Python para Informáticos

Explorando a Informação

Version 2.7.2

Autor: Charles Severance
Tradução: @victorjabur

Histórico de Publicação:

Maio 2015: Checagem editorial obrigado a Sue Blumenberg.

Outubro 2013: Revisão principal dos Capítulos 13 e 14 para mudar para JSON e usar OAuth. Novo capítulo adicionado na Visualização.

Setembro 2013: Livro publicado na Amazon CreateSpace

Janeiro 2010: Livro publicado usando a máquina da Universidade de Michigan Espresso Book.

Dezembro 2009: Revisão principal dos capítulos 2-10 de *Think Python: How to Think Like a Computer Scientist* e escrita dos capítulos 1 e 11-15 para produzir *Python for Informatics: Exploring Information*

Junho 2008: Revisão prncipal, título alterado para *Think Python: How to Think Like a Computer Scientist*.

Agosto 2007: Revisão principal, título alterado para *How to Think Like a (Python) Programmer*.

Abril 2002: Primeira edição de *How to Think Like a Computer Scientist*.

O código fonte LaTeX para a *Think Python: How to Think Like a Computer Scientist* versão deste livro está disponível em `http://www.thinkpython.com`.

Prefácio

Python para Informáticos: Adaptação de um livro aberto

É muito comum que acadêmicos, em sua profissão, necessitem publicar continuamente materiais ou artigos quando querem criar algo do zero. Este livro é um experimento em não partir da estaca zero, mas sim "remixar" o livro entitulado *Think Python: How to Think Like a Computer Scientist* escrito por Allen B. Downey, Jeff Elkner, e outros.

Em dezembro de 2009, quando estava me preparando para ministrar a disciplina **SI502 - Programação para Redes** na Universidade de Michigan para o quinto semestre e decidi que era hora de escrever um livro de Python focado em explorar dados ao invés de entender algoritmos e abstrações. Minha meta em SI502 é ensinar pessoas a terem habilidades na manipulação de dados para a vida usando Python. Alguns dos meus estudantes estavam planejando tornarem-se profissionais em programação de computadores. Ao invés disso, eles escolheram ser bibliotecários, gerentes, advogados, biólogos, economistas, etc., e preferiram utilizar habilmente a tecnologia nas áreas de suas escolhas.

Eu nunca consegui encontrar o livro perfeito sobre Python que fosse orientado a dados para utilizar no meu curso, então eu comecei a escrever o meu próprio. Com muita sorte, em uma reunião eventual três semanas antes de eu começar a escrever o meu novo livro do zero, em um descanso no feriado, Dr. Atul Prakash me mostrou o *Think Python* livro que ele tinha usado para ministrar seu Curso de Python naquele semestre. Era um texto muito bem escrito sobre Ciência da Computação com foco em explicações diretas e simples de se aprender.

Toda a estrutura do livro foi alterada, visando a resolução de problemas de análise de dados de um modo tão simples e rápido quanto possível, acrescido de uma série de exemplos executáveis e exercícios sobre análise de dados desde o início.

Os capítulos 2–10 são similares ao livro *Think Python* mas precisaram de muitas alterações. Exemplos com numeração e exercícios foram substituídos por exercícios orientados a dados. Tópicos foram apresentados na ordem necessária para construir soluções sofisticadas em análise de dados. Alguns tópicos tais como `try` e `except` foram movidos mais para o final e apresentados como parte do capítulo de condicionais. Funções foram necessárias para simplificar a complexidade na manipulação dos programas introduzidos anteriormente nas primeiras

lições em abstração. Quase todas as funções definidas pelo usuário foram removidas dos exemplos do código e exercícios, com exceção do Capítulo 4. A palavra "recursão"[1] não aparece no livro inteiro.

Nos capítulos 1 e 11–16, todo o material é novo, focado em exemplos reais de uso e exemplos simples de Python para análise de dados incluindo expressões regulares para busca e transformação, automação de tarefas no seu computador, recuperação de dados na internet, extração de dados de páginas web, utilização de web services, transformação de dados em XML para JSON, e a criação e utilização de bancos de dados utilizando SQL (Linguagem estruturada de consulta em bancos de dados).

O último objetivo de todas estas alterações é a mudança de foco, de Ciência da Computação para uma Informática que inclui somente tópicos que podem ser utilizados em uma turma de primeira viagem (iniciantes) que podem ser úteis mesmo se a escolha deles não for seguir uma carreira profissional em programação de computadores.

Estudantes que acharem este livro interessante e quiserem se aprofundar devem olhar o livro de Allen B. Downey's *Think Python*. Porque há muita sinergia entre os dois livros, estudantes irão rapidamente desenvolver habilidades na área com a técnica de programação e o pensamento em algoritmos, que são cobertos em *Think Python*. Os dois livros possuem um estilo de escrita similar, é possível mover-se para o livro *Think Python* com o mínimo de esforço.

Com os direitos autorais de *Think Python*, Allen me deu permissão para trocar a licença do livro em relação ao livro no qual este material é baseado de GNU Licença Livre de Documentação para a mais recente Creative Commons Attribution — Licença de compartilhamento sem ciência do autor. Esta baseia-se na documentação aberta de licenças mudando da GFDL para a CC-BY-SA (i.e., Wikipedia). Usando a licença CC-BY-SA, os mantenedores deste livro recomendam fortemente a tradição "copyleft" que incentiva os novos autores a reutilizarem este material da forma como considerarem adequada.

Eu sinto que este livro serve de exemplo sobre como materiais abertos (gratuitos) são importantes para o futuro da educação, e quero agradecer ao Allen B. Downey e à editora da Universidade de Cambridge por sua decisão de tornar este livro disponível sob uma licença aberta de direitos autorais. Eu espero que eles fiquem satisfeitos com os resultados dos meus esforços e eu desejo que você leitor esteja satisfeito com *nosso* esforço coletivo.

Eu quero fazer um agradecimento ao Allen B. Downey e Lauren Cowles por sua ajuda, paciência, e instrução em lidar com este trabalho e resolver os problemas de direitos autorais que cercam este livro.

Charles Severance
www.dr-chuck.com

[1] Com exceção, naturalmente, desta linha.

Ann Arbor, MI, USA
9 de Setembro de 2013

Charles Severance é um Professor Associado à Escola de Informação da Universidade de Michigan.

Tradução:
@victorjabur

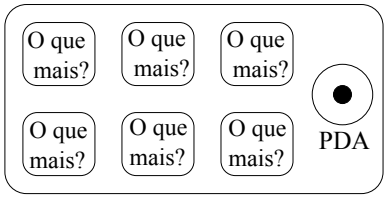
Sumário

Capítulo 1

Por que você deve aprender a escrever programas ?

Escrever programas (ou programação) é uma atividade muito criativa e recompensadora. Você pode escrever programas por muitas razões, que vão desde resolver um difícil problema de análise de dados a se divertir ajudando alguém a resolver um problema. Este livro assume que *qualquer pessoa* precisa saber como programar, e uma vez que você sabe como programar, você irá imaginar o que você quer fazer com suas novas habilidades.

Nós estamos cercados no nosso dia a dia por computadores, desde notebooks até celulares. Nós podemos achar que estes computadores são nossos "assistentes pessoais" que podem cuidar de muitas coisas a nosso favor. O hardware desses computadores no nosso dia a dia é essencialmente construído para nos responder a uma pergunta, "O que você quer que eu faça agora ?"



Programadores adicionam um sistema operacional e um conjunto de aplicações ao hardware e nós terminamos com um Assistente Pessoal Digital que é muito útil e capaz de nos ajudar a fazer diversas coisas.

Nossos computadores são rápidos, tem vasta quantidade de memória e pode ser muito útil para nós, somente se conhecermos a linguagem falada para explicar para um computador o que nós gostaríamos de fazer "em seguida". Se nós conhecemos esta linguagem, nós podemos pedir ao computador para fazer tarefas repetitivas a nosso favor. Curiosamente, as coisas que os computadores podem fazer melhor são frequentemente aquelas coisas que humanos acham chatas e entediantes.

Por exemplo, olhe para os três primeiros parágrafos deste capítulo e me diga qual é a palavra mais usada e quantas vezes. Contá-las é muito doloroso porque não é o tipo de problema que mentes humanas foram feitas para resolver. Para um computador o oposto é verdade, ler e entender o texto de um pedaço de papel é difícil, mas contar palavras dizendo a você quantas vezes ela aparece é muito fácil:

```
python palavras.py
Digite o nome do arquivo: palavras.txt
para 16
```

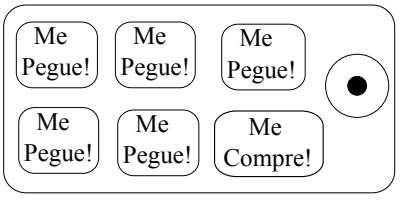
Nosso "assistente de análise pessoal de informações" rapidamente conta para nós que a palavra "para" foi utilizada dezesseis vezes nos primeiros três parágrafos deste capítulo.

Este fato de que os computadores são bons em coisas que humanos não são é a razão pela qual você precisa tornar-se qualificado em falar a "linguagem do computador". Uma vez que você aprende esta nova linguagem, pode delegar tarefas mundanas para o seu parceiro (o computador), ganhando mais tempo para fazer coisas que você foi especialmente adaptado para fazer. Você agrega criatividade, intuição e originalidade para o seu parceiro.

1.1 Criatividade e motivação

Embora este livro não se destine a programadores profissionais, programação profissional pode ser um trabalho muito gratificante, tanto financeiramente quanto pessoalmente. Construir programas úteis, elegantes, inteligentes para que outros utilizem é uma atividade criativa. Seu computador ou assistente pessoal digital (PDA) geralmente contém muitos programas diferentes feitos por diversos grupos de programadores, todos competindo por sua atenção e seu interesse. Eles tentam dar o seu melhor para atender suas necessidades e dar a você uma boa experiência de usabilidade no processo. Em algumas situações, quando você executa um trecho de software, os programadores são diretamente recompensados por sua escolha.

Se nós pensarmos em programas como resultado criativo de grupos de programadores, então talvez a figura a seguir seja uma versão mais sensata de nosso PDA:

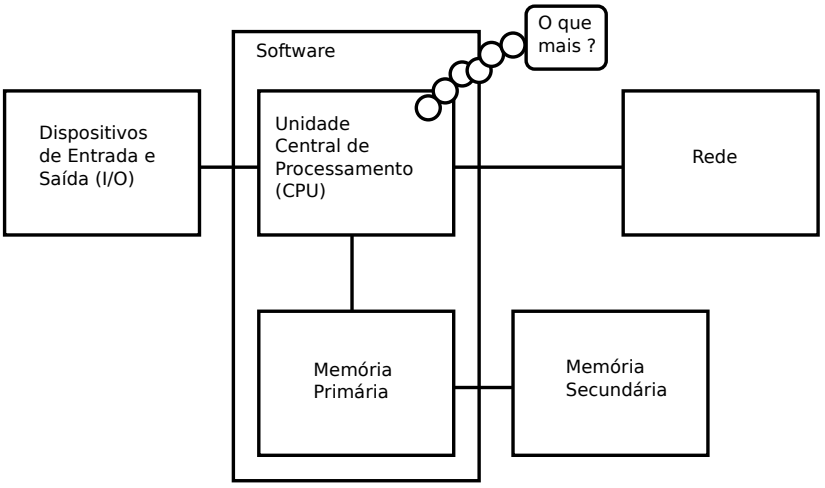


Por enquanto, nossa motivação primária não é ganhar dinheiro ou agradar usuários finais, para nós sermos mais produtivos na manipulação de dados e informações que nós encontraremos em nossas vidas. Quando você começar, você será tanto

o programador quanto o usuário final de seus programas. Conforme você ganhar habilidades como programador e melhorar a criatividade em seus próprios programas, mais você pode pensar em programar para os outros.

1.2 Arquitetura física do Computador - Hardware

Antes de começar a estudar a linguagem, nós falamos em dar instruções aos computadores para desenvolver software, nós precisamos aprender um pouco mais sobre como os computadores são construídos. Se você desmontar seu computador ou celular e olhar por dentro, você encontrará as seguintes partes:



As definições resumidas destas partes são:

- A **Unidade Central de Processamento** (ou CPU) é a parte do computador que é feita para sempre te perguntar: "O que mais ?" Se seu computador possui uma frequência de 3.0 Gigahertz, significa que a CPU irá te perguntar "O que mais ?" três bilhões de vezes por segundo. Você irá aprender como conversar tão rápido com a CPU.

- A **Memória Principal** é utilizada para armazenar informação que a CPU precisa com muita pressa. A memória principal é aproximadamente tão rápida quanto a CPU. Mas a informação armazenada na memória principal se perde quando o computador é desligado (volátil).

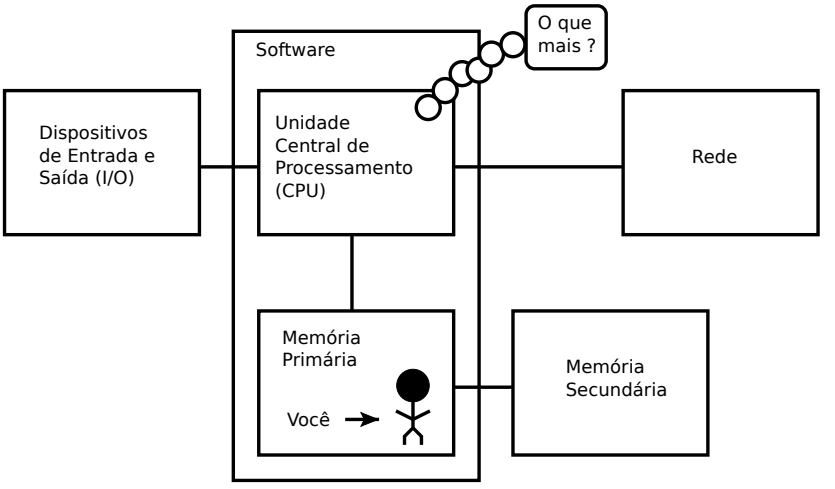
- A **Memória Secundária** é também utilizada para armazenar informação, mas ela é muito mais lenta que a memória principal. A vantegem da memória secundária é que ela pode armazenar informação que não se perde quando o computador é desligado. Exemplos de memória secundária são discos rígidos (HD), pen drives, cartões de memória (sd card) (tipicamente) encontradas no formato de USB e portáteis.

- Os **Dispositivos de Entrada e Saídas** são simplesmente nosso monitor (tela), teclado, mouse, microfone, caixa de som, touchpad, etc. Eles são todas as formas com as quais interagimos com o computador.

- Atualmente, a maioria dos computadores tem uma **Conexão de Rede** para buscar informação em uma rede. Nós podemos pensar a rede como um lugar muito lento para armazenar e buscar dados que podem não estar "disponíveis". Em essência, a rede é mais lenta e às vezes parece uma forma não confiável de **Memória Secundária**.

É melhor deixar a maior parte dos detalhes de como estes componentes funcionam para os construtores dos computadores, isso nos ajuda a ter alguma terminologia que podemos utilizar para conversar sobre as diferentes partes nos programas que vamos escrever.

Como um programador, seu trabalho é usar e orquestrar cada um destes recursos para resolver um problema que você precisa resolver e analisar os dados que você obtém da solução. Como um programador você irá "conversar" com a CPU e contar a ela o que fazer em um próximo passo. Algumas vezes você irá dizer à CPU para usar a memória principal, a memória secundária, a rede ou os dispositivos de entrada e saída.



Você precisa ser a pessoa que responde à pergunta "O que mais ?" para a CPU. Mas seria muito desconfortável se você fosse encolhido para uma altura de apenas 5 mm e inserido dentro de um computador e ainda ter que responder uma pergunta três bilhões de vezes por segundo. Então, ao invés disso, você deve escrever suas instruções previamente. Nós invocamos as instruções armazenadas de um **programa** e o ato de escrita dessas instruções precisa ser correto.

1.3 Entendendo programação

No restante deste livro, nós iremos tentar fazer de você uma pessoa com habilidades na arte da programação. No final você será um **programador**, no entanto não um programador profissional, mas pelo menos você terá os conhecimentos para analisar os problemas de dados/informações e desenvolver um programa para resolver tais problemas.

Resumidamente, você precisa de duas qualidades para ser um programador:

- Primeiramente, você precisa conhecer uma linguagem de programação (Python) - você precisa conhecer o vocabulario e a gramática. Você precisa saber pronunciar as palavras desta nova linguagem corretamente e conhecer como construir sentenças bem formadas nesta linguagem.

- Segundo, você precisa "contar uma estória". Na escrita da estória, você combina palavras e sentenças para convencer o leitor. É necessário qualidade e arte na construção da estória, adquiri-se isto através da prática de contar estórias e obter um feedback. Na programação, nosso programa é a "estória" e o problema que você quer resolver é a "idéia".

Uma vez que você aprende uma linguagem de programação, como o Python, você irá achar muito mais fácil aprender a segunda linguagem de programação, tal como JavaScript ou C++. A nova linguagem de programação possuirá um vocabulário e gramática bastante diferente, mas as habilidades na resolução do problemas serão as mesmas em qualquer linguagem.

Você aprenderá o "vocabulário" e "sentenças" do Python rapidamente. Levará muito tempo para você tornar-se hábil em escrever programas coerentes para resolver um novo problema. Nós ensinamos programação assim como ensinamos a escrever. Nós estaremos lendo e explicando programas, nós escreveremos programas simples, e então nós aumentaremos a complexidade dos programas ao longo do tempo. Em algum momento, você "deslancha" e vê os padrões por si próprio e pode visualizar com maior naturalidade como escrever um programa para resolver o problema. Uma vez que você chega neste pontoprogramar torna-se um processo muito agradável e criativo.

Nós iniciamos com o vocabulário e a estrutura de programas em Python. Seja paciente com os exemplos simples, relembre quando você iniciou a leitura pela primeira vez.

1.4 Programas e Sentenças

Diferentemente dos idiomas humanos, o vocabulário do Python é atualmente muito pequeno. Nós chamamos isto de "vocabulário" e "palavras reservadas".

Estas palavras tem um significado especial no Python. Quando o Python encontra estas palavras em um programa, elas possuem um e somente um significado para o Python. Quando você escreve seus programas você irá definir suas próprias palavras com significado, são chamadas **variáveis**. Você pode escolher muitos nomes diferentes para as suas variáveis, mas você não pode usar qualquer palavra reservada do Python como o nome de uma variável.

Quando nós treinamos um cachorro, nós usamos palavras especiais, tais como: "sentado", "fique" e "traga". Quando você conversar com um cachorro e não usar qualquer uma destas palavras reservadas, eles ficarão olhando para você com um olhar curioso até que você diga uma palavra reservada. Por exemplo, se você disser: "Eu desejo que mais pessoas possam caminhar para melhorar a sua saúde", o que os cachorros vão ouvir será: "blah blah blah **walk** blah blah blah blah." Isto porque "caminhar" é uma palavra reservada na linguagem dos cachorros. Muitos podem sugerir que a linguagem entre humanos e gatos não tem palavras reservadas[1].

As palavras reservadas na linguagem onde os humanos conversam com o Python, incluem o seguinte:

```
and       del       from      not       while
as        elif      global    or        with
assert    else      if        pass      yield
break     except    import    print
class     exec      in        raise
continue  finally   is        return
def       for       lambda    try
```

É isto, e ao contrário do cachorro, o Python é completamente treinado. Quando você diz "try", o Python irá tentar todas as vezes que você pedir sem desobedecer.

Nós aprenderemos as palavras reservadas e como elas são usadas mais adiante, por enquanto nós iremos focar no equivalente ao Python de "falar" (na linguagem humano-para-cachorro). Uma coisa legal sobre pedir ao Python para falar é que nós podemos até mesmo pedir o que nós queremos através de uma mensagem entre aspas:

```
print 'Hello world!'
```

E finalmente nós escrevemos a nossa primeira sentença sintaticamente correta em Python. Nossa sentença inicia com uma palavra reservada **print** seguida por uma cadeia de caracteres textuais de sua escolha entre aspas simples.

1.5 Conversando com Python

Agora que você tem uma palavra e uma simples sentença que nós conhecemos em Python, nós precisamos saber como iniciar uma conversação com Python para testar nossas habilidades na nova linguagem.

[1] http://xkcd.com/231/

Antes de você conversar com o Python, você deve primeiramente instalar o programa Python em seu computador e aprender como inicializá-lo. Isto é muita informação para este capítulo, então eu sugiro que você consulte www.pythonlearn.com onde se encontra instruções e screencasts de preparação e inicialização do Python em sistemas Windows e Macintosh. Em algum momento, você estará no interpretador Python e estará executando o modo interativo e aparecer algo assim:

```
Python 2.6.1 (r261:67515, Jun 24 2010, 21:47:49)
[GCC 4.2.1 (Apple Inc. build 5646)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

O >>> prompt é a forma do interpretador Python perguntar o que você deseja: "O que você quer que eu faça agora ?" Python está pronto para ter uma conversação com você. Tudo o que você deve conhecer é como falar a linguagem Python.

Digamos, por exemplo, que você não conhece nem mesmo as mais simples palavras ou sentenças da linguagem Python. Você pode querer usar a linha padrão que os astronautas usam quando eles estão em uma terra distante do planeta e tentam falar com os habitantes do planeta:

```
>>> Eu venho em paz, por favor me leve para o seu líder
  File "<stdin>", line 1
    Eu venho em paz, por favor me leve para o seu líder
          ^
SyntaxError: invalid syntax
>>>
```

Isto não deu muito certo. A menos que você pense algo rapidamente, os habitantes do planeta provavelmente irão apunhalá-lo com uma lança, colocando-o em um espeto, assando-o no fogo, e comê-lo no jantar.

A sorte é que você trouxe uma cópia deste livro em sua viagem, e caiu exatamente nesta página, tente novamente:

```
>>> print 'Ola Mundo!'
Ola Mundo!
```

Isso parece bem melhor, então você tenta se comunicar um pouco mais:

```
>>> print 'Voce deve ser um Deus lendario que veio do ceu'
Voce deve ser um Deus lendario que veio do ceu
>>> print 'Nos estivemos esperando voce por um longo tempo'
Nos estivemos esperando voce por um longo tempo
>>> print 'Nossa lenda nos conta que voce seria muito apetitoso com mostarda'
Nossa lenda nos conta que voce seria muito apetitoso com mostarda
>>> print 'Nos teremos uma festa hoje a noite a menos que voce diga
  File "<stdin>", line 1
    print 'Nos teremos uma festa hoje a noite a menos que voce diga
                                                                   ^
SyntaxError: EOL while scanning string literal
>>>
```

A conversa foi bem por um momento, até que você cometeu o pequeno erro no uso da linguagem e o Python trouxe a lança de volta.

Até o momento, você deve ter percebido que o Python é incrivelmente complexo e poderoso e muito exigente em relação à sintaxe que você utiliza para se comunicar com ele, Python *não* é inteligente. Você está na verdade tendo uma conversa com você mesmo, mas usando uma sintaxe apropriada.

De certa forma, quando você usa um programa escrito por alguém, a conversa ocorre entre você e os programadores, neste caso o Python atuou como um intermediário. Python é uma forma para que os criadores de programas se expressem sobre como uma conversa deve proceder. E em poucos capítulos, você será um dos programadores usando Python para conversar com os usuários de seus programas.

Antes de sairmos da nossa primeira conversa com o interpretador do Python, você deve conhecer o modo correto de dizer "ate-logo" quando interagir com os habitantes do Planeta Python.

```
>>> ate-logo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'ate' is not defined

>>> se voce nao se importa, eu preciso ir embora
  File "<stdin>", line 1
    se voce nao se importa, eu preciso ir embora
          ^
SyntaxError: invalid syntax

>>> quit()
```

Você pode perceber que o erro é diferente nas duas primeiras tentativas incorretas. No primeiro erro, por tratar-se de uma palavra simples, o Python não pode encontrar nenhuma função ou variável com este nome. No segundo erro, existe um erro de sintaxe, não sendo reconhecida a frase como válida.

O jeito correto de se dizer "ate-logo" para o Python é digitar **quit()** no prompt do interpretador interativo. É provável que você tenha perdido certo tempo tentado fazer isso, ter um livro em mãos irá tornar as coisas mais fáceis e pode ser bastante útil.

1.6 Terminologia: interpretador e compilador

Python é uma linguagem **alto nível** cujo objetivo é ser relativamente fácil para humanos ler e escrever e para computadores ler e processar. Outras linguagens alto nível incluem Java, C++, PHP, Ruby, Basic, Perl, JavaScript, e muito mais. O atual hardware dentro da Unidade Central de Processamento (CPU) não é capaz de entender nenhum destes comando em alto nível.

A CPU entende a linguagem que chamamos de **linguagem de máquina**. Linguagem de máquina é muit simples e francamente cansativa de se escrever porque ela é representada em zeros e uns:

```
01010001110100100101010000001111
11100110000011101010010101101101
...
```

Linguagem de máquina parece simples olhando-se de um modo superficial, dado que são apenas zeros e uns, mas sua sintaxe é muito mais complexa e mais distante que o Python. Poucos programadores escrevem linguagem de máquina. Invés disso, nós usamos vários tradutores para permitir que os programadores escrevam linguagem de máquina em linguagens de alto nível como o Python ou o JavaScript e estes tradutores convertem os programas para linguagem de máquina para execução pela CPU.

Visto que linguagem de máquina é vinculada ao hardware do computador, linguagem de máquina não é **portável** entre diferentes tipos de hardware. Programas que foram escritos em linguagens de alto nível podem mover-se entre diferentes computadores usando um interpretador diferente em cada máquina ou então recompilando o código para criar uma versão de linguagem de máquina do programa para a nova máquina.

Os tradutores das linguagens de programação se enquadram em duas características gerais: (1) interpretadores e (2) compiladores

Um **interpretador** lê o código fonte de um programa da forma como foi escrito pelo programador, analisa, e interpreta as instruções em tempo de execução. Python é um interpretador e quando ele está rodando Python no modo interativo, nós podemos digitar uma linha de Python (uma sentença) e o Python processa imediatamente e está ponto para receber outra linha de Python.

Algumas das linhas de Python diz a ele que você quer armazenar algum valor para resgatar depois. Nós precisamos dar um nome para um valor de forma que possa ser armazenado e resgatado através deste nome simbólico. Nós usamos o termo **variável** para se referir aos apelidos que nós demos ao dado que foi armazenado.

```
>>> x = 6
>>> print x
6
>>> y = x * 7
>>> print y
42
>>>
```

Neste exemplo, nós pedimos ao Python para armazenar o valor seis e usar um apelido **x** assim nós podemos resgatar o valor mais tarde. Nós verificamos que o Python realmente lembrou dos valores quando usamos a função **print**. Então nós perguntamos ao Python para resgatar **x** e multiplicar por sete a armazenar de novo

em uma variável **y**. Então nós pedimos ao Python para imprimir o valor corrente em **y**.

Mesmo que nós digitemos estes comandos em uma única linha de Python por vez, o Python está processando elas em sequência, mantendo a ordem, de forma que as instruções seguintes consigam recuperar os dados criados pelas anteriores. Nós escrevemos nosso primeiro parágrafo simples com quatro sentenças com uma ordem lógica e com um significado.

É da natureza de um **interpretador** ser capaz de ter uma conversa interativa, como foi mostrado acima. Um **compilador** precisa ter em mãos o programa completo em um arquivo, e então ele roda um processo para traduzir um código fonte em alto nível para uma linguagem de máquina e então o compilador coloca o resultado deste processo em um outro arquivo para posterior execução.

Se você está em um sistema Windows, frequentemente este programa executável em código de máquina tem o sufixo ".exe" ou ".dll", o qual é chamado de "executável" ou "biblioteca de link dinâmico", respectivamente. Em Linux e Macintosh, não há um sufixo que marca um arquivo unicamente como executável.

Se você abrir um arquivo executável em um editor de texto, verá algo completamente doido e ilegível.

```
^?ELF^A^A^A^@^@^@^@^@^@^@^@^B^@^C^@^A^@^@^@\xa0\x82
^D^H4^@^@^@\x90^]^@^@^@^@^@^@4^@ ^@^G^@(^@$^@!^@^F^@
^@^@4^@^@^@4\x80^D^H4\x80^D^H\xe0^@^@^@\xe0^@^@^@^E
^@^@^@^D^@^@^@^C^@^@^@^T^A^@^@^T\x81^D^H^T\x81^D^H^S
^@^@^@^S^@^@^@^D^@^@^@^A^@^@^@^A\^D^HQVhT\x83^D^H\xe8
....
```

Não é nada fácil ler ou escrever código de máquina, assim é bom que tenhamos **interpretadores** e **compiladores** que nos permitam escrever em linguagens de alto nível assim como o Python ou o C.

Agora neste ponto em nossa discussão de compiladores e interpretadores, você deve estar com algumas dúvidas sobre o funcionamento do interpretador Python. Em que linguagem é escrito ? É escrito em uma linguagem compilada ? Quando nós digitamos "python", o que exatamente acontece ?

O interpretador Python é escrito em uma linguagem de alto nível, chamada "C". Você pode dar uma olhada no código fonte atual do interpretador através do endereço `www.python.org` e trabalhar como você quiser com o código fonte. Então Python é um programa compilado em uma linguagem de máquina. Quando você instalou Python em seu computador (ou o fornecedor instalou), você copiou um código de máquina do programa Python traduzido para o seu sistema. Em Windows, o código de máquina executável para o Python encontra-se em um arquivo com um nome a seguir:

```
C:\Python27\python.exe
```

Isto é mais do que você realmente precisa conhecer para ser um programador Python, mas às vezes, isto ajuda a entender questões que intrigam justamente no início.

1.7 Escrevendo um programa

Digitar comandos em um Interpretador Python é uma boa maneira de experimentar as características da linguagem, mas isto não é recomendado para resolver problemas mais complexos.

Quando nós queremos escrever um programa, usamos um editor de texto para escrever as instruções Python em um arquivo, o qual chamamos de **script**. Por convenção, scripts Python tem nomes que terminam com `.py`.

Para executar o script, você tem que dizer ao interpretador do Python o nome do arquivo. Em uma janela de comandos Unix ou Windows, você digita `python hello.py` como a seguir:

```
csev$ cat hello.py
print 'Ola Mundo!'
csev$ python hello.py
Ola Mundo!
csev$
```

O "csev$" é o prompt do sistema operacional, e o "cat hello.py" é para nos mostrar que o arquivo "hello.py" tem uma linha de programa Python para imprimir uma string.

Nós chamamos o interpretador Python e pedimos a ele para ler o código fonte do arquivo "hello.py" ao invés dele nos perguntar quais são as próximas linhas de modo interativo.

Você notará que não é preciso ter o **quit()** no fim do programa Python no arquivo. Quando o Python está lendo o seu código fonte de um arquivo, ele sabe que deve parar quando chegar ao fim do arquivo.

1.8 O que é um programa ?

A definição de um **programa** em sua forma mais básica é uma sequência de comandos Python que foram criados para fazer algo. Mesmo o nosso simples script **hello.py** é um programa. É um programa de uma linha e não é particularmente útil, mas na estrita definição, é um programa Python.

Pode ser mais fácil entender o que é um programa, imaginando qual problema ele foi construído para resolver, e então olhar para o programa que resolve um problema.

Vamos dizer que você está fazendo uma pesquisa de computação social em posts do Facebook e está interessado nas palavras mais frequentes em uma série de posts. Você não pode imprimir o stream de posts do Facebook e debruçar-se sobre o texto procurando pela palavra mais comum, mas pode levar um longo tempo e ser muito propenso a erros. Você pode ser inteligente para escrever um programa Python para tratar disso rapidamente a com acurácia, então você pode passar seu final de semana fazendo algo divertido.

Por exemplo, olhe para o seguinte texto sobre o palhaço e o carro. Olhe para o texto e imagine qual é a palavra mais comum e quantas vezes ela aparece:

```
O palhaço correu atrás do carro e o carro correu para a tenda
e a tenda caiu em cima do palhaço e do carro
```

Então imagine que você está fazendo esta tarefa olhando para milhões de linhas de texto. Francamente será mais rápido para você aprender Python e escrever um programa Python para contar as palavras do que você manualmente escanear as palavras.

A notícia ainda melhor é que eu já fiz para você um simples programa para encontrar a palavra mais comum em um arquivo texto. Eu escrevi, testei e agora eu estou dando isto para que você use e economize algum tempo.

```
name = raw_input('Enter file:')
handle = open(name, 'r')
text = handle.read()
words = text.split()
counts = dict()

for word in words:
   counts[word] = counts.get(word,0) + 1

bigcount = None
bigword = None
for word,count in counts.items():
    if bigcount is None or count > bigcount:
        bigword = word
        bigcount = count

print bigword, bigcount
```

Você nem precisa conhecer Python para usar este programa. Você precisará chegar até o capítulo 10 deste livro para entender completamente as impressionantes técnicas Python que foram utilizadas para fazer o programa. Você é o usuário final, você simplesmente usa o programa e admira-se com a inteligência e em como ela poupou seus esforços manuais. Você simplesmente digitou o código em um arquivo chamado **words.py** e executou ou então fez o download do código fonte no site `http://www.pythonlearn.com/code/` e executou.

Este é um bom exemplo de como o Python e sua linguagem podem atuar como um intermediário entre você (o usuário final) e eu (o programador). Python é uma

forma para trocarmos úteis sequências de instruções (i.e., programas) em uma linguagem comum que pode ser usada por qualquer que instalar Python em seu computador. Então nenhum de nós está conversando *com o Python* mas sim nos comunicando uns com os outros *através* de Python.

1.9 A construção de blocos de programas

Em poucos capítulos, nós iremos aprender mais sobre o vocabulário, estrutura das sentenças, dos parágrafos e da história do Python. Nós iremos aprender sobre as capacidades poderosas do Python e como compor estas capacidades juntas para criar programas úteis.

Há alguns padrões conceituais de baixo nível que nós usamos para construir programas. Estas construções não são apenas para programas Python, elas são parte de todas as linguagens de programação desde linguagens de baixo nível até as de alto nível.

input: Obter dados do "mundo externo". Estes dados podem ser lidos de um arquivo ou mesmo de algum tipo de sensor como um microfone ou um GPS. Em nossos primeiros programas, nosso input virá de um usuário que digita dados no teclado.

output: Exibe os resultados do programa em uma tela ou armazena-os em um arquivo ou talvez os escreve em algum dispositivo tal como um alto falante para tocar música ou falar o texto.

execução sequencial: Executa instruções uma após a outra respeitando a sequência encontrada no script.

execução condicional: Avalia certas condições e as executa ou pula a sequência de instruções.

execução repetitiva: Executa algumas instruções repetitivamente, geralmente com alguma variação.

reúso: Escrever um conjunto de instruções uma única vez, dar um nome a elas e reusar estas instruções em várias partes de um programa.

Parece simples demais para ser verdade, e naturalmente que isto nunca é tão simples. É como dizer que caminhar é simplesmente "colocar um pé na frente do outro". A "arte" de escrever um programa é compor e costurar estes elementos básicos muitas vezes para produzir algo que seja útil aos usuários.

O programa de contar palavras acima usa todos estes padrões exceto um.

1.10 O que pode dar errado ?

Como vimos em nossa última conversa com o Python, devemos nos comunicar de modo preciso quando escrevemos código Python. O mínimo desvio ou erro fará com que o Python pare de executar o seu programa.

Programadores iniciantes mutas vezes tomam o fato de que o Python não deixa espaço para erros, como prova de que ele é malvado e cruel. Enquanto o Python parece gostar de todo mundo, ele os conhece pessoalmente e guarda um ressentimento contra eles. Devido a este ressentimento, o Python leva avalia nossos programas perfeitamente escritos e os rejeita como "incorretos" apenas para nos atormentar.

```
>>> primt 'Ola mundo!'
  File "<stdin>", line 1
    primt 'Ola mundo!'
                  ^
SyntaxError: invalid syntax
>>> primt 'Ola mundo'
  File "<stdin>", line 1
    primt 'Ola mundo'
                 ^
SyntaxError: invalid syntax
>>> Eu te odeio Python!
  File "<stdin>", line 1
    Eu te odeio Python!
       ^
SyntaxError: invalid syntax
>>> se você vier aqui fora, vou te dar uma lição
  File "<stdin>", line 1
    se você vier aqui fora, vou te dar uma lição
             ^
SyntaxError: invalid syntax
>>>
```

Não se ganha muita coisa discutindo com o Python. Ele é somente uma ferramenta. Ele não tem emoções e também não fica feliz e pronto para te servir quando você precisa dele. Suas mensagens de erro parecem ásperas, mas elas apenas tentam nos ajudar. Ele recebeu o seu comando e simplesmente não conseguiu entender o que você digitou.

Python se parece muito com um cachorro, te ama incondicionalmente, consegue entender apenas algumas poucas palavras, olha para você com um olhar doce na face (>>>), e fica esperando você dizer algo que ele entenda. Quando o Python diz "SyntaxError: invalid syntax", está simplesmente abanando o rabo e dizendo, "Parece que você disse algo que eu não consegui entender, por favor, continue conversando comigo (>>>)."

Conforme seu programa vai se tornando mais sofisticado, você encontrará três tipos genéricos de erro:

Erros de Sintaxe: Estes são os primeiros erros que você cometerá e os mais fáceis de se consertar. Um erro de sintaxe significa que você violou as "regras gramaticais" do Python. Python dá o seu melhor para apontar a linha correta e o caractere que o confundiu. A única parte complicada dos erros de sintaxe é que às vezes os erros que precisam de conserto na verdade ocorrem um pouco antes de onde o Python *indica* e isso confunde um pouco. Desta forma, a linha e caractere que o Python indica no erro de sintaxe pode ser que seja apenas um ponto de início e precisa da sua investigação.

Erros de Lógica: Um erro de lógica é quando o seu programa tem uma boa sintaxe mas há um erro na ordem das instruções ou às vezes um erro em como uma instrução se relaciona com as demais. Um bom exemplo de erro de lógica pode ser, "tome um gole de sua garrafa de água, coloque-a na mochila, caminhe para a biblioteca, e depois coloque a tampa de volta na garrafa."

Erros de Semântica: Um erro de semântica é quando a descricao dos passos estão sintaticamente corretos, na ordem certa, mas há existe um erro no programa. O programa está perfeitamente correto, mas ele não faz o que você *deseja* que ele faça. Um exemplo simples poderia ser quando você instrui uma pessoa a chegar até um restaurante e diz, "quando você cruzar a estação de gás, vire à esquerda e ande por um quilômetro e o restaurante estará no prédio vermelho à sua esquerda." Seu amigo está muito atrasado e liga para você para dizer que está em uma fazenda, passando atrás de um celeiro, sem o sinal da existência de um restaurante. Então você diz "você virou à esquerda ou à direita na estação de gás ?" e ele diz: "Eu segui suas instruções perfeitamente, as escrevi em um papel, e dizia para virar à esquerda e andar por um quilômetro até a estação de gás." Então você diz: "Eu sinto muito, embora minhas instruções estivessem sintaticamente corretas, elas infelizmente tinham um pequeno erro semântico indetectável."

Novamente em todos os três tipos de erros, o Python está se esforçando para fazer tudo aquilo que você pediu.

1.11 A jornada do aprendizado

Enquanto você progride para o restante do livro, não tenha medo se os conceitos não parecem se encaixar tão bem em um primeiro momento. Quando você aprender a falar, verá que os problemas que você enfrentou valeram a pena e não se lembrará mais das dificuldades. Estará tudo certo se você levar seis meses para se mover de um simples vocabulário até simples sentenças e levar mais 5-6 anos para mover-se de sentenças a parágrafos, e uns anos mais para estar habilitado a escrever uma interessante e curta estória com suas próprias mãos.

Nós queremos que você aprenda Python muito mais rápido, então nós ensinamos tudo ao mesmo tempo nos próximos capítulos. Mas aprender uma nova linguagem

leva tempo para ser absorver e entender antes de se tornar natural. Este processo pode gerar alguma confusão conforme nós visitamos e revisitamos os tópicos para tentar dar a você uma visão completa, nós definimos pequenos fragmentos que aos poucos irão formando a visão completa. Este livro é dividido em capítulos sequenciais e à medida que você avança vai aprendendo diversos assuntos, não se sinta preso na sequência do livro, avance capítulos e depois recue se for preciso, o que importa é o seu aprendizado e em como você sente que deve ser. Ao estudar superficialmente materiais mais avançados sem entender completamente os detalhes, você pode obter um melhor entendimento do "porque?" programar. Revisando materiais mais básicos e até mesmo refazendo exercícios anteriores, você irá perceber que aprendeu muito, até mesmo com aqueles materiais que pareciam impenetráveis de tão difíceis.

Normalmente, quando você aprende sua primeira linguagem de programação, ocorrem vários momentos "Ah Hah!". Aqueles em que você está trabalhando arduamente e quando para para prestar atenção e dar um descanso percebe que está construindo algo maravilhoso.

Se algo estiver particularmente difícil, saiba que não vale a pena ficar acordado a noite inteira encarando o problema. Faça uma pausa, tire um cochilo, faça um lanche, compartilhe o seu problema com alguém (com seu cão talvez) e então retorne ao problema com a mente descansada. Eu asseguro a você que uma vez que você aprender os conceitos de programação neste livro, irá olhar para trás e perceber que tudo foi muito fácil, elegante e tão simples que tomou de você apenas um tempo para absorver o aprendizado.

1.12 Glossário

bug: Um erro em um programa.

unidade central de processamento: O coração de qualquer computador. É ela que executa o software que nós escrevemos; também chamada de "CPU" ou de "processador".

compilação: Traduzir um programa escrito em uma linguagem de alto nível em uma linguagem de baixo nível tudo de uma vez, em preparação para uma posterior execução.

linguagem de alto nível: Uma linguagem de programação como o Python que é desenhada para ser fácil para humanos ler e escrever.

modo interativo: Um modo de usar o interpretador Python digitando comandos e expressões no prompt.

interpretação: Executar um programa em uma linguagem de alto nível traduzindo uma linha por vez.

linguagem de baixo nível: Uma linguagem de programação que é desenhada para que seja fácil um computador executar; também chamada "código de máquina" ou "linguagem assembly".

código de máquina: A linguagem mais baixo nível que pode existir em software, é a linguagem que é diretamente executada pela unidade central de processamento (CPU).

memória principal: Armazena programas e dados. A memória principal perde informação quando a energia é desligada.

parse: Examinar um programa e analisar a estrutura sintática.

portabilidade: Uma propriedade de um programa que roda em mais de um tipo de computador.

instrução print: Uma instrução que faz com que o interpretador Python exiba um valor na tela.

resolução de problema: O processo de formular um problema, encontrar a solução e expressá-la.

programa: Um conjunto de instruções que especifica uma computação.

prompt: Quando um programa exibe uma mensagem e aguarda o usuário digitar algo para o programa.

memória secundária: Armazena programas e dados e retém a informação mesmo quando a energia é desligada. Geralmente mais devagar em relação à memória principal. Exemplos de memória secundária são discos rígidos e memória flash nos pendrives USB.

semântica: O significado de um programa.

erro semântico: Um erro em um programa que faz algo diferente daquilo que o programador desejava.

código fonte: Um programa em uma linguagem de alto nível.

1.13 Exercícios

Exercício 1.1 Qual é a função da memória secundária em um computador ?

a) Executar todas as computações e lógica de um programa
b) Obter páginas web da internet
c) Armazenar informação por um longo período – mesmo se faltar energia
d) Receber o input de um usuário

Exercício 1.2 O que é um programa ?

Exercício 1.3 Qual é a diferença entre um compilador e um interpretador ?

Exercício 1.4 Qual destas opções a seguir contém "código de máquina"?

a) O interpretador Python
b) O teclado
c) Arquivo de código fonte Python
d) Um documento do processador de texto

Exercício 1.5 O que está errado no código a seguir:

```
>>> primt 'Ola mundo!'
  File "<stdin>", line 1
    primt 'Ola mundo!'
                      ^
SyntaxError: invalid syntax
>>>
```

Exercício 1.6 Em qual lugar do computador existe uma variável "X" armazenada depois que a seguinte linha de Python finaliza ?

```
x = 123
```

a) Unidade central de processamento
b) Memória Principal
c) Memória Secundária
d) Dispositivos de Entrada
e) Dispositivos de Saída

Exercício 1.7 O que o seguinte programa irá imprimir:

```
x = 43
x = x + 1
print x
```

a) 43
b) 44
c) x + 1
d) Um erro porque x = x + 1 não é matematicamente possível

Exercício 1.8 Explique cada item a seguir usando como exemplo uma capacidade humana: (1) Unidade central de processamento, (2) Memória principal, (3) Memória secundária, (4) Dispositivo de entrada, e (5) Dispositivo de saída. Por exemplo, "Qual é a capacidade humana equivalente a Unidade central de processamento"?

Exercício 1.9 Como se conserta um "Erro de Sintaxe"?

Capítulo 2

Utilizando Banco de Datos e Structured Query Language (SQL)

2.1 O que é um banco de dados?

Um **banco de dados** é um tipo de arquivo organizado para armazenamento de dados. A maioria dos bancos de dados são orgazanizados como um dicionário, no sentido de que eles realizam o mapeamento por chaves e valores. A grande diferença é que os bancos de dados estão em disco (ou outros dispositivos de armazenamentos permanentes), então eles continuam armazenando os dados mesmo depois que o programa termina. Porque um banco de dados é armazenado de forma permanente, isto permite armazenar muito mais dados que um dicionário, que é limitado ao tamanho da memória no computador.

Como um dicionário, um banco de dados é um software desenvolvido para manter a inserção e acesso aos dados de forma muito rápida, até para grandes volumes de dados. O banco de dados mantém sua performance através da construção de **indices** assim que o dado é adicionado, isto permite ao computador acessar rapidamente uma entrada em particular.

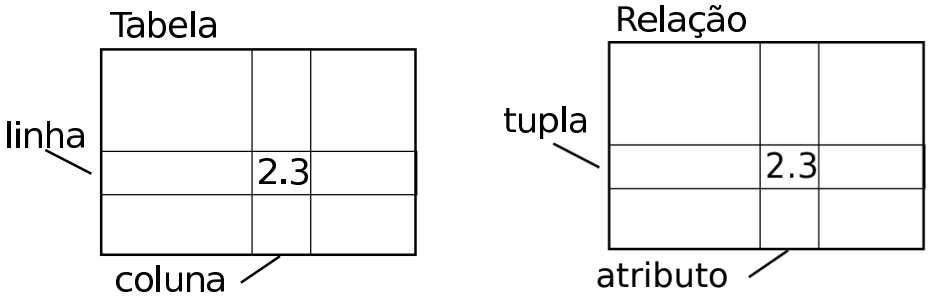
Existem diferentes tipos de sistemas de bancos de dados que são utilizados para diferentes propósitos, alguns destes são: Oracle, MySQL, Microsoft SQL Server, PostgreSQL, e SQLite. Focaremos no uso do SQLite neste livro pois é um banco de dados comum e já está integrado ao Python. O SQLite foi desenvolvido com o propósito de ser *embarcado* em outras aplicações para prover suporte a banco de dados junto à aplicação. Por exemplo, o navegador Firefox utiliza o SQLite internamente, assim como muitos outros produtos.

`http://sqlite.org/`

SQLite é adequado para alguns problemas de manipulação de dados que podemos ver na informática como a aplicação de indexação do Twitter que descrevemos neste capítulo.

2.2 Conceitos de bancos de dados

Quando você olha para um banco de dados pela primeira vez, parece uma planilha (como uma planilha de cálculo do LibreOffice) com múltiplas folhas. A estrutura de dados básica que compõem um banco de dados são: **tabelas**, **linhas**, e **colunas**.



Na descrição técnica de um banco de dados relacional o conceito de tabela, linha e coluna são referências formais para **relação**, **tupla**, e **atributo**, respectivamente. Usaremos os termos menos formais neste capítulo.

2.3 Plugin do Firefox de Gerencia SQLite

O foco deste capítulo é o uso do Python para trabalhar com dados com o SQLite, muitas operações podem ser feitas de forma mais conveniente utilizando um *plugin* do Firefox, o **SQLite Database Manager** que está disponível gratuitamente através do *link*:

`https://addons.mozilla.org/en-us/firefox/addon/sqlite-manager/`

Utilizando o navegador você pode facilmente criar tabelas, inserir, editar ou executar consultas SQL nos dados da base de dados.

De certa forma, o gerenciador de banco de dados é similar a um editor de texto quando utilizado arquivos de texto. Quando você quer fazer uma ou mais operações com um arquivo de texto, você pode simplesmente abrir o arquivo em um editor de texto e fazer as alterações que desejar. Quando você tem que fazer muitas alterações para fazer, normalmente você pode escrever um programa em Python simples para executar esta tarefa. Você encontrará os mesmos padrões quando for trabalhar com banco de dados. Você fará operações em um gerenciador de banco de dados e as operações mais complexas serão mais convenientes se forem feitas com Python.

2.4 Criado uma tabela em um banco de dados

Bancos de dados precisam de estruturas mais bem definidas do quê listas ou dicionários em Python[1].

Quando criamos uma **tabela** em um banco de dados, precisamos informar ao banco de dados previamente o nome de cada **coluna** na tabela e o tipo de dados que planejamos armazenar em cada **coluna**. Quando o sistema de banco de dados conhece o tipo de dado em cada coluna, ele pode definir a forma mais eficiente de armazenar e consultar o dado baseado no tipo do dado.

Você pode visualizar os diversos tipos de dados que são suportados pelo SQLite através do seguinte endereço:

`http://www.sqlite.org/datatypes.html`

Definir a estrutura dos seus tipos de dados pode parecer inconveniente no começo, mas a recompensa é o acesso rápido aos dados mesmo quando o banco de dados contém um grande número de informações.

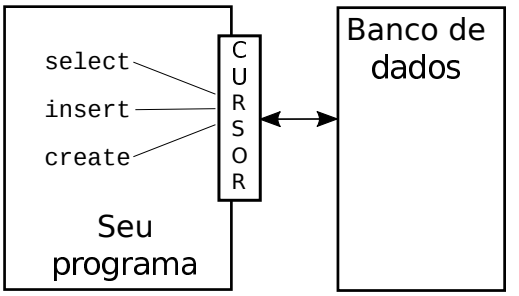
O seguinte código cria um arquivo de banco de dados com uma tabela, chamada `Tracks` e com duas colunas:

```
import sqlite3

conn = sqlite3.connect('music.sqlite3')
cur = conn.cursor()

cur.execute('DROP TABLE IF EXISTS Tracks ')
cur.execute('CREATE TABLE Tracks (title TEXT, plays INTEGER)')

conn.close()
```

A operação `connect` cria uma "conexão" com o banco de dados armazenado no arquivo `music.sqlite3` no diretório corrente. Se o arquivo não existir, este será criado. O motivo para isto ser chamado de "conexão" é que algumas vezes o banco de dados está em um "servidor de banco de dados" separado da aplicação propriamente dita. Em nossos exemplos o banco de dados está armazenado localmente em um arquivo no mesmo diretório que o código Python está sendo executado.

Um **cursos** é como um identificador de arquivo que podemos utilizar para realizar operações sobre as informações armazenadas em um banco de dados. Ao chamar a função `cursor()`, conceitualmente, é similar ao chamar a função `open()` quando estamos trabalhando com arquivos de texto.

[1] Atualmente o SQLite permite uma maior flexibilidade em relação aos tipos de dados que são armazenados em uma coluna, mas vamos manter os tipos de dados restritos neste capítulo, assim os mesmos conceitos aprendidos aqui podem ser aplicados a outros sistemas de banco de dados como MySQL.



Uma vez que temos o cursos, podemos começar a executar comandos no conteúdo armazenado no banco de dados utilizando o método `execute()`.

Os comandos de um banco de dados são expressos em uma linguagem especial que foi padronizada por diferentes fornecedores de bancos de dados, que nos permite aprender uma única linguagem. A linguagem dos bancos de dados é chamada de **Structured Query Language**[2] ou referenciada pelo acrônimo **SQL** `http://en.wikipedia.org/wiki/SQL`

Em nossos exemplos, estamos executando dois comandos SQL no banco de dados que criamos. Convencionaremos que os comandos SQL serão mostrados em maiúsculas e as partes que não são palavras reservadas do SQL (como os nomes das tabelas e colunas) serão mostrados em minúsculas.

O primeiro comando SQL remove a tabela `Tracks` do banco de dados se ela existir. Este padrão nos permite executar o mesmo programa para criar a tabela `Tracks` repetidas vezes sem que cause erro. Perceba que o comando `DROP TABLE` remove a tabela e todo o seu conteúdo do banco de dados (i.e., não é possível desfazer esta operação)

```
cur.execute('DROP TABLE IF EXISTS Tracks ')
```

O segundo comando cria a tabela `Tracks` com uma coluna chamada `title` com o tipo texto e uma coluna chamada `plays` com o tipo inteiro.

```
cur.execute('CREATE TABLE Tracks (title TEXT, plays INTEGER)')
```

Agora que criamos a tabela `Tracks`, podemos inserir algum dado dentro dela utilizando a operação SQL `INSERT`. Novamente, estamos estabelecendo uma conexão com o banco de dados e obtendo o `cursos`. E então executamos o comando SQL utilizando o cursor.

O comando SQL `INSERT` indica qual tabela estamos utilizando, e em seguida, cria uma nova linha listando quais campos utilizaremos para incluir `(title, plays)` seguido pelo comando `VALUES` com os valores que desejamos adicionar na nova linha. Especificamos os valores utilizando pontos de interrogação `(?, ?)` para indicar que os valores serão passados como tuplas `( 'My Way', 15)` como um segundo parâmetro da chamada `execute()`.

[2] Em Português, pode ser chamada de Linguagem de Consulta Estruturada

```
import sqlite3

conn = sqlite3.connect('music.sqlite3')
cur = conn.cursor()

cur.execute('INSERT INTO Tracks (title, plays) VALUES ( ?, ? )',
    ( 'Thunderstruck', 20 ) )
cur.execute('INSERT INTO Tracks (title, plays) VALUES ( ?, ? )',
    ( 'My Way', 15 ) )
conn.commit()

print 'Tracks:'
cur.execute('SELECT title, plays FROM Tracks')
for row in cur :
   print row

cur.execute('DELETE FROM Tracks WHERE plays < 100')
conn.commit()

cur.close()
```

Primeiro nós adicionamos com `INSERT` duas linha na nossa tabela e usamos `commit()` para forçar a escrita da informação no arquivo do banco de dados.

Tracks

title	plays
Thunderstruck	20
My Way	15

Depois usamos o comando `SELECT` para buscar a linha que acabamos de inserir na tabela. Com o comando `SELECT`, indicamos que coluna gostaríamos `(title, plays)` e de qual tabela queremos buscar a informação. Depois de confirmar a executação do comando `SELECT`, o cursor pode ser utilizado como repetição através de um comando `for`. Por questões de eficiência, o cursor não lê toda a informação da base de dados quando executamos o comando `SELECT`. Ao invés disto, a informação é lida sob demanda enquanto iteramos através da linha com o comando `for`.

A saída do programa fica da seguinte forma:

```
Tracks:
(u'Thunderstruck', 20)
(u'My Way', 15)
```

A iteração do `for` encontrou duas linhas, e cada linha é uma tupla em Python com o primeiro valor como `title` e o segundo como o número de `plays`. Não se preocupe com o fato de a *strings* são mostrados com o caractere `u'` no começo. Isto é uma indicação que a *string* estão em **Unicode**, o que indica que são capazes de armazenar um conjunto de caractere não-Latin.

No final do programa, executamos o comando SQL `DELETE` para remover as linhas que acabamos de criar, assim podemos executar o programa repetidas vezes. O `DELETE` pode ser utilizado com a condição `WHERE` que permite selecionar através de uma expressão o critério permitindo pesquisar no banco de dados somente as linhas que correspondem com a expressão utilizada. Neste exemplo a expressão construida se aplica em todas as linhas, para que possamos executar o programa outras vezes. Depois de executar o `DELETE` chamamos o `commit()` para forçar que o dado seja removido do banco de dados.

2.5 Resumo de Structured Query Language (SQL)

Estamos utilizando SQL junto com os exemplos de Python e até agora cobrimos muitos comandos SQL básicos. Nesta seção, vamos olhar a linguagem SQL com mais atenção e apresentaremos uma visão geral da sintaxe do SQL.

Existem diferentes fornecedores de bancos de dados, a linguagem SQL foi padronizada, desta forma podemos nos comunicar de maneira portável entre os diferentes sistemas de banco de dados dos diferentes fornecedores.

Basicamente um banco de dados relacional é composto por tabelas, linhas e colunas. As colunas geralmente possuem tipos, como textos, números ou informação de data. Quando criamos uma tabela, indicamos os nomes e tipos das colunas:

```
CREATE TABLE Tracks (title TEXT, plays INTEGER)
```

Para inserir uma linha em uma tabela, utilizamos o comando SQL `INSERT`:

```
INSERT INTO Tracks (title, plays) VALUES ('My Way', 15)
```

A declaração do `INSERT` especifica o nome da tabela, e então, uma lista dos campos/colunas que gostaríamos de definir na nova linha, e por fim, através do campo `VALUES` passamos uma lista de valores correspondentes a cada campo.

O comando `SELECT` é utilizado para buscar as linhas e colunas de um banco de dados. A declaração do `SELECT` permite que você especifique qual coluna gostaria de buscar, bem como utilizando a condição do `WHERE`, permite selecionar qual linha gostaríamos de visualizar. Isto também possibilita o uso de uma condição opcional, `ORDER BY`, para ordenar as linhas retornadas.

```
SELECT * FROM Tracks WHERE title = 'My Way'
```

O uso do `*` indica que o banco de dados deve retornar todas as colunas para cada linha que casa com a condição `WHERE`.

Atenção, diferente de Python, a condição `WHERE`, em SQL, utiliza o sinal de igual simples (=), para indicar uma condição de igualdade, ao invés de um sinal duplo (==) <, >, <=, >=, !=,

assim como é possível utilizar as condições `AND` e `OR` e parênteses para construir expressões lógicas.

Você pode pedir que as linhas retornadas sejam ordenadas por um dos campos como apresentados no exemplo a seguir:

```
SELECT title,plays FROM Tracks ORDER BY title
```

Para remover uma linha, é preciso combinar a condição `WHERE` com a condição `DELETE`. O `WHERE` irá determinar quais linhas serão removidas:

```
DELETE FROM Tracks WHERE title = 'My Way'
```

É possível alterar/atualizar uma ou mais colunas e suas linhas de uma tabela utilizando a condição SQL `UPDATE`, da seguinte forma:

```
UPDATE Tracks SET plays = 16 WHERE title = 'My Way'
```

A condição `UPDATE` especifica uma tabela e depois uma lista de campos e valores que serão alterados após o comando `SET`, e utilizando uma condição `WHERE`, opctional, é possível selecionar as linhas que serão atualizadas. Uma condição `UPDATE` irá mudar todas as linhas que casam com a condição `WHERE`. Se a condição `WHERE` não for especificada, o `UPDATE` será aplicado em todas as linhas da tabela.

Os quatros comandos básicos de SQL (INSERT, SELECT, UPDTE e DELETE) permitem as quatro operações básicas necessárias para criação e manutenção das informações em um banco de dados.

2.6 Rastreando o Twitter utilizando um banco de dados

Nesta seção criaremos um programa simples para rastreamento que navegará através de contas de usuários do Twitter e construirá um banco de dados destas referentes as estes usuários. *Nota: Tenha muito cuidado ao executar este programa. Você não irá querer extrair muitas informações ou executar o programa por muito tempo e acabar tendo sua conta do Twitter bloqueada.*

Um dos problemas, em qualquer tipo de programas de rastreamento, é que precisa ser capaz de ser interrompido e reiniciado muitas vezes e você não quer perder informações que você já tenha recuperado até agora. Não quer sempre reiniciar a recuperação dos dados desde o começo, então armazenamos as informações tão logo seja recuperada, assim o programa poderá reiniciar a busca do ponto onde parou.

Vamos começar recuperando os amigos de uma pessoa no Twitter e seus status, iterando na lista de amigos, e adicionando cada um ao banco de dados para que possa ser recuperado no futuro. Depois de listar os amigos de uma pessoa, verificamos na nossa base de dados e coletamos os amigos de um dos amigos da

primeira pessoa. Vamos fazendo isto repetidas vezes, escolhendo umas das pessoas "não visitadas", recuperando sua lista de amigos, e adicionando amigos que não tenhamos visto anteriormente a nossa lista, para visitar futuramente.

Também rastrearemos quantas vezes vimos um amigo em particular na nossa base para ter uma ideia da sua "popularidade".

Armazenando nossa lista de contas conhecidas, no banco de dados no disco do nosso computador, e se já recuperamos a conta ou não, e quanto esta conta é popular, podemos parar e recomeçar nosso programa quantas vezes quisermos.

Este programa é um pouco complexo. É baseado em um exercício apresentado anteriormente neste livro, que utiliza a API do Twitter

O seguinte código apresenta o programa que realiza o rastreamento no Twitter:

```
import urllib
import twurl
import json
import sqlite3

TWITTER_URL = 'https://api.twitter.com/1.1/friends/list.json'

conn = sqlite3.connect('spider.sqlite3')
cur = conn.cursor()

cur.execute('''
CREATE TABLE IF NOT EXISTS Twitter
(name TEXT, retrieved INTEGER, friends INTEGER)''')

while True:
    acct = raw_input('Enter a Twitter account, or quit: ')
    if ( acct == 'quit' ) : break
    if ( len(acct) < 1 ) :
        cur.execute('SELECT name FROM Twitter WHERE retrieved = 0 LIMIT 1')
        try:
            acct = cur.fetchone()[0]
        except:
            print 'No unretrieved Twitter accounts found'
            continue

    url = twurl.augment(TWITTER_URL,
                {'screen_name': acct, 'count': '20'} )
    print 'Retrieving', url
    connection = urllib.urlopen(url)
    data = connection.read()
    headers = connection.info().dict
    # print 'Remaining', headers['x-rate-limit-remaining']
    js = json.loads(data)
    # print json.dumps(js, indent=4)

    cur.execute('UPDATE Twitter SET retrieved=1 WHERE name = ?', (acct, ) )

    countnew = 0
```

```
    countold = 0
    for u in js['users'] :
        friend = u['screen_name']
        print friend
        cur.execute('SELECT friends FROM Twitter WHERE name = ? LIMIT 1',
            (friend, ) )
        try:
            count = cur.fetchone()[0]
            cur.execute('UPDATE Twitter SET friends = ? WHERE name = ?',
                (count+1, friend) )
            countold = countold + 1
        except:
            cur.execute('''INSERT INTO Twitter (name, retrieved, friends)
                VALUES ( ?, 0, 1 )''', ( friend, ) )
            countnew = countnew + 1
    print 'New accounts=',countnew,' revisited=',countold
    conn.commit()

cur.close()
```

Nossa base de dados está armazenada no arquivo `spider.sqlite3` e possui uma tabela chamada `Twitter`. Cada linha na tabela `Twitter` tem uma coluna para o nome da conta, se recuperamos os amigos desta conta, e quantas vezes esta conta foi "seguido".

Na repetição principal do programa, perguntamos ao usuário uma conta de Twitter ou "quit" para sair do programa. Se o usuário informar um usuário do Twitter, o programa começa a recuperar a lista de amigos e os status para aquele usuário e adiciona cada amigo na base de dados se não possuir. Se o amigo já está na lista, nós adicionamos "1" no campo `friends` da base de dados.

Se o usuário pressionar `enter`, pesquisamos na base a próxima conta que não rastreamos ainda, e então rastreamos os amigos e status com aquela conta e adicionamos na base de dados ou atualizamos, incrementando seu contador de `friends`.

Uma vez que rastreamos a lista de amigos e status, iteramos entre todas os ítens `user` retornados no JSON e rastreamos o `screen_name` para cada usuário. Então utilizamos a declaração `SELECT` para ver se já armazenamos este `screen_name` em particular na base e recuperamos o contador de amigos (`friends`), se este registro existir.

```
    countnew = 0
    countold = 0
    for u in js['users'] :
        friend = u['screen_name']
        print friend
        cur.execute('SELECT friends FROM Twitter WHERE name = ? LIMIT 1',
            (friend, ) )
        try:
            count = cur.fetchone()[0]
            cur.execute('UPDATE Twitter SET friends = ? WHERE name = ?',
                (count+1, friend) )
            countold = countold + 1
```

```
        except:
            cur.execute('''INSERT INTO Twitter (name, retrieved, friends)
                VALUES ( ?, 0, 1 )''', ( friend, ) )
            countnew = countnew + 1
    print 'New accounts=',countnew,' revisited=',countold
    conn.commit()
```

Uma vez que o cursor tenha executado o `SELECT`, nós devemos recuperar as linhas. Podemos fazer isto com uma declaração de `for`, mas uma vez que estamos recuperando uma linha (`LIMIT 1`), podemos utilizar o método `fetchone()` para buscar a primeira (e única) linha que é o resultado da operação `SELECT`. Sendo o retorno `fetchone()` uma linha como uma **tupla** (ainda que haja somente um campo), pegamos o primeiro valor da tupla utilizando índice `[0]` para pegar o contador de amigos atual dentro da variável `count`.

Se a busca for bem sucedida, utilizamos a declação `UPDATE` com a clausula `WHERE` para adicionar 1 na coluna `friends` para a linha que corresponde com a conta do amigo. Note que existem dois espaços reservados (i.e., pontos de interrogações) no SQL, e o segundo parâmetro para o `execute()` é uma tupla que armazena o valor para substituir no SQL no lugar dos pontos de interrogações.

Se o bloco `try` falhar, é provavelmente por que nenhum resultado corresponde a clausula em `WHERE name = ?` do SELECT. Então no block `except`, utilizamos a declaração `INSERT` para adicionar o `screen_name` do amigo a tabela com a indicação que ainda não rastreamos o `screen_name` e setamos o contador de amigos com 0 (zero).

Assim, a primeira vez que o programa é executado e informamos uma conta do Twitter, a saída do programa é a seguinte:

```
Enter a Twitter account, or quit: drchuck
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 20  revisited= 0
Enter a Twitter account, or quit: quit
```

Como esta é a primeira vez que executamos o programa, o banco de dados está vazio e criamos o banco no arquivo `spider.sqlite3`, adicionamos a tabela chamada `Twitter` na base de dados. Então nós rastreamos alguns amigos e os adicionamos a base, uma vez que ela está vazia.

Neste ponto podemos escrever um *dumper* simples para olhar o que está no nosso arquivo `spider.sqlite3`:

```
import sqlite3

conn = sqlite3.connect('spider.sqlite3')
cur = conn.cursor()
cur.execute('SELECT * FROM Twitter')
count = 0
for row in cur :
   print row
```

```
   count = count + 1
print count, 'rows.'
cur.close()
```

Este programa abre o banco de dados e seleciona todas as colunas de todas as linhas na tabela `Twitter`, depois itera em cada linha e imprime o valor dentro de cada uma.

Se executarmos este programa depois da primeira execução do nosso rastreador *spider* do Twitter, sua saída será como a seguinte:

```
(u'opencontent', 0, 1)
(u'lhawthorn', 0, 1)
(u'steve_coppin', 0, 1)
(u'davidkocher', 0, 1)
(u'hrheingold', 0, 1)
...
20 rows.
```

Veremos uma linha para cada `screen_name`, que não tenhamos recuperado o dado daquele `screen_name`, e todos tem um amigo.

Agora nosso banco de dados reflete quais amigos estão relacionados com a nossa primeira conta do Twitter (**drchuck**) utilizada para rastreamento. Podemos executar o programa novamente e mandar rastrear a próxima conta "não processada" e recuperar os amigos, simplemente pressionando `enter` ao invés de informar uma conta do Twitter, conforme o exemplo a seguir:

```
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 18  revisited= 2
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 17  revisited= 3
Enter a Twitter account, or quit: quit
```

Uma vez que pressionamos `enter` (i.e., não especificamos uma conta do Twitter), o seguinte código é executado:

```
    if ( len(acct) < 1 ) :
        cur.execute('SELECT name FROM Twitter WHERE retrieved = 0 LIMIT 1')
        try:
            acct = cur.fetchone()[0]
        except:
            print 'No unretrieved twitter accounts found'
            continue
```

Utilizamos a declaração SQL `SELECT` para recuperar o nome do primeiro (`LIMIT 1`) usuário que ainda tem seu "recuperamos este usuário" com o valor setado em zero. Também utilizamos o padrão `fetchone()[0]` dentro de um bloco try/except para extrair também um `screen_name` do dado recuperado ou apresentamos uma mensagem de erro e iteramos novamente.

Se tivermos sucesso ao recuperar um `screen_name` não processado, vamos extrair seus dados da seguinte maneira:

```
url = twurl.augment(TWITTER_URL, {'screen_name': acct, 'count': '20'} )
print 'Retrieving', url
connection = urllib.urlopen(url)
data = connection.read()
js = json.loads(data)

cur.execute('UPDATE Twitter SET retrieved=1 WHERE name = ?', (acct, ) )
```

Ao recuperar os dados com sucesso, utilizaremos a declaração `UPDATE` para setar a coluna `retrieved` para 1 para indicar que completamos a extração dos amigos relacionados com esta conta. Isto no permite recuperar o mesmo dado diversas vezes e nos permite prosseguir através da lista de amigos no Twitter.

Se executarmos o programa novamente, e pressionarmos `enter` duas vezes seguidas para recuperar os próximos amigos do amigo e depois executarmos o programa de *dumping*, ele nos mostrará a seguinte saída:

```
(u'opencontent', 1, 1)
(u'lhawthorn', 1, 1)
(u'steve_coppin', 0, 1)
(u'davidkocher', 0, 1)
(u'hrheingold', 0, 1)
...
(u'cnxorg', 0, 2)
(u'knoop', 0, 1)
(u'kthanos', 0, 2)
(u'LectureTools', 0, 1)
...
55 rows.
```

Podemos ver que gravamos de forma apropriada que visitamos os usuários `lhawthorn` e `opencontent`. E que as contas `cnxorg` e `kthanos` já tem dois seguidores. Desde que tenhamos recuperado os amigos de três pessoas (`drchuck`, `opencontent`, e `lhawthorn`) nossa tabela tem agora 55 linhas de amigos recuperados.

Cada vez que executamos o programa e pressionamos `enter` ele pegará a próxima conta não visitada (e.g., a próxima conta será `steve_coppin`), recuperar seus amigos, marcá-los como recuperados, e para cada um dos amigos de `steve_coppin` também adicionaremos eles para no fim da base de dados e atualizaremos seus amigos que já estiverem na base de dados.

Assim que os dados do programa estejam armazenados no disco em um banco de dados o rastreamento pode ser suspenso e reiniciado tantas vezes quando quiser sem a perda de informações.

2.7 Modelagem de dados básica

O verdadeiro poder de um banco de dados relacional é quando criamos múltiplas tabelas e criamos ligações entre elas. Decidir como dividir os dados da sua aplicação em diferentes tabelas e estabelecer a relação entre estas tabelas é o que chamamos de **modelagem de dados**. O documento que mostra a estrutura das tabelas e suas relações é chamado de **modelo de dados**.

Modelagem de dados é uma habilidade relativamente sofisticada e nesta seção nós iremos somente introduzir os conceitos mais básicos da modelagem de dados relacionais. Para maiores detalhes sobre modelagem de dados você pode começar com:

`http://en.wikipedia.org/wiki/Relational_model`

Digamos que para a nossa aplicação de rastreamento do Twitter, ao invés de só contar os amigos das pessoas, nós queiramos manter uma lista de todas as relações de entrada, então poderemos encontrar uma lista de todos que seguem uma pessoa em particular.

Já que todos, potencialmente, terão tantas contas que o sigam, nós não podemos simplesmente adicionar uma coluna para nossa tabela `Twitter`. Então criamos uma nova tabela que mantém o controle dos pares de amigos. A seguir temos uma forma simples de criar tal tabela:

```
CREATE TABLE Pals (from_friend TEXT, to_friend TEXT)
```

Toda vez que encontrarmos uma pessoa que `drchuck` está seguindo, nós iremos inserir uma linha da seguinte forma:

```
INSERT INTO Pals (from_friend,to_friend) VALUES ('drchuck', 'lhawthorn')
```

Como estamos processando 20 amigos da conta do *Twitter* do `drchuck`, inseriremor 20 registros com "drchuck" como primeiro parâmetro e assim acaberemos duplicando a *string* muitas vezes no banco de dados.

Esta duplicação de dados viola uma das melhores práticas da **normatização de banco de dados** que basicamente afirma que nunca devemos colocar o mesmo dado mais de uma vez em um banco de dados. Se precisarmos inserir um dado mais de uma vez, criamos uma referência numérica **key** (chave) para o dado, e utilizamos a chave para referenciar o dado.

Na prática, uma *string* ocupa muito mais espaço do que um inteiro, no disco e na memória do nosso computador, e leva mais tempo do processor para comparar e ordenar. Se tivermos somente algumas centenas de entradas, a base de dados e o tempo de processamento dificilmente importarão. Mas se tivermos um milhão de pessoas na nossa base de dados e uma possibilidade de 100 milhões de conexões de amigos, é importante permitir examinar os dados o mais rápido possível.

Nós armazenaremos nossas contas do Twitter em uma tabela chamada `People` ao invés de utilizar a tabela `Twitter` utilizada no exemplo anterior. A tabela `People`

tem uma coluna adicional para armazenar uma chave associada a linha para este usuário.

Podemos criar a tabela `People` com esta coluna `id` adicional com o seguinte comando:

```
CREATE TABLE People
    (id INTEGER PRIMARY KEY, name TEXT UNIQUE, retrieved INTEGER)
```

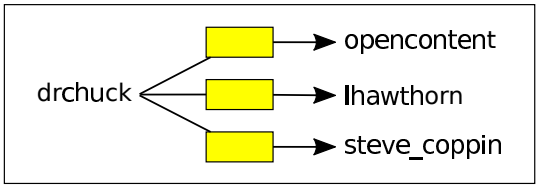
Perceba que nós não estamos mais mantendo uma conta de amigo em cada linha da tabela `People`. Quando selecionamos `INTEGER PRIMARY KEY` como o tipo da nossa coluna `id`, estamos indicando que gostaríamos que o SQLite gerencie esta coluna e defina uma chave numérica única automagicamente para cada linha que inserirmos. Também adicionamos uma palavra-chave `UNIQUE` para indicar que não permitiremos ao SQLite inserir duas linhas com o mesmo valor para `name`.

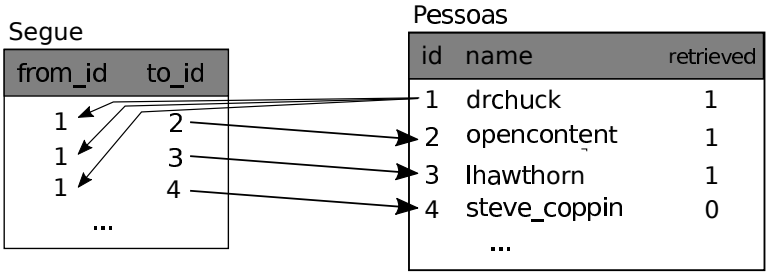
Agora, ao invés de criar a tabela `Pals` acima, criaremos uma tabela chamada `Follows` com duas colunas com o tipo inteiro `from_id` e `to_id` e associaremos na tabela onde a *combinação* de `from_id` e `to_id` devem ser únicos nesta tabela (i.e., não podemos inserir linhas duplicadas) na nossa base de dados.

```
CREATE TABLE Follows
    (from_id INTEGER, to_id INTEGER, UNIQUE(from_id, to_id) )
```

Quando adicionamos a condição `UNIQUE` a nossa tabela, estamos definindo um conjunto de regras e pedindo a base de dados para cumprir estas regras quando tentarmos inserir algum registro. Estamos criando estas regras como uma conveniencia no nosso programa, como veremos a seguir. As regras nos impede de cometer enganos e facilita na escrita dos nossos códigos.

Em essencia, criando a tabela `Follows`, estamos modelando uma "relação" onde uma pessoa "segue" outro alguém e representamos isto com um par de números indicando que (a) as pessoas estão conectadas e (b) a direção do relacionamento.





2.8 Programming with multiple tables

We will now redo the Twitter spider program using two tables, the primary keys, and the key references as described above. Here is the code for the new version of the program:

```
import urllib
import twurl
import json
import sqlite3

TWITTER_URL = 'https://api.twitter.com/1.1/friends/list.json'

conn = sqlite3.connect('friends.sqlitesqlite3')
cur = conn.cursor()

cur.execute('''CREATE TABLE IF NOT EXISTS People
    (id INTEGER PRIMARY KEY, name TEXT UNIQUE, retrieved INTEGER)''')
cur.execute('''CREATE TABLE IF NOT EXISTS Follows
    (from_id INTEGER, to_id INTEGER, UNIQUE(from_id, to_id))''')

while True:
    acct = raw_input('Enter a Twitter account, or quit: ')
    if ( acct == 'quit' ) : break
    if ( len(acct) < 1 ) :
        cur.execute('''SELECT id, name FROM People
            WHERE retrieved = 0 LIMIT 1''')
        try:
            (id, acct) = cur.fetchone()
        except:
            print 'No unretrieved Twitter accounts found'
            continue
    else:
        cur.execute('SELECT id FROM People WHERE name = ? LIMIT 1',
            (acct, ) )
        try:
            id = cur.fetchone()[0]
        except:
            cur.execute('''INSERT OR IGNORE INTO People (name, retrieved)
                VALUES ( ?, 0)''', ( acct, ) )
            conn.commit()
            if cur.rowcount != 1 :
                print 'Error inserting account:',acct
                continue
            id = cur.lastrowid

    url = twurl.augment(TWITTER_URL,
       {'screen_name': acct, 'count': '20'} )
    print 'Retrieving account', acct
    connection = urllib.urlopen(url)
    data = connection.read()
    headers = connection.info().dict
    print 'Remaining', headers['x-rate-limit-remaining']

    js = json.loads(data)
```

```
    # print json.dumps(js, indent=4)

    cur.execute('UPDATE People SET retrieved=1 WHERE name = ?', (acct, ) )

    countnew = 0
    countold = 0
    for u in js['users'] :
        friend = u['screen_name']
        print friend
        cur.execute('SELECT id FROM People WHERE name = ? LIMIT 1',
            (friend, ) )
        try:
            friend_id = cur.fetchone()[0]
            countold = countold + 1
        except:
            cur.execute('''INSERT OR IGNORE INTO People (name, retrieved)
                VALUES ( ?, 0)''', ( friend, ) )
            conn.commit()
            if cur.rowcount != 1 :
                print 'Error inserting account:',friend
                continue
            friend_id = cur.lastrowid
            countnew = countnew + 1
        cur.execute('''INSERT OR IGNORE INTO Follows (from_id, to_id)
            VALUES (?, ?)''', (id, friend_id) )
    print 'New accounts=',countnew,' revisited=',countold
    conn.commit()

cur.close()
```

This program is starting to get a bit complicated, but it illustrates the patterns that we need to use when we are using integer keys to link tables. The basic patterns are:

1. Create tables with primary keys and constraints.
2. When we have a logical key for a person (i.e., account name) and we need the `id` value for the person, depending on whether or not the person is already in the `People` table we either need to: (1) look up the person in the `People` table and retrieve the `id` value for the person or (2) add the person to the `People` table and get the `id` value for the newly added row.
3. Insert the row that captures the "follows" relationship.

We will cover each of these in turn.

2.8.1 Constraints in database tables

As we design our table structures, we can tell the database system that we would like it to enforce a few rules on us. These rules help us from making mistakes and introducing incorrect data into out tables. When we create our tables:

```
cur.execute('''CREATE TABLE IF NOT EXISTS People
    (id INTEGER PRIMARY KEY, name TEXT UNIQUE, retrieved INTEGER)''')
cur.execute('''CREATE TABLE IF NOT EXISTS Follows
    (from_id INTEGER, to_id INTEGER, UNIQUE(from_id, to_id))''')
```

We indicate that the `name` column in the `People` table must be `UNIQUE`. We also indicate that the combination of the two numbers in each row of the `Follows` table must be unique. These constraints keep us from making mistakes such as adding the same relationship more than once.

We can take advantage of these constraints in the following code:

```
cur.execute('''INSERT OR IGNORE INTO People (name, retrieved)
    VALUES ( ?, 0)''', ( friend, ) )
```

We add the `OR IGNORE` clause to our `INSERT` statement to indicate that if this particular `INSERT` would cause a violation of the "`name` must be unique" rule, the database system is allowed to ignore the `INSERT`. We are using the database constraint as a safety net to make sure we don't inadvertently do something incorrect.

Similarly, the following code ensures that we don't add the exact same `Follows` relationship twice.

```
cur.execute('''INSERT OR IGNORE INTO Follows
    (from_id, to_id) VALUES (?, ?)''', (id, friend_id) )
```

Again, we simply tell the database to ignore our attempted `INSERT` if it would violate the uniqueness constraint that we specified for the `Follows` rows.

2.8.2 Retrieve and/or insert a record

When we prompt the user for a Twitter account, if the account exists, we must look up its `id` value. If the account does not yet exist in the `People` table, we must insert the record and get the `id` value from the inserted row.

This is a very common pattern and is done twice in the program above. This code shows how we look up the `id` for a friend's account when we have extracted a `screen_name` from a `user` node in the retrieved Twitter JSON.

Since over time it will be increasingly likely that the account will already be in the database, we first check to see if the `People` record exists using a `SELECT` statement.

If all goes well[3] inside the `try` section, we retrieve the record using `fetchone()` and then retrieve the first (and only) element of the returned tuple and store it in `friend_id`.

If the `SELECT` fails, the `fetchone()[0]` code will fail and control will transfer into the `except` section.

[3] In general, when a sentence starts with "if all goes well" you will find that the code needs to use try/except.

```
        friend = u['screen_name']
        cur.execute('SELECT id FROM People WHERE name = ? LIMIT 1',
            (friend, ) )
        try:
            friend_id = cur.fetchone()[0]
            countold = countold + 1
        except:
            cur.execute('''INSERT OR IGNORE INTO People (name, retrieved)
                VALUES ( ?, 0)''', ( friend, ) )
            conn.commit()
            if cur.rowcount != 1 :
                print 'Error inserting account:',friend
                continue
            friend_id = cur.lastrowid
            countnew = countnew + 1
```

If we end up in the `except` code, it simply means that the row was not found, so we must insert the row. We use `INSERT OR IGNORE` just to avoid errors and then call `commit()` to force the database to really be updated. After the write is done, we can check the `cur.rowcount` to see how many rows were affected. Since we are attempting to insert a single row, if the number of affected rows is something other than 1, it is an error.

If the `INSERT` is successful, we can look at `cur.lastrowid` to find out what value the database assigned to the `id` column in our newly created row.

2.8.3 Storing the friend relationship

Once we know the key value for both the Twitter user and the friend in the JSON, it is a simple matter to insert the two numbers into the `Follows` table with the following code:

```
cur.execute('INSERT OR IGNORE INTO Follows (from_id, to_id) VALUES (?, ?)',
    (id, friend_id) )
```

Notice that we let the database take care of keeping us from "double-inserting" a relationship by creating the table with a uniqueness constraint and then adding `OR IGNORE` to our `INSERT` statement.

Here is a sample execution of this program:

```
Enter a Twitter account, or quit:
No unretrieved Twitter accounts found
Enter a Twitter account, or quit: drchuck
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 20  revisited= 0
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 17  revisited= 3
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 17  revisited= 3
Enter a Twitter account, or quit: quit
```

We started with the drchuck account and then let the program automatically pick the next two accounts to retrieve and add to our database.

The following is the first few rows in the People and Follows tables after this run is completed:

```
People:
(1, u'drchuck', 1)
(2, u'opencontent', 1)
(3, u'lhawthorn', 1)
(4, u'steve_coppin', 0)
(5, u'davidkocher', 0)
55 rows.
Follows:
(1, 2)
(1, 3)
(1, 4)
(1, 5)
(1, 6)
60 rows.
```

You can see the id, name, and visited fields in the People table and you see the numbers of both ends of the relationship in the Follows table. In the People table, we can see that the first three people have been visited and their data has been retrieved. The data in the Follows table indicates that drchuck (user 1) is a friend to all of the people shown in the first five rows. This makes sense because the first data we retrieved and stored was the Twitter friends of drchuck. If you were to print more rows from the Follows table, you would see the friends of users 2 and 3 as well.

2.9 Three kinds of keys

Now that we have started building a data model putting our data into multiple linked tables and linking the rows in those tables using **keys**, we need to look at some terminology around keys. There are generally three kinds of keys used in a database model.

- A **logical key** is a key that the "real world" might use to look up a row. In our example data model, the name field is a logical key. It is the screen name for the user and we indeed look up a user's row several times in the program using the name field. You will often find that it makes sense to add a UNIQUE constraint to a logical key. Since the logical key is how we look up a row from the outside world, it makes little sense to allow multiple rows with the same value in the table.

- A **primary key** is usually a number that is assigned automatically by the database. It generally has no meaning outside the program and is only used to link rows from different tables together. When we want to look up a row

in a table, usually searching for the row using the primary key is the fastest way to find the row. Since primary keys are integer numbers, they take up very little storage and can be compared or sorted very quickly. In our data model, the `id` field is an example of a primary key.

- A **foreign key** is usually a number that points to the primary key of an associated row in a different table. An example of a foreign key in our data model is the `from_id`.

We are using a naming convention of always calling the primary key field name `id` and appending the suffix `_id` to any field name that is a foreign key.

2.10 Using JOIN to retrieve data

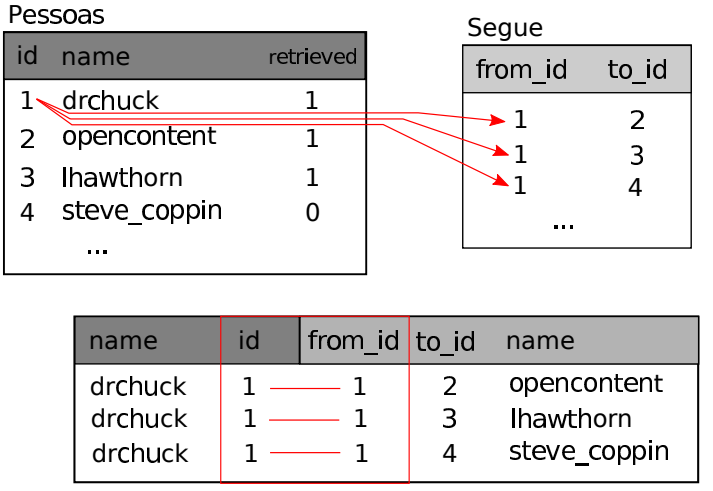
Now that we have followed the rules of database normalization and have data separated into two tables, linked together using primary and foreign keys, we need to be able to build a `SELECT` that reassembles the data across the tables.

SQL uses the `JOIN` clause to reconnect these tables. In the `JOIN` clause you specify the fields that are used to reconnect the rows between the tables.

The following is an example of a `SELECT` with a `JOIN` clause:

```
SELECT * FROM Follows JOIN People
    ON Follows.from_id = People.id WHERE People.id = 1
```

The `JOIN` clause indicates that the fields we are selecting cross both the `Follows` and `People` tables. The `ON` clause indicates how the two tables are to be joined: Take the rows from `Follows` and append the row from `People` where the field `from_id` in `Follows` is the same the `id` value in the `People` table.



The result of the JOIN is to create extra-long "metarows" which have both the fields from `People` and the matching fields from `Follows`. Where there is more

than one match between the `id` field from `People` and the `from_id` from `People`, then JOIN creates a metarow for *each* of the matching pairs of rows, duplicating data as needed.

The following code demonstrates the data that we will have in the database after the multi-table Twitter spider program (above) has been run several times.

```
import sqlite3

conn = sqlite3.connect('spider.sqlite3')
cur = conn.cursor()

cur.execute('SELECT * FROM People')
count = 0
print 'People:'
for row in cur :
   if count < 5: print row
   count = count + 1
print count, 'rows.'

cur.execute('SELECT * FROM Follows')
count = 0
print 'Follows:'
for row in cur :
   if count < 5: print row
   count = count + 1
print count, 'rows.'

cur.execute('''SELECT * FROM Follows JOIN People
    ON Follows.to_id = People.id WHERE Follows.from_id = 2''')
count = 0
print 'Connections for id=2:'
for row in cur :
   if count < 5: print row
   count = count + 1
print count, 'rows.'

cur.close()
```

In this program, we first dump out the `People` and `Follows` and then dump out a subset of the data in the tables joined together.

Here is the output of the program:

```
python twjoin.py
People:
(1, u'drchuck', 1)
(2, u'opencontent', 1)
(3, u'lhawthorn', 1)
(4, u'steve_coppin', 0)
(5, u'davidkocher', 0)
55 rows.
Follows:
(1, 2)
(1, 3)
```

```
(1, 4)
(1, 5)
(1, 6)
60 rows.
Connections for id=2:
(2, 1, 1, u'drchuck', 1)
(2, 28, 28, u'cnxorg', 0)
(2, 30, 30, u'kthanos', 0)
(2, 102, 102, u'SomethingGirl', 0)
(2, 103, 103, u'ja_Pac', 0)
20 rows.
```

You see the columns from the `People` and `Follows` tables and the last set of rows is the result of the `SELECT` with the `JOIN` clause.

In the last select, we are looking for accounts that are friends of "opencontent" (i.e., `People.id=2`).

In each of the "metarows" in the last select, the first two columns are from the `Follows` table followed by columns three through five from the `People` table. You can also see that the second column (`Follows.to_id`) matches the third column (`People.id`) in each of the joined-up "metarows".

2.11 Summary

This chapter has covered a lot of ground to give you an overview of the basics of using a database in Python. It is more complicated to write the code to use a database to store data than Python dictionaries or flat files so there is little reason to use a database unless your application truly needs the capabilities of a database. The situations where a database can be quite useful are: (1) when your application needs to make small many random updates within a large data set, (2) when your data is so large it cannot fit in a dictionary and you need to look up information repeatedly, or (3) when you have a long-running process that you want to be able to stop and restart and retain the data from one run to the next.

You can build a simple database with a single table to suit many application needs, but most problems will require several tables and links/relationships between rows in different tables. When you start making links between tables, it is important to do some thoughtful design and follow the rules of database normalization to make the best use of the database's capabilities. Since the primary motivation for using a database is that you have a large amount of data to deal with, it is important to model your data efficiently so your programs run as fast as possible.

2.12 Debugging

One common pattern when you are developing a Python program to connect to an SQLite database will be to run a Python program and check the results using the

SQLite Database Browser. The browser allows you to quickly check to see if your program is working properly.

You must be careful because SQLite takes care to keep two programs from changing the same data at the same time. For example, if you open a database in the browser and make a change to the database and have not yet pressed the "save" button in the browser, the browser "locks" the database file and keeps any other program from accessing the file. In particular, your Python program will not be able to access the file if it is locked.

So a solution is to make sure to either close the database browser or use the **File** menu to close the database in the browser before you attempt to access the database from Python to avoid the problem of your Python code failing because the database is locked.

2.13 Glossary

attribute: One of the values within a tuple. More commonly called a "column" or "field".

constraint: When we tell the database to enforce a rule on a field or a row in a table. A common constraint is to insist that there can be no duplicate values in a particular field (i.e., all the values must be unique).

cursor: A cursor allows you to execute SQL commands in a database and retrieve data from the database. A cursor is similar to a socket or file handle for network connections and files, respectively.

database browser: A piece of software that allows you to directly connect to a database and manipulate the database directly without writing a program.

foreign key: A numeric key that points to the primary key of a row in another table. Foreign keys establish relationships between rows stored in different tables.

index: Additional data that the database software maintains as rows and inserts into a table to make lookups very fast.

logical key: A key that the "outside world" uses to look up a particular row. For example in a table of user accounts, a person's email address might be a good candidate as the logical key for the user's data.

normalization: Designing a data model so that no data is replicated. We store each item of data at one place in the database and reference it elsewhere using a foreign key.

primary key: A numeric key assigned to each row that is used to refer to one row in a table from another table. Often the database is configured to automatically assign primary keys as rows are inserted.

relation: An area within a database that contains tuples and attributes. More typically called a “table”.

tuple: A single entry in a database table that is a set of attributes. More typically called “row”.

Capítulo 3

Automação de tarefas comuns no seu computador

Temos lido dados de arquivos, redes, serviços e banco de dados. Python também pode navegar através de todas as pastas e diretórios no seu computador e ler os arquivos também.

Neste capítulo, nós iremos escrever programas que analisam por através do seu computador e executam alguma operações em cada arquivo. Arquivos são organizados nos diretórios (também chamado de pastas).

Scripts Python simples podem fazer o trabalho de tarefas simples que deve ser feito para centenas ou milhares de arquivos espalhados por uma árvore de diretórios ou todo o seu computador Para navegar através de todos os diretórios e arquivos em uma árvore nós utilizamos `os.walk` e um laço de repetição `for`. Isto é similar em como `open` nos permite escrever um laço de repetição para ler o contúdo de um arquivo, `socket` nos permiteescrever um laço de repetição para ler o conteúdo de umaconexão e `urllib` nospermite abrir um documento web e navegar por meio de um laço de repetição no seu conteúdo.

3.1 Nomes e caminhos de arquivos

Todo programa em execução tem um "diretório atual"que é o diretório padrão para a maioria das operações. Por exemplo, quando você abre um arquivo para ler, Python procura por ele no diretório atual.

O módulo `os` disponibiliza funções para trabalhar com arquivos e diretórios (`os` do inglês "operating system"que significa sistema operacional). `os.getcwd` retorna o nome do diretório atual:

```
>>> import os
>>> cwd = os.getcwd()
```

```
>>> print cwd
/Users/csev
```

`cwd` significa **diretório atual de trabalho**. O resultado neste exemplo é `/Users/csev`, que é o diretório home do usuário chamado `csev`.

Uma sequência de caracteres como `cwd` que identifica um arquivo é chamado de path. Um **relative path** inicia do diretório atual; Um **absolute path** inicia do diretório raiz no sistema de arquivo.

Os caminhos que temos visto até agora são nomes de arquivos simples, por isso são relativo ao diretório atual. Para encontrar o caminho absoluto para um arquivo, você pode usar `os.path.abspath`:

```
>>> os.path.abspath('memo.txt')
'/Users/csev/memo.txt'
```

`os.path.exists` verifica se um determinado arquivo existe:

```
>>> os.path.exists('memo.txt')
True
```

Se existir, `os.path.isdir` verifica se é um diretório:

```
>>> os.path.isdir('memo.txt')
False
>>> os.path.isdir('music')
True
```

Similar, `os.path.isfile` verifica se é um arquivo.

`os.listdir` retorna uma lista com os arquivos (e outros diretórios) do diretório dado:

```
>>> os.listdir(cwd)
['music', 'photos', 'memo.txt']
```

3.2 Exemplo: Limpando um diretório de foto

Há algum tempo atrás, desenvolvi um pequeno software tipo Flickr que recebe fotos do meu celular e armazena essas fotos no meu servidor. E escrevi isto antes do Flickr existir e continuo usando por que eu quero manter copias das minhas imagens originais para sempre.

Eu também gostaria de enviar uma simples descrição numa mensagem MMS ou como um título de uma mensagem de email. Eu armazenei essas mensagens em uma arquivo de texto no mesmo diretório do arquivo da imagem. Eu vim com uma estrutura de diretorios baseada no mês, ano, dia e hora que a foto foi tirada, abaixo um exemplo de nomenclatura para uma foto e sua descrição:

```
./2006/03/24-03-06_2018002.jpg
./2006/03/24-03-06_2018002.txt
```

Após sete anos, eu tenho muitas fotos e legendas. Ao longo dos anos como eu troquei de celular, algumas vezes, meu código para extrair a legenda para uma menssagem quebrou e adicionou um bando de dados inuteis no meu servidor ao invés de legenda.

Eu queria passar por estes arquivos e descobrir qual dos arquivos de texto foram realmente legendas e quais eram lixo e, em seguida, apagar os arquivos que são lixo. A primeira coisa a fazer era conseguir um simples inventário dos arquivos de texto que eu tinha em uma das subpastas usando o seguinte programa:

```
import os
count = 0
for (dirname, dirs, files) in os.walk('.'):
   for filename in files:
       if filename.endswith('.txt') :
           count = count + 1
print 'Files:', count

python txtcount.py
Files: 1917
```

A chave que torna possivel pouco codigo é a biblioteca `os.walk` em Python. Quando nós chamamos `os.walk` e inicializamos uma diretório, ele ”anda”através de todos os diretórios e subdiretórios recursivamente. O caractere . indica para iniciar no diretório corrente e navegar para baixo.

Assim que encontra cada diretório, temos três valores em uma tupla no corpo do laço de repetição `for`. O primeiro valor é o diretório corrente o segundo é a lista de sub-diretórios e o treceito valor é a lista de arquivos no diretório corrente.

Nós não temos que procutar explicitamente dentro de cada diretório por que nó podemos contar com `os.walk` para visitar eventualmente todas as pastas mas, nós queremos procurar em cada arquivo, então, escrevemos um simples laço de repetição `for` para examinar cada um dos arquivos no diretório corrente. Vamos verificar se cada arquivo termina com ”.txt”e depois contar o número de arquivos através de toda a árvore de diretórios que terminam com o sufixo ”.txt”.

Uma vez que nós temos uma noção da quantidade de arquivos terminados com ”.txt”, a proxima coisa a fazer é tentar determinar automaticamente no Python quais arquivos são maus e quais são bons. Para isto, escreveremos um programa simples para imprimir os arquivos e seus tamanhos.

```
import os
from os.path import join
for (dirname, dirs, files) in os.walk('.'):
   for filename in files:
       if filename.endswith('.txt') :
           thefile = os.path.join(dirname,filename)
           print os.path.getsize(thefile), thefile
```

Agora, em vez de apenas contar os arquivos, criamos um nome de arquivo concatenando o nome do diretório com o nome do arquivo dentro do diretório usando `os.path.join`.

É importante usar o `os.path.join` para concatenar a sequência de caracteres por que no Windows usamos a barra invertida para construir os caminhos de arquivos e no Linux ou Apple nós usamos a barra (`/`) para construir o caminho do arquivo.

O `os.path.join` conhece essas diferenças e sabe qual sistema esta rodando dessa forma, faz a concatenação mais adequada considerando o sistema. Então, o mesmo código em Python roda tando no Windows quanto em sistemas tipo Unix.

Uma vez que temos o nome completo do arquivo com o caminho do diretório, nós usamos o utilitário `os.path.getsize` para pegar e imprimir o tamanho, produzindo a seguinte saída.

```
python txtsize.py
...
18 ./2006/03/24-03-06_2303002.txt
22 ./2006/03/25-03-06_1340001.txt
22 ./2006/03/25-03-06_2034001.txt
...
2565 ./2005/09/28-09-05_1043004.txt
2565 ./2005/09/28-09-05_1141002.txt
...
2578 ./2006/03/27-03-06_1618001.txt
2578 ./2006/03/28-03-06_2109001.txt
2578 ./2006/03/29-03-06_1355001.txt
...
```

Analisando a saída, nós percebemos que alguns arquivos são bem pequenos e muitos dos arquivos são bem grande e com o mesmo tamanho (2578 e 2565) Quando

Quando observamos alguns desses arquivos maiores manualmente, parece que os grandes arquivos não são nada mais que HTML genérico idênticos que vinham de e-mails enviados para meu sistema a partir do meu próprio telefone:

```
<html>
        <head>
                <title>T-Mobile</title>
...
```

Passando através do arquivo parece que não há boas informações desses arquivos para que possamos provavelmente eliminá-los. Mas antes de excluir os arquivos, vamos escrever um programa para procurar por arquivos que são mais do que uma linha de tempo e mostrar o conteúdo do arquivo.

Não vamos nos incomodar mostrando os arquivos que são exatamente 2578 ou 2565 caracteres pois sabemos que estes não têm informações úteis. Assim podemos escrever o seguinte programa:

```
import os
from os.path import join
for (dirname, dirs, files) in os.walk('.'):
   for filename in files:
       if filename.endswith('.txt') :
           thefile = os.path.join(dirname,filename)
           size = os.path.getsize(thefile)
           if size == 2578 or size == 2565:
               continue
           fhand = open(thefile,'r')
           lines = list()
           for line in fhand:
               lines.append(line)
           fhand.close()
           if len(lines) > 1:
                print len(lines), thefile
                print lines[:4]
```

Nós usamos um `continue` para ignorar arquivos com dois "Maus tamanhos ", então, abrimos o resto dos arquivos e lemos as linhas do arquivo em uma lista Python, se o arquivo tiver mais que uma linha nós imprimimos a quantidade de linhas e as primeiras três linhas do arquivo.

Parece que filtrando esses dois tamanhos de arquivo ruins, e supondo que todos os arquivos de uma linha estão corretos, nós temos abaixo alguns dados bastante limpos:

```
python txtcheck.py
3 ./2004/03/22-03-04_2015.txt
['Little horse rider\r\n', '\r\n', '\r']
2 ./2004/11/30-11-04_1834001.txt
['Testing 123.\n', '\n']
3 ./2007/09/15-09-07_074202_03.txt
['\r\n', '\r\n', 'Sent from my iPhone\r\n']
3 ./2007/09/19-09-07_124857_01.txt
['\r\n', '\r\n', 'Sent from my iPhone\r\n']
3 ./2007/09/20-09-07_115617_01.txt
...
```

Mas existe um ou mais padrões chatos de arquivo: duas linhas brancas seguidas por uma linha que diz "Sent from my iPhone"que tem escorregado em meus dados. Então, fizemos a seguinte mudança no programa para lidar com esses arquivos, bem.

```
           lines = list()
           for line in fhand:
               lines.append(line)
           if len(lines) == 3 and lines[2].startswith('Sent from my iPho
```

```
            continue
        if len(lines) > 1:
             print len(lines), thefile
             print lines[:4]
```

Nós simplesmente verificamos se nós temos um arquivo com três linhas, e se a terceira linha inicia com o texto específico nós a pulas. Agora quando nós rodamos o programa, nós somente vemos apenas quatro restantes arquivos de multi-linha e todos esses arquivos parecem bastante razoável:

```
python txtcheck2.py
3 ./2004/03/22-03-04_2015.txt
['Little horse rider\r\n', '\r\n', '\r']
2 ./2004/11/30-11-04_1834001.txt
['Testing 123.\n', '\n']
2 ./2006/03/17-03-06_1806001.txt
['On the road again...\r\n', '\r\n']
2 ./2006/03/24-03-06_1740001.txt
['On the road again...\r\n', '\r\n']
```

Se você olhar para o padrão global deste programa, nós refinamos sucessivamente como aceitamos ou rejeitamos arquivos e uma vez encontrado um padrão que tenha ”bad”nós usamos `continue` para ignorar os maus arquivos para que pudéssemos refinar nosso código para encontrar mais padrões que eram ruim.

Agora estamos nos preparando para excluir os arquivos, nós vamos inverter a lógica e ao invés de imprimirmos os bons arquivos vamos imprimir os maus arquivos que estamos prestes a excluir.

```
import os
from os.path import join
for (dirname, dirs, files) in os.walk('.'):
   for filename in files:
       if filename.endswith('.txt') :
           thefile = os.path.join(dirname,filename)
           size = os.path.getsize(thefile)
           if size == 2578 or size == 2565:
               print 'T-Mobile:',thefile
               continue
           fhand = open(thefile,'r')
           lines = list()
           for line in fhand:
               lines.append(line)
           fhand.close()
           if len(lines) == 3 and lines[2].startswith('Sent from my iPhone'):
               print 'iPhone:', thefile
```

```
            continue
```

Podemos ver agora uma lista de possiveis arquivos os quais queremos apagar e por quê esses arquivos estão para exclusão

We can now see a list of candidate files that we are about Podemos ver agora uma lista de possiveis arquivos os quais queremos apagar e por quê esses arquivos estão para exclusão. O Programa produz a seguinte saída:

```
python txtcheck3.py
to delete and why these files are up for deleting.
The program produces the following output:

\begin{verbatim}
python txtcheck3.py
...
T-Mobile: ./2006/05/31-05-06_1540001.txt
T-Mobile: ./2006/05/31-05-06_1648001.txt
iPhone: ./2007/09/15-09-07_074202_03.txt
iPhone: ./2007/09/15-09-07_144641_01.txt
iPhone: ./2007/09/19-09-07_124857_01.txt
...
```

Podemos verificar pontualmente esses arquivos para certificar que nao iremos inserir um bug em nosso programa ou talvez a nossa logica nao pegou arquivos que nao queriamos

Uma vez satisfeitos que esta é lista de arquvos que queremos excluir, fazemos a seguinte mudança no programa.

```
            if size == 2578 or size == 2565:
                print 'T-Mobile:',thefile
                os.remove(thefile)
                continue
...
            if len(lines) == 3 and lines[2].startswith('Sent from my iPho
                print 'iPhone:', thefile
                os.remove(thefile)
                continue
```

Nesta versão do programa, iremos fazer ambos, imprimir o arquivo e remover os arquivos ruins com `os.remove`

```
python txtdelete.py
T-Mobile: ./2005/01/02-01-05_1356001.txt
T-Mobile: ./2005/01/02-01-05_1858001.txt
...
```

Apenas por diversão, rodamos o programa uma segunda vez e o programa não vai produzir nehuma saída desde que os arquivos ruins não existam.

Se rodar novamente `txtcount.py` podemos ver que removemos 899 arquivos ruins:

```
python txtcount.py
Files: 1018
```

Nesta seção, temos seguido uma sequencia onde usamos Python primeiro para navegar através dos diretórios e arquivos procurando padrões. Usamos Python devagar para ajudar a determinar como fariamos para limpar nosso diretorio. Uma vez descoberto quais arquivos são bons e quais não são,nós usamos o Python para excluir os arquivos e executar a limpeza.

O problema pode ser necessário para resolver pode ser bastante simples e pode depender somente procurar pelos nomes dos arquivos, ou talvez você precise ler todos os arquivos. alguma vezesvocê irá precisar ler todos os arquivos e fazer uma mudança para alguns dos arquivos. Todos estes são bastante simples uma vez que você entender como tt os.walk e os outros utilitários tt OS podem ser usados.

3.3 Argumentos de linha de comando

Nos capitulos anteriores tivemos uma série de programas que solicitavam por um nome de arquivo usando `raw_input` e então, ler os dados de um arquivo e processar os dados como segue:

```
name = raw_input('Enter file:')
handle = open(name, 'r')
text = handle.read()
...
```

Nós podemos simplificar este programa um pouco pegando p nome do arquivo de um comando de linha qiando iniciar o Python. Até agora nós simplesmente executamos nossos programas em Python e respondemos a solicitação como segue:

```
python words.py
Enter file: mbox-short.txt
...
```

Nós podemos lugar adicional sequencia de carcteres após o arquivo Pyhton e acessar aqueles **command-line arguments** em nosso programa Python. Aqui é um simples programa que demonstra leitura de arqumentos a partir de uma linha de comando.

```
import sys
print 'Count:', len(sys.argv)
print 'Type:', type(sys.argv)
for arg in sys.argv:
   print 'Argument:', arg
```

Os conteúdos de `sys.argv` são uma lista de sequencia de caracteres on aprmeira sequencia é o nome do programa Python e as outras sequencias são argumentos no comenado de linha aós o arquivo Python.

O seguinte mostra nosso programa lendo uma série de argumentos de linha de comando de uma linha de comando:

```
python argtest.py hello there
Count: 3
Type: <type 'list'>
Argument: argtest.py
Argument: hello
Argument: there
```

Há três argumentos são passados em nosso programa como uma lista de três elementos. O primeiro elemento da lista é o nome do arquivo (artest.py) e os outros são os dois argumentos de linha de comando após o nome do arquivo.

Nós podemos reescrever nosso programa para ler o arquivo pegando o nome do arquivo do argumento de linha de comando como segue:

```
import sys

name = sys.argv[1]
handle = open(name, 'r')
text = handle.read()
print name, 'is', len(text), 'bytes'
```

Nós égamos o segundo arqgumento da linha de comando com o nome do arquivo (pulando o nome do programa na entrada `[0]`). Nósabrimos o arquivo e lemos o conteudo como segue:

```
python argfile.py mbox-short.txt
mbox-short.txt is 94626 bytes
```

Usando argumento de linha de comando como entrada podemos tornar isto facil para reutilizar no seu priograna Python, especialment quando você somente precisa para entrada uma ou duas sequencias de caracteres.

3.4 Pipes

A maioria dos sistemas operacionais oferecem uma interface de linha de comando, conhecido também com **shell**. Shells normalmente normalmente disponibilizam comandos para navegar entre arquivos do sistemas e lançar aplicativos. Por exemplo, no Unix, você pode mudar de diretório com `cd`, mostrar na tela o conteudo de um diretório com `ls` e lançar um web browser digitando (por exemplo) `firefox`.

Qualquer programa que consiga lançar a partir do shell também pode ser lançado à partir do Python usando um **pipe**. Um pipe é um objeto que representa um processo em execução.

Por exemplo, o comando Unix[1] `ls -l` normalmente mostrar o conteudo do diretório atual(no formato longo). Você pode lançar `ls` com `os.open`:

```
>>> cmd = 'ls -l'
>>> fp = os.popen(cmd)
```

Um argumento é uma sequencia de caracteres que contem um comando shell. O valor de retorno é um poneiro para um arquivo que se comporta exatamente como um arquivo aberto.Você pode ler a saída do tt ls um processo linha de cada vez com tt readline ou obter a coisa toda de uma vez com tt ler:

```
>>> res = fp.read()
```

When you are done, you close the pipe like a file:

```
>>> stat = fp.close()
>>> print stat
None
```

O valor de retorno é a situação final do processo `ls`; `None` significa que ele terminou normalmente (sem erros).

3.5 Glossário

absolute path: Uma seqüência de caracteres que descreve onde um arquivo ou diretório é armazenado que começa no " topo da árvore de diretórios " de modo que ele pode ser usado para acessar o arquivo ou diretório, independentemente do diretório de trabalho atual.

checksum: Ver também **hashing**. O termo "checksum" vem da necessidade de verificar se os dados foi como foi ilegível enviada através de uma rede ou gravados em um meio de backup e, em seguida, leia novamente. Quando os dados são gravados ou enviados, o sistema de envio calcula uma soma de verificação e também envia a soma de verificação. Quando o os dados são lidos ou recebidos, o sistema de recepção re-computa a soma de verificação com base nos dados recebidos e compara-o com o recebido de soma de verificação. Das somas de verificação não corresponderem, devemos assumir que os dados foi ilegível, uma vez que foi transferido. checksum

command-line argument: Parametros na linha de comando após o nome do arquivo Python.

current working directory: O diretório atual que você esta "em". Você pode mudar seu diretório de trabalho usando o comando `cd` na maioria dos sistemas em suas interfaces de linha de comando. Quando você abre um arquivo em Python usando apenas o nome do arquivo sem caminho informações, o arquivo deve estar no diretório de trabalho atual onde está a executar o programa.

[1] Ao usar pipes para interagir com comando do sistema operacional como `ls`, isso é importante para você saber quais sistemas operacionais você esta usando e somente comandos pipeque são suportados élo seu sistema operacional

hashing: Leitura através de um potencialmente grande quantidade de dados e produzir uma soma de verificação original para os dados. As melhores funções hash produzem muito poucos ”colisões”onde você pode dar dois diferentes fluxos de dados para a função hash e receber de volta o mesmo hash. MD5, SHA1 e SHA256 são exemplos de funções hash mais usadas.

pipe: Um pipe é uma conexão com um programa em execução. usando um pipe, você pode escrever um programa para enviar os dados para outro programa ou receber dados a partir desse programa. Um pipe é semelhante a um **socket** aceitar que um pipe só pode ser usado para conectar os programas em execução no mesmo computador (ou seja, não através de uma rede). pipe

relative path: Uma sequência de caracteres que descreve onde um arquivo ou diretório é armazenado em relação ao trabalho atual diretório.

shell: Uma interface de linha de comando para um sistema operacional. Também chamado em alguns sistemas operacionais de ”programa terminal”Nesta interface você escreve um comando com para metros na linha e pressiona ”enter”para executar o comando.

walk: um termo que usamos para descrever a noção de visitando a entrada de arvorede diretóroio, sub-diretorios, sub-sub-diretórios, até que tenhamso visitado todos os diretórios. Nós chamamos isto de ”Caminhando na árvore de diretório”

3.6 Exercícios

Numa grande coleção de arquivos MP3 pode existir mais de uma cópia de um mesmo som, armazenado em diferentes diretórios ou com diferentes nomes de arquivo. O objetivo deste exercicio é procurar por essas duplicatas.

1. Escreva um programa que caminhe no diretório e todos esses subdiretórios para todos arquivoscom o sufixo (like `.mp3`) e listar par de arquvios com o mesmo tamanho. Dica: Use um dicionário onde a chave do dicionário é o tamanho do arquivo do `os.path.getsize` e o valor no dicionário é o nome do caminho concatenado com o nome do arquivo. Como você encontrar cada arquivo, verifique se você já tem um arquivo que tem o mesmo tamanho do arquivo atual. Se assim for, você tem um duplicar o tamanho do arquivo, então imprimir o tamanho do arquivo e os dois nomes de arquivo (um a partir do hash e outro arquivo que você está olhando).

2. Adaptar o programa anterior para procurar arquivos que com conteúdo duplicado usando um hash ou o algoritmo **checksum**. Por exemplo, MD5 (Message-Digest algorithm 5) leva um longo arbitrariamente- “ mensagem ” e retorna um 128-bit “checksum”. a probabilidade é muito pequena que dois arquivos com diferentes conteúdos retornem o mesmo checksum.

Você pode ler sobre MD5 em wikipedia.org/wiki/Md5 . O seguinte trecho de código abre um arquivo, o lê, e computa seu checksum.

```
import hashlib
...
            fhand = open(thefile,'r')
            data = fhand.read()
            fhand.close()
            checksum = hashlib.md5(data).hexdigest()
```

Você deve criar um dicionário onde a soma de verificação é a chave e o nome do arquivo é o valor. Quando você calcular o checksum e já está no dicionário como uma chave, você tem dois arquivos com conteúdo duplicado, então imprimir o arquivo no dicionário e o arquivo que você acabou de ler. Aqui está um exemplo de saída a partir de uma corrida em uma pasta de arquivos de imagem:

```
./2004/11/15-11-04_0923001.jpg ./2004/11/15-11-04_1016001.jpg
./2005/06/28-06-05_1500001.jpg ./2005/06/28-06-05_1502001.jpg
./2006/08/11-08-06_205948_01.jpg ./2006/08/12-08-06_155318_02.jpg
```

Aparentemente eu às vezes enviou a mesma foto mais de uma vez ou fez uma cópia de uma foto de vez em quando sem excluir o original.