderson.

Para usuários do Emacs

> Há uma boa introdução sobre como tornar o Emacs um poderoso IDE para Python por Ryan McGuire.

Usando um Arquivo Fonte

Agora vamos voltar a programação. Há uma tradição de que quando você aprende uma nova linguagem de programação, o primeiro programa que você escreve e roda é o 'Olá Mundo' - tudo que ele faz é apenas dizer 'Olá Mundo' quando você o roda. Como Simon Cozens [1] diz, ele é uma 'tradicional invocação para os deuses da programação ajudarem você a aprender melhor a linguagem' :) .
Inicie o editor escolhido, digite o seguinte programa e salve-o como `olamundo.py`
Se você está usando o IDLE, clique sobre `File` → `New Window` e então entre com o seguinte programa. Então clique sobre `File` → `Save`.

```
#!/usr/bin/python
#Filename: olamundo.py
print('Olá Mundo')
```

Rode este programa abrindo o shell (terminal Linux ou prompt do DOS) e entre com o comando `python olamundo.py`.
Se você está usando o IDLE, use o menu `Run` → `Run Module` ou o atalho `F5`.
A saída é semelhante a seguinte:

```
$ python olamundo.py


Olá Mundo
```

Se você obter uma saída como acima, parabéns! - você rodou com sucesso seu primeiro programa para Python.
No caso de um erro, pro favor digite o programa acima *exatamente* como mostrado e rode o programa novamente. Note que Python é case-sensitive i.e.`print` não é o mesmo que `Print` - note o `p` minúsculo na primeira instrução e o `P`maiúsculo na segunda. Além disso, assegure-se que não haja espaços ou tabulações antes do primeiro carácter de cada linha - nós veremos por que isso é importante mais tarde.

Como ele Funciona

Vamos considerar as duas primeiras linhas do programa. Elas são chamadas de*comentários* - qualquer coisa a direita do símbolo `#` é um comentário e é usado principalmente como notas para o leitor do programa. Python não usa os comentário, exceto para o caso especial da primeira linha. Ela é chamada de *linha de organização* - sempre que os dois primeiros caracteres do arquivo fonte são `#!` seguidos pela localização de um programa, isto diz para seu sistema Linux/Unix que este programa deve ser rodado com este interpretador quando você *executar* o programa. Isto é explicado em detalhes na próxima seção. Note que você sempre pode rodar o programa em qualquer plataforma especificando o interpretador diretamente na linha de comando, como o comando`python olamundo.py` .

Importante

> Use comentário em seu programa de forma sensata, para explicar algum detalhe importante de seu programa - isto é útil para que os leitores de seu programa possam entender facilmente o que ele está fazendo. Lembre-se, está pessoa pode ser você mesmo depois de seis meses!

Os comentário são seguidos por uma *instrução* de Python. Aqui nós chamamos a*função* `print` que apenas imprime o texto `'Olá Mundo'`. Nós aprenderemos sobre funções em um capítulo posterior, o que você deve entender agora é que o que você colocar nos parênteses será impresso na tela. Neste caso, nós fornecemos`'Olá Mundo'` que se refere a uma string - não se preocupe, nós vamos explorar essas terminologias em detalhes mais tarde.

Programas Executáveis em Python

Isto funciona apenas para usuários Linux/Unix, mas usuários do Windows podem estar curiosos sobre a primeira linha do programa. Inicialmente, devemos dar ao programa a permissão para executar usando o comando `chmod` e então *rodar* o programa fonte.

```
$ chmod a+x olamundo.py

$ ./olamundo.py

Olá Mundo
```

O comando chmod é usado aqui para mudar ('ch'ange) o modo ('mod'e) do arquivo dando permissão para todos ('a'll) os usuário do sistema o executar (e'x'ecute). Então, nós executamos o programa diretamente especificando a localização do arquivo fonte. Nós usamos o `./` para indicar que o programa está no diretório atual.
Para tornar as coisas mais divertidas, você pode renomear o arquivo para`olamundo` e rodá-lo com `./olamundo` e ele ainda funcionará, uma vez que o sistema sabe que ele tem de rodar o programa usando o interpretador cuja localização é especificada na primeira linha do arquivo fonte.
Agora você é capaz de rodar o programa desde que você conheça o caminho exato para o arquivo fonte - mas e se você desejar rodar o arquivo de qualquer lugar? Você pode fazer isso armazenando o programa em um dos diretórios listados na variável de ambiente `PATH`. Sempre que você roda qualquer programa, o sistema procura por aquele programa em cada diretório listado na variável de ambiente `PATH`e então roda aquele programa. Nós podemos tornar este programa disponível em qualquer lugar simplesmente copiando este arquivo fonte para um dos diretórios listados no `PATH`.

```
$ echo $PATH

/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:/home/swaroop/bin

$ cp olamundo.py /home/swaroop/bin/olamundo

$ olamundo

Olá Mundo
```

Nós podemos mostrar a variável `PATH` usando o comando `echo` e incluindo o prefixo `$` para indicar para o shell que nós precisamos do valor desta variável. Nós vemos que `/home/swaroop/bin` é um dos diretórios na variável PATH onde*swaroop* é o nome de usuário que eu estou usando em meu sistema. Provavelmente existe um diretório similar para seu nome de usuário para seu sistema. Alternativamente, você pode adicionar um diretório de sua escolha para a variável `PATH` - isto pode ser feito rodando `PATH=$PATH:/home/swaroop/mydir`onde `'/home/swaroop/mydir'` é o diretório que eu desejo adicionar na variável`PATH`.
Este método é muito útil se você deseja escrever scripts úteis que você queira rodar de qualquer lugar, a qualquer hora. É como criar seu próprio comando, semelhante ao `cd` ou qualquer outro comando que você use no terminal linux ou no prompt do DOS.

Cuidado

> Em Python, um programa ou um script significam a mesma coisa.

Obtendo Ajuda

Se você precisar de informação sobre alguma função ou instrução em Python, então você pode usar a função embutida `help`. Isto é muito útil especialmente quando estiver usando o prompt do interpretador. Por exemplo, rode`help(print)` - isto mostra a ajuda para a função print que é usada para imprimir coisas na tela.

Nota

Pressione `q` para sair da ajuda.

De forma similar, você pode obter informação sobre quase qualquer coisa em Python. Use `help()` para aprender mais usando o próprio `help`!

No caso de você precisar de ajuda para operadores como `return`, então você deve colocá-los dentro de aspas como em `help('return')` dessa forma Python não se confunde com o que nós queremos fazer.

Sumário

Agora você é capaz ou de escrever,salvar e rodar programas em Python facilmente. Agora que você é um usuário Python, vamos aprender alguns conceitos de Python.

Python pt-br:Fundamentos

Imprimir 'Olá mundo' não é suficiente, ou é? Você quer fazer mais que isso - você quer inserir dados, manipulá-los e obter alguma resposta a partir deles. Nós podemos conseguir isso em Python usando constantes e variáveis.

Constantes Literais

Um exemplo de uma constante literal é um número como `5`, `1.23`, `9.25e-3` ou uma string (sequência de caracteres) como `'Esta é uma string'` ou `"É uma string!"`. Ela é denominada literal porque é *literal* - você usa seu valor literalmente. O número `2` sempre representa a si mesmo e nada além disso - ele é uma*constante* pois seu valor não pode ser mudado. Logo, todos esses valores referem-se a constantes literais.

Números

Os números em Python são são de três tipos - inteiros, ponto flutuante e complexos.

- `2` é um exemplo de inteiro, os inteiros são os números redondos.
- `3.23` e `52.3E-4` são exemplos de números de ponto flutuante (ou *floats*, para abreviar). A notação `E` indica as potências de 10. Neste caso, `52.3E-4` significa`52.3 * 10`-4.
- `(-5+4j)` e `(2.3 - 4.6j)` são exemplos de números complexos.

Nota para programadores experientes

> Não há o tipo 'long int' (inteiro longo) em separado. O tipo inteiro padrão pode assumir qualquer valor grande.

Strings

Uma string é uma *sequência* de *caracteres*. As strings são basicamente um amontoado de palavras. As palavras podem estar em inglês ou em qualquer língua que seja suportada pelo padrão Unicode, que atende a quase todas as línguas do mundo.

Nota para programadores experientes

> Não há strings "somente em ASCII" porque o padrão Unicode engloba o ASCII.
>
> Por convenção, todas as stringas estão em UTF-8.

Eu posso garantir que você usará strings em quase todos os programas que escrever em Python, portanto preste atenção à próxima parte sobre como usar strings em Python.

Aspas Unitárias

Você pode especificar as strings usando aspas unitárias (ou apóstrofes) tais como `'Use aspas unitárias em mim'`. Todos os espaços em branco, isto é, espaços e tabulações são preservados no estado em que se encontram.

Aspas Duplas

As strings em aspas duplas trabalham exatamente da mesma maneira que as strings em aspas unitárias. Eis um exemplo: `"Qual é o seu nome?"`

Aspas Triplas

Você pode especificar strings que ocupam várias linhas usando aspas triplas - (""" ou '''). Você pode usar aspas unitárias e aspas duplas livremente para formar as aspas triplas. Eis um exemplo:

```
'''Esta é uma string multi-linha. Esta é a primeira linha.

 Esta é a segunda linha.

 "Qual é o seu nome?", eu perguntei.

 Ele disse "Bond, James Bond."

 '''
```

Seqüências de Escape

Suponha que você queira obter uma string que contenha um apóstrofe (`'`), como você escreverá essa string? Por exemplo, a string é `What's your name?`. Você não pode escrever `'What's your name?'` porque o Python ficará confuso sobre onde a string começa e onde termina. Logo, você terá que especificar que este apóstrofe não indica o fim da string. Isso pode ser feito com a ajuda do que é denominada uma *sequência de escape*. Você especifica o apóstrofe como `\'` - note a barra invertida. Agora, você pode escrever a string como `'What\'s your name?'`.

Uma outra maneira de escrever essa string específica seria `"What's your name?"`, isto é, usando aspas duplas. Da mesma maneira, você pode usar uma sequência de escape para inserir aspas duplas em uma string limitada por aspas duplas. Você também pode inserir a própria barra invertida usando a seqüência de escape `\\`.

O que fazer se você quer escrever uma string de duas linhas? Uma solução é usar uma string limitada por aspas triplas conforme foi ensinado previamente ou você pode utilizar uma seqüência de escape para o caracter de nova linha - `\n`para indicar o início da nova linha. Eis um exemplo, `Essa é a primeira linha\nEssa é a segunda linha`. Uma outra seqüência de escape útil a ser conhecida é a tabulação - `\t`. Há muitas outras seqüências de escape, mas eu mencionei aqui somente as mais úteis.

É importante observar que numa string, uma única barra invertida no fim da linha indica que a string continua na próxima linha, mas nenhuma linha nova é adicionada. Por exemplo:

```
"Essa é a primeira frase.\

Essa é a segunda frase."
```

equivale a `"Essa é a primeira frase. Essa é a segunda frase."`.

Strings Brutas

Se você precisa escrever algumas strings onde nenhum processamento especial tais como as seqüências de escape são manipuladas, então o que você precisa é escrever uma string *bruta* prefixando um `r` ou um `R` à string . Eis um exemplo,`r"Novas linhas são indicadas por \n"`.

Strings São Imutáveis

Isso significa que uma vez que você tenha criado uma string, você não pode mudá-la. Embora isso pareça como algo ruim, não é realmente. Nós veremos porque isso não é uma limitação nos diversos programas que nós analisaremos mais adiante.

Concatenação de Literais do Tipo String

Se você colocar duas strings literais lado a lado, elas são automaticamente concatenadas pelo Python. Por exemplo, `'Qual é ' 'o seu nome?'` é automaticamente convertido em `"Qual é o seu nome?"`.

Nota para programadores C/C++

> Não há tipo de dado `char` (caracter) separado em Python. Não existe nenhum motivo real para isto e eu tenho certeza que você não esquecerá isto.

Nota para programadores Perl/PHP

> Lembre-se que strings com aspas simples e duplas são a mesma coisa - elas não diferem entre si

Nota para Usuários de Expressão Regular

> Sempre use strings brutas quando estiver manipulando expressões regulares. Do contrário, será necessário muito uso de caracteres de escape. Por exemplo, referências a barras invertidas podem ser feitas como `'\\1'` ou`r'\1'`.

O método 'format'

As vezes nós iremos querer construir strings de uma outra informação. Isto é onde o método `format()` é útil

```
#!/usr/bin/python
# Filename: str_format.py

age = 25
name = 'Swaroop'

print('{0} is {1} years old'.format(name, age))
print('Why is {0} playing with that python?'.format(name))
```

Saída:

```
$ python str_format.py

Swaroop is 25 years old

Why is Swaroop playing with that python?
```

Como isto funciona:
Uma string pode utilizar certas especificações e sub consequentemente, o método *format* pode ser chamado de um substituto para estas especificações correspondendo os argumentos ao método `format`. Observe que está é a primeira vez que usamos `{0}` e isto corresponde á váriavel`name` que é o primeiro argumento ao método 'format'. Similarmente, a segunda especificação é `{1}` que corresponde á `age` que é o segundo argumento ao método 'format'. O que o Python faz aqui é substituir cada valor do argumento no lugar da especificação. Que pode ter especificações mais detalhadas como:

```
>>> '{0:.3}'.format(1/3)  # decimal (.) precision of 3 for float
'0.333'
>>> '{0:_^11}'.format('hello')  # fill with underscores (_) with the text centered (^) to 11 width
'___hello___'
>>> '{name} wrote {book}'.format(name='Swaroop', book='A Byte of Python')  # keyword-based
'Swaroop wrote A Byte of Python'
```

Detalhes desta especificação de formatação são explicados na Python Enhancement Proposal No. 3101. File:Example.jpg

Variáveis

Usar apenas constantes literais pode se tornar algo chato - nós precisamos de alguma maneira de armazenar qualquer informação, bem como manipulá-la. É aí onde as *variáveis* entram na jogada. Variáveis são exatamento o que seu nome significa - seu valor pode variar, isto é, vôce pode armazenar qualquer coisa usando uma variável. Variáveis são apenas partes da memória do seu computador nas quais você armazena alguma informação. Diferentemente das constantes literais, você precisa de alguma maneira de acessar estas variáveis e, portanto, você dá nomes a elas.

Nomenclatura de Identificadores

Variáveis são exemplos de identificadores. *Identificadores* são nomes dados de maneira a identificar *algo*. Existem algumas regras que você terá que seguir para dar nome aos identificadores:

- O primeiro caracter do identificador precisa ser uma letra do alfabeto (maiúsculo ASCII ou minúsculo ASCII ou caracter Unicode) ou um *underscore*('_').
- O restante do nome do identificador pode consistir de letras (maiúsculo ASCII ou minúsculo ASCII ou caracter Unicode), *underscores* ('_') ou dígitos (0-9).
- Nomes de idenficadores são *case-sensitive*. Por exemplo, `myname` e `myName` **não**são o mesmo. Note o minúsculo `n` no primeiro e o maiúsculo `N` no segundo.

- Exemplos de nomes de identificadores *válidos* são `i`, `__my_name`, `name_23`,`a1b2_c3` and `resumÃƒÆ'Ã†â€™Ãƒâ€ Ã¢â‚¬â„¢ÃƒÆ'Ã¢â‚¬Å¡Ãƒâ€šÃ‚Â©_count`.
- Exemplos de nomes de idenficadores *inválidos* são `2things`, `este contém espaços` e `my-name`.

Tipos de Dados

Variáveis podem manter valores de diferentes tipos, chamados de **tipos de dados**. Os tipos básicos são números e strings, que nós já discutimos. Nos próximos capítulos, veremos como criar seus próprios tipos usando classes.

Objetos

Lembre-se de que Python se refere a qualquer coisa usada em um programa como um *objeto*, em sentido geral. Ao invés de dizer 'o *alguma coisa*', nós de dizemos 'o *objeto*'.

Nota para os usuários de Programação Orientada a Objetos

> Python é fortemente orientada a objetos, no sentido que tudo é um objeto, incluindo números, strings e funções.

Agora veremos como usar variáveis juntamento com constantes literais. Salve o exemplo a seguir e execute o programa.

Como escrever programas em Python

> Daqui em diante, o procedimento padrão para salvar e executar um programa Python é a seguinte:
>
> 1. Abra o seu editor favorito.
> 2. Digite o código do programa dado no exemplo.
> 3. Salve-o como um arquivo com o nome mencionado no comentário. Eu sigo a convenção de ter todos os programas em Python salvos com a extensão `.py`.
> 4. Execute o interpretador com o comando `python program.py` ou use o IDLE para executar os programas. Você também pode usar o método executável como explicado anteriormente.

Exemplo: Utilizando variáveis e constantes literais

```
# Filename : var.py

i = 5
print(i)
i = i + 1
print(i)

s = '''Esta é uma string de múltiplas linhas.
Esta é a segunda linha.'''
print(s)
```

Output:

```
$ python var.py

5

6

Esta é uma string de múltiplas linhas.
```

Esta é a segunda linha.
Como funciona:
Como funciona:

Veja como este programa funciona. Primeiro, vamos atribuir o valor constante literal `5` à variável `i` com o operador de atribuição (`=`). Esta linha é chamada de instrução, pois indica que algo deve ser feito e neste caso, nós estamos conectando o nome da variável `i` ao valor `5`. Em seguida, imprimimos o valor de `i`com o comando `print`, que, obviamente, apenas imprime o valor da variável na tela.
Então somamos `1` ao valor armazenado em `i` e armazenmos de volta. Em seguida, imprimimos e, como esperado, obtemos o valor tt>6</tt>.
Da mesma forma, atribuímos a string literal para a variável `s` e depois a imprimimos.

Nota para programadores de linguagens estáticas

As variáveis são usadas simplesmente atribuindo-lhes um valor. Nenhuma declaração ou definição de tipo de dados é necessária ou utilizada.

Linhas Lógicas e Físicas

Uma linha física aquela que você *vê* quando escreve o programa. Uma linha lógica é que o *Python vê* como uma única instrução. Python implicitamente assume que cada *linha física* corresponde a uma *linha lógica*.
Um exemplo de uma linha lógica é uma instrução como `print('Hello World')` - se este estava em uma linha por si só (como você vê no editor), então isso também corresponde a uma linha física.
Implicitamente, Python incentiva o uso de uma única instrução por linha, o que torna o código mais legível.
Se você quiser especificar mais de uma linha lógica em uma única linha física, então você tem que especificar explicitamente usando um ponto-e-vírgula (`;`) que indica o fim de uma linha lógica ou instrução. Por exemplo,

```
i = 5

print(i)
```

é o mesmo que

```
i = 5;

print(i);
```

e o mesmo pode ser escrito como

```
i = 5; print(i);
```

ou ainda

```
i = 5; print(i)
```

No entanto, eu **recomendo fortemente** que você se atenha a **escrever somente uma única linha lógica para cada linha física**. Use mais de uma linha física para uma única linha lógica apenas se a linha lógica for realmente comprida. A ideia é evitar o ponto-e-vírgula tanto quanto possível, uma vez que isso conduz a um código mais legível. De fato, eu *nunca* utilizei ou sequer vi um ponto-e-vírgula num programa Python.
Segue um exemplo da escrita de uma linha lógica se estendendo por muitas linhas físicas. Nos referimos a isso com **junção explícita de linhas**

```
s = 'Isto é uma string. \

Isto continua a string.'

print(s)
```

Isso nos dá a saída:

```
Isto é uma string. Isto continua a string.
```

Similarmente,

```
print\


(i)
```

é o mesmo que

```
print(i)
```

Em alguns casos não é necessário utilizar barra invertida. Nesses casos a linha lógica usa parênteses, colchetes ou chaves. Isto se chama **junção implícita de linha**. Você pode vê-la em ação quando escrevermos programas usando listasem capítulos posteriores.

Indentação

Espaços são importantes em Python. Na verdade, **espaços no início da linha são importantes**. Isto é chamado de **indentação**. Espaços (espaços e tabs) no início da linha lógica são usados para determinar o nível de indentação da linha lógica, o qual por sua vez é usado para determinar o agrupamento de instruções

Isto significa que as instruções que vão juntas **devem** ter a mesma indentação. Cada conjunto dessas instruções é chamada de **bloco**. Veremos exemplos de como os blocos são importantes nos capítulos seguintes.

Uma coisa que você deve lembrar é que indentação inadequada pode ocasionar erros. Por exemplo:

```
i = 5
 print('O valor é ', i) # Erro! Perceba um espaço no início da linha
 print('Eu repito, o valor é ', i)
```

Quando você executa isso, obtém o seguinte erro:

```
    File "whitespace.py", line 4


      print('O valor é ', i) # Erro! Perceba um espaço no início da linha


      ^


IndentationError: unexpected indent
```

Perceba que há um espaço no início da segunda linha. O erro indicado pelo Python nos diz que a sintaxe do programa é inválida, isto é, o programa não foi escrito direito. O que isso quer dizer é que *você não pode iniciar novos blocos de instruções arbitrariamente* (exceto pelo bloco principal padrão que você vem usando o tempo todo, é claro). Casos nos quais você pode usar novos blocos serão detalhados em capítulos posteriores, tais como o capítulo sobre controle de fluxo.

Como indentar

> **Não** use uma mistura de tabs e espaços para a indentação, pois isso não funciona adequadamente em diferentes plataformas. Eu *recomendo fortemente* que você use um *único tab* ou *quatro espaços* para cada nível de indentação.
>
> Escolha qualquer um desses dois estilos de indentação. Mais importante, escolha um e use **consistentemente**, isto é, use *somente* aquele estilo de indentação.

Nota para programadores de linguagens estáticas

> Python sempre utiliza indentação para blocos e nunca utiliza chaves. Execute `from __future__ import braces` para aprender mais.

Resumo

Agora que nós discutimos vários detalhes vitais, podemos seguir para coisas mais interessantes como declarações de controle de fluxo. Certifique-se de que você já está confortável com o que você leu nesse capítulo.

Python pt-br:Operadores e Expressoes

Introdução

A maioria das instruções (linhas lógicas) que você escreverá irão conter**expressões**. Um exemplo simples de expressão é `2 + 3`. Uma expressão pode ser divida em operadores e operandos.
Operadores definem que operação será realizada e podem ser representados por símbolos como `+` ou por palavras-chave especiais. Operadores requerem dados para funcionar e tais dados são chamados *operandos*. Neste caso, `2` e `3` são os operandos.

Operadores

Iremos dar uma breve olhada nos operadores e sua utilização:
Perceba que você pode testar as espressões dadas nos exemplos usando o interpretador interativo. Por exemplo, para testar a expressão `2 + 3`, use o interpretador interativo do Python:

```
>>> 2 + 3
5
>>> 3 * 5
15
>>>
```

Operadores e sua utilização

Operador	**Nome**	**Explicação**	**Exemplos**
+	Adição	Soma dois objetos	`3 + 5` retorna `8`. `'a' + 'b'`retorna `'ab'`.
-	Subtração	Torna um número negativo ou a subtração de um número por outro	`-5.2` retorna um número negativo. `50 - 24` retorna `26`.
*	Multiplicação	Retorna o produto de dois números ou uma string repetida uma certa quantidade de vezes.	`2 * 3` retorna `6`. `'la' * 3` retorna`'lalala'`.
**	Potência	Retorna x elevado à potência de y	`3 ** 4` retorna `81` (i.e. `3 * 3 * 3 * 3`)
/	Divisão	Divide x por y	`4 / 3` retorna`1.3333333333333333`.
//	Divisão Inteira	Retorna a parte inteira do quociente	`4 // 3` retorna `1`.
%	Modulo	Retorna o resto da divisão	`8 % 3` retorna `2`. `-25.5 % 2.25`retorna `1.5`.
<<	Deslocamento de bits à esquerda	Desloca os bits do número para a esquerda pelo número de bits especificado. (Cada número é representado na memória por bits ou digitos binários i.e. 0	`2 << 2` retorna `8`. `2` é representado por `10` em bits. Deslocando os bits do número à esquerda por 2 bits retorna `1000`, que representa o número decimal `8`.

		and 1)	
>>	Deslocamento de bits à direita	Desloca os bits do número para a direita pelo número de bits especificado.	`11 >> 1` retorna `5`. `11` é representado em bits por `1011`que quando os bits são deslocados para a direita por 1 bit retorna `101`, que é o número decimal `5`.
&	Operador bit a bit AND	Bits configurados nos dois operadores são configurados no resultado	`5 & 3` retorna `1`.
|	Operador bit a bit OR	Bits configurados em um ou outro operador são configurados no resultado	`5 | 3` retorna `7`
^	Operador bit a bit XOR	Bits configurados em um ou outro operador, mas não em ambos, são configurados no resultado	`5 ^ 3` retorna `6`
~	Operador bit a bit NOT	Bits configurados no operador não são configurados no resultado e vice-versa.	`~5` retorna `-6`.
<	Menor que	Retorna se x é menor que y. Todos os operadores de comparção retornam`True`(verdadeiro) ou `False`(falso). Note a capitalização dos nomes.	`5 < 3` retorna `False` e `3 < 5`retorna `True`. Comparações podem ser encadeadas arbitrariamente: `3 < 5 < 7` retorna `True`.
>	Maior que	Retorna se x é maior que y	`5 > 3` retorna `True`. Se os dois operadores forem números, eles são antes convertidos a um tipo comum. De outra maneira, será sempre retornado `False`.
<=	Menor ou igual a	Retorna se x é menor ou igual a y	`x = 3; y = 6; x <= y` retorna`True`.
>=	Maior ou igual a	Retorna se x é maior ou igual a y	`x = 4; y = 3; x >= 3` retorna`True`.
==	Igual a	Compara se os objetos são iguais	`x = 2; y = 2; x == y` retorna`True`. `x = 'str'; y = 'stR'; x == y`retorna `False`. `x = 'str'; y = 'str'; x == y`retorna `True`.
!=	Diferente de	Compara se os objetos são diferentes	`x = 2; y = 3; x != y` retorna`True`.
not	Operador booleano NOT	Se x é `True`, ele retorna `False`. Se x é `False`, ele	`x = True; not y` retorna `False`.

		retorna `True`.	
and	Operador booleano AND	`x and y` retorna `False` se x é `False`, senão ele retorna a avaliação de y	`x = False; y = True; x and y` retorna `False` desde que x seja False. Neste caso, Python não irá avaliar y desde que ele saiba que o lado esquerdo da espressão 'and' é `False` o que implica que toda a espressão será `False` independente do outro valor. isso é chamado short-circuit evaluation.
or	Operador booleano OR	Se x é `True`, ele retorna True, senão ele retorna a avaliação de y	`x = True; y = False; x or y` retorna `True`. A Short-circuit evaluation se aplica aqui também.

Precendência de Operadores

Se você tem uma expressão como `2 + 3 * 4`, será realizada primeiro a adição ou a multiplicação? A matemática do nosso ensino médio nos diz que a multiplicação deve ser realizada primeiro. Isso significa que o operador de multiplicação possui maior precedência que o operador de adição.
A tabela a seguir nos dá a ordem de precedência de operadores para o Python, do de menor precedência (least binding) ao de maior precedência (most binding). Isto significa que em uma dada expressão, Python irá primeiramente avaliar os operadores listados na base da tabela antes dos operadores listados ao topo. A tabela a seguir (tirada de Manual de referência Python) é proporcionada visando maior entendimento. É muito melhor usar parênteses para agrupar operadores e operandos apropriadamente de maneira que fique clara a precedência. Isso torna o programa mais legível. Veja #Ordem de Avaliaçãoabaixo para detalhes.

Precedência de Operadores

Operador	**Descrição**
lambda	Expressão Lambda
or	Operador Booleano OR
and	Operador Booleano AND
not x	Operador Booleano NOT
in, not in	Testes de membros/existência
is, is not	Testes de identidade
<, <=, >, >=, !=, ==	Comparações
|	Operador bit-a-bit OR
^	Operador bit-a-bit XOR
&	Operador bit-a-bit AND
<<, >>	Deslocamentos de bits
+, -	Adição e subtração
*, /, //, %	Multiplicação, Divisão, Divisão inteira e Resto
+x, -x	Positivo, Negativo
~x	Operador bit-a-bit NOT
**	Exponenciação
x.atributo	Referencia a atributo
x[índice]	Subscrição

x[índice1:índice2]	Repartição
f(argumentos ...)	Chamada de função
(expressões, ...)	Uniao ou exibição de tupla
[expressões, ...]	Exibição de listas
{chave:dado, ...}	Exibição de dicionário

Os operadores que ainda não vimos serão explicados em capítulos posteriores.
Operadores com a *mesma precedência* são listados na mesma linha na tabela acima. Por exemplo, `+` e – possuem a mesma precedência.

Ordem de avaliação

Por padrão, a tabela de precedência dos operadores decide que operadores são avaliados primeiro. Para tornar uma expressão mais legível podemos usar parênteses. Por exemplo,`2 + (3 * 4)` é, definitivamente, mais fácil de entender que `2 + 3 * 4` que requer conhecimento da ordem de precedência dos operadores. Assim como todo o resto, parênteses devem ser usados conscientemente (não sobrecarregue) e não devem ser redundantes (como em `2 + (3 + 4)`).
Se você quer mudar a ordem em que eles são avaliados, você pode, novamente, usar parênteses. Por exemplo, se você quer que a adição seja realizada antes da multiplicação em uma expressão, então você pode escrever algo como `(2 + 3) * 4`.

Associatividade

Operadores são, geralmente, associados da esquerda para a direita, isto é, operadores com a mesma procedência são avaliados da esquerda para a direita. Por exemplo, `2 + 3 + 4` é avaliado como `(2 + 3) + 4`. Alguns operadores, como os de atribuição, possuem a associatividade da direira para a esquerda, isto é, `a = b = c` é tratado como `a = (b = c)`.

Expressões

Usando Expressões

Exemplo:

```
#!/usr/bin/python
# Nome do Arquivo: expressao.py

length = 5
breadth = 2

area = length * breadth
print('Área é', area)
print('Perímetro é', 2 * (length + breadth))
```

Saída:

```
$ python expressao.py

Área é 10

Perimetro é 14
```

Como Funciona:
O tamanho e amplitude do retângulo são armazenados nas variáveis length e breadth. Nós os utilizamos para calcular a área e o perímetro do retângulo com a ajuda de expressões. Armazenamos o resultado da expressão `length * breadth`na variável `area` e então o imprimimos usando a função `print`. No segundo caso, usamos o valor da expressão `2 * (length + breadth)` diretamente na função print.
Perceba, também, como o Python faz a 'impressão elegante' (pretty-print) da saída. Mesmo que não tenhamos especificao um espaço entre `'Área é'` e a variável `area`, Python coloca-o para nós de uma maneira que tenhamos uma saída limpa e bonita, e o programa é muito mais legível dessa maneira

(desde que não tenhamos que nos preocupar com espaçamento na string que utilizamos para a saída). Isto é um exemplo de como o Python torna mais fácil a vida do programador.

Sumário

Vimos como utilizar operadores, operandos e expressões - estes são os blocos de construção básicos de qualquer programa. A seguir, veremos como fazer uso deles em nossos programas através de indicação.

Python pt-br:Controle de Fluxo

Introdução

Nos programas que vimos até agora, houveram uma série de declarações e o Python executa-os na mesma ordem. E se você quisesse alterar o fluxo de seu funcionamento? Por exemplo, você quer que o programa tome algumas decisões e faça diferentes coisas dependendo das diferentes situações, como imprimir 'Bom Dia' ou 'Boa Tarde' dependendo da hora do dia?
Como você deve ter pensando, isto é alcançado usando as instruções de controle de fluxo no Python - `if`, `for` e `while`.

A instrução IF

A instrução `if` é usada para verificar uma condição e *se* a condição é verdadeira, será executado um bloco de instruções (chamado de *bloco-if(if-block)*), *senão* será processado outro bloco de instruções (chamado de *bloco-else(else-block)*). A cláusula *else*é opcional.
Exemplo:

```
#!/usr/bin/python
# Nome do aquivo: if.py

number = 23
guess = int(input('Entre com um número inteiro : '))

if guess == number:
    print('Parabéns, você advinhou.') # Novo bloco começa aqui
    print('(mas você não ganhou nenhum prêmio!)') # Novo bloco termina aqui
elif guess < number:
    print('Não, era um pouco maior que isso') # Outro bloco
    # Você pode fazer o que quiser em um bloco ...
else:
    print('Não, era um pouco menor que isso')
    # Você deve adivinhar > número a alcançar aqui

print('Feito')
# Esta última instrução é sempre executada, depois da instrução if ser executada
```

Saída:

```
$ python if.py

Entre com um número inteiro : 50

Não, era um pouco menor que isso

Feito



$ python if.py

Entre com um número inteiro : 22

Não, era um pouco maior que isso

Feito
```

```
$ python if.py

Entre com um número inteiro : 23

Parabéns, você advinhou.

(mas você não ganhou nenhum prêmio!)

Feito
```

Como Funciona:
Neste programa, recebemos tentativas de advinhações do usuário e verificamos se este é igual ao número que temos. Setamos a variável `number` para qualquer inteiro que desejarmos, digamos `23`. Então, pegamos a tentativa de advinhação do usuário usando a função `input()`. Funções são peças de programas reutilizáveis. Iremos ler mais sobre elas no próximo capítulo.
Nós fornecemos uma string para função enbutida `input` que a imprime na tela e aguarda uma entrada do usuário. Uma vez que entramos com algum valor e apertamos a tecla `enter`, a função `input()` retorna o valor que entramos, como uma string. Nós, então, convetemos essa string para um inteiro usando `int` e depois armazenamos na variável `guess`. Na verdade, o `int` é uma classe, mas tudo o que voê precisa saber agora é que você pode usá-la para converter uma string em um número inteiro (assumindo que a string contém um número inteiro válido no texto).
A seguir, comparamos a tentativa de adivinhação do usuário com o número que escolhemos. Se eles forem iguais, imprimimos uma mensagem de sucesso. Note que utilizamos níveis de indentaçãopara dizer ao Python que instruções pertencem a qual bloco. É pos isso que a indentação é tão importante no Python. Eu espero que você esteja mantendo a regra da "indentação consistente". Você está?
Perceba que a instrução `if` contém 'dois pontos' no final - nós estamos indicando ao Python que a seguir há um bloco de instruções.
Então, checamos se a tentativa de advinhação do usuário é menor que o número da variável number, e se for verdadeiro, informamos ao usuário para tentar com um número um pouco maior que o inserido. O que usamos aqui é a clausula `elif`que, na verdade, combina duas instruções `if else-if else` relacionadas em uma instrução `if-elif-else` combinada. Isso torna o programa mais fácil e reduz quantidade de indentações requeridas.
As instruções `elif else` devem, também, possuir 'dois pontos' no final da linha lógica, seguido pelo seu bloco de instruções correspondente (com indentação apropriada, é claro).
Você pode ter outra instrução `if` dentro de um bloco-if de uma instrução if `if` a assim por diante - isto é chamado de instrução `if` aninhada
Lembre que as partes `elif` e `else` são opcionais. Uma instrução `if` mínima válida é:

```
if True:
    print('Sim, é verdadeiro')
```

Depois que python terminou de executar a instrução `if` completamente, junto com as cláusulas `elif` e `else` associadas, ele passa para a próxima instrução no bloco contendo a instrução `if`. Neste caso, é o bloco principal onde a execução do programa inicia e a próxima instrução é `print('Feito')`. Depois disso, Python vê o final do programa e simplesmente termina.
Ainda que este seja um programa muito simples, eu estive apontando várias coisas que você deve notar em programas assim. Todas elas são bem avançadas (e surpreendentemente simples para todos vocês com conhecimento em C/C++) e requerem que você esteja inicialmente ciente de todas elas, mas depois disso, você irá se familiarizar e isso irá se tornar 'natural' para você.

Nota para os programadores de C/C++

> Não há a instrução `switch` no Python. Você pode usar uma instrução`if..elif..else` para fazer a mesma coisa (e em alguns casos, usar umdicionário para fazê-lo rapidamente)

A instrução while

A instrução `while` permite que você execute repetidamente um bloco de instruções enquanto uma condição for verdadeira. Uma instrução `while` é um exemplo do que é chamado de instrução de *looping*. Uma instrução `while` pode ter uma cláusula `else` opcional.

Exemplo:

```
#!/usr/bin/python
# Nome do arquivo: while.py

number = 23
running = True

while running:
    guess = int(input('Entre com um número inteiro : '))

    if guess == number:
        print('Parabéns, você advinhou.')
        running = False  # Isto faz o loop while parar
    elif guess < number:
        print('Não, é um pouco maior que este.')
    else:
        print('Não, é um pouco menor que este.')
else:
    print('O loop while terminou.')
    # Faça qualquer outra coisa que quiser aqui

print('Fim')
```

Saída:

```
$ python while.py

Entre com um número inteiro : 50

Não, é um pouco menor que este.

Entre com um número inteiro : 22

Não, é um pouco maior que este.

Enter an integer : 23

Parabéns, você advinhou.

O loop while terminou.

Fim
```

Como funciona:

Neste programa, nós ainda estamos jogando o jogo da advinhação, mas a vantagem é que o usuário pode continuar tentando advinhar até que ele acerte o número - não ha necessidade de rodar novamente o programa para cada tentativa de adivinhação, como fizemos na seção anterior. Isto demonstra o uso da instrução `while`.

Nós movemos o`input` e a instrução `if` para dentro do loop `while` e setamos a variável `running` para `True` antes do loop while. Primeiro, nós checamos se a

variável `running` é `True`(verdadeiro) e então seguimos para executar executar o <emphasis>bloco while</emphasis> correspondente. Depois que o bloco é executado, a condição é novamente checada que neste caso é a variável`running`. Se isso é verdade, nós executamos o bloco while novamente, senão continuamos para executar o bloco else optional e então seguir para a próxima instrução.
O bloco `else` é executado quando a condição do loop `while` se torna `False`(falso) - esta pode até ser a primeira vez que a condição é verificada. Se há alguma cláusula `else` para um loop `while`, ele é sempre executado a menos que você tenha um loop `while` que se executado para sempre sem sair sequer uma vez!
Os valores `True`(verdadeiro) e `False`(falso) são chamados tipos Booleanos e você pode considerá-los equivalentes aos valores `1` e `0` respectivamente.
O bloco else é, na verdade, redundante a partir que você pode colocar estas instruções no mesmo bloco (como a instrução `while`) depois da instrução `while`para conseguir o mesmo efeito.

Nota para programadores de C/C++

Lembre que você pode ter uma cláusula else `else` para o loop `while`.

O loop for

A instrução `for..in` é outra instrução de loop que *itera* sobre uma sequência de objetos, por exemplo, percorre cada item em uma sequência. Iremos ver mais sobre sequências em detalhes em capítlos posteriores. O que você precisa saber agora é que uma sequência é apenas uma coleção ordenada de itens.
Exemplo:

```
#!/usr/bin/python
# Nome do arquivo: for.py

for i in range(1, 5):
    print(i)
else:
    print('O loop for terminou.')
```

Saída:

```
$ python for.py

1

2

3

4

O loop for terminou.
```

Como funciona:
Neste programa, estamos imprimindo uma *sequência* de números. Nós geramos esta sequência de números usando a função interna `range`.
O que fazemos é fornecer dois números e `range` retorna uma sequência de números iniciando do primeiro número e segue até o segundo número. Por exemplo, `range(1,5)` nos dá a sequência `[1, 2, 3, 4]`. Por padrão, `range` dá um passo contando de 1 em 1. Se fornecer-mos um terceiro número para `range`, então ele se torna o tamanho do passo. Por exemplo, `range(1,5,2)` nos dá`[1,3]`. Lembre-se que range extende-se *até* o segundo número, ou seja, ele **não**inclui o segundo número.
O loop `for` então itera sobre esta faixa - `for i in range(1,5)` é equivalente a`for i in [1, 2, 3, 4]` que é a mesma coisa que atribuir cada número (ou objeto) na sequência a i, um de cada vez, e então executar o bloco de instrições para cada valor de `i`. Neste caso, nós apenas imprimimos o valor no bloco de instruções.

Lembre-se que a parte `else` é opcional. Quando incluída, ela será sempre executada uma vez após o loop `for` ter terminado, a não ser que uma instruçãobreak seja encontrada.
Lembre-se que o loop `for..in` funciona para qualquer sequência. Aqui, temos uma lista de números gerados pela função interna `range`, mas, no geral, podemos usar qualquer tipo de sequência de qualquer tipo de objeto! Iremos explorar essa idéia em detalhes em capítulos posteriores.

Nota para programadores de C/C++/Java/C#

> O loop `for` do Python é radicalmente diferente do loop `for` das linguagens C/C++. Programadores da linguagem C# irão notar que o loop `for` no Python é similar ao loop `foreach` em C#. Programadores da linguagem Java irão notar que o mesmo é similar a `for (int i : IntArray)` em Java 1.5 .
> Em C/C++, se você quer escrever `for (int i = 0; i < 5; i++)`, então em Python você escreve apenas `for i in range(0,5)`. Como você pode ver, o loop `for` é mais simples, mais expressivo e menos propenso a erros no Python.

A instrução break

A instrução `break` é usada para *quebrar* uma instrução de loop, ou seja, para a execução de uma instrução de loop, mesmo que a condição não tenha se tornado `False` ou a sequência de itens tenha sido iterada completamente.
Uma informação importante é que se você *quebra* um loop `for` ou `while`, qualquer bloco `else` correspondente **não** é executado.
Exemplo:

```
#!/usr/bin/python
# Nome do arquivo: break.py

while True:
    s = input('Entre com alguma coisa: ')
    if s == 'sair':
        break
    print('Tamanho da string é ', len(s))
print('Feito')
```

Saída:

```
$ python break.py

Entre com alguma coisa: Programar é divertido

Tamanho da string é 21

Entre com alguma coisa: Quando o trabalho está feito

Length of the string is 28

Entre com alguma coisa: se você quer tornar seu trabalho também divertido:

Tamanho da string é 51

Entre com alguma coisa:       use Python!

Tamanho da string é 12

Entre com alguma coisa: sair

Feito
```

Como funciona:
Neste programa, nós recebemos os dados de entrada do usuário repetidamente e imprimimos o tamanho de cada entrada toda vez. Estamos provendo uma condição especial para parar o programa verificando se a entrada do usuário é `'sair'`. Paramos o programa com <emphasis>break</emphasis>, fazendo com que saiamos do loop e alcancemos o fim do programa.
O tamanho da entrada pode ser encontrado utilizando a função interna `len`.
Lembre-se que a instrução `break` pode também ser usada com o loop `for`.

Python Poético do Swaroop

Os dados de entrada que utilizei aqui é um mini poema que escrevi chamado**Python Poético do Swaroop**:

```
Programar é divertido

Quando o trabalho está feito

se você quer tornar seu trabalho também divertido:

    use Python!
```

A Instrução continue

A instrução `continue` é usada para dizer ao Python que pule as intruções restantes do bloco de loop corrente e para continuar (*continue*) na iteração seguinte do loop.
Exemplo:

```
#!/usr/bin/python
# Arquivo: continue.py

while True:
    s = input('Entre com algo : ')
    if s == 'sair':
        break
    if len(s) < 3:
        print('Muito pequeno')
        continue
    print('A entrada é de tamanho suficiente')
    # Faça outros tipos de processo aqui...
```

Saída:

```
$ python continue.py

Entre com algo : a

Muito pequeno

Entre com algo : 12

Muito pequeno

Entre com algo : abc

A entrada é de tamanho suficiente

Entre com algo : sair
```

Como funciona:

Neste programa, aceitamos entradas do usuário, mas as processamos somente se ela possuírem pelo menos 3 caracteres. Então, utilizamos a função interna `len`para pegar o tamanho e se este for menor que 3, pulamos o resto das instruções no bloco utilizando a instrução `continue`. Senão, o resto das instruções no loop são executadas e poderemos fazer qualquer tipo de processamento que quiser-mos aqui.
Note que a instrução `continue` funciona, também, com o loop `for`.

Sumário

Vimos como utilizar as três instruções de controle de fluxo - `if`, `while` and `for`, em conjunto com suas instruções `break` e `continue` associadas. Estas são algumas das partes mais utilizadas do Python e outras, tornar-se confortável com elas é essencial.

Python pt-br:Funcoes

Introdução

Funções são pedaços reutilizáveis de programas. Elas permitem dar um nome para um bloco de código e rodá-lo usando seu nome em qualquer lugar do programa, quantas vezes forem necessárias. Isto é conhecido como *chamar* a função. Nós já usamos muitas funções internas como `len` e `range`.

O conceito de função é provavelmente o bloco de construção mais importante de qualquer programa não-trivial (em qualquer linguagem de programação), portanto vamos explorar vários aspectos das funções neste capítulo.

Funções são **def**inidas usando a palavra-chave `def`. A mesma é seguida pelo seu nome *identificador*, um par de parênteses que podem conter ou não nomes de variáveis separados por vírgula, e por dois pontos ao final da linha. Em seguida o bloco de código que faz parte desta função. Um exemplo abaixo irá mostrar que isto é realmente muito simples:

Exemplo:

```
#!/usr/bin/python
# Filename: funcao1.py

def digaOla():
    print('Olá Mundo!') # bloco pertencente à função
# Fim da função

digaOla() # chamando a função
digaOla() # chamando a função denovo
```

Output:

```
$ python funcao.py

Olá Mundo!

Olá Mundo!
```

Como isto funciona:

Definimos a função chamada `digaOla` usando a sintaxe como explicado abaixo. Esta função não recebe parâmetros e conseqüentemente não há variáveis declaradas entre parênteses. Para as funções, parâmetros são somente uma entrada que podemos passar diferentes valores e receber resultados correspondentes.

Repare que podemos chamar a mesma função duas vezes, o que significa que não temos que escrever o mesmo código de novo.

Parâmetros da Função

Uma função pode receber parâmetros, que são valores fornecidos à função para que a mesma possa *fazer* algo útil com os mesmos. Estes parâmetros são como variáveis, exceto que os valores destas variáveis são definidos quando chamamos a função e já estão atribuídos quando a função é executada. Parâmetros são especificados dentro do par de parênteses na definição da função, separados por vírgulas. Quando chamamos a função, passamos os valores da mesma maneira. Note a terminologia utilizada - os nomes dados na definição da função são chamados de *parâmetros* enquanto que os valores que fornecemos na chamada da função são chamados de *argumentos*.

Exemplo:

```
#!/usr/bin/python
# Filename: func_param.py

def printMax(a, b):
    if a > b:
        print(a, 'is maximum')
```

```
    else:
        print(b, 'is maximum')

printMax(3, 4) # valores literais passados para a função

x = 5
y = 7

printMax(x, y) # passando variáveis como argumentos
```

Saída:

```
$ python func_param.py

4 is maximum

7 is maximum
```

Como funciona:
Aqui definimos uma função chamada `printMax` que recebe dois parâmetros chamados `a` e `b`. Encontramos o maior número usando um simples comando`if..else` e então imprimindo o maior número.
No primeiro uso de `printMax`, nos passamos diretamente os números (argumentos). No segundo, chamamos a função usando variáveis. `printMax(x, y)` faz com que o valor do argumento `x` seja atribuído ao parâmetro `a` e o valor do argumento `y` atribuído ao parâmetro `b`. A função printMax funciona da mesma maneira em ambos os casos.

Variáveis Locais

Quando você declara variáveis dentro de uma definição de função, elas não estão relacionadas de forma alguma com outras variáveis com os mesmos nomes usados fora da função, ou seja, variáveis *locais* em relação à função. Isto é o que se chama de *escopo* da variável. Todas as variáveis têm o escopo do bloco no qual são declaradas, a partir do ponto da definição do seu nome.
Exemplo:

```
#!/usr/bin/python
# Filename: func_local.py

def func(x):
    print('x is', x)
    x = 2
    print('Variável local x mudou para', x)

x = 50
func(x)
print('x continua', x)
```

Saída:

```
$ python func_local.py

x is 50

Variável local x mudou para 2

x continua 50
```

Como funciona:

Na função, a primeira vez que usamos o *valor* do nome `x`, o Python utiliza o valor do parâmetros declarado na função.
Em seguida, atribuímos o valor `2` a `x`. O nome `x` é local à nossa função. Então, quando mudamos o valor de `x` na função, o `x` definido no bloco principal permanece inalterado.
No último comando `print`, confirmamos que o valor de `x` no bloco principal está de fato inalterado.

Usando o comando global

Se você quer atribuit um valor a um nome definido no nível mais alto do programa, ou seja, fora de qualquer tipo de escopo tal como funções ou classes, então você tem que dizer ao Python que o nome não é local, mas sim *global*. Fazemos isso usando o comando `global`. É impossível atribuir um valor a uma variável definida fora de uma função sem utilizar o comando `global`.
Você pode usar os valores dessas variáveis globais definidas fora da função, assumindo que não haja nenhuma variável com o mesmo nome dentro da função. No entanto, isso não é recomendado e deve ser evitado, uma vez que não fica claro para o leitor do programa onde está a definição da variável. Usar o comando `global` torna claro que a variável foi definida no block mais externo.
Exemplo:

```
#!/usr/bin/python
# Filename: func_global.py

def func():
    global x

    print('x é', x)
    x = 2
    print('Variável global x mudou para', x)

x = 50
func()
print('O valor de x é', x)
```

Saída:

```
$ python func_global.py

x is 50

Variável global x mudou para 2

O valor de x é 2
```

Como funciona:
O comando `global` é usado para declarar que `x` é uma variável global. Assim, quando vamos atribuir um valor a `x` dentro da função, essa mudança é refletida quando usamos o valor de `x` no bloco principal.
Você pode especificar mais de uma variável global usando o mesmo comando`global`. Por exemplo, `global x, y, z`.

Usando o comando nonlocal

Vimos acima como acessar variáveis nos escopos local e global. Existe outro tipo de escopo chamado "nonlocal", que é um meio termo entre esses dois tipos de escopo. Escopos nonlocal ocorrem quando se definem funções dentro de funções.
Uma vez que tudo em Python é código executável, você pode definir funções em qualquer lugar.
Vejamos um exemplo:

```
#!/usr/bin/python
# Filename: func_nonlocal.py

def func_outer():
```

```
    x = 2
    print('x é', x)

    def func_inner():
        nonlocal x
        x = 5

    func_inner()
    print('O x local mudou para', x)

func_outer()
```

Saída:

```
$ python func_nonlocal.py

x é 2

O x local mudou para 5
```

Como funciona:
Quando estamos dentro de `func_inner`, o 'x' definido na primeira linha de`func_outer` não está, relativamente, nem no escopo local, nem no global. Declaramos que estamos usando esse x por meio de `nonlocal x` e assim temos acesso àquela variável.
Tente trocar o `nonlocal x` para `global x` e também remover o comando e observar a diferença de comportamento nesses dois casos.

Valores padrão de argumentos

Para algumas funções, você pode querer que alguns dos seus parâmetros sejam*opcionais* e usar valores padrão pré-definidos se o usuário não quiser fornecer valores para esses parâmetros. Isto é feito com a ajuda de valores padrão de argumentos. Você pode especificar valores padrão de argumentos para parâmetros colocando o operador de atribuição (`=`) após o nome do parâmetro na definição da função, seguifo pelo valor padrão.
Note que o valor do argumento padrão dever ser uma constante. Mais precisamente, o valor do argumento padrão deve ser imutável. Isso é explicado em detalhes nos capítulos seguintes. Por enquanto, basta lembrar disso.
Exemplo:

```
#!/usr/bin/python
# Filename: func_default.py

def say(message, times = 1):
    print(message * times)

say('Olá')
say('Mundo', 5)
```

Saída:

```
$ python func_default.py

Olá

MundoMundoMundoMundoMundo
```

Como funciona:
A função chamada `say` é usada para imprimir uma string quantas vezes forem necessárias. Se nós não fornecemos um valor, então, por padrão, a string é impressa apenas uma vez. Conseguimos isso especificando um valor de argumento padrão `1` para o parâmetro `times`.

No primeiro uso de `say`, fornecemos apenas a string e ele a imprime uma vez. No segundo uso de `say`, fornecemos tanto a string quanto o argumento 5 `5`determinando que queremos *dizer* a mensagem 5 vezes.

Importante

> Apenas os parâmetros que estiverem no final da lista de parâmetros podem receber valores padrão de argumento, ou seja, você não pode ter um parâmetro com valor de argumento padrão antes de um parâmetro sem um valor de argumento padrão na ordem dos parâmetros declarados na lista de parâmetros da função.
>
> O motivo é que os valores são atribuídos aos parâmetros por posição. Por exemplo, `def func(a, b=5)` é válido, mas `def func(a=5, b)` é *não válido*.

Argumentos Nomeados

Se você tem funções com muitos parâmetros e quer especificar apenas alguns deles, então você pode passar valores para esse parâmetros nomeando-os. Isto é o que chamamos de *argumentos nomeados*. Usamos o nome (palavra-chave) ao invés da posição (como viemos usando até agora) para especificar os argumentos para a função.
Existem duas *vantagens* nessa abordagem: primeiro, é mais fácil usar a função, uma vez que não precisamos nos preocupar com a ordem dos argumentos. Segundo, podemos dar valores apenas para os parâmetros que quisermos, desde que os demais tenham valores padrão de argumento.
Exemplo:

```
#!/usr/bin/python
# Filename: func_key.py

def func(a, b=5, c=10):
    print('a é', a, 'e b é', b, 'e c é', c)

func(3, 7)
func(25, c=24)
func(c=50, a=100)
```

Saída:

```
$ python func_key.py

a é 3 e b é 7 e c é 10

a é 25 e b é 5 e c é 24

a é 100 e b é 5 e c é 50
```

Como funciona:
A função chamada `func` tem um parâmetro sem valores padrão de argumento, seguida por dois parâmetros sem valores padrão de argumento.
No primeiro uso, `func(3, 7)`, o parâmetro `a` recebe o valor `3`, o parametro `b`recebe o valor `5` e `c` recebe o valor padrão de `10`.
No segundo uso `func(25, c=24)`, a variável `a` recebe o valor de 25 devido à posição do argumento. Então, o parâmetro `c` fica com o valor de `24` devido ao nome, ou seja, aos argumentos nomeados. A variável `b` fica com o valor padrão de `5`.
No terceiro uso `func(c=50, a=100)`, utilizamos somente argumentos nomeados para especificar os valores. Note que estamos especificando o valor para o parâmetro `c` antes do valor do parâmetro `a` ainda que `a` tenha sido definido antes de `c` na definição da função.

Parâmetros VarArgs

TODO

> Devo escrever sobre este assunto num capítulo posterior, uma vez que ainda não falamos sobre listas e dicionários?

Às vezes você pode querer definir uma função que possa receber *qualquer*número de parâmetros. Isso pode ser conseguido usando os asteriscos:

```
#!/usr/bin/python
# Filename: total.py

def total(initial=5, *numbers, **keywords):
    count = initial
    for number in numbers:
        count += number
    for key in keywords:
        count += keywords[key]
    return count

print(total(10, 1, 2, 3, vegetables=50, fruits=100))
```

Saída:

```
$ python total.py

166
```

Como funciona:
Quando declaramos um parâmetro prefixado com um asterisco, tal como `*param`, então todos os argumentos posicionais daquele ponto até o final são armazenados numa lista chamada 'param'.
De maneira similiar, quando declaramos um parâmetro prefixado com dois asteriscos, tal como `**param`, então todos os argumentos nomeados, a partir daquele ponto até o final, serão armazenados em um dicionário chamado 'param'.
Vamos explorar as listas e dicionários em um capítulo mais à frente.

Parâmetros apenas por palavra-chave

Se quisermos especificar que certos parâmetros por palavra-chave estejam disponíveis somente por palavra-chave e *não* como argumentos posicionais, estes pode ser declarados após um parâmetro marcado com asterisco:

```
#!/usr/bin/python
# Filename: keyword_only.py

def total(initial=5, *numbers, vegetables):
    count = initial
    for number in numbers:
        count += number
    count += vegetables
    return count

print(total(10, 1, 2, 3, vegetables=50))
print(total(10, 1, 2, 3))
# Levanta um erro, pois não fornecemos um valor de argumento padrão para 'vegetables'
```

Saída:

```
$ python keyword_only.py

66

Traceback (most recent call last):
```

```
  File "test.py", line 12, in <module>

print(total(10, 1, 2, 3))

TypeError: total() needs keyword-only argument vegetables
```

Como funciona:
Declarar parâmetros após um parâmetro marcado com asterisco resulta em argumentos apenas por palavra-chave. Se estes argumentos não são fornecidos com um valor padrão, então chamadas à função levantarão um erro se o argumento por palavra-chave não é fornecido, como visto acima.
Se você quer ter apenas parâmetros por palavra-chave mas não precisa de um parâmetro marcado com asterisco, então simplesmente use um asterisco sem qualquer nome, tal como `def total(initial=5, *, vegetables)`.

O comando return

O comando `return` é usado para *retornar* da execução de uma função, isto é, sair da função. Podemos, opcionalmente, *retornar um valor* vindo da função.
Exemplo:

```
#!/usr/bin/python
# Filename: func_return.py

def maximum(x, y):
    if x > y:
        return x
    else:
        return y

print(maximum(2, 3))
```

Saída:

```
$ python func_return.py

3
```

Como funciona:
A função `maximum` retornar o maior valor entre os parâmetros, neste caso, os números fornecidos para a função. Ela usa um simples comando `if..else` para encontrar o maior valor e *retorna* esse valor.
Note que um comando `return` sem um valor é equivalente a `return None`. `None` é um tipo especial em Python que representa um valor nulo. Por exemplo, é usado para indicar que uma variável não possui nenhum valor quando tem o valor `None`.
Cada função implicitamente contém um comando `return None` no final, a menos que você tenha escrito seu seu próprio comando `return`. Você pode confirmar isso executando a `print someFunction()` onde a função `someFunction` não usa o comando `return`, tal como:

```
def someFunction():
    pass
```

O comando `pass` é usado em Python para indicar um bloco de comandos vazio.

Nota

> Existe uma função embutida chamada `max` que já implementa a funcionalidade 'encontrar máximo'. Portanto, use esta função sempre que possível.

DocStrings

Python possui um recurso chamado *strings de documentação*, usualmente conhecidas pelo seu nome mais curto, *docstrings*. DocStrings são uma ferramenta importante da qual você deve fazer uso, uma vez que ela ajuda a documentar o programa e o torna mais fácil de entender. Python has a nifty feature called*documentation strings*, usually referred to by its shorter name *docstrings*. DocStrings are an important tool that you should make use of since it helps to document the program better and makes it

more easy to understand. Surpreendentemente, podemos até obter a docstring de, digamos, uma função, durante a execução do programa!
Exemplo:

```
#!/usr/bin/python
# Filename: func_doc.py

def printMax(x, y):
    '''Imprime o maior entre dois números.

    Os dois valores devem ser inteiros.'''
    x = int(x) # converte para inteiro, se possível
    y = int(y)

    if x > y:
        print(x, 'é o máximo')
    else:
        print(y, 'é o máximo')

printMax(3, 5)
print(printMax.__doc__)
```

Saída:

```
$ python func_doc.py

5 é o máximo

Imprime o maior entre dois números.

    Os dois valores devem ser inteiros.
```

Como funciona:
Uma string na primeira linha lógica de uma função é o *docstring* para essa função. Note-se que docstrings também se aplicam a módulos e classes, que vamos aprender sobre nos respectivos capítulos. A convenção seguida para um docstring é uma string de múltiplas linhas onde a primeira linha começa com uma letra maiúscula e termina com um ponto. Em seguida, a segunda linha fica em branco, seguida de uma explicação detalhada a partir da terceira linha. É *fortemente recomendado* que você siga esta convenção para todos os seus docstrings em todas as suas funções não-triviais.
Podemos acessar a docstring da função `printMax` usando o atributo (nome pertencente a) `__doc__` (note o *duplo sublinhado*) da função. Lembre-se que Python trata *tudo* como um objeto e isso inclui funções. Vamos saber mais sobre os objetos no capítulo sobre classes.
Se você já usou o `help()` em Python, então você já viu o uso de docstrings! O que ele faz é apenas buscar o atributo `__doc__` dessa função e o exibir para você de uma forma elegante. Você pode experimentá-lo na função acima - basta incluir`help(printMax)` no seu programa. Lembre-se de pressionar a tecla `q` para sair do`help()`.
Ferramentas automatizadas podem recuperar a documentação do seu programa desta maneira. Portanto, eu *recomendo fortemente* que você use docstrings para qualquer função não-trivial que você escreva. O comando `pydoc` que vem com sua distribuição Python funciona de forma semelhante a `help()` usando docstrings.

Anotações

Funções possuem outra funcionalidade avançada chamada anotações, que são um modo estiloso de anexar informação adicional para cada um dos parâmetros e também para o valor de retorno. Como o interpretador Python não interpreta estas anotações (tal funcionalidade é responsabilidade de bibliotecas de terceiros), pularemos esta funcionalidade desta discussão. Se você está interessado em ler mais sobre anotações, por favor veja a Proposta de Melhoria do Python No. 3107.

Sumário

Vimos vários aspectos das funções mas note que não cobrimos todos os seus aspectos. Entretanto, cobrimos a maioria das funções de Python que você precisará dia após dia.
E seguida, veremos como usar e criar módulos de Python.

Python pt-br:Modulos

Introdução

Você viu como pode reutilizar códigos em seu programa através da definição apenas uma vez de funções. Que tal se você quisesse reutilizar um certo número de funções em outros programas que você escrever? Como você poderia ter adivinhado, a resposta está em módulos. Um módulo é basicamente um arquivo contendo todas as funções e variáveis que você definiu.
Para reutilizar o módulo em outros programas, o nome do módulo **deve** ter uma extensão `.py`.
Um módulo pode ser *importado* por um outro programa pra fazer uso da sua funcionalidade. Assim é também como nós podemos usar a biblioteca padrão de Python. Primeiro, nós veremos como utilizar os módulos da biblioteca padrão.
Exemplo:

```
#!/usr/bin/python
# Nome do arquivo: using_sys.py

import sys

print('Os argumentos da linha de comando são:')
for i in sys.argv:
    print(i)

print('\n\nO PYTHONPATH é', sys.path, '\n')
```

Saída:

```
$ python using_sys.py nos somos argumentos

Os argumentos da linha de comando são:

test.py

nos

somos

argumentos




O PYTHONPATH é ['C:\\tmp', 'C:\\Python30\\python30.zip',

'C:\\Python30\\DLLs', 'C:\\Python30\\lib', 'C:\\Python30\\lib\\plat-win',

'C:\\Python30', 'C:\\Python30\\lib\\site-packages']
```

Como Funciona:
Primeiro nós *importamos* o módulo `sys` usando o comando `import`. Basicamente, isso se traduz em nós dizermos a Python que queremos empregar este módulo. O módulo `sys` contém funcionalidade relacionada ao interpretador de Python e o seu ambiente, ou seja, o *sys*tema.
Quando Python executa o comando `import sys`, procura pelo módulo `sys.py` em um dos diretórios listados na sua variável `sys.path`. Se o arquivo é encontrado, então os comandos do bloco principal

(main) do módulo são executados e o módulo é tornado *disponível* para seu uso. Observe que a inicialização é realizada apenas a *primeira* vez que nós importamos o módulo.
A variável `argv` no módulo `sys` é acessada utilizando-se a notação por pontos (dotted) isto é `sys.argv`. Ela claramente indica que este nome é parte do módulo`sys`. Uma outra vantagem dessa abordagem é que o nome não colide com qualquer variável `argv` usada em seu programa.
A variável `sys.argv` é uma *lista* de strings (listas (lists) são explicadas em detalhes em um capítulo posterior). Especificamente, `sys.argv` contém a lista de*argumentos da linha de comando*, ou seja, os argumentos que são passados para o seu programa usando a linha de comando.
Se você estiver usando uma IDE para escrever e executar esses programas, procure por uma forma de especificar os argumentos da linha de comando ao seu programa nos menus.
Aqui, quando nós executamos `python using_sys.py nos somos argumentos` nós executamos o módulo `using_sys.py` com o comando `python` e as outras coisas que seguem são argumentos passados ao programa. Python guarda os argumentos da linha de comando na variável `sys.argv` para nosso uso.
Lembre-se, o nome do script que está em execução é sempre o primeiro argumento na lista `sys.argv`. assim, neste caso nós
teremos `'using_sys.py'`como `sys.argv[0]`, `'nos'` como `sys.argv[1]`, `'somos'` como `sys.argv[2]` e`'argumentos'` como `sys.argv[3]`. Note que Python começa a contagem a partir de 0 e não de 1.
A variável `sys.path` contém a lista de nomes de diretórios de onde os módulos são importados. Observe que a primeira string em `sys.path` está vazia - esta string vazia indica que o diretório corrente é também parte de `sys.path` a qual é a mesma que a variável de ambiente `PYTHONPATH`. Isso significa que você pode importar diretamente módulos localizados no diretório corrente. Se não for o caso, você terá que colocar seu módulo em um dos diretórios listados em`sys.path`.

Arquivos .pyc Byte-compilados

A importação de módulos é uma ação relativamente custosa, de modo que Python realiza alguns truques para torná-la mais rápida. Uma maneira é a criação de arquivos *byte-compilados* com a extensão `.pyc` que é uma forma intermediária em que Python transforma o programa (lembre-se da seção de introdução em Como Python Funciona?). Este arquivo `.pyc` é útil quando você importar o módulo a próxima vez de um programa diferente - será muito mais rápido pois uma parte do procedimento requerido para a importação do módulo já está feita. Além disso, estes arquivos byte-compilados são independentes da plataforma.

Nota

> Esses arquivos `.pyc` são usualmente criados no mesmo diretório dos arquivos `.py` correspondentes. Se Python não tiver permissão para escrever arquivos naquele diretório, então os arquivos `.pyc` não serão criados.

O Comando from .. import

Se você quiser importar diretamente a variável `argv` no seu programa (para evitar ficar digitando `sys.` todas as vezes), então você pode usar o comando`from sys import argv`. Se você quiser importar todos os nomes usados no módulo `sys`, então você pode usar o oomando `from sys import *`. Isso funciona com qualquer módulo.
Em geral, você **deve evitar** o uso desse comando e, ao invés, utilizar o comando`import`, pois dessa forma você evitará colisão de nomes e seu programa ficará mais legível.

O __name__ de um Módulo

Todo módulo tem um nome e comandos em um módulo podem encontrá-lo. Isso é conveniente na situação particular em que se proecisa descobrir se o módulo está sendo executado por si só ou está sendo importado. Como mencionado anteriormente, quando um módulo é importado pela primeira vez, o bloco principal (main) daquele módulo é executado. Nó podemos usar esse conceito para executar o bloco apenas se o programa for usado por si só e não quando for importado por outro módulo. Isso pode ser alcançado usando o atributo`__name__` do módulo.
Exemplo:

```
#!/usr/bin/python
# Nome do arquivo: using_name.py
```

```
if __name__ == '__main__':
    print('Este programa está sendo executado por si só')
else:
    print('Eu estou sendo importado de outro módulo')
```

Saída:

```
$ python using_name.py

Este programa está sendo executado por si só



$ python

>>> import using_name

Eu estou sendo importado de outro módulo

>>>
```

Como Funciona:

Todo módulo em Python tem o seu `__name__` definido e se este é `'__main__'`, isso implica que o módulo está sendo executado por si só pelo usuário e nós poderemos adotar as ações apropriadas.

Produzindo os Seus Próprios Módulos

A criação de seus próprios módulos é fácil, pois você já tem feito isso o tempo todo! Isso é porque todo programa em Python é também um módulo. Você apenas tem que se assegurar que possua a extensão `.py`. O próximo exemplo deve tornar isso claro.

Exemplo:

```
#!/usr/bin/python
# Nome do arquivo: mymodule.py

def sayhi():
    print('Olá, este é meu módulo falando.')

__version__ = '0.1'

# Fim de mymodule.py
```

O código acima é um *módulo* de amostra. Como você pode ver, não existe nada particularmente especial a respeito dele em comparação com os nossos programas usuais em Python. Veremos agora como usar este módulo em nossos outros programas em Python.

Lembre-se que os módulos devem ser colocados no mesmo diretório que o programa que os importa, ou então deve estar em um dos diretórios listados em `sys.path`.

```
#!/usr/bin/python
# Nome do arquivo: mymodule_demo.py

import mymodule

mymodule.sayhi()
print 'Versão', mymodule.__version__
```

Saída:

```
$ python mymodule_demo.py

Olá, este é mymodule falando.
```

```
Versão 0.1
```

Como Funciona:
Note que nós empregamos a notação dos pontos para acessar os membros do módulo. Python utiliza bastante a mesma notação que lhe dá a característica 'Pythonica' a ela, e modo que não temos que ficar aprendendo novas maneiras de fazer as coisas.
Aqui está a versão utilizando a sintaxe `from..import`.

```
#!/usr/bin/python
# Nome do arquivo: mymodule_demo2.py

from mymodule import sayhi, __version__
# Alternativa:
# from mymodule import *

sayhi()
print('Versão', __version__)
```

A saída de `mymodule_demo2.py` é a mesma que a saída de `mymodule_demo.py`.
Note que já houvesse um nome `__version__` declarado no módulo que importa mymodule, haveria uma colisão. Isso é altamente provável pois é prática comum para cada módulo declarar-se a sua versão usando esse nome. Daí é sempre recomendado optar pelo comando `import`, mesmo se tornar o seu programa um pouco mais longo.

Zen de Python

> Um dos princípios-guia de Python é que 'Explicito é melhor do que implícito'. Execute `import this` para aprender mais.

A Função dir

Você pode usar a função interna (built-in) `dir` para listar os indetificadores que um objeto define. Por exemplo, para um módulo, os identificadores incluem as funções, classes e variáveis definidas naquele módulo.
Quando você fornece o nome do módulo à função `dir()`, ela retorna a lista dos nomes definidos naquele módulo. Quando nenhum argumento é fornecido, ela retorna a lista de nomes definidos no módulo corrente.
Exemplo:

```
$ python

>>> import sys # Obtenha a lista de atributos, neste caso, do módulo sys

>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__package__', '__s
tderr__', '__stdin__', '__stdout__', '_clear_type_cache', '_compact_freelists',
'_current_frames', '_getframe', 'api_version', 'argv', 'builtin_module_names', '
byteorder', 'call_tracing', 'callstats', 'copyright', 'displayhook', 'dllhandle'
, 'dont_write_bytecode', 'exc_info', 'excepthook', 'exec_prefix', 'executable',
'exit', 'flags', 'float_info', 'getcheckinterval', 'getdefaultencoding', 'getfil
esystemencoding', 'getprofile', 'getrecursionlimit', 'getrefcount', 'getsizeof',
 'gettrace', 'getwindowsversion', 'hexversion', 'intern', 'maxsize', 'maxunicode
', 'meta_path', 'modules', 'path', 'path_hooks', 'path_importer_cache', 'platfor
m', 'prefix', 'ps1', 'ps2', 'setcheckinterval', 'setprofile', 'setrecursionlimit
', 'settrace', 'stderr', 'stdin', 'stdout', 'subversion', 'version', 'version_in
fo', 'warnoptions', 'winver']

>>> dir() # obtenha a lista dos atributos do módulo corrente
['__builtins__', '__doc__', '__name__', '__package__', 'sys']
```

```
>>> a = 5  # cria uma nova variável 'a'

>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'a', 'sys']

>>> del a  # delete/remove um nome

>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'sys']

>>>
```

Como Funciona:

Primeiro nós vemos o uso de `dir` sobre o módulo importado. Podemos ver a enorme lista de atributos que ele contém.

Em seguida, nós vemos a função `dir` sem nenhum parâmetro passado a ela. Por padrão, ela retorna a lista dos atributos do módulo corrente. Note que a lista dos módulos importados também faz parte dessa lista.

A fim de observar a ação de `dir`, nós definimos uma nova variável e atribuímos um valor a ela e então verificamos que existe um valor adicional na lista, com o mesmo nome da variável. Nós removemos a variável/atributo do módulo corrente usando o comando `del` e a alteração é refletida novamente na saída da função`dir`.

Uma nota sobre `del` - este comando é usado para *deletar* a variável/nome e depois de executado o comando, neste caso `del a`, voce não poderá acessar a variável a - é como se ela nunca tivesse existido antes.

Note que a função `dir()` funciona para *qualquer* objeto. Por exemplo, execute`dir(print)` para conhecer os atributos da função print, ou `dir(str)` para os atributos da classe str.

Packages

Nesse ponto, você já deve ter começado a observar a hierarquia da organização de seus programas. As variáveis usualmente vão dentro das funções. Funções e variáveis globais usualmente vão dentro dos módulos. Que tal se você quisesse organizar os módulos? Este o momento em que entram em cena as packages (pacotes).

Packages são apenas pastas contendo módulos com um arquivo `__init__.py`que indica a Python que esta pasta é especial porque contém módulos de Python.

Vamos dizer que você quer criar uma package denominada 'world' com as subpackages 'asia', 'africa', etc. e estas subpackages por sua vez contêm módulos como 'india', 'madagascar', etc.

Esta é como você estruturaria as pastas:

```
- <alguma pasta presente em the sys.path>/

    - world/

        - __init__.py

        - asia/

            - __init__.py

            - india/

                - __init__.py

                - foo.py
```

```
- africa/
    - __init__.py
    - madagascar/
        - __init__.py
        - bar.py
```

Packages são apenas uma conveniência para organizar módulos hierarquicamente. Nós veremos muitos exemplos disso em biblioteca padrão.

Resumo

Da mesma forma que funções são partes reutilizáveis de programas, os módulos são programas reutilizáveis. Packages são uma outra hierarquia para organizar módulos. A biblioteca padrão que vem com Python representa um exemplo de tais conjuntos de packages e módulos.
Nós vimos como usar esses módulos e criar os nossos próprios módulos.
A seguir, aprenderemos sobre interessantes conceitos denominados estruturas de dados.

Python pt-br:Estruturas de Dados

Introdução

Estruturas de dados são basicamente isso - são *estruturas* que podem conter alguns *dados* juntos. Em outras palavras, elas são utilizadas para guardar uma coleção de dados relacionados entre si.
Existem quatro estruturas de dados internas (built-in) em Python - lista, tupla, dicionário e conjunto (set). Nós veremos como usar cada uma delas e de que modo elas nos facilitam a vida.

Lista

Uma `list` (lista) é uma estrutura de dados que contém uma coleção ordenada de itens, ou seja, você pode guardar uma *sequência* de itens em uma lista. Isso é fácil de imaginar se você pensar em uma lista de compras na qual você tem uma lista de itens para comprar, exceto que você provavelmente terá cada item em uma linha separada, ao passo que Python coloca uma vírgula entre eles.
A lista de itens deverá estar dentro de colchetes (square brackets), de modo que Python entende que você está especificando uma lista. Uma vez que você criou a lista, pode adicionar, remover ou fazer buscas por itens dela. Desde que nós podemos adicionar e remover itens, dizemos que uma lista é um tipo *mutável* de dados, ous seja, este tipo pode ser alterado.

Rápida Introdução a Objetos e Classes

Embora eu tenha retardado a discussão sobre objetos e classes até agora, uma pequena explanação torna-se imediatamente necessária, tal que você possa entender melhor as listas. Exploraremos esse tópico em mais detalhes em seu próprio capítulo.
Uma lista é um exemplo de utilização de objetos e classes. Quando nós usamos a variável `i` e atribuímos um valor, digamos, inteiro `5` a ela, você pode raciocinar como se criasse um **objeto** (isto é uma instância) `i` de uma **classe** (ou seja tipo)`int`. De fato, você pode ler `help(int)` para melhor entender isso.
Uma classe pode ter **métodos** ou seja funções definidas para uso com respeito unicamente àquela classe. Você pode usar essas peças de funcionalidade apenas quando tem um objeto daquela classe. Por exemplo, Python fornece um método `append` para a classe `list`, o que permite a você adicionar um item ao final da lista. Por exemplo, `minhalista.append('um item')` adicionará aquela string ao final da lista `minhalista`. Note o emprego da notação dos pontos para acessar métodos dos objetos.
Uma classe pode também possuir **campos** que não são nada mais do que variáveis definidas para uso com respeito unicamente àquela classe. Você pode utilizar aquelas variáveis/nomes apenas quando tem um objeto daquela classe. Campos são também acessados com a notação dos pontos, por exemplo,`minhalista.campo`.
Exemplo:

```
#!/usr/bin/python
# Nome do arquivo: using_list.py

# Esta é a minha lista de compras
shoplista = ['maçã', 'manga', 'cenoura', 'banana']

print('Eu tenho', len(shoplista), 'itens para comprar.')

print('Estes itens são:', end=' ')
for item in shoplista:
    print(item, end=' ')

print('\nTambém tenho que comprar arroz.')
shoplista.append('arroz')
print('Minha lista de compras é agora', shoplista)
```

```
print('Vou colocar a minha lista em ordem agora')
shoplista.sort()
print('A minha lista ordenada é', shoplista)

print('O primeiro item que comprarei é', shoplista[0])
olditem = shoplista[0]
del shoplista[0]
print('Eu comprei o', olditem)
print('Minha lista de compras é agora', shoplista)
```

Saída:

```
$ python using_list.py

Eu tenho 4 itens para comprar.

Estes itens são: maçã manga cenoura banana

Também tenho que comprar arroz.

Minha lista de compras é agora ['maçã', 'manga', 'cenoura', 'banana', 'arroz']

Vou colocar a minha lista em ordem agora

A minha lista ordenada é ['arroz', 'banana', 'cenoura', 'maçã', 'manga' ]

O primeiro item que comprarei é arroz

Eu comprei o arroz

Minha lista de compras é agora 'banana', 'cenoura', 'maçã', 'manga' ]
```

Como Funciona:

A variável `shoplista` é uma lista de compras para alguém que está indo ao mercado. Em `shoplista` nós guardamos strings do nomes dos itens a serem comprados, mas você pode adicionar *qualquer espécie de objeto* a uma lista, incluindo números e até mesmo outras listas.

Nós também empregamos o laço `for..in` para iterar sobre os itens da lista. Agora você deve ter percebido que uma lista é também uma sequência. A especialidade das sequências será discutida em uma seção posterior.

Observe o uso da palavra-chave `end` como argumento na função `print`, para indicar que nós queremos terminar a saída com um espaço, ao invés da costumeira quebra-de-linha.

Em seguida, nós adicionamos um item à lista usando o método `append` do objeto lista, como já foi discutido antes. Então, nós verificamos que o item foi efetivamente acrescentado à lista através da impressão do conteúdo da lista, obtida simplesmente passando a lista ao comando `print`, que a imprime sem problemas.

Daí nós ordenamos a lista por meio do método `sort` do objeto lista. É importante compreender que este método afeta a a própria lista e que não retorna uma lista modificada - isso é diferente da maneira pela qual a string funciona. É o que nós queremos salientar quando dizemos que as listas são *mutáveis* e as strings são*imutáveis*.

A seguir, quando nós terminamos de comprar um item no mercado, queremos retirá-lo da lista. Alcançamos isso por meio do comando `del`. Aqui nós mencionamos qual o item da lista que desejamos suprimir e o comando `del` o remove da lista para nós. Especificamos que queremos remover o primeiro item da lista então usamos `del shoplist[0]` (lembre-se que Python começa a contagem a partir do 0.)

Se você quiser conhecer todos os métodos definidos para o objeto lista, veja`help(list)` para os detalhes.

Tupla

Tuplas (tuples) são exatamente como listas, exceto que são **imutáveis**, como as strings, isto é, você não pode modificar tuplas. Tuplas são definidas especificando itens separados por vírgulas dentro de um par de parênteses. As tuplas são costumeiramente usadas em casos em que um comando ou uma função definida pelo usuário pode seguramente assumir que uma coleção de valores, ou seja, a tupla de valores não será alterada.

Exemplo:

```
#!/usr/bin/python
# Nome do arquivo: using_tuple.py

zoo = ('lobo', 'elefante', 'pinguim')
print('O Número de animais no zoo é', len(zoo))

novo_zoo = ('macaco', 'golfinho', zoo)
print('O Número de animais no novo zoo é', len(novo_zoo))
print('Todos os animais no novo zoo são', novo_zoo)
print('Os animais trazidos do antigo zoo são', novo_zoo[2])
print('O último animal trazido do antigo zoo é', novo_zoo[2][2])
```

Output:

```
$ python using_tuple.py

O Número de animais no zoo é 3

O Número de animais no novo zoo é 3

Todos os animais no novo zoo são ('macaco', 'golfinho', ('lobo', 'elefante',
'pinguim'))

Os animais trazidos do antigo zoo são ('lobo', 'elefante', 'pinguim')

O último animal trazido do antigo zoo é pinguim
```

Como Funciona:

A variável `zoo` refere-se a uma tupla de itens. Nós vemos que a função `zoo` pode ser utilizada para se obter o comprimento da tupla. Isso também indica que a tupla também é uma sequência.

Nós estamos mudando aqueles animais para um novo zoo, uma vez que o antigo está para ser fechado. Dessa forma, a tupla `novo_zoo` contém alguns animais que já estavam lá, juntamente com aqueles trazidos do velho zoo. De volta a realidade, observe que uma tupla dentro de outra tupla não perde a sua indentidade.

Podemos acessar os itens em uma tupla especificando a posição do item dentro de um par de colchetes exatamente como nós fizemos para listas. Esse é chamado operador de *indexação*. Nós acessamos o terceiro item em `novo_zoo`pela especificação `novo_zoo[2]` e acessamos o terceiro item dentro do terceiro item na tupla `novo_zoo` especificando `novo_zoo[2][2]`. É muito simples uma vez que você domine o jeito.

Tuplas com 0 ou 1 item

> Uma tupla vazia é construída com um par de parênteses como `vazia = ()`. Entretanto, com um único item já não é tão simples. Você deve especificar usando uma vírgula em seguida ao primeiro (e único) item, tal que Python possa diferenciar entre uma tupla e um par de parênteses que cercam uma expressão, ou seja, você tem que especificar `singleton = (2 , )`, se quiser uma tupla contendo o item `2`.

Nota para programadores em Perl

> Uma lista dentro de uma lista não perde sua identidade, listas não são "achatadas", como em Perl. O mesmo se aplica a uma tupla dentro de uma tupla, ou uma tupla em uma lista, ou uma lista em uma tupla, etc. Do ponto de vista de Python, são apenas objetos alocados usando um outro objeto, apenas isso.

Dicionário

Um dicionário (dictionary) é como uma agenda de endereços na qual você pode encontrar ou endereço ou detalhes de contato de uma pessoa, conhecendo apenas o nome dela, isto é, associamos as **keys (chaves)** (nome) com **values (valores)** (detalhes). Note que a key deve ser única, da mesma forma que você não poderá encontrar a informação correta se houverem duas pessoas com exatamente o mesmo nome.

Observe que você pode utilizar somente objetos imutáveis (como strings) como keys de um dicionário, mas pode usar objetos imutáveis e mutáveis como values do dicionário. Isso basicamente se traduz em que você deve usar apenas objetos simples como keys.

Pares de keys e values são especificados em um dicionário pela notação `d = {key1 : value1, key2 : value2 }`. Observe que os pares key/value estão separados por uma vírgula e todo o conjunto está dentro de um par de chaves.

Lembre-se que os pares key/value em um dicionário não estão de forma alguma ordenados. Se quiser um ordem particular, você mesmo terá que ordená-los antes de usá-los.

Os dicionários que você utilizará são instâncias/objetos da classe `dict`.

Exemplo:

```
#!/usr/bin/python
# Nome do arquivo: using_dict.py

# 'ab' é diminutivo para 'a'ddress'b'ook

ab = {   'Swaroop'    : 'swaroop@swaroopch.com',
         'Larry'      : 'larry@wall.org',
         'Matsumoto'  : 'matz@ruby-lang.org',
         'Spammer'    : 'spammer@hotmail.com'
     }

print("Swaroop's address is", ab['Swaroop'])

# Deletando um par key/value
del ab['Spammer']

print('\nThere are {0} contacts in the address-book\n'.format(len(ab)))

for name, address in ab.items():
     print('Contact {0} at {1}'.format(name, address))

# Adiconando um par key/value
ab['Guido'] = 'guido@python.org'

if 'Guido' in ab: # OU ab.has_key('Guido')
     print("\nGuido's address is", ab['Guido'])
```

Saída:

```
$ python using_dict.py

Swaroop's address is swaroop@swaroopch.com
```

```
There are 3 contacts in the address-book


Contact Swaroop at swaroop@swaroopch.com

Contact Matsumoto at matz@ruby-lang.org

Contact Larry at larry@wall.org


Guido's address is guido@python.org
```

Como Funciona:
Nós criamos um dicionário `ab` usando a notação já discutida. Então acessamos os pares key/value especificando a key usando o operador de indexação como discutido no contexto de listas e tuplas. Observe a sintaxe simples.
Nós podemos deletar os pares key/value usando nosso velho amigo - o comando`del`. Simplesmente especificamos o dicionário o operador de indexação para a key a ser removida e passamos tudo ao comando `del`. Não há a necessidade de conhecer o value correspondente à key para esta operação.
Em seguida, nós acessamos cada par key/value usando o método `items` do dicionário, o qual retorna uma lista de tuplas, em que cada tupla contém um par de itens - a key seguida por um value. Nós recuperamos esse par e o atribuímos as variáveis `name` e `address` correspondentemente para cada par, usando o laço de `for..in` e então imprimimos esses valores no bloco de for.
Podemos adicionar novos pares key/value, simplesmente empregando o operador de indexação para acessar a key e atribuirmos aquele value, como fizemos para Guido no caso acima.
Nó podemos verificar se um par key/value existe, usando o operador `in` ou mesmo o método `has_key` da classe `dict`. Você pode ver a documentação para a lista completa de métodos da classe `dict`, usando `help(dict)`.

Argumentos em Keyword (palavras-chave) e Dicionários

> Em um tópico distinto, se você tiver empregado argumentos em keywords em suas funções, você já usou dicionários! Pense um pouco sobre isso - o par key/value é especificado por você na lista de parâmetros da definição da função e quando você acessa variáveis dentro de sua função, trata-se apenas de uma acesso a key de um dicionário (que é chamado de *tabela de símbolos* em terminologia de compiladores).

Sequências

Listas, tuplas, strings e arquivos são exemplos de sequências, mas o que as sequências tem de tão especial? Duas das principais características de uma sequência são a operação de **indexação'***que nos permite recuperarmos diretamente um item particular e a operação de slicing* (fatiamento) que nos permite recuperarmos uma fatia de uma sequência, ou seja, uma parte da sequência.
Exemplo:

```
#!/usr/bin/python
# Nome do arquivo: seq.py

shoplist = ['apple', 'mango', 'carrot', 'banana']
name = 'swaroop'

# Operação de Indexação ou 'Subscrição'
print('Item 0 is', shoplist[0])
print('Item 1 is', shoplist[1])
print('Item 2 is', shoplist[2])
```

```
print('Item 3 is', shoplist[3])
print('Item -1 is', shoplist[-1])
print('Item -2 is', shoplist[-2])
print('Character 0 is', name[0])

# Slicing (fatiamento) sobre uma lista
print('Item 1 to 3 is', shoplist[1:3])
print('Item 2 to end is', shoplist[2:])
print('Item 1 to -1 is', shoplist[1:-1])
print('Item start to end is', shoplist[:])

# Slicing (fatiamento) sobre uma string
print('characters 1 to 3 is', name[1:3])
print('characters 2 to end is', name[2:])
print('characters 1 to -1 is', name[1:-1])
print('characters start to end is', name[:])
```

Saída:

```
$ python seq.py

Item 0 is apple

Item 1 is mango

Item 2 is carrot

Item 3 is banana

Item -1 is banana

Item -2 is carrot

Character 0 is s

Item 1 to 3 is ['mango', 'carrot']

Item 2 to end is ['carrot', 'banana']

Item 1 to -1 is ['mango', 'carrot']

Item start to end is ['apple', 'mango', 'carrot', 'banana']

characters 1 to 3 is wa

characters 2 to end is aroop

characters 1 to -1 is waroo

characters start to end is swaroop
```

Como Funciona:

Primeiramente, nós vemos como utiizar os índices para obter itens individuais de uma sequência. Isso é também conhecido como *operação de subscrição*. Quando você especifica um número dentro de colchetes a

uma sequência, como mostrado acima, Python irá recuperar para você o item correspondente aquela posição na sequência. Lembre-se que Python começa a contagem dos números a partir do 0.
Daí, `shoplist[0]` retorna o primeiro item e `shoplist[3]` retorna o quarto item na sequência `shoplist`.
O índice pode também ser um número negativo, em cujo caso, a posição é calculada a partir do fim da sequência. Assim, `shoplist[-1]` refere-se ao último item da sequência e `shoplist[-2]` recuoera o penúltimo item da sequência.
A operação de slicing é empregada especificando-se o nome da sequência seguido de um par opcional de números separados por 'dois pontos' dentro de colchetes. Note que isso é bem similar a operação de indexação que você vem usando até agora. Lembre-se que os números são opcionais mas o 'dois pontos' não é.
O primeiro número (antes do 'dois pontos') na operação de slicing refere-se a posição de onde a fatia começa e o segundo número (depois do 'dois pontos') indica onde a fatia terminará. Se o primeiro número não for especificado, Python começará no início da sequência. Se o segundo número não existir, Python terminará no final da sequência. Note que a fatia retornada *começa* na posição de início e termina imediatamente antes da posição do *fim*, ou seja, a posição de início é incluída, mas a posição do fim é excluída na fatia da sequência.
Então, `shoplist[1:3]` retorna uma fatia de uma sequência que começa na posição 1, inclui a posição 2 mas para na posição 3 e assim uma *fatia* de dois itens é retornada. Similarmente, `shoplist[:]` retorna uma cópia da sequência toda.
Você pode também efetuar slicing com números negativos. Os números negativos são usados para posições a partir do fim da sequência. Por exemplo,`shoplist[:-1]` retornará uma fatia da sequência que exclui o último item da sequência mas contém todo o resto dela.
Experimente várias combinações de especificações de slicing usando interativamente o interpretador de Python, ou seja, o prompt, de modo que você possa ver os resultados imediatamente. Uma grande coisa a respeito de sequências e que você pode acessar tuplas, listas e strings todas da mesma maneira!

Conjuntos

Conjuntos (Sets) são coleções *não ordenadas* de objetos simples. Eles são usados quando a existência de um objeto em uma coleção é mais importante do que a ordem em que está ou o número de vezes em que ocorre.
Usando sets, você pode testar para pertencimento a um conjunto, verificar se um conjunto é um subconjunto de outro, encontrar a interseção entre dois conjuntos, e assim por diante.

```
>>> bri = set(['brasil', 'russia', 'india'])
>>> 'india' in bri
True
>>> 'usa' in bri
False
>>> bric = bri.copy()
>>> bric.add('china')
>>> bric.issuperset(bri)
True
>>> bri.remove('russia')
>>> bri & bric # OR bri.intersection(bric)
{'brasil', 'india'}
```

Como Funciona:
O exemplo é razoavelmente auto-explicativo, pois envolve teoria básica de conjuntos ensinada na escola.

Referências

Quando você cria um objeto e o atribui a uma variável, esta apenas se *refere* ao objeto e não representa o objeto em si mesmo! Isto é, o nome da variável aponta para aquela parte da memória do seu computador na qual o objeto está armazenado. Isso é chamado *vinculação (binding)* do nome ao objeto.
Em geral você não precisa ficar preocupado com isso, mas existe um efeito sutil devido a referências ao qual você precosa ficar atento:
Exemplo:

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

```
# Nome do arquivo: reference.py

print('Atribuição Simples')
shoplist = ['apple', 'mango', 'carrot', 'banana']
mylist = shoplist # mylist é apenas um outro nome apontando para o mesmo objeto!

del shoplist[0] # Eu comprei o primeiro item , então eu o removo da lista

print('shoplist is', shoplist)
print('mylist is', mylist)
# note como tanto shoplist e mylist ambas imprimem a mesma lista, sem 'apple', confirmando que apontam para o mesmo objeto.

print('Copie por meio de uma fatia completa')
mylist = shoplist[:] # efetue uma cópia por meio de uma fatia completa
del mylist[0] # remova o primeiro item

print('shoplist is', shoplist)
print('mylist is', mylist)
# note que agora as duas listas são distintas
```

Saída:

```
$ python reference.py

Atribuição Simples

shoplist is ['mango', 'carrot', 'banana']

mylist is ['mango', 'carrot', 'banana']

Copie por meio de uma fatia completa

shoplist is ['mango', 'carrot', 'banana']

mylist is ['carrot', 'banana']
```

Como Funciona:

A maior parte da explicação está disponível nos próprios comentários.

O que você precisa lembrar é que quiser fazer uma cópia de uma lista ou tais espécies de sequências ou objetos complexos (não *objetos* simples como inteiros), então você deve utilizar a operação de slicing para efetuar uma cópia. Se você apenas atribuir o nome da variável a outro nome, então ambos *se referirão* ao mesmo objeto e isso pode ser um problema se você não for cuidadoso.

Nota para programadores em Perl

> Lembre-se que um comando de atribuição para listas **não** cria uma cópia. Você deverá usar a operação de slicing para efetuar uma cópia da sequência.

Mais Sobre Strings

Nós já discutimos strings em detalhes antes. O que mais há para se conhecer? Bem, você sabia que strings são também objetos e que tem métodos que fazem de tudo, desde verificar partes de uma string até retirar os espaços em branco!

As strings que você usa em seu programa são todas objetos da classe `str`. Alguns métodos úteis dessa classe são demonstrados no próximo exemplo. Para uma lista completa de tais métodos, veja em `help(str)`.

Exemplo:

```
#!/usr/bin/python
# Nome do arquivo: str_methods.py

name = 'Swaroop'  # Este é um objeto string

if name.startswith('Swa'):
    print('Sim, a string começa com "Swa"')

if 'a' in name:
    print('Sim, ela contém a string "a"')

if name.find('war') != -1:
    print('Sim, ela contém a string "war"')

delimitador = '_*_'
mylist = ['Brasil', 'Rússia', 'Índia', 'China']
print(delimitador.join(mylist))
```

Output:

```
$ python str_methods.py

Sim, a string começa com "Swa"

Sim, ela contém a string "a"

Sim, ela contém a string "war"

Brasil_*_Rússia_*_Índia_*_China
```

Como Funciona:
Aqui, nós vemos um bocado de métodos da string em ação. O método `startswith` é usado para descobrir se a string começa com uma dada string. O operador `in` é usado para verificar se uma dada string é uma parte de uma string.
O método `find` é usado para achar a posição de uma dada string em uma string ou retorna -1 se não for bem sucedido em achar a substring. A classe `str` tem também um elegante método para `join` os itens de uma sequência, com a string atuando como delimitador entre cada item da sequência e retorna uma string muito maior produzida com isso.

Resumo

Nós esxploramos detalhadamente as várias estruturas internas (built-in) de Python. Estas estruturas serão essenciais para escrever programas de um tamanho razoável.
Agora que nós já temos sob controle o básico de Python, veremos a seguir como projetar e escrever programas do mundo real em Python.

Python pt-br:Resolucao de Problemas

Nós exploramos várias partes da linguagem Python e agora daremos uma olhada em como essas partes se ajustam entre si, projetando e escrevendo um programa que faz algo útil. A idéia é aprender a escrever o seu próprio script em Python.

O Problema

O problema é *"Eu desejo escrever um programa que cria um backup de todos os meus arquivos importantes"*
Embora esse seja um problema simples, não existe informação suficiente para que possamos dar início a uma solução. Um pouco mais de **análise** é necessária. Por exemplo, como poderíamos especificar *quais* os arquivos cujo backup deveria ser feito? *Como* eles estão alocados? *Onde* eles estão alocados?
Depois de analisarmos apropriadamente o problema, nós **projetamos** o nosso programa. Nós elaboramos uma lista de coisas a respeito de como o nosso programa deveria operar. Neste caso em particular, criei a lista que se segue, com base em como *eu* quero que ele funcione. Se você desenvolver o projeto, poderá surgir com outro tipo de análise, uma vez que cada pessoa tem o seu próprio modo de fazer as coisas, o que é perfeitamente legítimo.

1. Os arquivos e diretórios objeto do backup estão especificados em uma lista.
2. O backup deve ser colocado em um diretório principal de backup.
3. O backup deve ser colocado em uma arquivo do tipo zip.
4. O nome do arquivo zip é a data e a hora correntes.
5. Usaremos o comando `zip` disponível por default em qualquer distribuição Linux/Unix padrão. Os usuários de Windows podem instalar da página do projeto e adicionar `C:\Arquivos de Programas\GnuWin32\bin` a variável de ambiente do sistema PATH, de modo análogo ao que fizemos para que o comando python fosse reconhecido. Note que você pode utilizar qualquer comando de arquivamento que desejar, desde que possua uma interface através de linha de comando, de forma que possamos passar argumentos a ele por meio de nosso script.

A Solução

Uma vez que o projeto de nosso programa encontra-se agora razoavelmente estável, podemos escrever o código que representa uma implementação de nossa solução.

```
#!/usr/bin/python
# Nome do arquivo: backup_ver1.py

import os
import time

# 1. Os arquivos e os diretórios a sofrerem backup estão especificados em uma lista.
origem = ['"C:\\Meus Documentos"', 'C:\\Code']
# Note que nós tivemos que usar aspas duplas dentro da string devido aos nomes com espaços dentro dela.

# 2. O backup deve ser colocado em um diretório de backup principal.
dir_alvo = 'E:\\Backup'  # Lembre-se de alterar para aquele que você estiver utilizando.

# 3. O backup deve ser colocado em uma arquivo do tipo zip.
# 4. O nome do arquivo zip é a data e a hora correntes.
alvo = dir_alvo + os.sep + time.strftime('%Y%m%d%H%M%S') + '.zip'

# 5. Usamos o comando zip para colocar os arquivos em um arquivo zip.
comando_zip = "zip -qr {0} {1}".format(alvo, ' '.join(origem))

# Execute o backup
```

```
if os.system(comando_zip) == 0:
    print('Backup bem-sucedido em', alvo)
else:
    print('Backup FALHOU')
```

Saída:

```
$ python backup_ver1.py


Backup bem-sucedido em E:\Backup\20080702185040.zip
```

Agora, nós nos encontramos em uma fase de **testes**, quando nós verificamos que o nosso programa funciona de forma apropriada. Se ele não se comportar como esperado, então nós temos que **debugar** o nosso programa, ou seja, remover os*bugs* (erros) do programa.
Se o programa acima não funcionar para você, coloque um `print(comando zip)` imediatamente antes da chamada para `os.system` e execute o programa. Agora copie/cole o comando_zip impresso no 'prompt' do shell e veja se ele executa adequadamente por si mesmo. Se o comando falhar, verifique no manual do comando zip o que poderia estar errado. Se o comando for bem sucedido, então verifique se o programa em Python corresponde exatamente ao prograna escrito acima.
Como funciona:
Você notará como nós convertemos o nosso projeto em um procedimento passo-a-passo.
Nós fazemos uso dos módulos `os` e `time`, antes de tudo importando-os. Então, nós especificamos os arquivos e diretórios cujo backup deve ser realizado na lista tt>origem</tt>. O diretório alvo é onde nós armazenamos todos os arquivos do backup e este está especificado pela variável `dir_alvo`. O nome do arquivo zip que iremos criar é a data e hora correntes, os quais nós obtemos utilizando a função `time.strftime()`. Ele receberá também a extensão zip e será posto no diretório `dir_alvo`.
Observe o uso da variável `os.sep` - esta fornece o separador de diretórios de acordo com o sistema operacional, ou seja, será `'/'` em Linux, Unix, será `'\\'`em Windows e `':'` em Mac OS. O uso de `os.sep`, ao invés desses caracteres diretamente, tornará o nosso programa portável e capaz de operar sob aqueles sistemas.
A função `time.strftime()` recebe uma especificação tal como a que usamos no programa acima. A especificação `%Y` será substituída pelo ano sem o século. A especificação `%m` será substituída pelo mes como um número decimal entre `01` e`12` e assim por diante. A lista completa de tais especificações pode ser encontrada no Python Reference Manual(Manual de Referência de Python).
Nós criamos o nome do arquivo alvo zip utilizando o operador adição, o qual*concatena* as strings, isto é, combina conjuntamente as strings e retorna uma nova string. Então, nós criamos uma string `comando_zip` que contém o comando que iremos executar. Você pode verificar se esse comando funciona, executando-o no shell (terminal de Linux ou no 'prompt' do DOS).
Ao comando `zip` que nós estamos usando, são passadas algumas opções e parâmetros. A opção The `-q` é empregada para indicar que o comando zip deve operar de forma silenciosa (**q**uiet). A opção -r especifica que o comando zip deve trabalhar **r**ecursivamente sobre os diretórios, ou seja, deverá incluir todos os subdiretórios e arquivos. As duas opções são combinadas e especificadas como `-qr`. As opções são seguidas pelo nome de um arquivo zip a ser criado seguido pela lista de arquivos e diretórios a realizar o backup. Nós convertemos a lista`origem` em uma string, usando o método `join` de strings, o qual já vimos como empregar.
Então, nós finalmente *executamos* o comando usando a função `os.system`, que executa o comando como se o estivesse fazendo pelo *sistema* operacional, isto é, no shell - ele retorna `0` se o comando foi bem sucedido, retornando um número de erro, em caso contrário.
Dependendo do resultado do comando, nós imprimimos a mensagem apropriada se o backup falhou ou se foi bem sucedido.
Então é isso, nós criamos um script para realizar um backup de nossos arquivos importantes!

Nota aos Usuários de Windows

> Ao invés de sequências de escape com duplo 'backslash' você pode também eepregar raw strings. Por exemplo, use `'C:\\Documentos'` ou`r'C:\Documentos'`.

Entretanto, **não** utilize `'C:\Documentos'`, uma vez que você vai acabar encontrando uma sequência desconhecida de escape `\D`.

Agora que nós temos um script de backup operacional, podemos usá-lo em qualquer situação em que desejamos obter um backup de arquivos. Os usuários de Linux/Unix são aconselhados a empregar o metodo_executavel como discutido anteriormente de modo que possam executar o script de backup a qualquer momento e em qualquer lugar. Esta é chamada a fase de **operação** ou de **emprego** do software.

O programa acima funciona apropriadamente, mas (usualmente) os primeiros programas não operam exatamente como você espera. Por exemplo, poderiam haver problemas se você não tivesse projetado o programa adequadamente ou se você tivesse cometido um erro ao digitar o código, etc. Apropriadamente, você terá que voltar à fase de projeto ou terá que 'debugar' o seu programa.

Segunda Versão

A primeira versão do nosso script funciona. Entretanto, nós podemos implementar certos refinamentos a ele, de modo a que possa funcionar ainda melhor dia após dia. Esta é chamada a fase de **manutenção** do software.

Um dos refinamentos que eu achei que seria útil é um mecanismo mais adequado de atribuição do nome para os arquivos - usar a *hora* como o nome do arquivo dentro de um diretório com a *data* corrente como um diretório no diretório principal de backup. A primeira vantagem é que os seus backups estarão armazenados de uma forma hierárquica e assim muito mais fácil de gerenciar. A segunda vantagem é que o comprimento dos nomes dos arquivos será muito menor. A terceira vantagem é que diretórios separados o ajudarão a facilmente verificar se você já realizou o backup do dia, uma vez que o diretório só será criado se você tiver efetuado o backup daquele dia.

```
#!/usr/bin/python
# Nome do arquivo: backup_ver2.py

import os
import time

# 1. Os arquivos e os diretórios cujo backup será efetuado estarão especificados em uma lista.
origem = ['"C:\\Meus Documentos"', 'C:\\Code']
# Note que nós tivemos que usar aspas duplas dentro da string devido aos nomes com espaços dentro dela.

# 2. O backup deve ser colocado em um diretório de backup principal.
dir_alvo = 'E:\\Backup'  # Lembre-se de alterar para aquele que você estiver utilizando.

# 3. O backup deve ser colocado em uma arquivo do tipo zip.
# 4. O dia corrente é o nome do subdiretório no diretório principal.
hoje = dir_alvo + os.sep + time.strftime('%Y%m%d')
# A hora corrente é o nome do arquivo zip
agora = time.strftime('%H%M%S')

# Crie o diretório se ainda não estiver lá
if not os.path.exists(hoje):
    os.mkdir(hoje)  # crie diretório
    print('Bem-Sucedida Criação do Diretório', hoje)

# O nome do arquivo zip
alvo = hoje + os.sep + agora + '.zip'

# 5. Usamos o comando zip para colocar os arquivos em um arquivo zip
comando_zip = "zip -qr {0} {1}".format(alvo, ''.join(origem))

# Execute o backup
```

```
if os.system(comando_zip) == 0:
    print('Backup Bem-Sucedido em', alvo)
else:
    print('Backup FALHOU')
```

Saída:

```
$ python backup_ver2.py

Bem-Sucedida Criação do Diretório E:\Backup\20080702

Backup Bem-Sucedido em E:\Backup\20080702\202311.zip



$ python backup_ver2.py

Backup Bem-Sucedido em E:\Backup\20080702\202325.zip
```

Como funciona:

A maior parte do programa permanece a mesma. As alterações são aquelas nas quais verificamos se existe um diretório com o dia corrente como nome dentro do diretório principal utilizando a função `os.exists`. Se não existir, nós o criamos por meio da função `os.mkdir`.

Terceira Versão

A segunda versão funciona muito bem quando eu efetuo muitos backups, mas quando há um número muito grande de backups, eu estou achando difícil diferenciar para que razão são os backups! Por exemplo, eu poderia ter feito algumas alterações significativas a um programa ou apresentação, então eu gostaria a razão para aquelas alterações ao nome do arquivo zip. Isso pode ser facilmente obtido associando-se um comentário fornecido pelo usuário ao nome do arquivo zip.

Nota

> O programa a seguir não funciona, então não precisa ficar alarmado, por favor acompanhe a discussão, pois nela existe uma lição.

```
#!/usr/bin/python
# Nome do arquivo: backup_ver3.py

import os
import time

# 1. Os arquivos e os diretórios cujo backup será efetuado estarão especificados em uma lista.
origem = ['"C:\\Meus Documentos"', 'C:\\Code']
# Note que nós tivemos que usar aspas duplas dentro da string devido aos nomes com espaços dentro dela.

# 2. O backup deve ser colocado em um diretório de backup principal.
dir_alvo = 'E:\\Backup'  # Lembre-se de alterar para aquele que você estiver utilizando.

# 3. O backup deve ser colocado em uma arquivo do tipo zip.
# 4. O dia corrente é o nome do subdiretório no diretório principal.
hoje = dir_alvo + os.sep + time.strftime('%Y%m%d')
# A hora corrente é o nome do arquivo zip
agora = time.strftime('%H%M%S')

# Receba um comentário do usuário para criar o nome do arquivo zip
coment = input('Entre com um comentário --> ')
if len(coment) == 0:  # verifique se entrou um comentário
```

```
    alvo = hoje + os.sep + agora + '.zip'
else:
    alvo = hoje + os.sep + agora + '_' +
        coment.replace(' ', '_') + '.zip'

# Crie o diretório se ainda não estiver lá
if not os.path.exists(hoje):
    os.mkdir(hoje) # crie diretório
    print('Bem-Sucedida Criação do Diretório', hoje)

# 5. Usamos o comando zip para colocar os arquivos em um arquivo zip
comando_zip = "zip -qr {0} {1}".format(alvo, ' '.join(origem))

# Execute o backup
if os.system(comando_zip) == 0:
    print('Backup Bem-Sucedido em', alvo)
else:
    print('Backup FALHOU')
```

Saída:

```
$ python backup_ver3.py

  File "backup_ver3.py", line 25

    alvo = hoje + os.sep + agora + '_' +

                                         ^


SyntaxError: invalid syntax
```

Como (não) funciona:

Este programa não funciona! Python está dizendo que existe um erro de sintaxe o que significa que o script não satisfaz a estrutura que Python espera encontrar. Quando nós observamos o erro dado por Python, ele também nos aponta o local onde detetou o erro. Então nós começamos a *debugar* o nosso programa a partir daquela linha.

Mediante uma observação mais cuidadosa, nós verificamos que a única linha lógica foi dividida em duas linhas físicas, mas nós não especificamos que essas duas linhas físicas estão conectadas uma a outra. Basicamente, Python encontrou o operador de adição (+) sem qualquer operando naquela linha lógica e daí não sabe como prosseguir. Lembre-se que nós podemos especificar que a linha lógica continua na próxima linha física através do uso de um 'backslash' no final da linha física. Assim, nós efetuamos essa correção no nosso programa. A correção do programa quando nós encontramos erros é denominada **conserto de bugs**.

Quarta Versão

```
#!/usr/bin/python
# Nome do arquivo: backup_ver4.py

import os
import time

# 1. Os arquivos e os diretórios cujo backup será efetuado estarão especificados em uma lista.
origem = ['"C:\\Meus Documentos"', 'C:\\Code']
# Note que nós tivemos que usar aspas duplas dentro da string devido aos nomes com espaços dentro dela.

# 2. O backup deve ser colocado em um diretório de backup principal.
```

```
dir_alvo = 'E:\\Backup'  # Lembre-se de alterar para aquele que você estiver utilizando.

# 3. O backup deve ser colocado em uma arquivo do tipo zip.
# 4. O dia corrente é o nome do subdiretório no diretório principal.
hoje = dir_alvo + os.sep + time.strftime('%Y%m%d')
# A hora corrente é o nome do arquivo zip
agora = time.strftime('%H%M%S')

# Receba um comentário do usuário para criar o nome do arquivo zip
coment = input('Entre com um comentário --> ')
if len(coment) == 0:  # verifique se entrou um comentário
    alvo = hoje + os.sep + agora + '.zip'
else:
    alvo = hoje + os.sep + agora + '_' +\
        coment.replace(' ', '_') + '.zip'

# Crie o diretório se ainda não estiver lá
if not os.path.exists(hoje):
    os.mkdir(hoje)  # crie diretório
    print('Bem-Sucedida Criação do Diretório', hoje)

# 5. Usamos o comando zip para colocar os arquivos em um arquivo zip
comando_zip = "zip -qr {0} {1}".format(alvo, ' '.join(origem))

# Execute o backup
if os.system(comando_zip) == 0:
    print('Backup Bem-Sucedido em', alvo)
else:
    print('Backup FALHOU')
```

Saída:

```
$ python backup_ver4.py

Entre com um comentário --> novos exemplos adicionados

Backup Bem-Sucedido em E:\Backup\20080702\202836_novos exemplos_adicionados.zip



$ python backup_ver4.py

Entre com um comentário -->

Backup Bem-Sucedido em E:\Backup\20080702\202839.zip
```

Como funciona:

Este programa agora funciona! Vamos discutir os melhoramentos reais que fizemos na versão 3. Nós coletamos o comentário do usuário usando a função`input` e então verificamos se o usuário efetivamente entrou com alguma coisa, por meio da determinação do comprimento da entrada usando a função `len`. Se o usuário apenas pressionou `enter` sem realmente haver entrado com algo (talvez tenha sido apenas um backup de rotina ou não haviam mudanças significativas), então nós procedemos como havíamos feito antes.

Entretanto, se um comentário foi fornecido, então ele é associado ao nome do arquivo imediatamente antes da extensão `.zip`. Observe que nós estamos substituindo espaços nos comentários por 'underscores' (sublinhados) - isto porque gerenciar arquivos sem espaços é muito mais fácil.

Mais Refinamentos

A quarta versão é um script que funciona satisfatoriamente para a maior parte dos usuários, mas existe espaço para melhorias. Por exemplo, você poderia incluir um nível de *verbosidade* para o programa, no qual você poderia especificar uma opção `-v` para tornar o seu programa mais informativo.

Uma outra possível melhoria seria permitir arquivos e diretórios adicionais a serem passados ao script na linha de comando. Nós podemos obter esses nomes da lista `sys.argv` e adicioná-los a nossa lista `sorigem` utilizando o método`extend` da classe `list`.

O refinamento mais importante seria a não utilização da via através de `os.system`para a criação dos arquivos e ao invés disso lançar mão dos módulos 'built-in'`zipfile` ou `tarfile` para criá-los. Estes são parte da biblioteca padrão e prontamente disponíveis para seu uso sem as dependências externas do programa zip que deve estar disponível no seu computador.

Entretanto, eu usei o caminho de `os.system` para criar um backup nos exemplos acima por razões puramente pedagógicas, de modo que o exemplo fosse suficientemente simples para ser entendido por qualquer pessoa mas real o suficiente para ser útil.

Você poderia tentar escrever a quinta versão que utiliza o módulo zipfile ao invés do recurso ao `os.system`?

O Processo de Desenvolvimento de Software

Nós passamos pelas várias **fases** no processo de escrever um software. Essas fases podem ser resumidas como segue:

1. Descubra o Quê (Análise)
2. Saiba Como (Projeto)
3. Faça (Implementação)
4. Teste (Testando and 'Debugando')
5. Use (Operação or Emprego)
6. Mantenha (Refinamento)

Uma forma recomendada de escrever programas é o procedimento que nós seguimos ao criar o script de backup: Realize a análise e o projeto. Comece por implementar uma versão mais simples. Teste-a e conserte os seus 'bugs'. Use a versão para certificar-se que opera como esperada. Agora, acrescente algumas funcionalidades que você queira e continue a repetir o ciclo Faça-Teste-Use tantas vezes quanto necessário. Lembre-se, **Software é desenvolvido, não vem pronto**.

Resumo

Nós vimos como criar os nossos próprios programas/scripts em Python e os vários estágios envolvidos ao escrever esses programas. Você poderá achar útil criar o seu próprio programa da mesma maneira que nós fizemos nesse capítulo, de modo que você se fique à vontade não apemnas com Python, mas também com a resolução de problemas.

A seguir, discutiremos a programação orientada a objetos.

Python pt-br:Programacao Orientada a Objetos

Introdução

Em todos os programas que escrevemos até agora, projetamos nosso programa em termos de funções, isto é, blocos de declarações que manipulam dados. Esta é a chamada maneira de programar *orientada a procedimentos*. Existe outra maneira de organizar seu programa que combina dados e funcionalidades e as empacota dentro de algo chamado objeto. Este é o chamado paradigma de programação *orientada a objetos*. Na maioria das vezes você pode usar a programação orientada a procedimentos, mas quando estamos escrevendo grandes programas ou temos um problema que se adequa melhor a esse método, você pode usar técnicas de programação orientadas a objetos.
Classes e objetos são os dois principais aspectos da programação orientada a objetos. Uma **classe** cria um novo *tipo* onde **objetos** são *instâncias* da classe. Uma analogia é que você pode ter variáveis do tipo `int` o que significa dizer que variáveis que armazenam inteiros são variáveis que são instâncias (objetos) da classe `int`.

Nota para Programadores de Linguagens Estáticas

> Note que mesmo os inteiros são tratados como objetos (da classe `int`). Isso é diferente do C++ e do Java (antes da versão 1.5) onde os inteiros são tipos primitivos nativos.
> Veja `help(int)` para mais detalhes sobre a classe.
> Programadores C# e Java 1.5 encontrarão semelhanças com os conceitos de*boxing e unboxing*.

Objetos podem armazenar dados utilizando variáveis comuns que *pertencem* ao objeto. Variáveis que pertencem a um objeto o classe são chamadas de **campos**. Objetos também podem ter funcionalidade por meio de funções que *pertencem* a uma classe. Tais funções são chamadas de **métodos** da classe. Esta terminologia é importante porque nos ajuda a diferencias entre funções e variáveis que são independentes e as que pertencem a uma classe ou objeto. Coletivamente, os campos ou métodos podem ser considerados *atributos* da classe.
Os campos podem ser de dois tipos - eles podem pertencer a cada instância/objeto ou podem pertencer à classe em si. Elas são chamadas de**variáveis de instância** e **variáveis de classe**, respectivamente.
Uma classe é criada usando a palavra-chave `class`. Os campos e métodos da classe são listados em um bloco indentado.

O self

Os métodos tem apenas uma diferença específica em relação a funções comuns: eles devem ter um primeiro nome extra que deve ser incluído no início da lista de parâmetros, mas você *não* deve atribuir um valor a esse parâmetro quando chama o método, pois o Python o fornece implicitamente. Essa variável em particular refere-se ao *próprio* objeto e, por convenção, é chamada de `self`.
Embora seja possível dar qualquer nome a este parâmetro, é *fortemente recomendado* que você use o nome `self`. Existem muitas vantagens um usar um nome padrão - qualquer leitor do seu programa irá reconhecê-lo imediatamente e mesmo IDEs especializadas podem ajudá-lo se você usar `self`.

Nota para Programadores C++/Java/C#

> O `self` em Python é equivalente ao ponteiro `this` em C++ e à referência `this`em Java e C#.

Você deve estar intrigado sobre como Python atribui o valor para `self` e o porquê você não precisa atribuir um valor para ele. Um exemplo tornará mais claro. Digamos que você tenha uma classe chamada `MyClass` e uma instância desta classe chamada `MyObject`. Quando você invoca um método deste objeto como `MyObject.method(arg1, arg2)`, isto é convertido automaticamente pelo Python em `MyClass.method(MyObject, arg1, arg2)` - este é o sentido especial de`self`.
Isto também significa que, se você tem um método que não possui argumentos, então você ainda assim deverá ter um argumento - o `self`.

Classes

A classe mais simples possível é mostrada no seguinte exemplo.

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

```
# Arquivo: simplestclass.py

class Person:
    pass  # Um bloco vazio

p = Person()
print(p)
```

Saída:

```
$ python simplestclass.py

<__main__.Person object at 0x019F85F0>
```

Como Funciona:
Criamos uma nova classe usando a instrução `class` e o nome da classe. Na sequência, temos um bloco de instruções que formam o corpo da classe. Neste caso, temos um bloco vazio que é indicado pela instrução `pass`.
Em seguida, criamos um objeto/instância desta classe usando o nome da classe seguido por um par de parênteses. (Aprenderemos mais sobre instanciação na próxima seção). Para verificação, confirmamos o tipo da variável simplementes imprimindo-a (`print`). Isso nos diz que temos uma instância da classe `Person` no módulo `__main__`.
Observe que o endereço de memória onde seu objeto está armazenado também é impresso. O endereço terá um valor diferente em seu computador, uma vez que o Python poderá armezenar o objeto onde quer que haja espaço.

Métodos de Objetos

Já discutimos que classes/objetos podem ter métodos exatamente como funções, exceto pelo fato de que temos uma variável extra - `self`. Vejamos um exemplo.

```
#!/usr/bin/python
# Arquivo: method.py

class Person:
    def sayHi(self):
        print('Hello, how are you?')

p = Person()
p.sayHi()

# Este pequeno exemplo também pode ser escrito como Person().sayHi()
```

Saída:

```
$ python method.py

Hello, how are you?
```

Como Funciona:
Aqui vemos o `self` em ação. Observe que o método `sayHi` não possui qualquer parâmetro mas, ainda assim, temos o `self` na definição do método.

O método __init__

Existem vários nomes de métodos que possuem significado especial em classes Python. Veremos agora o significado do método `__init__`.
O método `__init__` é invocado pronta e automaticamente quando um objeto de uma classe é instanciado. Este método é útil para realizar quaisquer *iniciações*que você desejar fazer com o seu objeto. Observe os sublinhados duplos, no início e no final do nome.
Exemplo:

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

```
# Arquivo: class_init.py

class Person:
    def __init__(self, name):
        self.name = name
    def sayHi(self):
        print('Hello, my name is', self.name)

p = Person('Swaroop')
p.sayHi()

# Este exemplo também pode ser escrito como Person('Swaroop').sayHi()
```

Saída:

```
$ python class_init.py

Hello, my name is Swaroop
```

Como Funciona:

Aqui, definimos o método `__init__` com um parâmetro `name` (além do usual parâmetro `self`). Então, simplesmente criamos um novo atributo também chamado `name`. Observe que tratam-se de variáveis diferentes, embora tenham o mesmo nome ('name'). A notação por ponto nos permite diferenciar uma variável da outra.

Mais importante ainda, observe que nós não invocamos o método `__init__`explicitamente, mas apenas passamos os argumentos entre os parênteses após o nome da classe, quando criamos uma nova instância. Este é o significado especial deste método.

Agora, podemos usar o atributo `self.name` em nossos métodos, como é demonstrado no método `sayHi`.

Nota para Programadores C++/Java/C#

O método `__init__` é análogo a um *construtor* em C++, Java ou C#.

Variáveis de Classe e Objeto

Já discutimos a respeito do lado funcional de classes e objetos (i.e. métodos), agora vamos aprender sobre dados. Dados (i.e. campos ou atributos) são nada mais que variáveis comuns que estão *associadas* ao *espaço de nomes*(*namespace*) de uma classe e/ou objeto. Isto significa que esses nomes são válidos apenas no contexto de uma classe e/ou objeto particular. Por isso são chamados de *espaços de nomes* (*name spaces*).

Existem dois tipos de *atributos* (ou *campos*) - variáveis de classe e variáveis de objeto, que são classificados dependendo de quem - a classe ou o objeto -*possui* essas variáveis.

Variáveis de classe são compartilhadas - elas podem ser acessadas por todas as instâncias daquela classe. Há apenas uma cópia de uma variável de classe e quando qualquer instância daquela classe modifica o seu valor, essa mudança será vista por todas as outras instâncias.

Variáveis de objeto são particulares a cada instância individual de uma classe. Neste caso, cada objeto possui a sua própria cópia, ou seja, elas não são compartilhadas e não estão relacionadas de qualquer maneira à outras variáveis com o mesmo em outras instâncias. Um exemplo tornará mais fácil de entender:

```
#!/usr/bin/python
# Arquivo: objvar.py

class Robot:
    '''Representa um robô com um nome.'''

    # Uma variável de classe, para contar o número de robôs.
    population = 0

    def __init__(self, name):
```

```
        '''Inicia os dados.'''
        self.name = name
        print('(Iniciando {0})'.format(self.name))

        # Quando este robô é criado ele é contabilizado
        # na população geral de robôs
        Robot.population += 1

    def __del__(self):
        '''Estou morrendo.'''
        print('{0} sendo destruído!'.format(self.name))

        Robot.population -= 1

        if Robot.population == 0:
            print('{0} era o último.'.format(self.name))
        else:
            print('Existe(m) ainda {0:d} robô(s) trabalhando.'.format(Robot.population))

    def sayHi(self):
        '''Saudações do robô.

        Sim, eles podem fazer isso.'''
        print('Saudações, meus mestres me chamam {0}.'.format(self.name))

    def howMany(klass):
        '''Imprime a população atual.'''
        print('Temos um total de {0:d} robô(s).'.format(Robot.population))
    howMany = classmethod(howMany)

droid1 = Robot('R2-D2')
droid1.sayHi()
Robot.howMany()

droid2 = Robot('C-3PO')
droid2.sayHi()
Robot.howMany()

print("\nRobôs podem realizar algum trabalho aqui.\n")

print("Robôs terminaram seus trabalhos. Então vamos destruí-los.")
del droid1
del droid2

Robot.howMany()
```

Saída:

```
    (Iniciando R2-D2)

    Saudações, meus mestres me chamam R2-D2.

    Temos um total de 1 robô(s).

    (Iniciando C-3PO)
```

```
Saudações, meus mestres me chamam C-3PO.

Temos um total de 2 robô(s).



Robôs podem realizar algum trabalho aqui.



Robôs terminaram seus trabalhos. Então vamos destruí-los.

R2-D2 sendo destruído!

Existe(m) ainda 1 robô(s) trabalhando.

C-3PO sendo destruído!

C-3PO era o último.

Temos um total de 0 robô(s).
```

Como Funciona:

Este é um longo exemplo, mas ajuda a demonstrar a natureza das variáveis de classe e de instância. Aqui, `population` pertence à classe `Robot` e, portanto, é uma variável de classe. A variável `name` pertence ao objeto (instância, pois é atribuída usando `self`) e, portanto, é uma variável de instância.

Assim, nos referímos à variável de classe `population` como `Robot.population` e não como `self.population`. Por outro lado, nos referímos à variável de instância`name` usando a notação `self.name`, nos métodos daquele objeto. Lembre-se desta simples diferença entre variáveis de classe e de instância. Note também que, uma variável de instância com o mesmo nome de uma variável de classe irá se sobrepor à variável de classe, escondendo-a!

O método `howMany` é, na verdade, um método de classe, e não um método de instância. Portanto, o primeiro argumento é `klass`. Note que escrevemos `klass`com 'k', por que não podemos reutilizar a palavra `class` que é a palavra-chave usada para criar nova classes.

Ainda não terminamos de definir 'howMany', pois temos que tornar explícito que trata-se de um método de classe. Para isso, *empacotamos* o método usando a função interna `classmethod`.

Podemos obter o mesmo efeito usando decorators:

```
@classmethod
def howMany(klass):
    '''Imprime a população atual.'''
    print('Temos um total de {0:d} robô(s).'.format(Robot.population))
```

Pense em *decorators* como sendo atalhos para uma chamada explícita a uma instrução, como vimos neste exemplo.

Note que o método `__init__` é usado para iniciar a instância de `Robot` com um nome. Na iniciação, incrementamos em 1 a população (`population`) de robôs, uma vez que temos mais um robô adicionado. Observe também que os valores de `self.name` são específicos a cada objeto demonstrado a natureza das variáveis de instância.

Lembre-se que você deve referir-se às variáveis e métodos de uma mesma instância **somente** em conjunto com `self`. Isto é chamado de *referência ao atributo*.

Neste exemplo, também vemos o uso de **docstrings** para classes e também para métodos. Podemos acessar a documentação da classe em tempo de execução usando `Robot.__doc__` e a documentação de um método usando`Robot.sayHi.__doc__`.

Assim como o método `__init__`, há um outro método especial `__del__` que é invocado com quando um objeto está para ser eliminado, isto é, quando não será mais utilizado e será eliminado para que o

sistema reutilize o trecho de memória que aquela instância estiver ocupando. Neste método, simplesmente decrementamos a população geral de robôs em 1 usando `Person.population -= 1`.
O método `__del__` será executado quando o objeto não estiver mais em uso e não há garantias de *quando* o método será invocado. Se você deseja vê-lo em ação, devemos utilizar a instrução `del` explicitamente, que é exatamente o que fizemos.

Nota para Programadores C++/Java/C#

> Todos os membros de classe (incluindo dados - variáveis) são *públicos* e todos os métodos são *virtual* em Python.
>
> Há uma exceção: Se você definir nomes para os membros prefixados com*dois caracteres de sublinha* como em `__privatevar`, Python altera esse nome internamente (técnica chamada *name-mangling*) para tornar a variável, efetivamente, privada.
> Assim, segue-se a convenção de que, qualquer variável que deva ser utilizada apenas pela classe ou pela instância deve iniciar com um sublinhado e, todos os outros nomes são públicos e podem ser utilizados por outras classes/objetos. Lembre-se de que esta é apenas uma convenção e não é exigido pelo Python (exceto para prefixos que iniciem com dois caracteres de sublinhado).
>
> Note também que o método `__del__` é análogo ao conceito de métodos*destructor*.

Herança

Um dos maiores benefícios da programação orientada a objetos é **reusabilidade**de código, e uma maneiras de obter isto é através do mecanismo de *herança*. Herança pode ser imaginada como a implementação de um relacionamento *tipo e subtipo* entre classes.
Suponha que você deva escrever um programa para controle de professores e alunos em uma escola. Eles possuem características comuns tais como nome, idade e endereço. Eles também possuem características específicas tais como salário, matérias e leaves para professores e notas e mensalidades para alunos.
Você pode criar duas classes independentes, uma para tipo, mas adicionar uma nova caraterística comum significará adicionar nas duas classes independentes. Isto torna-se rapidamente desajeitado.
Um modo mais eficiente é criar uma classe comum chamada `SchoolMember` e então fazer outras duas classes professor (`Teacher`) e aluno (`Student`) *herdarem*suas caracteríistias, isto é, elas serão seus sub-tipos, e então poderemos adicionar características específicas para cada sub-tipo.
Existem muitas vantagens nesta abordagem. Se adicionarmos ou mudarmos qualquer funcionalidade em `SchoolMember`, isto será automaticamente refletido em seus sub-tipos. Por exemplo, você poderá adicionar um novo campo ID para professores e alunos simplesmente adicionando esse campo na classe`SchoolMember`. Entretanto, mudanças em sub-tipos não afetarão outros sub-tipos. Outra vantagem é que você pode se referir aos objetos professores e alunos como objetos `SchoolMember`, o que pode ser útil em algumas situações, tal como contar o número total de membros. Isto é chamado de **polimorfismo**, onde um sub-tipo pode ser substituído em qualquer situação onde um super-tipo é esperado, isto é, o objeto pode ser manipulado como se fosse uma instância da classe pai.
Observe também que nós *reutilizamos* o código da classe pai (super-tipo) e por isso não precisamos repetí-lo nos sub-tipos, como teríamos que fazer no caso de usarmos classes independentes.
A classe `SchoolMember` nesta situação é conhecida como *classe base* (*base class*) ou *super classe*.. As classes `Teacher` e `Student` são chamadas de *classes derivadas*(*derived classes*) ou *sub-classes*.
Vamos implementar o exemplo citado:

```
#!/usr/bin/python
# Arquivo: inherit.py

class SchoolMember:
    '''Representa qualquer membro da escola.'''
    def __init__(self, name, age):
        self.name = name
        self.age = age
        print('(Iniciado SchoolMember: {0})'.format(self.name))
```

```
    def tell(self):
        '''Imprime os detalhes desta instância.'''
        print('Nome:"{0}" Idade:"{1}"'.format(self.name, self.age), end=" ")

class Teacher(SchoolMember):
    '''Representa um professor.'''
    def __init__(self, name, age, salary):
        SchoolMember.__init__(self, name, age)
        self.salary = salary
        print('(Iniciado Teacher: {0})'.format(self.name))

    def tell(self):
        SchoolMember.tell(self)
        print('Salário: "{0:d}"'.format(self.salary))

class Student(SchoolMember):
    '''Representa um aluno.'''
    def __init__(self, name, age, marks):
        SchoolMember.__init__(self, name, age)
        self.marks = marks
        print('(Iniciado Student: {0})'.format(self.name))

    def tell(self):
        SchoolMember.tell(self)
        print('Nota: "{0:d}"'.format(self.marks))

t = Teacher('Mrs. Shrividya', 40, 30000)
s = Student('Swaroop', 25, 75)

print() # imprime uma linha em branco

members = [t, s]
for member in members:
    member.tell() # funciona tanto para Teachers como para Students
```

Saída:

```
$ python inherit.py

(Iniciado SchoolMember: Mrs. Shrividya)

(Iniciado Teacher: Mrs. Shrividya)

(Iniciado SchoolMember: Swaroop)

(Iniciado Student: Swaroop)



Nome:"Mrs. Shrividya" Idade:"40" Salário: "30000"

Nome:"Swaroop" Idade:"25" Nota: "75"
```

Como Funciona:

Para usar herança, especificamos os nomes das classes base em uma tupla, logo após o nome da classe em sua declaração. Em seguida, observamos que o método `__init__` da classe base é explicitamente invocado usando a variável`self`, de tal maneira que podemos iniciar a porção da classe base do objeto. É muito importante lembrar-se disto - Python não invocará automaticamente o construtor da classe base, por isso você deverá invocá-lo explicitamente.
Observamos também que podemos invocar métodos da classe base prefixando o nome da classe na chamada do método e então passar como argumento a variável `self` além de quaisquer outros argumentos.
Note que podemos tratar instâncias de `Teacher` ou `Student` simplesmente como instâncias de `SchoolMember`, tal como quando usamos o método `tell` da classe`SchoolMember`.
Ainda, note que é invocado o método `tell` do sub-tipo e não o método `tell` da classe `SchoolMember`.
Uma maneira de enteder isto é que Python *sempre* procura pelo método invocado, primeiro no tipo real do objeto, o que é verificado neste caso. Se o método invocado não puder ser encontrado, Python irá procurar pelo método nas classes base (especificadas na tupla onde a classe é declarada) uma-a-uma na ordem em que foram escritas.
Uma nota sobre terminologia - se mais de uma classe é fornecida na tupla de herança (na declaração da classe), isto é chamado de *herança múltipla*.

Metaclasses

Há muito mais no vasto tópico da programação orientada a objetos, mas tocaremos levemente em apenas alguns poucos tópicos apenas para ficarmos sabendo que existem.
Assim como usamos classes para criar objetos, podemos utilizar metaclasses para criar classes.
Metaclasses são usadas para modificar ou criar novos comportamentos em classes.
Vamos tomar um exemplo. Suponha que desejemos ter certeza que sempre criaremos instâncias de subclasses da classe `SchoolMember` e que não criaremos instâncias da classe `SchoolMember` em si.
Podemos garantir isso usando um conceito chamado de *classe base abstrata*(*abstract base class*). Isso significa que a classe será *abstrata* o que quer dizer que a classe em si é um conceito, e não deve ser utilizada como uma classe real (ou seja, não devem ser criados objetos - instâncias - dessa classe).
Podemos declarar que a nossa classe é uma classe base abstrata usando a metaclasse interna chamada `ABCMeta` (ABC - *abstract base class*).

```
#!/usr/bin/env python
# Arquivo: inherit_abc.py

from abc import *

class SchoolMember(metaclass=ABCMeta):
    '''Representa um membro qualquer da escola.'''
    def __init__(self, name, age):
        self.name = name
        self.age = age
        print('(Iniciado SchoolMember: {0})'.format(self.name))

    @abstractmethod
    def tell(self):
        '''Imprime os dados da instância.'''
        print('Nome:"{0}" Idade:"{1}"'.format(self.name, self.age), end=" ")

class Teacher(SchoolMember):
    '''Representa um professor.'''
    def __init__(self, name, age, salary):
        SchoolMember.__init__(self, name, age)
        self.salary = salary
        print('(Iniciado Teacher: {0})'.format(self.name))
```

```
    def tell(self):
        SchoolMember.tell(self)
        print('Salário: "{0:d}"'.format(self.salary))

class Student(SchoolMember):
    '''Representa um aluno.'''
    def __init__(self, name, age, marks):
        SchoolMember.__init__(self, name, age)
        self.marks = marks
        print('(Iniciado Student: {0})'.format(self.name))

    def tell(self):
        SchoolMember.tell(self)
        print('Nota: "{0:d}"'.format(self.marks))

t = Teacher('Mrs. Shrividya', 40, 30000)
s = Student('Swaroop', 25, 75)

m = SchoolMember('abc', 10)

m.tell()

print() # imprime uma linha em branco

members = [t, s]
for member in members:
    member.tell() # funciona tanto para Teacher como para Student
```

Saída:

```
(Iniciado SchoolMember: Mrs. Shrividya)

(Iniciado Teacher: Mrs. Shrividya)

(Iniciado SchoolMember: Swaroop)

(Iniciado Student: Swaroop)

Traceback (most recent call last):

  File "C:\Users\swaroop\python\usingabc.py", line 44, in <module>

    m = SchoolMember('abc', 10)

TypeError: Can't instantiate abstract class SchoolMember with abstract methods tell
```

Como funciona:
Podemos declarar o método `tell` da classe `SchoolMember` como sendo um método abstrato, assim, automaticamente, não poderemos criar instâncias da classe`SchoolMember`.
Entretanto, podermos criar instâncias de `Teacher` e `Student` como se fossem instâncias de `SchoolMember`, por que eles são subclasses.

Sumário

Acabamos de explorar os vários aspectos de classes e objetos bem como várias terminologias relacionadas. Vimos também os benefícios e as armadilhas da programação orientada a objetos. Python é

fortemente orientado a objetos e compreender cuidadosamente estes conceitos o ajudarão muito a longo prazo.
A seguir, aprenderemos a lidar com entrada e saída (*input*/*output*) e sobre como acessar arquivos em Python.

Python pt-br:Entrada e Saida

Haverá situações em que seu programa deverá interagir com o usuário. Por exemplo, você poderia desejar obter alguma entrada do usuário e então imprimir algo. Nós podemos fazer isso usando as funções `input()` e `print()`.
Para a saída, nós também podemos usar vários métodos da classe `str` (string). Por exemplo, você pode usar o método `rjust` para tornar uma string justificada a direita em uma largura especificada.
Veja `help(str)`.
Outro tipo comum de entrada/saída é o uso de arquivos. A habilidade de criar, ler e escrever arquivos é essencial para muitos programas e nós exploraremos este aspecto neste capítulo.

Entrada do usuário

```
#!/usr/bin/python
# user_input.py

def reverse(texto):
    return texto[::-1]

def is_palindrome(texto):
    return texto == reverse(texto)

algo = input('Entre com o texto: ')
if (is_palindrome(algo)):
    print("Sim, é um palíndromo")
else:
    print("Não, não é um palíndromo")
```

Saída:

```
$ python user_input.py

Entre com o texto: senhor

Não, não é um palíndromo



$ python user_input.py

Entre com o texto: ana

Sim, é um palíndromo



$ python user_input.py

Entre com o texto: somavamos

Sim, é um palíndromo
```

Como funciona:

Nós usamos um 'slicing' para inverter o texto. Nós já vimos como podemos fazê-lo 'slices' de sequências usando o código `seq[a:b]` iniciando da posição `a` para a posição `b`. Nós também podemos fornecer um terceiro argumento que determina os *passos* pelos quais o 'slicing' é feito. O passo padrão é `1`, pois assim ele retorna uma parte contínua do texto. Dando um passo negativo, i.e. `-1` o texto será retornado de forma invertida.
A função `input()` recebe uma string como argumento e a mostra para o usuário. Então ela espera que o usuário digite algo e pressione a tecla de retorno (Enter). Feito isso, a função `input()` irá retornar o texto. Nós pegamos este texto e o invertemos. Se o texto original e o invertido forem iguais, então o texto é um palíndromo.
Lição de casa:
Verificar se um texto é um palíndromo deveria ignorar pontuação, espaços e caixas altas e baixas. Por exemplo, "Socorram-me, subi no ônibus em Marrocos!" também é um palíndromo, mas nosso atual programa não diz isso. Você pode modificar o programa para reconhecer este palíndromo?

Arquivos

Você pode abrir e usar arquivos para leitura ou gravação criando um objeto da classe `file` e usando seus métodos `read`, `readline` ou `write` para ler ou escrever no arquivo. A possibilidade de ler ou escrever no arquivo depende do modo que você especificou em sua abertura. Quando você acabar de usar o arquivo, você pode chamar o método `close()` para dizer para o Python que nós terminamos de usar o arquivo.
Exemplo:

```
#!/usr/bin/python
# Filename: usando_arquivo.py

poema = '''\
Programar é divertido
Quando o trabalho está pronto
se você quer tornar seu trabalho divertido:
      use Python!
'''

f = open('poema.txt', 'w') # abrir para escrever ('w'riting)
f.write(poema) # escreve o texto no arquivo
f.close() # fecha o arquivo

f = open('poema.txt') # se nenhum modo é especificado, o modo leitura ('r'ead) é aberto por padrão
while True:
      linha = f.readline()
      if len(linha) == 0: # Tamanho 0 indica EOF (fim do arquivo - End Of File)
            break
      print(linha, end='')
f.close() # fecha o arquivo
```

Saída:

```
$ python usando_arquivo.py

Programar é divertido

Quando o trabalho está pronto

se você quer tornar seu trabalho divertido:

          use Python!
```

Como funciona:

Inicialmente, abrimos um arquivo usando a função embutida `open` e especificando o nome do arquivo e o modo no qual queremos abri-lo. Os modos podem ser de leitura (`'r'`), gravação (`'w'`) ou adicionar (`'a'`). Nós também podemos trabalhar com um arquivo de texto (`'t'`) ou um arquivo binário (`'b'`). Na verdade existem muito mais módulos disponíveis, digitar `help(open)` dará a você mais detalhes sobre eles. Por padrão, `open()` consideradas o arquivo como um 't'exto e o abre no modo leitura ('r'ead).
Em nosso exemplo, nós primeiro abrimos o arquivo em modo texto para gravação, então usamos o método `write` do objeto arquivo para escrever o texto e, finalmente, fechamos o arquivo com o método `close`.
Em seguida, nós abrimos o mesmo arquivo novamente para leitura. Nós não especificamos o modo pois o modo padrão é 'arquivo texto para leitura'. Nós lemos cada linha usando o método `readline` em um loop. Este método retorna uma linha completa, incluindo o carácter de nova linha no final. Quando uma string *vazia* é retornada, significa que nós atingimos o fim do arquivo e, então, 'interrompemos' (break) o loop.
Por padrão, a função `print()` imprime o texto com uma nova linha na tela. Nós evitamos a nova linha especificando `end=''` pois a linha que é lida do arquivo já termina com um carácter de nova linha. Finalmente, nós fechamos o arquivo.
Agora, verifique o conteúdo do arquivo `poema.txt` para se certificar de que o programa escreveu e leu o arquivo.

Pickle

Python fornece um módulo padrão chamado `pickle`, que permite que você armazene **qualquer** objeto de Python em um arquivo e então o pegue novamente mais tarde. Isto é chamado de armazenar o objeto*permanentemente*.
Exemplo:

```
#!/usr/bin/python
# Filename: pickling.py

import pickle

# o nome do arquivo onde armazenaremos o objeto
arquivo_listadecompras = 'listadecompras.data'
# a lista de coisas a comprar
listadecompras = ['maçã', 'manga', 'cenoura']

# Escrevendo no arquivo
f = open(arquivo_listadecompras, 'wb')
pickle.dump(listadecompras, f) # descarrega o objeto em um arquivo
f.close()

del listadecompras # destrói a variável listadecompras

# Lê o que foi armazenado
f = open(arquivo_listadecompras, 'rb')
lista_armazenada = pickle.load(f) # carrega o objeto do arquivo
print(lista_armazenada)
```

Saída:

```
$ python pickling.py

['maçã', 'manga', 'cenoura']
```

Como funciona:
Para armazenar um objeto em um arquivo, nós devemos inicialmente abrir o arquivo com o módulo `open` em modo de gravação ('w'rite) e 'b'inário, para então chamar a função `dump` do módulo `pickle`. Este processo é chamado *pickling*.

A seguir, nós recuperamos o objeto, usando a função `load` do módulo `pickle` que retorna o objeto. Este processo é chamado *unpickling*.

Sumário

Nós discutimos vários tipos de entrada/saída e também o manuseio de arquivos usando o módulo *pickle*. A seguir, nós exploraremos o conceito de exceções.

Python pt-br:Excecoes

Introdução

Exceções ocorrem quando determinadas situações "excepcionais" ocorrem em seu programa. Por exemplo, que tal se você estiver para ler um arquivo e este não existir? Ou que tal se você acidentalmente deletou-o durante a execução do programa? Tais situações são manipuladas uitizando **exceções**.
De modo análogo, que tal se o seu programa tivesse comandos inválidos? Isso é manipulado por Python, que **levanta** (**raise**) as mãos e avisa a você que existe um **erro**.

Erros

Suponha uma simples chamada da função `print<tt>. Que tal se nós escrevêssemos errado como <tt>print` ou `Print<tt>? Observe a capitalização. Neste caso, Python *levanta* um erro de sintaxe.`

```
>>> Print('Hello World')

Traceback (most recent call last):

  File "<pyshell#0>", line 1, in <module>

    Print('Hello World')

NameError: name 'Print' is not defined

>>> print('Hello World')

Hello World
```

`Observe que um <tt>NameError` é levantado e também que a localização onde o erro foi detetado é impressa. Isso é o que um *error handler* para este erro faz.

Exceções

Nós **tentaremos** (**try**) ler a entrada do usuário. Digite `ctrl-d` e veja o que acontece.

```
>>> s = input('Entre com alguma coisa --> ')

Entre com alguma coisa -->

Traceback (most recent call last):

  File "<pyshell#2>", line 1, in <module>

    s = input('Entre com alguma coisa --> ')

EOFError: EOF when reading a line
```

Python levanta um erro chamado `EOFError` o qual basicamente significa que encontrou um símbolo *end of file* (fim do arquivo) (representado por `ctrl-d`) quando não esperaria vê-lo.

Manipulando Exceções

Nós podemos manipular exceções usando o comando `try..except`. Nós basicamente colocamos os comandos costumeiros dentro do bloco do try e dipomos os nossos manipuladores de erros no bloco do except.

```
#!/usr/bin/python
# Nome do arquivo: try_except.py

try:
    texto = input('Entre com alguma coisa --> ')
except EOFError:
    print('Por que você jogou um EOF em mim?')
except KeyboardInterrupt:
    print('Você cancelou a operação.')
else:
    print('Você entrou com {0}'.format(texto))
```

Saída:

```
$ python try_except.py

Entre com alguma coisa -->     # Press ctrl-d

Por que você jogou um EOF em mim?



$ python try_except.py

Entre com alguma coisa -->     # Press ctrl-c

Você cancelou a operação.



$ python try_except.py

Enter something --> nenhuma exceção

Você entrou com nenhuma exceção
```

Como Funciona:
Nós colocamos todos os comandos que poderiam levantar exceções dentro do bloco do `try` e então os 'handlers' para os erros/exceções na cláusula/bloco`except`. a cláusula `except` pode manipular um único erro ou exceção especificada, ou uma lista de erros/exceções entre parênteses. Se nenhum nome de erro ou exceção for fornecido, manipulará *todos* os erros ou exceções.
Note que deve haver pelo menos uma cláusula `except` associada com cada cláusula `try`. Ou não haveria razão para um bloco de try.
Se um erro ou exceção não é manipulada, então é invocado o Python handler padrão que interrompe a execução do programa e emite uma mensagem de erro. Nós vimos isso em ação acima.
Nós podemos também ter uma cláusula `else` associada a um bloco `try..catch`. A cláusula `else` é executada se nenhuma exceção ocorrer.
No próximo exemplo, nós veremos como obter um objeto exceção de modo que nós possamos recuperar informação adicional.

Levantando Exceções

Você pode **levantar** exceções com o comando `raise` fornecendo o nome do erro/exceção e o objeto exceção que deverá ser *lançado* (*thrown*).
O erro ou exceção que você pode levantar deverá ser uma classe que direta ou indiretamente deve ser uma classe derivada da classe `Exception`.

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

```
# Nome do arquivo: raising.py

class ShortInputException(Exception):
    '''Uma classe exceção definida pelo usuário.'''
    def __init__(self, length, atleast):
        Exception.__init__(self)
        self.length = length
        self.atleast = atleast

try:
    text = input('Entre com alguma coisa --> ')
    if len(text) < 3:
        raise ShortInputException(len(text), 3)
    # Demais tarefas podem continuar como usualmente aqui...
except EOFError:
    print('Por que você jogou um EOF em mim?')
except ShortInputException as ex:
    print('ShortInputException: A entrada teve o comprimento {0}, o esperado era pelo menos {1}'\
          .format(ex.length, ex.atleast))
else:
    print('Nenhuma exceção foi levantada.')
```

Saída:

```
$ python raising.py

Entre com alguma coisa -->  a

ShortInputException: A entrada teve o comprimento 1, o esperado era pelo menos 3



$ python raising.py

Enter something --> abc

Nenhuma exceção foi levantada.
```

Como Funciona:
Aqui nós estamos criando o nosso próprio tipo de Exceção. Este novo tipo de exceção é chamado `ShortInputException`. Ele tem dois campos - `length` que é o comprimento da entrada fornecida e `atleast` que é o mínimo comprimento da entrada que o programa estava esperando.
Na cláusula `except`, nós mencionamos a classe de erro que será guardada `as`(como) o nome da variable que irá manter o objeto erro/exceção correspondente. Isto é análogo aos parâmetros e argumentos em uma chamada de função. Dentro dessa particular cláusula `except` clause, nós usamos os campos `length` e `atleast` do objeto exceção para imprimir uma mensagem apropriada ao usuário.

Try .. Finally

Suponha que você estivesse lendo um arquivo em seu programa. Como você poderia assegurar-se que o objeto arquivo estivesse fechado adequadamente, fosse ou não levantada uma exceção? Isso pode ser obtido usando-se o bloco`finally`. Note que você pode empregar uma cláusula `except` juntamente com o bloco `finally` para o mesmo correspondente bloco `try`. Você terá que imergir um dentro do outro se quiser utilizar ambos.

```
#!/usr/bin/python
# Nome do arquivo: finally.py
```

```
import time

try:
    f = open('poema.txt')
    while True: # nossa usual linguagem de leitura de arquivo
        line = f.readline()
        if len(line) == 0:
            break
        print(line, end='')
        time.sleep(2) # Assegurar que levará algum tempo executando
except KeyboardInterrupt:
    print('!! Você cancelou a leitura do arquivo.')
finally:
    f.close()
    print('(Limpando tudo: Fechado o arquivo)')
```

Saída:

```
$ python finally.py

Programming is fun

When the work is done

if you wanna make your work also fun:

!! Você cancelou a leitura do arquivo.

(Limpando tudo: Fechado o arquivo)
```

Como Funciona:
Nós efetuamos a tradicional rotina de leitura do arquivo, mas introduzimos arbitrariamente uma pausa ('sleeping') de 2 segundos depois da impressão de cada linha, usando a função `time.sleep` de modo que o programa execute lentamente (Python é muito rápido por natureza). Enquanto o programa estiver executando, digite `ctrl-c` para interromper/cancelar o programa.
Observe que a exceção `KeyboardInterrupt` é lançada e o programa termina. Entretanto, antes da saída do programa, a cláusula `finally` é executada e o objeto arquivo é sempre fechado.

O Comando With

Adquirir um recurso no bloco `try` e liberá-lo no bloco `finally` é um padrão comum. Daí existe também o comando `with` que faz com que isso seja realizado de um modo bem mais claro:

```
#!/usr/bin/python
# Nome do arquivo: using_with.py

with opened("poema.txt") as f:
    for line in f:
        print(line, end='')
```

Como Funciona:
A saída deverá ser a mesma do exemplo anterior. A diferença aqui está em que nós estamos usando a função `opened` com o comando `with`.
Isto não congestiona o processamento do arquivo e focaliza no que estamos fazendo com ele. Nós deixamos o fechamento do arquivo a ser realizado automaticamente por `opened`.

Resumo

Nós discutimos o uso dos comandos `try..except` e `try..finally`. Vimos como criar o nosso próprio tipo de exceção e também como levantar exceções.
Em seguida, exploraremos a Python Standard Library (Bibioteca Padrão do Python).

Python pt-br:Biblioteca Padrao

Introdução

A Biblioteca Padrão do Python contém um vasto número de módulos úteis e é parte de todas as instalações de Python. É importante conhece-la, uma vez que muitos problemas podem ser resolvidos rapidamente se você estiver familiarizado com o grande número de coisas que estas bibliotecas podem fazer.
Nós iremos explorar alguns dos módulos mais usados desta biblioteca. Você pode encontar detalhes completos para todos os módulos da Biblioteca Padrão do Python na seção 'Library Reference' da documentação que vem com sua instalação do Python.
Vamos explorar alguns módulos úteis.

Nota

> Se você achar estes tópicos muito avançados, você pode pular este capítulo. No entanto, eu realmente recomendo que você volte para estudar este capítulo quando você estiver familiarizado com a programação usando Python.

Módulo sys

O módulo `sys` contém funcionalidades específicas do sistema. Nós já vimos que a lista `sys.argv` contém os argumentos da linha de comando.
Suponha que nós desejemos checar a versão do Python que está sendo usada para se assegurar que estamos usando a versão 3. O módulo `sys` nos dá tal funcionalidade.

```
>>> import sys
>>> sys.version_info
(3, 0, 0, 'beta', 2)
>>> sys.version_info[0] >= 3
True
```

Como funciona:
O módulo `sys` tem a tupla `version_info` que nos dá informações sobre a versão do Python. O primeiro valor é a versão principal. Nós podemos verificar a versão, por exemplo, para garantir que o programa só rode sob o Python 3.0:

```
#!/usr/bin/python
# Filename: checarversao.py
import sys, warnings
if sys.version_info[0] < 3:
    warnings.warn("Necessário Python 3.0 para rodar este programa",
        RuntimeWarning)
else:
    print('Prosseguir normalmente')
```

Saída:

```
$ python2.5 checarversao.py

checarversao.py:6: RuntimeWarning: Necessário Python 3.0 para rodar este programa

  RuntimeWarning)



$ python3 checarversao.py

Prosseguir normalmente
```

Como funciona:
Nós usamos um outro módulo da biblioteca padrão chamado `warnings`, que é usado pra mostrar avisos para o usuário final. Se a versão do Python não for 3.0 ou superior, nós mostramos o aviso correspondente.

Módulo logging

E se você desejar armazenar mensagens de depuração ou outras mensagens importantes, de forma que você possa verificar se seu programa está rodando da forma que você esperava? Como armazenar estas mensagens? Isso pode ser feito usando o módulo `logging`.

```
#!/usr/bin/python
# Filename: use_logging.py
import os, platform, logging

if platform.platform().startswith('Windows'):
    logging_file = os.path.join(os.getenv('DIRETÓRIOINICIAL'), os.getenv('CAMINHO'), 'test.log')
else:
    logging_file = os.path.join(os.getenv('HOME'), 'test.log')

logging.basicConfig(
    level=logging.DEBUG,
    format='%(asctime)s : %(levelname)s : %(message)s',
    filename = logging_file,
    filemode = 'w',
)

logging.debug("Início do programa")
logging.info("Fazendo alguma coisa")
logging.warning("Terminando agora")
```

Saída:

```
$python use_logging.py

Logging to C:\Users\swaroop\test.log
```

Se nós verificarmos o conteúdo de `test.log`, veremos algomo como isto:

```
2008-09-03 13:18:16,233 : DEBUG : Início do programa

2008-09-03 13:18:16,233 : INFO : Fazendo alguma coisa

2008-09-03 13:18:16,233 : WARNING : Terminando agora
```

Como funciona:
Nós usamos três módulos da biblioteca padrão - o módulo `os` para interagir com o sistema operacional, o módulo `platform` para informações sobre a plataforma, i.e. o sistema operacional, e o módulo `logging` para *registrar* informações.
Inicialmente, nós verificamos qual o sistema operacional que estamos usando, verificando a string retornada por `platform.platform()` (para mais informações, veja `import platform; help(platform)` no interpretador Python). Se for o Windows, nós apontamos o diretório inicial, o caminho e o nome do arquivo no qual queremos armazenar a informação. Juntando estas três partes, nós teremos a localização completa do arquivo. Para outras plataformas, nós precisamos conhecer apenas a pasta home do usuário e assim teremos a localização completa do arquivo.
Nós usamos a função `os.path.join()` para juntar estas três partes da localização. O motivo de usar uma função especial ao inves de apenas somar as strings, é que esta função se assegura de que a localização completa esteja de acordo com o formato esperado pelo sistema operacional.

Nós configuramos o módulo `logging` para escrever todas as mensagens no arquivo especificado, num formato específico.
Finalmente, nós podemos armazenar mensagens úteis para depuração, informação, avisos ou mensagens críticas. Uma vez que o programa tenha rodado, nós podemos verificar este arquivo e, assim, saberemos o que aconteceu no programa, mesmo que nenhuma informação tenha sido mostrada para o usuário durante seu funcionamento.

Módulos urllib e json

Não seria divertido se nós pudéssemos escrever nossos próprios programas de busca para internet? Vamos descobrir como fazê-los.
Isto pode ser feito usando-se alguns módulos. Primeiro o módulo `urllib` que podemos usar para obter qualquer página da internet. Poderemos usar o Yahoo! Search para pesquisar resultados e, por sorte, eles podem nos dar os resultados num formato chamado JSON, que podemos analisar facilmente graças ao módulo embutido `json` da biblioteca padrão.

Nota

Este programa ainda não funciona devido ao que parece ser um bug no Python 3.0 beta 2.

```
#!/usr/bin/python
# Filename: yahoo_search.py

import sys
if sys.version_info[0] != 3:
    sys.exit('Este programa precisa do Python 3.0')

import json
import urllib, urllib.parse, urllib.request, urllib.response

# Obtenha seu próprio APP ID em http://developer.yahoo.com/wsregapp/
YAHOO_APP_ID = 'jl22psvV34HELWhdfUJbfDQzlJ2B57KFS_qs4I8D0Wz5U5_yCI1Awv8.lBSfPhwr'
SEARCH_BASE = 'http://search.yahooapis.com/WebSearchService/V1/webSearch'

class YahooSearchError(Exception):
    pass

# Obtido de http://developer.yahoo.com/python/python-json.html
def search(query, results=20, start=1, **kwargs):
    kwargs.update({
        'appid': YAHOO_APP_ID,
        'query': query,
        'results': results,
        'start': start,
        'output': 'json'
    })
    url = SEARCH_BASE + '?' + urllib.parse.urlencode(kwargs)
    result = json.load(urllib.request.urlopen(url))
    if 'Error' in result:
        raise YahooSearchError(result['Error'])
    return result['ResultSet']

query = input('Pelo que você quer procurar? ')
for result in search(query)['Result']:
    print("{0} : {1}".format(result['Title'], result['Url']))
```

Como funciona:

Nós podemos obter resultados de um determinado site, fornecendo o texto que se deseja procurar em um formato específico. Nós devemos especificar diversas opções, que nós juntamos usando o formato `key1=value1&key2=value2`, que é usado pela função `urllib.parse.urlencode()`.
Então, por exemplo, abra este link em seu navegador e você vera 20 resultados, iniciando do primeiro, para a busca das palavras "byte of python". Este endereço nos dá os resultados no formato JSON.
Nós realizamos a conexão a este URL usando a função`urllib.request.urlopen()` e passando o arquivo obtido para `json.load()` que irá ler seu conteúdo e simultâneamente converte-li em um objeto Python.
Então nós usamos um loop nestes resultados para mostrá-los ao usuário final.

O Módulo da Semana

Há muito mais para ser explorado na biblioteca padrão, tal como depuração, uso de opções em linha de comando, expressões comuns e muito mais.
A melhor forma de se aprofundar na biblioteca padrão é ler os excelentes artigos 'Python Module of the Week' de Doug Hellmann.

Sumário

Nós vimos algumas das funcionalidades de alguns módulos da Biblioteca Padrão do Python. É recomendável que você navegue pela documentação da Biblioteca Padrão do Python para ter uma ideia de tudo que os módulos são capazes de fazer.
A seguir, nós veremos vários aspectos sobre Python que farão nosso tour mais *completo*.

Python pt-br:Mais

Introdução

Até agora nós cobrimos a maior parte dos diversos aspectos que nós usaremos. Neste capítulo, nós abordaremos algumas funcionalidades adicionais que tornarão o nosso conhecimento de Python mais completo.

Passando Tuplas Adiante

Algumas vezes você gostaria de poder retornar com dois valores diferentes de uma função? Você pode. Tudo que precisa fazer é usar uma tupla.

```
>>> def obtenha_detalhes_erro():
... return (2, 'detalhes do segundo erro')
...
>>> errnum, errstr = obtenha_detalhes_erro()
>>> errnum
2
>>> errstr
'detalhes do segundo erro'
```

Note que o uso de a,b = <alguma expressao> interpreta o resulatado da expressão como uma tupla com dois valores.

Se quiser interpretar o resultado como (a, <todo o resto>), então você precisa acrescentar um asterisco, como se fossem parâmetros de uma função:

```
>>> a, *b = [1, 2, 3, 4]
>>> a
1
>>> b
[2, 3, 4]
```

Isso também significa que o modo mais rápido de trocar duas varíaveis entre si em Python é:

```
>>> a = 5; b = 8
>>> a, b = b, a
>>> a, b
(8, 5)
```

Métodos Especiais

Existem certos métodos tais como __init__ e __del__ que possuem significado especial em classes.

Os métodos especiais são empregados para emular certos comportamentos dos tipos 'built-in'. Por exemplo, se você quiser usar a operação de indexação x[indice] para a sua classe (exatamente como você faria para listas e tuplas), então tudo o que voc deve fazer é implementar o método __getitem__() e está feito. Se você refletir a respeito, isto é o que Python faz no caso da própria classe das listas!

Alguns métodos especiais úteis estão listados na tabela que segue. Se você quiser conhecer mais a respeito de todos os métodos especiais, veja o manual.

Alguns Métodos Especiais

Nome

Explicação

__init__(self, ...)

Este método é chamado imediatamente antes que o novo objeto criado esteja pronto para utilização.

__del__(self)

Chamado imediatamente antes que o objeto seja destruído.

__str__(self)

Chamado quando nós usamos a função print ou quando str() é usado.

__lt__(self, other)

Chamado quando o operador 'menor do que' (<) é usado. Similarmente, existem métodos especiais para todos os operadores (+, >, etc.)

__getitem__(self, key)

Chamado quando a operação de indexação x[key] é usada.

__len__(self)

Chamado quando a função 'built-in' len() é utilizada para um objeto sequência.

Blocos de Um Único Comando

Nós já vimos que cada bloco de comandos é posto separadamente do resto pelo seu próprio nível de indentação. Bem, existem algumas exceções. Se o seu bloco de comandos contém apenas um único comando, então você poderá especificá-lo na mesma linha de, digamos, um comando condicional ou em laço. O seguinte exemplo deverá tornar isso mais claro:

```
>>> flag = True
>>> if flag: print 'Sim'
...
Sim
```

Observe que o comando único é empregado no local e na mesma linha e não como um bloco separado. Embora você possa fazer uso disso para tornar menor o seu programa, eu aconselho a evitar este procedimento em curto-circuito, exceto no caso de verficação de erro, uma vez que ficará muito mais fácil acrescentar um comando a mais se você estiver usando a indentação apropriada.

Formas Lambda

Uma forma lambda é empregada para criar novos objetos funções e retorná-los em tempo de execução ('runtime').

1. !/usr/bin/python
2. Nome do arquivo: lambda.py

```
def cria_repetidor(n):
    return lambda s: s * n



dobro = cria_repetidor(2)
print(dobro('palavra')) print(dobro(5))
```

Saída:

```
$ python lambda.py

palavrapalavra

10
```

Como Funciona:
Aqui nós desenvolvemos uma função cria_repetidor para fabricar novos objetos funções em tempo de execução e retorná-los. Um comando lambda é utilizado para criar o objeto função. Essencialmente lambda recebe um parâmetro seguido de apenas uma única expressão, a qual se torna o corpo da função e o valor dessa expressão é retornado pela nova função. Note que mesmo um comando print **(Python 2.x)** não pode ser usado no interior de uma forma lambda, apenas expressões.

A SER FEITO

```
   Podemos realizar uma list.sort() fornecendo uma função de comparação criada usando
lambda?
```

pontos = [{ 'x' : 2, 'y' : 3 }, { 'x' : 4, 'y' : 1 }]

1. pontos.sort(lambda a, b : cmp(a['x'], b['x']))

List Comprehension

'List comprehensions' são empregadas para obter-se uma nova lista de uma lista existente. Suponha que você tenha uma lista de números e deseje obter uma nova lista com todos os números multiplicados por dois apenas quando o número por sua vez for maior do que 2. 'List comprehensions' são ideais para tais situações.

1. !/usr/bin/python
2. Nome do arquivo: list_comprehension.py

listaum = [2, 3, 4] listadois = [2*i for i in listaum if i > 2] print(listadois)
Saída:
[2, 6, 8]
Como Funciona:
Aqui nós obtemos uma nova lista através da especificação da manipulação a ser efetuada quando alguma condição for satisfeita (if i > 2). Note que a lista original permanece inalterada.
A vantagem de usar 'list comprehensions' é que reduz a quantidade de código padrão ('boilerplate') requerido quando nós usamos laços para processar cada elemento de uma lista e adicioná-lo em uma nova lista.

Recebendo Tuplas e Dicionários em Funções

Existe um modo especial de atribuir parâmetros a uma função como uma tupla ou dicionário, usando os prefixos * e ** respectivamente. Isso é muito útil quando se considera um número variável de argumentos na função.

```
>>> def powersum(power, *args):
... Retorna a soma de cada argumento elevada à potência especificada.
... total = 0
... for i in args:
... total += pow(i, power)
... return total
...
>>> powersum(2, 3, 4)
25
>>> powersum(2, 10)
100
```

Uma vez que temos um prefixo * na variável args, todos os argumentos extra passados à função são atribuídos a args como uma tupla. Se ao invés disso, tivesse sido usado um prefixo **, os parâmetros extra seriam considerados como pares chave/valor de um dicionário.

Os Comandos Exec E Eval

A função exec é empregada para executar comandos em Python que estão armazenados em uma string ou arquivo, ao contrário do que se estivessem no próprio programa. Por exemplo, nós podemos gerar uma

string contendo comandos em Python em tempo de execução e então executá-los utilizando a função exec:

```
>>> exec('print("Hello World"')
Hello World
```

De modo análogo, a função eval é utilizada para avaliar expressões válidas em Python que estão armazenadas em uma string. Um exemplo simples é mostrado a seguir.

```
>>> eval('2*3')
6
```

O Comando Assert

O comando assert é empregado para assegurar-se de que alguma coisa é verdadeira. Por exemplo, se você está bem certo que terá pelo menos um elemento em uma lista que você está usando e quer verificar esse fato e levantar ('raise') um erro se isso não for verdade, então o comando assert é ideal para essa situação. Quando um comando assert falha, um AssertionError é levantado.

```
>>> mylist = ['item']
>>> assert len(mylist) >= 1
>>> mylist.pop()
'item'
>>> mylist
[]
>>> assert len(mylist) >= 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

O comando assert deve ser usado com cautela. Na maior parte dos casos, é melhor capturar exceções, seja manipulando o problema, ou emitindo uma mensagem de erro para o usuário e saindo do programa.

A Função Repr

A função repr é utilizada para obter-se uma representação por meio de uma string canônica do objeto. A parte interessante é que você terá eval(repr(objeto)) == objeto em quase todos os casos.

```
>>> i = []
>>> i.append('item')
>>> repr(i)
"['item']"
>>> eval(repr(i))
['item']
>>> eval(repr(i)) == i
True
```

Basicamente, a função repr é empregada para obter-se uma representação imprimível do objeto. Você pode controlar o que a sua classe retorna sob a função repr, definindo o método __repr__ em sua classe.

Resumo

Nós abordamos algumas funcionalidades adicionais de Python neste capítulo e mesmo assim não cobrimos todas elas. Porém, a essa etapa, nós apresentamos a maior parte do que você irá empregar na prática. Isso é suficiente para que você possa iniciar com quaisquer programas que for produzir.

A seguir, nós discutiremos como explorar Python ainda mais.

Python pt-br:Em Seguida

Se você leu este livro completamente até agora e praticou escrevendo muitos programas, então deve estar se sentindo confortável e familiarizado com Python. Você provavelmente também criou diversos programas em Python para realizar alguns experimentos e exercitar suas habilidades em Python. Se você não fez isso ainda, deveria tê-lo feito. A questão agora é "O Que Fazer Em Seguida?".

Eu sugiro que você ataque o seguinte problema:

Crie o seu próprio programa "address_book" usando linha de comando, com o qual você poderá adicionar, modificar, deletar ou realizar buscas pelos seus contatos, tais como amigos, familiares e colegas, bem como pelas suas informações, tais como endereços de email e/ou números de telefone. Os detalhes devem ser armazenados para posterior recuperação.

Isto se torna razoavelmente fácil se você pensar em termos de todo o material que nós encontramos até agora. Se você ainda assim deseja instruções de como proceder, aqui está uma sugestão.

Sugestão (Não leia)

> Crie uma classe que represente a informação sobre cada pessoa. Use um dicionário para guardar os objetos pessoais com o nome de cada uma como chave. Utilize o módulo pickle para guardar os objetos de forma persistente em seu disco rígido. Empregue os métodos internos do dicionário para adicionar, deletar e modificar as pessoas.

Uma vez que você seja capaz de realizar isso, você pode proclamar-se programador em Python. Agora, imediatamente envie um email me agradecendo por este grande livro ;-). Este passo é opcional mas recomendado. Da mesma forma, por favor, considere realizar uma doação, contribuir com melhoramentos ou voluntariar-se para traduções para apoiar o contínuo desenvolvimento deste livro.

Se você achou muito fácil aquele programa, aqui está um outro:

> Implemente o comando replace. Este comando substituirá uma string por outra em uma lista de arquivos fornecida.

O comando replace pode ser tão simples ou sofisticado quanto você desejar, desde simples substituição de strings, até a busca por padrões (expressões regulares).

Depois disso, aqui estão alguns caminhos para prosseguir a sua jornada com Python:

Exemplos de Programas

A melhor maneira de aprender uma linguagem de programação é escrever muitos programas e ler um monte deles:

- Repositário de programas Rosetta
- Exemplos de Python examples em java2s
- Python Cookbook é uma coleção extremamente valiosa de receitas ou sugestões em como resolver certas espécies de problemas utilizando Python. Esta é uma leitura indispensável para todos os usuários de Python.
- Python Brasil. Este é o principal site brasileiro de Python.

Perguntas e Respostas

- Oficial Python Dos e Don'ts (Faça Isso e Não Faça Aquilo)
- Python FAQ Oficial
- Lista de Norvig das Infrequently Asked Questions (Perguntas Raramente Feitas)
- Entrevista em Python Q & A
- Perguntas StackOverflow marcadas com python

Tips and Tricks (Sugestões e Truques)

- Python Tips & Tricks
- Advanced Software Carpentry using Python
- Charming Python é uma excelente série de artigos de David Mertz relacionados com Python.

Livros, Publicações, Tutoriais, Vídeos

O próximo passo lógico depois deste livro é a leitura do extraordinário livro de Mark Pilgrim Dive Into Python (Mergulhando em Python) o qual você pode também ler inteiramente on line. Esse livro explora tópicos como expressões regulares, processamento XML, serviços web, teste unitário (unit testing), etc. em detalhes.
Outros recursos adicionais são:

- vídeos ShowMeDo sobre Python
- Lista completa de Awaretek dos tutoriais em Python
- The Effbot's Python Zone
- Links no fim de cada Python-URL! email
- Python Papers

Discussão

Se você parece incapaz de resolver um problema de Python, e não sabe a quem perguntar, então a lista comp.lang.python discussion group é o melhor lugar para colocar a sua questão.
Certifique-se de fazer sua 'lição de casa' e de haver tentado resolver o seu problema por você mesmo antes de mais nada.
Make sure you do your homework and have tried solving the problem yourself first.

Novidades

Se você quiser conhecer as últimas novidades no mundo de Python, então acompanhe o Python Planet oficial e/ou o Python Planet não-oficial.

Instalando Bibliotecas

Existe um número enorme de bibliotecas open-source no Python Package Indexas quais você pode empregar em seus próprios programas.
Para instalar aquelas bibliotecas, você pode usar o excelente EasyInstall tool de Philip Eby.

Software Gráfico

Suponha que você queira criar seus próprios programas gráficos usando Python. Isso pode ser feito por meio de um biblioteca GUI (Graphical User Interface) com suas interfaces vínculos (bindings) com Python. Interfaces são o que permite que você escreva programas em Python e utilize as bibliotecas que, por sua vez, são escritas em C, C++ ou outras linguagens.
Existem um grande número de opções para GUI usando Python:

PyQt

Esta é a interface em Python para o Qt toolkit que é a fundação sobre a qual KDE foi produzido. Qt é extremamente fácil de usar e muito poderosa, especialmente devido ao Qt Designer e a incrivel documentação. PyOt é livre se você quiser criar software open-source (GPL) e pago, se você for desenvolver software proprietário fechado. Para iniciar, leia o PyQt tutorialou o PyQt book.

PyGTK

Esta é a interface em Python para o GTK+ toolkit que é a fundação sobre a qual Gnome foi produzido. GTK+ possui muitos pequenos problemas para o seu uso, mas uma vez que você se torna confortável, poderá criar aplicativos com GUI rapidamente. A interface gráfica Glade é indispensável. A documentação ainda deve ser emlhorada. GTK+ funciona bem em Linux, mas a sua versão para Windows está incompleta. Você pode criar softwares tanto livres, como proprietários usando GTK+. Para iniciar, leia o PyGTK tutorial.

wxPython

Este é a interface em Python para o wxWidgets toolkit. wxPython tem uma curva de aprendizagem associada. Entretanto, é bem portável e executa em Linux, Windows, MAC e mesmo em plataformas imersas (embedded). Existem muitas IDEs disponíveis em wxPython as quais incluem GUI designers, tais como SPE (Stani's Python Editor) e a construtora de GUIs wxGlade. Você pode criar software tanto livre como proprietário usando wxPython. Para iniciar, leia o wxPython tutorial.

TkInter

Este é dos mais antigos GUI toolkits em existência. Se você tem utilizado IDLE, já viu o programa TkInter em ação. Não oferece uma das melhores sensações para o usuário, devido a sua aparência antiquada. TkInter é portável e executa tanto em Linux/Unix, quanto em Windows. Mas importante que tuso, TkInter é parte integrante da distribuição padrão de Python. Para iniciar, leia o Tkinter tutorial.

Para mais opções, veja o wiki GuiProgramming no website oficial de Python.

Resumo das Ferramentas GUI

Infelizmente, não existe uma ferramenta GUI padrão em Python. Eu sugiro que você escolha uma das anteriores, dependendo de sua situação. O primeiro fator é se você está disposto a pagar para usar qualquer uma das ferramentas GUI. O segundo fator é se você quer que seu programa execute apenas em Windows ou em MAC e Linux, ou em todos eles. O terceiro fator, se Linux for a plataforma escolhida, se você é um usuário de KDE ou de GNOME.

Para uma análise mais detalhada e completa, veja a página 26 de The Python Papers, Volume 3, Issue 1.

Várias Implementações

EXistem usualmente duas partes em uma linguagem de programação - a linguagem e o software. A linguagem é "como" nós escrevemos algo. O software é "o que" realmente executa o nossos programas. Nós vimos uitlizando o software "CPython" para executar os nossos programas. Este é referenciado como CPython, pois é escrito na linguagem C e é o "Interpretador Python Clássico".

Existem adicionalmente outros softwares que podem executar seus programas em Python:

Jython

Uma implementação de Python que executa na plataforma Java. Isso significa que você pode usar as bibliotecas e classes em Java dentro da linguagem Python e vice-versa.

IronPython

Uma implementação de Python que executa em plataforma .NET. Isso significa que você pode usar as bibliotecas e classes .NET dentro da linguagem Python e vice-versa.

PyPy

Uma implementação de Python escrita em Python! Este é um projeto de pesquisa para tornar mais fácil e rápido o interpretador, uma vez que o próprio interpretador está escrito em uma linguagem dinâmica (em oposição a linguagens estáticas, tais como C, Java, ou C# nas implementações acima).

Stackless Python

Uma implementação que é especializada em performance baseada em threads.

Existem também outras tais como CLPython - uma implementação de Python escrita em Common Lisp e IronMonkey, que é uma implementação de IronPython para operar sobre um interpretador JavaScript, o que poderia significar que você poderia usar Python (ao invés de JavaScript) para escrever os seus programas web-browser ("Ajax").

Cada uma dessas implementações têm a sua própria área especializada na qual são úteis.

Resumo

Nós chegamos ao final deste livro, mas, como se diz comumente, este é "o começo do fim"! Você é, agora, um ávido usuário de Python e está sem dúvida pronto para resolver muitos problemas utilizando Python. Você pode iniciar por automatizar o seu computador para realizar toda espécie de coisas que sequer podiam ser imaginadas anteriormente, ou por escrever seus próprios jogos e muito, muito mais. Assim, pode começar!

Python pt-br:Apendice FLOSS

Software Gratuito e Livre (Free/Libré and Open Source Software - FLOSS)

FLOSS está baseado no conceito de uma comunidade, que por sua vez está fundamentado no conceito de compartilhamento, e particularmente compartilhamento de conhecimento. FLOSS são livres para uso, modificação e redistribuição.
Se você já leu este livro, então já está familiarizado com FLOSS, desde que você vem empregando **Python** ao longo dele e Python é um software de fonte aberta (open source)!
Aqui estão alguns exemplos de FLOSS para dar-lhe uma idéia do tipo de coisas que a construção e compartilhamento comunitário pode criar:

- **Linux**. Este é um sistema operacional FLOSS que o mundo inteiro está lentamente adotando! Começou com Linus Torvalds enquanto um estudante. Agora, está competindo com o Microsoft Windows. [Linux Kernel]
- **Ubuntu**. Esta é uma distribuição gerenciada pela comunidade, patrocinada pela Canonical e é a distribuição de Linux mais popular hoje em dia. Permite que você instale uma quantidade enorme de FLOSS disponíveis e isso em uma maneira fácil de usar e fácil de instalar. Melhor que tudo, você pode apenas reiniciar o seu computador e executar Linux diretamente do CD! Isto permite que você experimente completamente o novo SO antes de instalar no seu computador. [Ubuntu Linux]
- **OpenOffice.org**. Esta é uma excelente suite de escritório com componentes para processamento de documento, apresentação, planilhas, desenho, entre outras coisas. Você pode até mesmo abrir e editar arquivos de MS Word e MS PowerPoint com facilidade. Ela executa em quase todas as plataformas. [OpenOffice]
- **Mozilla Firefox**. Esta é a *próxima* geração de navegadores para a web que está competindo com o Internet Explorer. É bastante rápida e tem recebido aclamação crítica pelos suas ótimas e atraentes características. O conceito de extensões permite o emprego de qualquer espécie de plugin.
- O seu produto companheiro Thunderbird é um excelente cliente de email que torna muito fácil a leitura de emails. [Mozilla Firefox, Mozilla Thunderbird]
- **Mono**. Esta é uma implementação de fonte aberta da plataforma .NET da Microsoft. Permite que as aplicações .NET sejam criadas e executadas em Linux, Windows, FreeBSD, Mac OS e muitas outras plataformas. [Mono, ECMA,Microsoft .NET]
- **Servidor de web Apache**. Este é um popular servidor de web. De fato, é *o*mais popular servidor de web do planeta! Ele executa em um pouco mais do que a metade de todos os sites web que existem. Sim, isso mesmo - Apache gerencia mais sites web do que toda a concorrência (incluindo Microsoft IIS) combinada. [Apache]
- **MySQL**. Este é um servidor de bancos de dados extremamente popular. É muito famoso pela grande velocidade. É o M na famosa pilha LAMP subjacente na maior parte dos sites web na internet. [MySQL]
- **VLC Player**. Este é o tocador de vídeo que pode tocar qualuqer coisa de DivX a MP3 a OGG, a VCDs e DVDs a ... que disse que software aberto não seria divertido? ;-) [VLC media player]
- **GeexBox** é uma distribuição de Linux que é projetada para tocar filmes assim que você reinicia do CD! [GeexBox]

Esta lista tem apenas a intenção de dar-lhe uma breve idéia - existem muitos mais excelentes FLOSS disponíveis, tais como a linguagem PERL, a linguagem PHP, os sistemas de gerenciamento de conteúdo para sites web PLONE e Drupal, o servidor de banco de dados PostgreSQL, o jogo de corridas TORCS, a IDE KDevelop, o tocador de filmes Xine, o editor VIM, o editor Quanta+, o tocador de áudio Banshee, o programa de edição de imagens GIMP ... Esta lista poderia não terminar jamais.
Para obter as últimas notícias sobre o mundo FLOSS, verifique os seguintes sites web:

- linux.com
- LinuxToday
- NewsForge
- DistroWatch

Visite os seguintes sites web para maiores informações sobre FLOSS:

- SourceForge
- FreshMeat

Assim, vá em frente e explore o vasto, livre e aberto mundo de FLOSS!

Python pt-br:Apendice Sobre

Cólofon

Quse todos os softwares que eu usei na criação deste livro são *softwares livres e abertos*.

Nascimento do Livro

No primeiro esboço deste livro, eu usei Linux Red Hat 9.0 como a fundação da minha estrutura e no sexto esboço eu usei Linux Fedora Core 3 como a base do meu esquema.
Inicialmente, eu estava usando KWord para escrever o livro (como explicado emLição de História no prefácio).

Era Adolescente

Mais tarde, eu mudei para DocBook XML usando Kate mas eu achei muito monótono. Assim, eu mudei para OppenOffice, que era excelente com o nível de controle que fornecia para formatação, assim como a geração de PDF, mas produzia um péssimo HTML do documento.
Finalmente, eu descobri XEmacs e reescrevi o livro desde o começo em DocBook XML (novamente) depois de decidir que aquele formato era a solução de longo prazo.
No sexto esboço, eu decidi usar Quanta+ para realizar toda a edição. As stylesheets que vieram junto com o Linux Fedora Core 3 foram utilizadas, assim como as fontes padrão. Entreanto, eu havia escrito um documento CSS para fornecer cor e estilo às páginas HTML. Eu havia escrito também um analisador léxico grosseiro, em Python, que automaticamente forneceu ênfase na sintaxe para as todas as listagens dos programas.

Agora

Para este sétimo esboço, eu estou usando MediaWiki como a base do meu setup. Agora, eu edito tudo online e os leitores podem diretamente ler/editar/discutir dentro do site wiki.
Eu ainda uso Vim para edição graças a ViewSourceWith extensão para Firefoxque integra-se com Vim.