

Python para Informáticos

Explorando a Informação

Version 2.7.2

Autor: Charles Severance
Tradução: @victorjabur

Copyright © 2009- Charles Severance. Tradução: PT-BR © 2015- : @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 principal, 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*.

Este trabalho está licenciado sob a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 licença não portada. Esta licença está disponível em creativecommons.org/licenses/by-nc-sa/3.0/. Você pode ver as considerações nas quais o autor considera a utilização comercial e não comercial deste material assim como as exceções da licença no apêndice intitulado Detalhes dos Direitos Autorais.

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 intitulado *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

Prefácio	iii
1 Por que você deve aprender a escrever programas ?	1
1.1 Criatividade e motivação	2
1.2 Arquitetura física do Computador - Hardware	3
1.3 Entendendo programação	5
1.4 Palavras e Sentenças	5
1.5 Conversando com Python	6
1.6 Terminologia: interpretador e compilador	8
1.7 Escrevendo um programa	11
1.8 O que é um programa ?	11
1.9 A construção de blocos de programas	13
1.10 O que pode dar errado?	14
1.11 A jornada do aprendizado	15
1.12 Glossário	16
1.13 Exercícios	17
2 Variáveis, expressões e instruções	19
2.1 Valores e tipos	19
2.2 Variáveis	20
2.3 Nomes de variáveis e palavras reservadas	21
2.4 Instruções	22

2.5	Operadores e operandos	22
2.6	Expressões	23
2.7	Ordem das operações	23
2.8	O operador Módulo	24
2.9	Operações com Strings	24
2.10	Solicitando dados de entrada para o usuário	25
2.11	Comentários	26
2.12	Escolhendo nomes de variáveis mnemônicos	26
2.13	Debugando	28
2.14	Glossário	29
2.15	Exercícios	30
3	Execução Condicional	33
3.1	Expressões booleanas	33
3.2	Operador Lógico	34
3.3	Execução condicional	34
3.4	Execução alternativa	35
3.5	Condicionais encadeadas	36
3.6	Condicionais aninhados	37
3.7	Capturando exceções usando try e except	38
3.8	Short-circuit avaliação de expressões lógicas	39
3.9	Depuração	41
3.10	Glossário	42
3.11	Exercícios	43
4	Dicionários	45
4.1	Dicionário como um conjunto de contagens	47
4.2	Dicionários e arquivos	48
4.3	Laços de repetição e dicionário	50
4.4	Processamento avançado de texto	51
4.5	Depuração	53
4.6	Glossário	53
4.7	Exercícios	54

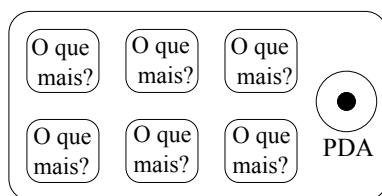
5	Banco de Dados e Structured Query Language (SQL)	57
5.1	O que é um banco de dados?	57
5.2	Conceitos de bancos de dados	58
5.3	Plugin do Firefox de Gerencia SQLite	58
5.4	Criado uma tabela em um banco de dados	59
5.5	Resumo de Structured Query Language (SQL)	62
5.6	Rastreando o Twitter utilizando um banco de dados	63
5.7	Modelagem de dados básica	69
5.8	Programando com múltiplas tabelas	71
5.9	Três tipos de chaves	75
5.10	Utilizando o JOIN para recuperar informações	76
5.11	Sumário	78
5.12	Depuração	79
5.13	Glossário	79
6	Automação de tarefas comuns no seu computador	81
6.1	Nomes e caminhos de arquivos	81
6.2	Exemplo: Limpando um diretório de fotos	82
6.3	Argumentos de linha de comando	87
6.4	Pipes	89
6.5	Glossário	90
6.6	Exercícios	91

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 podem ser muito úteis 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
```

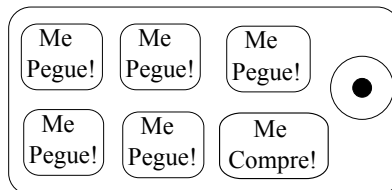
Nosso “assistente de análise pessoal de informações” rapidamente conta para nós que a palavra “para” foi utilizada dezesseis vezes nos primeiros três parágrafos deste capítulo.

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:

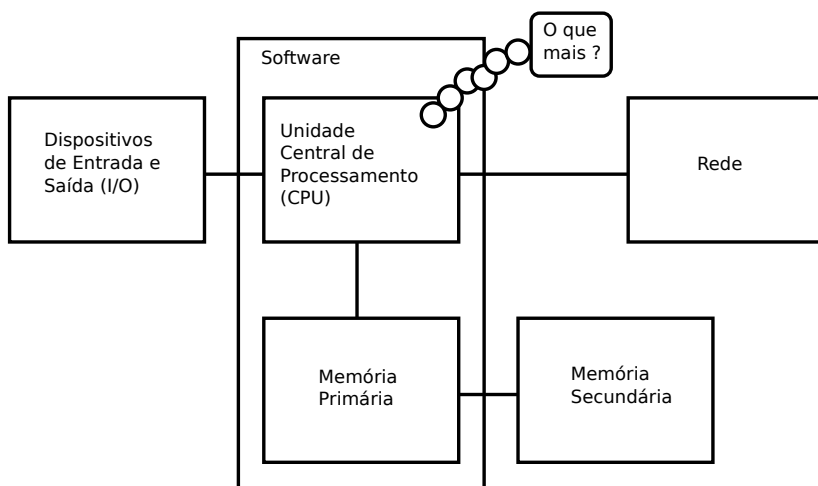


Por enquanto, nossa motivação primária não é ganhar dinheiro ou agradar usuários finais, mas sermos mais produtivos na manipulação de dados e informações que nós encontraremos em nossas vidas. Quando você começar, você será tanto o

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:



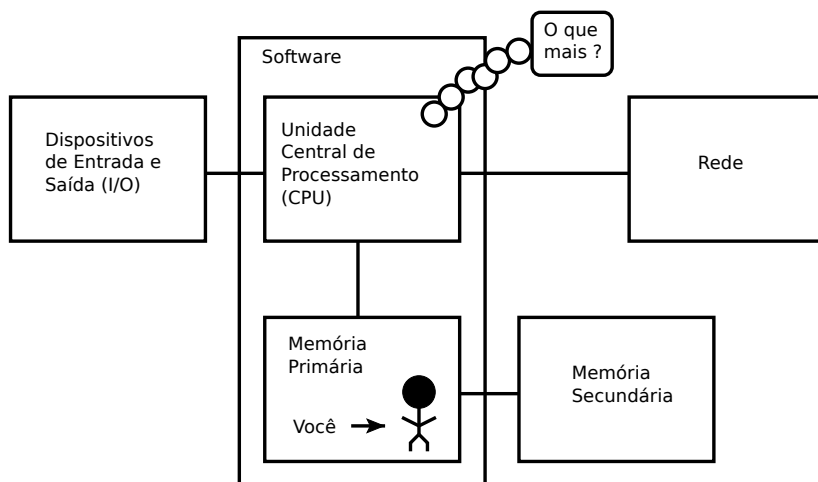
As definições resumidas destas partes são:

- 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 vantagem 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 essas partes conforme escrevemos nossos programas.

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 chamamos essas instruções armazenadas de **programa** e o ato de escrever essas instruções e garantir que essas estejam corretas de **programação**.

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 vocabulário 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 história”. Na escrita da história, você combina palavras e sentenças para convencer o leitor. É necessário qualidade e arte na construção da história, adquirir-se isso através da prática de contar histórias e obter um feedback. Na programação, nosso programa é a “história” 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 uma 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 leremos e explicaremos 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 ponto, programar 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, lembre quando você iniciou a leitura pela primeira vez.

1.4 Palavras e Sentenças

Diferentemente dos idiomas humanos, o vocabulário do Python é atualmente muito pequeno. Nós chamamos esse “vocabulário” de “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 escrever 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 cachorros e não usar qualquer uma dessas 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 **caminhar** 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 reservadas¹.

As palavras reservadas na linguagem pelas quais os humanos conversam com o Python, incluem as seguintes:

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 isso, 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 nossa 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 conversa 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, executando o modo interativo e aparecerá 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 conversa 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, colocá-lo em um espeto, assá-lo 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, 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 tentando 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 de **alto nível** cujo objetivo é ser relativamente fácil para humanos lerem e escreverem e para computadores lerem e processarem. Outras linguagens de 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 muito 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 intrincada que o Python. Poucos programadores escrevem em linguagem de máquina. Ao invés disso, nós usamos vários tradutores para permitir que os programadores escrevam em linguagem de máquina a partir de linguagens de alto nível como o Python ou o JavaScript. Essas linguagens convertem os programas para linguagem de máquina que, desse modo, são executados 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 a processa imediatamente e está pronto 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**, de modo a nós podermos 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**, multiplicá-lo por sete e armazenar de

Isso é mais do que você realmente precisa conhecer para ser um programador Python, mas às vezes, isso 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 pode imprimir o stream de posts do Facebook e debruçar-se sobre o texto procurando pela palavra mais comum, mas pode levar um tempo longo e ser muito propenso a erros. Você pode ser inteligente para escrever um programa Python para tratar disso rapidamente e 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 programa simples para encontrar a palavra mais comum em um arquivo texto. Eu escrevi, testei e agora eu estou dando isso 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 sequências úteis de instruções (i.e., programas) em uma linguagem comum que pode ser usada por qualquer um 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 muitas 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 avalia nossos programas perfeitamente escritos e os rejeita como “incorretos” apenas para nos atormentar.

```
>>> print 'Ola mundo!'
File "<stdin>", line 1
    print 'Ola mundo!'
    ^
SyntaxError: invalid syntax
>>> print 'Ola mundo'
File "<stdin>", line 1
    print '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 fica feliz e pronto para te servir quando você precisar 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 “re-

gras 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 para 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 descrição 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 não detectado.”

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 aprendeu a falar, não era um problema que em seus primeiros anos você fizesse sons fofos e desajeitados. Foi tudo certo se levou seis meses para se mover de um vocabulário simples até sentenças simples e levou mais 5-6 anos para se mover de sentenças a parágrafos, e uns anos mais para estar habilitado a escrever uma história curta e interessante 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 aprenda 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”.

compilar: 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.

interpretar: 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 de montagem”.

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 a expressar.

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, retendo 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 das 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:

```
>>> print 'Ola mundo!'
      File "<stdin>", line 1
        print '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

Variáveis, expressões e instruções

2.1 Valores e tipos

Um **valor** é uma das coisas básicas com a qual um programa trabalha, como uma letra ou um número. Os valores que vimos até agora são 1, 2, and 'Ola, Mundo!'

Estes valores pertencem a diferentes **tipos**: 2 é um inteiro, e 'Ola, Mundo!' é uma **string**, assim chamada por conter uma “cadeia” de letras. Você (e o interpretador) podem identificar strings porque elas aparecem entre aspas.

A instrução `print` também funciona com inteiros. Nós usamos o comando `python` para iniciar o interpretador.

```
python
>>> print 4
4
```

Se você não tem certeza que tipo tem um valor, o interpretador pode te dizer.

```
>>> type('Ola, Mundo!')
<type 'str'>
>>> type(17)
<type 'int'>
```

Não surpreendentemente, strings pertencem ao tipo `str` e inteiros pertencem ao tipo `int`. Menos, obviamente, números com ponto decimal pertencem a um tipo chamado `float`, uma vez que estes números são representados em um formato chamado **ponto flutuante**.

```
>>> type(3.2)
<type 'float'>
```

E quanto a valores como '17' e '3.2'? Eles se parecem com números, mas eles são, quando entre aspas, strings.

```
>>> type('17')
<type 'str'>
>>> type('3.2')
<type 'str'>
```

Eles são strings.

Quando você digita um número inteiro grande, você pode ficar tentado a utilizar vírgulas entre os grupos de três dígitos, como em 1,000,000. Este não é um número válido em Python, no entanto ele é válido:

```
>>> print 1,000,000
1 0 0
```

Bem, de toda forma, isto não é o que nós esperávamos! Python interpreta 1,000,000 como uma sequência de integers separados por vírgulas, o qual imprimi com espaços entre eles.

Este é o primeiro exemplo que vemos de um erro semântico: o código executa sem produzir uma mensagem de erro, mas ele não faz a coisa “certa”.

2.2 Variáveis

Uma das mais poderosas características de uma linguagem de programação é a capacidade de manipular **variáveis**. Uma variável é um nome que se refere a um valor.

Um **comando de atribuição** cria novas variáveis e dá valores a elas:

```
>>> message = 'E agora algo completamente diferente'
>>> n = 17
>>> pi = 3.1415926535897931
```

Este exemplo faz três atribuições. O primeiro atribui uma string a uma nova variável chamada `message`; o segundo atribui o integer 17 à variável `n`; o terceiro atribui valor (aproximado) de π à variável `pi`.

Para mostrar o valor de uma variável, você pode usar o comando `print`.

```
>>> print n
17
>>> print pi
3.14159265359
```

O tipo de uma variável é o tipo do valor ao qual ela se refere.

```
>>> type(message)
<type 'str'>
>>> type(n)
<type 'int'>
>>> type(pi)
<type 'float'>
```

2.3 Nomes de variáveis e palavras reservadas

Programadores geralmente escolhem nomes, que tenham algum significado, para suas variáveis e documentam para qual finalidade a variável será utilizada.

Nomes de variáveis podem ser arbitrariamente longos. Eles podem conter tanto letras quanto números, porém eles não podem começar com um número. É válido usar letras maiúsculas, porém é uma boa prática começar o nome de uma variável com uma letra minúscula (você verá o porquê, mais tarde).

O caractere sublinhado (`_`) pode aparecer no nome. Ele é frequentemente usado em nomes com múltiplas palavras, como `my_name` ou `airvelocidade_of_unladen_swallow`.

Nomes de variáveis podem começar como caractere sublinhado, mas nós, geralmente, evitamos isto, a menos que estejamos escrevendo uma biblioteca de código para outros usarem.

Se você der a uma variável um nome inválido, você receberá um erro de sintaxe.

```
>>> 76trombones = 'grande desfile'
SyntaxError: invalid syntax
>>> more@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Avancada Teoria Zymurgy'
SyntaxError: invalid syntax
```

`76trombones` é inválida porquê ela começa com um número. `more@` é inválida porquê ela contém um caractere inválido, `@`. Mas o quê há de errado com `class`? Acontece que a palavra `class` é uma Palavra Reservada do Python **keywords**. O interpretador usa as Palavras Reservadas para reconhecer a estrutura do programa, e elas não podem ser usadas como nomes de variáveis.

Python reserva 31 Palavras Reservadas ¹ para seu uso:

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

Você pode querer manter esta lista ao alcance das mãos. Se o interpretador reclamar sobre um de seus nomes de variável e você não souber o porquê, verifique se ela se encontra nesta lista.

¹Em Python 3.0, `exec` não é mais uma palavra reservada, mas `nonlocal` é.

2.4 Instruções

Uma **instrução** é uma unidade de código que o interpretador Python pode executar. Nós vimos dois tipos de instruções: impressão (`print`) e atribuição (`=`).

Quando você digita uma instrução no modo interativo, o interpretador a executa e mostra o resultado, se houver um.

Um script geralmente contém uma sequência de instruções. Se houver mais de uma instrução, os resultados aparecem um de cada vez conforme as instruções são executadas.

Por exemplo, o script

```
print 1
x = 2
print x
```

Produz a saída:

```
1
2
```

A instrução de atribuição não produz saída.

2.5 Operadores e operandos

Operadores são símbolos especiais que representam cálculos como adição e multiplicação. Os valores aos quais os operadores são aplicados são chamados de **operandos**.

Os operadores `+`, `-`, `*`, `/`, e `**` realizam, adição, subtração, multiplicação, divisão e exponenciação, como no exemplo a seguir:

```
20+32   hora-1   hora*60+minuto   minuto/60   5**2   (5+9)*(15-7)
```

O operador de divisão pode não fazer o que você espera:

```
>>> minuto = 59
>>> minuto/60
0
```

O valor de `minuto` é 59, e na aritmética convencional 59 dividido por 60 é 0.98333, não 0. A razão para esta discrepância é o fato de que o Python realiza um **floor division**²

Quando ambos os operandos são integers, o resultado é, também, um integer; floor division corta a parte fracionária, portanto, neste exemplo o resultado foi arredondado para zero.

²Em Python 3.0, o resultado desta divisão é do tipo `float`. Em Python 3.0, o novo operador `//` realiza uma divisão to tipo `integer`.

Se um dos operandos é um número do tipo ponto flutuante, Python realiza uma divisão de ponto flutuante, e o resultado é um `float`:

```
>>> minuto/60.0
0.98333333333333328
```

2.6 Expressões

Uma **expressão** é uma combinação de valores, variáveis e operadores. Um valor, por si só, é considerado uma expressão, e portanto, uma variável, então o que segue são todas expressões válidas (assumindo que a variável `x` tenha recebido um valor):

```
17
x
x + 17
```

Se você digita uma expressão no modo interativo, o interpretador a **avalia** e mostra o resultado:

```
>>> 1 + 1
2
```

Mas em um script, uma expressão por si só não faz nada! Isto é uma fonte comum de confusão para iniciantes.

Exercício 2.1 Digite a seguinte declaração no interpretador do Python para ver o que ele faz:

```
5
x = 5
x + 1
```

2.7 Ordem das operações

Quando mais de um operador aparece em uma expressão, a ordem de avaliação depende das **regras de precedência**. Para operadores matemáticos, Python segue a convenção matemática. O Acrônimo **PEMDAS** é uma modo útil para lembrar as regras:

- **Parênteses** têm a mais alta precedência e pode ser usado para forçar que uma expressão seja calculada na ordem que você deseja. Como as expressões entre parênteses são avaliadas primeiro, $2 * (3-1)$ é 4, e $(1+1) ** (5-2)$ é 8. Você também pode usar parênteses para tornar uma expressão mais fácil de ser lida, como em $(minute * 100) / 60$, mesmo que isto não mude o resultado.

- Exponenciação é a próxima precedência mais alta, então $2^{**}1+1$ é 3, não 4, e $3^{*}1^{**}3$ é 3, não 27.
- Multiplicação e Divisão têm a mesma precedência, a qual é mais alta que Adição e Subtração, que também têm a mesma precedência entre si. Então $2^{*}3-1$ é 5, não 4, e $6+4/2$ é 8, não 5.
- Operadores com a mesma precedência são avaliados da esquerda para direita. Portanto na expressão $5-3-1$ é 1, não 3 pois o $5-3$ acontece primeiro e então o 1 é subtraído de 2.

Na dúvida, sempre utilize parênteses em suas expressões para ter certeza de que os cálculos serão realizados na ordem que você deseja.

2.8 O operador Módulo

O **operador módulo** funciona em integers e fornece o resto da divisão, quando o primeiro operando é dividido pelo segundo. No Python, o operador módulo é um sinal de percentual (%). A sintaxe é a mesma dos outros operadores:

```
>>> quociente = 7 / 3
>>> print quociente
2
>>> resto = 7 % 3
>>> print resto
1
```

Portanto, 7 dividido por 3 é igual a 2, com resto 1.

O operador módulo apresenta-se surpreendentemente útil. Por exemplo, você pode checar se um número é divisível por outro—se $x \% y$ é zero, então x é divisível por y .

Você pode, também, testar se um número é divisível por outro. Por exemplo, $x \% 10$ nos mostra se o número x é divisível por 10. Similarmente, $x \% 100$ nos mostra se x é divisível por 100.

2.9 Operações com Strings

O operador $+$ funciona com strings, mas ele não é uma adição no sentido matemático. Ao invés disto, ele realiza **concatenação**, que significa juntar as strings, vinculando-as de ponta-a-ponta. Por exemplo:

```
>>> primeiro = 10
>>> segundo = 15
>>> print primeiro + segundo
25
>>> primeiro = '100'
```

```
>>> segundo = '150'
>>> print primeiro + segundo
100150
```

A saída deste programa é 100150.

2.10 Solicitando dados de entrada para o usuário

Algumas vezes gostaríamos de solicitar, do usuário, o valor para uma variável por meio do teclado. Python fornece uma função interna chamada `raw_input` que recebe dados de entrada a partir do teclado³. Quando esta função é chamada, o programa para e espera para que o usuário digite algo. Quando o usuário pressiona o Return ou Enter, o programa continua e a função `raw_input` retorna o que o usuário digitou, como uma string.

```
>>> entrada = raw_input()
Alguma coisa boba
>>> print entrada
Alguma coisa boba
```

Antes de receber os dados de entrada do usuário, é uma boa idéia imprimir uma mensagem, dizendo ao usuário que o dado deve ser informado. Você pode passar uma string para a função `raw_input` para ser mostrada para o usuário antes da parada para a entrada de dados:

```
>>> nome = raw_input('Qual é o seu nome?\n')
Qual é o seu nome?
Chuck
>>> print nome
Chuck
```

A sequência `\n` no final da mensagem representa uma **nova linha**, que é um caractere especial que causa a quebra de linha. É por este motivo que os dados de entrada informados pelo usuário aparecem abaixo da mensagem.

Se você espera que o usuário digite um integer, você pode tentar converter o valor retornado para `int` usando a função `int()`:

```
>>> pergunta = 'Qual é ... a velocidade de uma andorinha sem carga?\n'
>>> velocidade = raw_input(pergunta)
Qual é ... a velocidade de uma andorinha sem carga?
17
>>> int(velocidade)
17
>>> int(velocidade) + 5
22
```

Porém, se o usuário digita algo diferente de um conjunto de números, você recebe um erro:

³Em Python 3.0, esta função é chamada de `input`.

```
>>> velocidade = raw_input(pergunta)
Qual é ... a velocidade de uma andorinha sem carga?
Que tipo de andorinha, uma Africana ou uma Européia?
>>> int(velocidade)
ValueError: invalid literal for int()
```

Nós veremos como tratar este tipo de erro mais tarde.

2.11 Comentários

Como os programas ficam maiores e mais complicados, eles ficam mais difíceis de serem lidos. Linguagens formais são densas, e muitas vezes é difícil olhar para um pedaço de código e descobrir o que ele está fazendo, ou porquê.

Por esta razão, é uma boa ideia adicionar notas em seus programas para explicar, em linguagem natural, o que o programa está fazendo. Estas notas são chamadas de **comentários**, e, em Python, elas começam com o símbolo #:

```
# computa a porcentagem de hora que se passou
porcentagem = (minuto * 100) / 60
```

Neste caso, o comentário aparece sozinho em uma linha. Você pode, também, colocar o comentário no final da linha:

```
porcentagem = (minuto * 100) / 60      # porcentagem de uma hora
```

Todos os caracteres depois do #, até o fim da linha são ignorados—eles não têm efeito sobre o programa. Comentários são mais úteis quando documentam características não óbvias do código. É razoável assumir que o leitor pode descobrir *o que* o código faz; é muito mais útil explicar o *porquê*.

Este comentário é redundante e inútil dentro do código:

```
v = 5      # atribui o valor 5 para a variável v
```

Este comentário contém informações úteis que não estão no código.

```
v = 5      # velocidade em metros por segundo
```

Bons nomes de variáveis podem reduzir a necessidade de comentários, porém, nomes longos podem tornar expressões complexas difíceis de serem lidas, então devemos ponderar.

2.12 Escolhendo nomes de variáveis mnemônicos

Contanto que você siga as regras simples de nomenclatura de variáveis, e evite Palavras Reservadas, você tem muitas escolhas quando você nomeia suas variáveis. No início, esta escolha pode ser confusa, tanto quando você lê um programa,

quanto quando você escreve seus próprios programas. Por exemplo, os três programas a seguir são idênticos em termos do que realizam, mas muito diferente quando você os lê e tenta compreendê-los.

```
a = 35.0
b = 12.50
c = a * b
print c
```

```
horas = 35.0
taxa = 12.50
pagamento = horas * taxa
print pagamento
```

```
x1q3z9ahd = 35.0
x1q3z9afd = 12.50
x1q3p9afd = x1q3z9ahd * x1q3z9afd
print x1q3p9afd
```

O interpretador Python vê todos os três programas *exatamente como o mesmo*, mas os seres humanos veem e entendem esses programas de forma bastante diferente, entenderão mais rapidamente a **intenção** do segundo programa, porque o programador escolheu nomes de variáveis que refletem a sua intenção sobre os dados que serão armazenados em cada variável.

Nós chamamos esses nomes de variáveis sabiamente escolhidos de “nomes de variáveis mnemônicos”. A palavra *mnemônico*⁴ significa “auxiliar de memória”. Nós escolhemos os nomes de variáveis mnemônicos para nos ajudar a lembrar o motivo pelo qual criamos a variável, em primeiro lugar.

Isso tudo soa muito bem, e é uma boa ideia usar nomes de variável mnemônicos, eles podem atrapalhar a capacidade de análise e entendimento do código de um programador iniciante. Isto acontece porque os programadores iniciantes ainda não memorizaram as palavras reservadas (existem apenas 31 delas) e, por vezes, variáveis que têm nomes muito descritivos podem parecer parte da linguagem e não apenas nomes de variáveis bem escolhidas.

Dê uma olhada rápida no seguinte exemplo de código Python que percorre alguns dados. Nós vamos falar sobre loops em breve, mas por agora apenas tente imaginar como isto funciona:

```
for palavra in palavras:
    print palavra
```

O que está acontecendo aqui? Qual das palavras (for, palavra, in, etc.) são palavras reservadas e quais são apenas nomes de variáveis? O Python entende em um nível fundamental a noção de palavras? Programadores iniciantes têm dificuldade para separar quais partes do código *devem* ser o mesmo que este exemplo e que partes

⁴veja <http://en.wikipedia.org/wiki/Mnemonic> para uma descrição completa da palavra “mnemônico”.

do código são simplesmente as escolhas feitas pelo programador. O código a seguir é equivalente ao código acima:

```
for pedaco in pizza:
    print pedaco
```

É mais fácil para o programador iniciante olhar para este código e saber quais partes são palavras reservadas definidas pelo Python e quais partes são, simplesmente, nomes de variáveis escolhidos pelo programador. É bastante claro que o Python não tem nenhuma compreensão fundamental de pizza e pedaços e o fato de que uma pizza é constituída por um conjunto de um ou mais pedaços.

Mas se o nosso programa é verdadeiramente sobre a leitura de dados e a procura de palavras nos dados, `pizza` e `pedaco` são nomes de variáveis não muito mnemônicos. Escolhê-los como nomes de variável, distorce o significado do programa. Depois de um período muito curto de tempo, você vai conhecer as palavras reservadas mais comuns, então vai começar a ver as palavras reservadas saltando para você: **for** palavra **in** palavras:

```
print palavra
```

As partes do código que são definidas pelo Python (`for`, `in`, `print`, and `:`) estão em negrito e as variáveis escolhidas pelo programador (`word` and `words`) não estão em negrito. Muitos editores de textos compreendem a sintaxe do Python e vão colorir palavras reservadas de forma diferente para dar a você pistas e manter suas variáveis e palavras reservadas separadas. Depois de um tempo você começará a ler o Python e rapidamente determinar o que é uma variável e o que é uma palavra reservada.

2.13 Debugando

Neste ponto, o erro de sintaxe que você está mais propenso a cometer é um nome de variável ilegal, como `class` e `yield`, que são palavras reservadas ou emprego~estranho e RS\$, que contêm caracteres não permitidos.

Se você colocar um espaço em um nome de variável, o Python interpreta que são dois operandos sem um operador:

```
>>> nome ruim = 5
SyntaxError: invalid syntax
```

Para erros de sintaxe, as mensagens de erro não ajudam muito. As mensagens mais comuns são `SyntaxError: invalid syntax` and `SyntaxError: invalid token`, nenhuma das quais é muito informativa.

O erro de execução que você está mais propenso a a cometer é “use before def;”, isto é, tentando usar uma variável antes de atribuir um valor. Isso pode acontecer se você digitar um nome de variável errado:

```
>>> principal = 327.68
>>> interesse = principal * taxa
NameError: name 'taxa' is not defined
```

Nomes de variáveis são sensíveis a maiúsculo e minúsculo, desta forma, LaTeX não é o mesmo que latex.

Neste ponto, a causa mais provável de um erro de semântica é a ordem das operações. Por exemplo, para calcular $\frac{1}{2\pi}$, você pode ser tentado a escrever

```
>>> 1.0 / 2.0 * pi
```

Mas a divisão acontece primeiro, então você iria ficar com $\pi/2$, que não é a mesma coisa! Não há nenhuma maneira de o Python saber o que você quis escrever, então, neste caso você não receberia uma mensagem de erro; você apenas receberia uma resposta errada.

2.14 Glossário

atribuição: Uma instrução que atribui um valor a uma variável.

concatenar: Para juntar dois operandos ponta-a-ponta.

Comentário : Informação em um programa que é destinado a outros programadores (ou qualquer pessoa lendo o código fonte) e não tem qualquer efeito sobre a execução do programa.

Avaliar: Para simplificar uma expressão realizando as operações, a fim de se obter um único valor.

Expressão: Uma combinação de variáveis, operadores e valores que representa um valor de resultado único.

Ponto Flutuante: Um tipo que representa números com partes fracionárias.

Floor Division: A operação que divide dois números e corta a parte fracionária.

Integer: Um tipo que representa números inteiros.

Palavra Reservada: Uma palavra reservada usada pelo compilador para analisar um programa; você não pode usar palavras reservadas como `if`, `def`, e `while` como nomes de variáveis.

Mnemônico: Um auxiliar de memória. Nós, muitas vezes, damos nomes mnemônicos a variáveis para nos ajudar lembrar o que está armazenado na mesma.

Operador módulo: Um operador, denotado pelo sinal de porcentagem (%), que funciona em inteiros e produz o restante quando um número é dividido por outro.

Operando: Um dos valores sobre os quais um operador opera.

Operador: Um símbolo especial que representa uma cálculo simples, como adição, multiplicação ou concatenação de strings.

Regras de precedência: O conjunto de regras que regem a ordem na qual as expressões, envolvendo múltiplos operadores e operandos, são avaliadas.

Instrução: Uma seção de código que representa um comando ou ação. Até o momento, as instruções que temos visto são instruções de atribuição e impressão.

String: Um tipo que representa sequências de caracteres.

Tipo: Uma categoria de valores. Os tipos que vimos até o momento são inteiros (tipo `int`), números de ponto flutuante (tipo `float`) e strings (tipo `str`).

valor: Uma das unidades básicas de dados, como um número ou string, que um programa manipula.

variável: Um nome que se refere a um valor.

2.15 Exercícios

Exercício 2.2 Escreva um programa que utiliza `raw_input` para solicitar a um usuário o nome dele, em seguida, saudá-lo.

```
Digite o seu nome: Chuck
Ola Chuck
```

Exercício 2.3 Escreva um programa para solicitar ao usuário por, horas e taxa por hora, e então, calcular salário bruto.

```
Digite as horas: 35
Digite a taxa: 2.75
Pagamento: 96.25
```

Não estamos preocupados em fazer com que o nosso pagamento tenha exatamente dois dígitos depois da vírgula, por enquanto. Se você quiser, pode brincar com a função `round` do Python para adequadamente arredondar o pagamento resultante com duas casas decimais.

Exercício 2.4 Suponha que nós executamos as seguintes instruções de atribuição:

```
comprimento = 17
altura = 12.0
```

Para cada uma das seguintes expressões, escrever o valor da expressão e o tipo (do valor da expressão).

1. comprimento/2
2. comprimento/2.0
3. altura/3
4. 1 + 2 * 5

Utilize o interpretador do Python para conferir suas respostas.

Exercício 2.5 Escreva um programa que pede ao usuário por uma temperatura Celsius, converter a temperatura para Fahrenheit e imprimir a temperatura convertida.

Capítulo 3

Execução Condicional

3.1 Expressões booleanas

Uma **expressão booleana** é uma expressão que é `true` ou `false`. Os seguintes exemplos usam o operador `==`, que compara dois operadores e produz `True` se eles forem iguais e `False` caso contrário:

```
>>> 5 == 5
True
>>> 5 == 6
False
```

`True` e `False` são valores especiais que pertencem ao tipo `bool`; eles não são strings:

```
>>> type(True)
<type 'bool'>
>>> type(False)
<type 'bool'>
```

O operador `==` é um dos **operadores de comparação**; os outros são:

<code>x != y</code>	# x não é igual a y
<code>x > y</code>	# x é maior que y
<code>x < y</code>	# x é menor que y
<code>x >= y</code>	# x é maior ou igual a y
<code>x <= y</code>	# x é menor ou igual a y
<code>x is y</code>	# x é o mesmo que y
<code>x is not y</code>	# x não é o mesmo que y

Embora estas operações sejam provavelmente familiar para você, os símbolos Python são diferentes dos símbolos matemáticos para a mesma operação. Um erro comum é usar um único sinal de igual (`=`) em vez de um sinal de igual duplo (`==`). Lembre-se que o `=` é um operador de atribuição e `==` é um operador de comparação. Não existe tal coisa como `=<` ou `=>`.

3.2 Operador Lógico

Existem três **operadores lógicos**: `and`, `or`, and `not`. A semântica (significado) destes operadores é semelhante ao seu significado em inglês. Por exemplo,

```
x > 0 and x < 10
```

só é verdade se `x` for maior que 0 e menor que 10.

`n%2 == 0 or n%3 == 0` é verdadeiro se *qualquer* uma das condições é verdadeira, isto é, se o número é divisível por 2 ou 3.

Finalmente, o operador `not` nega uma expressão booleana, então `not (x > y)` é verdadeiro se `x > y` é falso; isto é, se `x` é menor do que ou igual a `y`.

Rigorosamente falando, os operandos dos operadores logicos devem ser expressões booleanas, mas Python não é muito rigoroso. Qualquer numero diferente de zero é interpretado como “verdadeiro.”

```
>>> 17 and True
True
```

Esta flexibilidade pode ser útil, mas existem algumas sutilezas que podem confundir o Python. Você pode querer evitá-los até você ter certeza que sabe o que está fazendo.

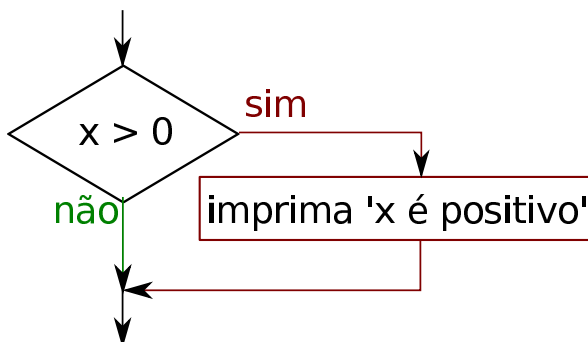
3.3 Execução condicional

Para escrever programas úteis, quase sempre precisamos da capacidade para verificar as condições e mudar o comportamento do programa em conformidade.

Instruções condicionais nos dão essa capacidade. A forma mais simples é a instrução `if`:

```
if x > 0 :
    imprima 'x é positivo'
```

A expressão booleana depois da declaração `if` é chamado de **condição**. Terminamos a instrução `if` com um caractere dois pontos (`:`) e a(s) linha(s) após a instrução `if` são indentadas.



Se a condição lógica é verdadeira, então a instrução indentada é executada. Se a condição lógica é falsa, a instrução indentada é ignorada.

Instruções `if` têm a mesma estrutura que as definições de funções ou loops `for`¹. A instrução é composta por uma linha de cabeçalho que termina com o caractere dois pontos (`:`) seguido por um bloco indentado. Instruções como esta são chamadas **declarações compostas** porque elas são compostas por mais de uma linha.

Não há limite para o número de instruções que podem aparecer no corpo, mas deve haver pelo menos uma. Às vezes, é útil ter um corpo sem instruções (usualmente como um corpo pacificador para o código que você não tenha escrito até o momento). Nesse caso, você pode usar a instrução `pass`, que não faz nada.

```
if x < 0 :  
    pass          # precisa lidar com valores negativos!
```

Se você digitar um `if` no interpretador Python, o prompt vai se modificar de três sinais `>>>` para três pontos `...` para indicar que você está no meio de um bloco de declarações, como mostrado abaixo:

```
>>> x = 3  
>>> if x < 10:  
...     print 'pequeno'  
...  
Small  
>>>
```

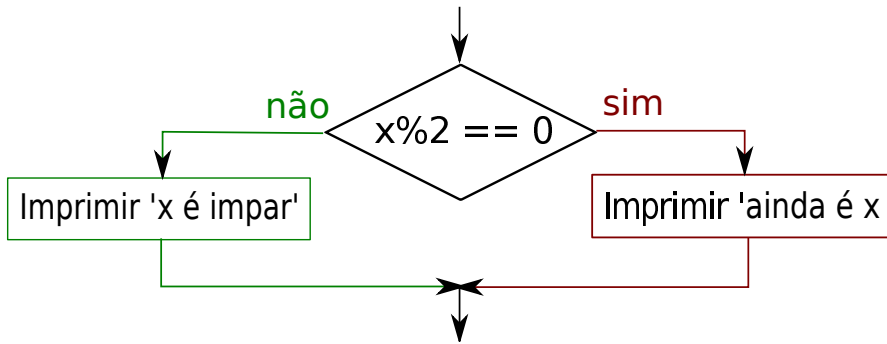
3.4 Execução alternativa

A segunda forma da instrução `if` é a **execução alternativa**, na qual há duas possibilidades e a condição determina qual delas será executada. A sintaxe se parece com esta:

```
if x%2 == 0 :  
    print 'x ainda é'  
else :  
    print 'x é estranho'
```

Se o resto da divisão de `x` por 2 for 0, nós sabemos que `x` é divisível, e o programa exibe uma mensagem para esse efeito. Se a condição for falsa, o segundo conjunto de instruções é executado.

¹Vamos aprender sobre as funções no Capítulo 4 e loops no Capítulo 5.



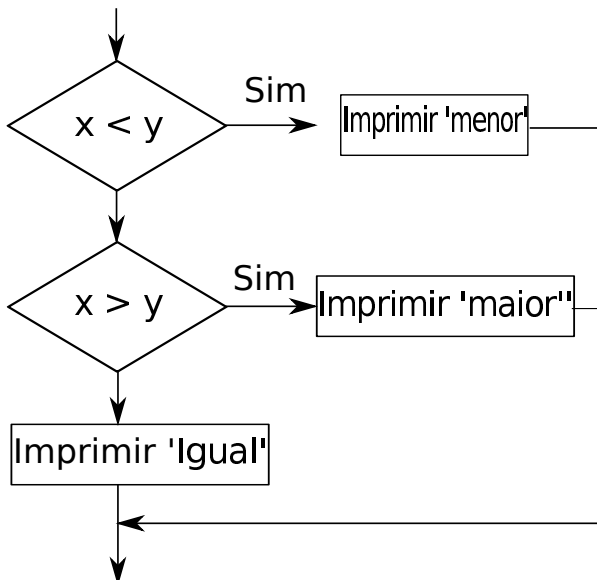
Uma vez que a condição deve ser verdadeira ou falsa, exatamente uma das alternativas será executada. As alternativas são chamadas de **branches**, porque elas dividem o fluxo de execução.

3.5 Condicionais encadeadas

Às vezes, há mais de duas possibilidades e precisamos de mais do que duas condições. Uma maneira de expressar uma computação como essa é uma **condição encadeada**:

```
if x < y:
    print 'x é menor que y'
elif x > y:
    print 'x é maior que y'
else:
    print 'x e y são iguais'
```

`elif` é uma abreviação de “else if.” Mais uma vez, exatamente uma condição será executada.



Não há limite para o número de instruções `elif`. Se houver uma cláusula `else`, ela deve estar no final, mas só pode existir uma única instrução deste tipo.

```
if choice == 'a':  
    print 'Escolha ruim'  
elif choice == 'b':  
    print 'Boa escolha'  
elif choice == 'c':  
    print 'Perto, mas não correto'
```

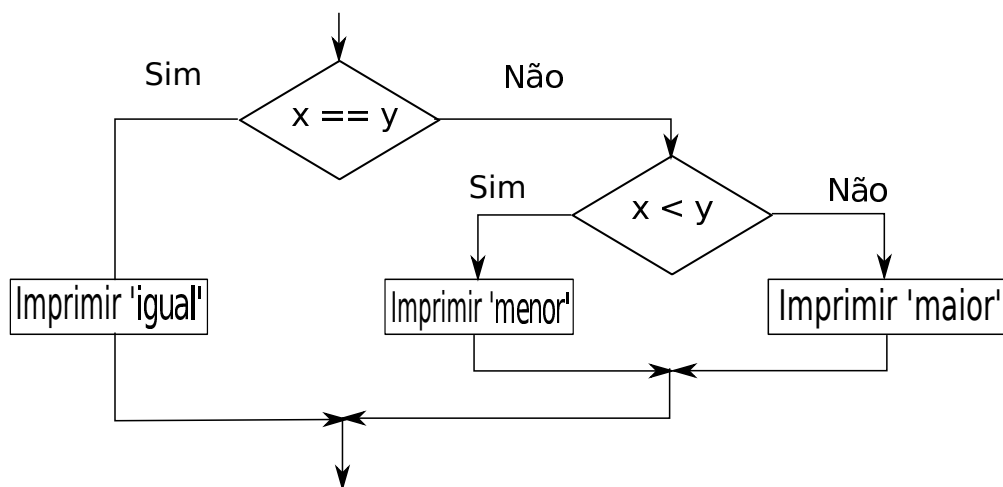
Cada condição é verificada em ordem. Se a primeira é falsa, a próxima será avaliada, e assim por diante. Se um deles é verdadeiro, o fluxo correspondente será executado, e a instrução termina. Mesmo se mais do que uma condição for verdadeira, apenas o primeiro fluxo verdadeiro é executado.

3.6 Condicionais aninhados

Uma instrução condicional também pode ser aninhada dentro de outra. Nós poderíamos ter escrito o exemplo de três ramificações como a seguir:

```
if x == y:  
    print 'x e y são iguais'  
else:  
    if x < y:  
        print 'x é menor que y'  
    else:  
        print 'x é maior que y'
```

A condicional externa contém duas ramificações. A primeira ramificação contém uma instrução simples. A segunda ramificação contém outra instrução `if`, que contém duas ramificações próprias. Aquelas duas ramificações são ambas instruções simples, embora pudessem ter sido instruções condicionais também.



Embora a indentação das instruções torna a estrutura visível, **condicionais aninhadas** fica difícil de ler muito rapidamente. Em geral, é uma boa idéia evitá-las

sempre que possível.

Os operadores lógicos muitas vezes fornecem uma maneira de simplificar as instruções condicionais aninhadas. Por exemplo, podemos reescrever o código a seguir usando um condicional simples:

```
if 0 < x:
    if x < 10:
        print 'x é um número positivo de um dígito.'
```

A instrução `print` é executada somente se ambas as condições forem verdadeiras, para que possamos obter o mesmo efeito com o operador `and`:

```
if 0 < x and x < 10:
    print 'x é um número positivo de um dígito.'
```

3.7 Capturando exceções usando `try` e `except`

Anteriormente, vimos um segmento de código onde foram utilizadas as funções `raw_input` e `int` para ler e validar um número inteiro informado pelo usuário. Também vimos como pode ser traiçoeiro utilizar isso:

```
>>> speed = raw_input(prompt)
Qual é ... a velocidade aerodinâmica de uma andorinha sem carga?
Você quer saber, uma andorinha Africana ou Européia?
>>> int(speed)
ValueError: invalid literal for int()
>>>
```

Quando estamos executando estas instruções no interpretador Python, temos um novo prompt do interpretador, acho que “oops”, e move-se para a próxima instrução.

No entanto, se você colocar esse código em um script Python e este erro ocorrer, seu script para imediatamente e nos retorna sua pilha de execução. Não foi executada a seguinte instrução.

Aqui está um programa de exemplo para converter uma temperatura Fahrenheit para uma temperatura em graus Celsius:

```
inp = raw_input('Digite a Temperatura Fahrenheit:')
fahr = float(inp)
cel = (fahr - 32.0) * 5.0 / 9.0
print cel
```

Se nós executarmos este código e informarmos uma entrada inválida, ele simplesmente falha com uma mensagem de erro não amigável:

```
python fahren.py
Digite a Temperatura Fahrenheit:72
22.2222222222
```



```
python fahren.py
Digite a Temperatura Fahrenheit:fred
Traceback (most recent call last):
  File "fahren.py", line 2, in <module>
    fahr = float(inp)
ValueError: invalid literal for float(): fred
```

Existe uma estrutura de execução condicional do Python para lidar com esses tipos esperados e inesperados de erros chamados “try / except”. A ideia de try e except é a de que você saiba que alguma sequência de instrução pode ter algum problema e você queira adicionar algumas instruções para serem executadas, caso um erro ocorra. Estas instruções adicionais (dentro do bloco except) são ignoradas se não ocorrer um erro.

Você pode associar os recursos try e except do Python como sendo uma “política segura” em uma sequência de instruções.

Podemos reescrever nosso conversor de temperaturas da seguinte forma:

```
inp = raw_input('Digite a Temperatura Fahrenheit:')
try:
    fahr = float(inp)
    cel = (fahr - 32.0) * 5.0 / 9.0
    print cel
except:
    print 'Por favor, digite um numero'
```

Python começa executando a sequência de instruções dentro do bloco try. Se tudo correr bem, ele ignora o bloco except e prossegue. Se uma exceção ocorre no bloco try, o Python pula para fora do bloco try e executa a sequência de instruções do bloco except.

```
python fahren2.py
Digite a Temperatura Fahrenheit:72
22.2222222222

python fahren2.py
Digite a Temperatura Fahrenheit:fred
Por favor, digite um numero
```

Tratar uma exceção com uma instrução try é chamado de **capturar** uma exceção. Neste exemplo, a cláusula except imprime uma mensagem de erro. Em geral, capturar uma exceção oferece a oportunidade de corrigir o problema, ou tentar novamente, ou pelo menos terminar o programa elegantemente.

3.8 Short-circuit avaliação de expressões lógicas

Quando o Python está processando uma expressão lógica, tal como $x \geq 2$ e $(x / y) > 2$, ele avalia a expressão da esquerda para a direita. Devido à definição do and, se x é inferior a 2, a expressão $x \geq 2$ é False e assim toda a expressão

é False independentemente de saber se $(x / y) > 2$ é avaliada como True ou False.

Quando o Python detecta que não existe nenhum ganho em se avaliar o resto de uma expressão lógica, ele para a sua avaliação e não faz os cálculos para o restante da expressão lógica. Quando a avaliação de uma expressão lógica para porque o valor global já é conhecido, a avaliação é chamada de **short-circuiting**.

Embora esta técnica pareça ter pouca importância, o comportamento de short-circuit leva a uma técnica inteligente chamada **guardian pattern**. Considere a seguinte sequência de código no interpretador Python:

```
>>> x = 6
>>> y = 2
>>> x >= 2 and (x/y) > 2
True
>>> x = 1
>>> y = 0
>>> x >= 2 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and (x/y) > 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
>>>
```

O terceiro cálculo falhou porque o Python estava avaliando (x/y) e y foi zero, o que causou um erro de execução. Mas o segundo exemplo *não* falhou porque a primeira parte da expressão $x \geq 2$ foi avaliada como False então a expressão (x/y) não foi executada devido à regra **short-circuit** e não houve erro.

Podemos construir a expressão lógica para colocar estrategicamente uma avaliação do tipo **guardian pattern** antes da avaliação que pode causar um erro, como segue:

```
>>> x = 1
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x >= 2 and (x/y) > 2 and y != 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
>>>
```

Na primeira expressão lógica, $x \geq 2$ é False, então a avaliação para no and. Na segunda expressão lógica, $x \geq 2$ é True mas $y \neq 0$ é False então nunca chegamos a avaliar a expressão (x/y) .

Na terceira expressão lógica, o $y \neq 0$ encontra-se *depois* do cálculo (x/y) de modo que a expressão termina com um erro.

Na segunda expressão, dizemos que $y \neq 0$ atua como um **guard** para garantir que só executaremos (x/y) se y for diferente de zero.

3.9 Depuração

O Python traceback é exibido quando ocorre um erro, ele contém diversas informações, mas pode ser um pouco confuso com tantos dados. A maioria das informações úteis geralmente são:

- Que tipo de erro ocorreu, e
- Onde ocorreu.

Erros de sintaxe geralmente são fáceis de encontrar, mas há algumas pegadinhas. Erros por espaço em branco podem ser difíceis, porque os espaços e tabs são invisíveis e geralmente os ignoramos.

```
>>> x = 5
>>> y = 6
      File "<stdin>", line 1
        y = 6
        ^
SyntaxError: invalid syntax
```

Neste exemplo, o problema é que a segunda linha é indentada por um espaço. Mas a mensagem de erro aponta para `y`, que é enganosa. Em geral, as mensagens de erro indicam onde o problema foi descoberto, mas o erro real pode estar no início do código, às vezes em uma linha anterior.

O mesmo ocorre para erros de execução. Suponha que você está tentando calcular uma relação sinal-ruído em decibéis. A fórmula é $SNR_{db} = 10 \log_{10}(P_{signal}/P_{noise})$. Em Python, você pode escrever algo como isto:

```
import math
signal_power = 9
noise_power = 10
ratio = signal_power / noise_power
decibels = 10 * math.log10(ratio)
print decibels
```

Mas quando você executá-lo, você recebe uma mensagem de erro ²:

```
Traceback (most recent call last):
  File "snr.py", line 5, in ?
    decibels = 10 * math.log10(ratio)
OverflowError: math range error
```

²Em Python 3.0, você não recebe uma mensagem de erro; o operador de divisão executa a divisão de ponto flutuante, mesmo com operandos do tipo inteiro.

A mensagem de erro indica a linha 5, mas não há nada errado com essa linha. Para encontrar o verdadeiro erro, pode ser útil imprimir o valor da variável `ratio`, que daria 0. O problema está na linha 4, porque dividir dois inteiros causa “floor division”. A solução é representar a potência do sinal e potência de ruído com valores de ponto flutuante.

Em geral, mensagens de erro dizem onde o problema foi descoberto, mas frequentemente não dizem onde ele foi causado.

3.10 Glossário

body: Uma sequência de instruções dentro de uma instrução composta

boolean expression: Uma expressão cujo valor é `True` ou `False`.

branch: Uma das sequências alternativas de instruções em uma instrução condicional.

condicional encadeada: Uma instrução condicional com uma série de ramificações alternativas.

operador de comparação: Um dos operadores que compara seus operandos: `==`, `!=`, `>`, `<`, `>=`, and `<=`.

instrução condicional: Uma declaração que controla o fluxo de execução dependendo de alguma condição

condição: A expressão booleana em uma declaração condicional que determina qual a condição é executado.

instrução composta: Uma declaração que consiste de um cabeçalho e um corpo. O cabeçalho termina com dois pontos (`:`). O corpo é indentado em relação ao cabeçalho.

guardian pattern: Onde nós construímos uma expressão lógica com comparações adicionais para aproveitar o comportamento de short-circuit.

operador lógico: Um dos operadores que combina expressões booleanas: `and`, `or`, e `not`.

condicional aninhada: Uma instrução condicional que aparece em um dos ramos de uma outra instrução condicional.

traceback: Uma lista das funções que estão em execução, impressa quando ocorre uma exceção.

short circuit: Quando o Python deixa de avaliar uma expressão lógica até o final e para porque já sabe o valor final para a expressão sem a necessidade de avaliar o resto da expressão.

3.11 Exercícios

Exercício 3.1 Reescrever o seu cálculo de pagamento para dar ao trabalhador 1.5 vezes o valor da hora para horas trabalhadas acima de 40 horas.

Digite as Horas: 45

Digite a Taxa: 10

Pagamento: 475.0

Exercício 3.2 Reescrever seu programa de pagamento usando `try` e `except` para que o programa trate entradas não numérica amigavelmente imprimindo uma mensagem e saindo do programa. A seguir mostra duas execuções do programa:

Digite as Horas: 20

Digite a Taxa: nove

Erro, por favor, informe entrada numérica

Digite as Horas: quarenta

Erro, por favor, informe entrada numérica

Exercício 3.3 Escreva um programa para solicitar uma pontuação entre 0.0 e 1.0. Se o resultado estiver fora da faixa, imprimir uma mensagem de erro. Se a pontuação estiver entre 0.0 e 1.0, imprimir uma nota utilizando a seguinte tabela:

Ponto	Nota
≥ 0.9	A
≥ 0.8	B
≥ 0.7	C
≥ 0.6	D
< 0.6	F

Digite a Pontuação: 0.95

A

Digite a Pontuação: perfeito

Pontuação incorreta

Digite a Pontuação: 10.0

Pontuação incorreta

Digite a Pontuação: 0.75

C

Digite a Pontuação: 0.5

F

Executar o programa repetidamente, como mostrado acima, para testar os diversos resultados para as diferentes entradas.

Capítulo 4

Dicionários

Um **dicionário** é como uma lista, porém mais abrangente. Em uma lista, os índices devem ser valores inteiros; em um dicionário, os índices podem ser de qualquer tipo (praticamente).

Pode-se considerar um dicionário como um mapeamento entre um conjunto de índices (chamados de **chaves**) e um conjunto de valores. Cada chave é mapeada a um valor. A associação entre uma chave e um valor é chamada de **par chave-valor** ou também como um **item**.

Como exemplo, construiremos um dicionário que mapeia palavras inglesas para palavras em espanhol, portanto chaves e valores são strings.

A função `dict` cria um novo dicionário sem itens. Pelo fato de `dict` ser o nome de uma função padrão da linguagem, esse termo não pode ser usado como nome de variável.

```
>>> eng2ptbr = dict()
>>> print eng2ptbr
{}
```

Os caracteres chaves, `{ }`, representam um dicionário vazio. Colchetes podem ser utilizados para adicionar itens ao dicionário:

```
>>> eng2ptbr['one'] = 'um'
```

Esta linha cria um item que mapeia da chave `'one'` para o valor `'um'`. Se exibirmos o dicionário novamente, veremos um par chave-valor com o caractere dois-pontos entre a chave e o valor:

```
>>> print eng2ptbr
{'one': 'um'}
```

Esse formato de saída também é um formato de entrada. Por exemplo, pode-se criar um novo dicionário com três itens:

```
>>> eng2ptbr = {'one': 'um', 'two': 'dois', 'three': 'tres'}
```

Mas se exibirmos `eng2ptbr`, podemos nos surpreender:

```
>>> print eng2ptbr
{'one': 'um', 'three': 'tres', 'two': 'dois'}
```

A ordem dos pares chave-valor não é a mesma. De fato, se esse mesmo exemplo for executado em outro computador, um resultado diferente pode ser obtido. Em linhas gerais, a ordem dos elementos em um dicionário é imprevisível.

Entretanto, isso não é um problema, uma vez que os elementos de um dicionário nunca são indexados por índices inteiros. Ao invés disso, usa-se as chaves para se buscar os valores correspondentes:

```
>>> print eng2ptbr['two']
'dois'
```

A ordem dos itens não importa, já que a chave `'two'` sempre é mapeada ao valor `'dois'`.

Se a chave não está no dicionário, uma exceção é levantada:

```
>>> print eng2ptbr['four']
KeyError: 'four'
```

A função `len` também pode ser usada em dicionários; ela devolve o número de pares chave-valor:

```
>>> len(eng2ptbr)
3
```

Pode-se utilizar o operador `in` para se verificar se algo está representado como uma *chave* no dicionário (não serve para verificar diretamente a presença de um valor).

```
>>> 'one' in eng2ptbr
True
>>> 'um' in eng2ptbr
False
```

Para verificar se algo está representado como um valor no dicionário, pode-se usar o método `values`, o qual devolve os valores como uma lista e, desse modo, o operador `in` pode ser usado:

```
>>> vals = eng2ptbr.values()
>>> 'um' in vals
True
```

O operador `in` usa algoritmos diferentes para listas e dicionários. Para listas é usado um algoritmo de busca linear. Conforme o tamanho da lista aumenta, o tempo de busca aumenta de maneira diretamente proporcional ao tamanho da lista. Para dicionários, Python usa um algoritmo chamado **tabela de hash**, a qual possui

uma propriedade notável—o operador `in` consome a mesma quantidade de tempo para se realizar a busca independente do número de itens existente no dicionário. Aqui não será explicado o porquê das funções de hash serem tão mágicas, mas informações adicionais sobre esse assunto podem ser lidas em pt.wikipedia.org/wiki/Tabela_de_disperso.

Exercício 4.1 Escreva um programa que leia as palavras do arquivo `words.txt` e armazene-as como chaves em um dicionário. Os valores não importam. Então, use o operador `in` como uma maneira rápida de verificar se uma string está no dicionário.

4.1 Dicionário como um conjunto de contagens

Suponha que dada uma string deseja-se saber quantas vezes aparece cada letra. Há várias maneiras para que isso seja feito:

1. Poderiam ser criadas 26 variáveis, cada uma contendo uma letra do alfabeto. Então, a string poderia ser travessada e, para cada caractere, seria incrementado o contador correspondente, provavelmente utilizando-se operadores condicionais encadeado.
2. Poderia ser criada uma lista com 26 elementos. Assim, cada caractere poderia ser convertido em um número (usando a função embutida `ord`), o qual seria usado como um índice na lista, e se incrementaria o contador apropriado.
3. Poderia ser criado um dicionário, onde os caracteres são as chaves e os valores são as contagens correspondentes. Ao se encontrar um caractere pela primeira vez, um item é adicionado ao dicionário. Em seguida, o valor de um dado item seria incrementado.

Essas opções realizam a mesma computação, porém cada uma a implementa de um modo diferente.

Uma **implementação** é um modo de se executar uma computação; algumas implementações são melhores do que outras. Por exemplo, uma das vantagens de se utilizar a implementação com dicionário é que não há a necessidade de se saber de antemão quais letras aparecem na string, sendo que as letras serão adicionadas ao dicionário conforme for demandado.

Eis como o código ficaria:

```
word = 'brontosaurus'
d = dict()
for c in word:
    if c not in d:
        d[c] = 1
```

```
    else:
        d[c] = d[c] + 1
print d
```

De fato está sendo construído um **histograma**, que é um termo estatístico para um conjunto de contagens (ou frequências).

O laço `for` caminha por toda a string. Em cada iteração, se o caractere `c` não está no dicionário, cria-se um novo item com chave `c` e valor inicial 1 (já que essa letra foi encontrada um vez). Se `c` já está no dicionário, o valor `d[c]` é incrementado.

Eis a saída do programa:

```
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

O histograma indica que as letras 'a' e 'b' aparecem uma vez; 'o' aparece duas vezes, e assim por diante.

Dicionários têm um método chamado `get`, que recebe como argumento uma chave e um valor padrão. Se a chave se encontra no dicionário, `get` devolve o valor correspondente; caso contrário, devolve o valor padrão. Por exemplo:

```
>>> counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
>>> print counts.get('jan', 0)
100
>>> print counts.get('tim', 0)
0
```

O método `get` pode ser usado para escrever o histograma de maneira mais concisa. Pelo fato de `get` automaticamente lidar com a ausência de uma chave no dicionário, quatro linhas de código podem ser reduzidas para uma e o bloco `if` pode ser removido.

```
word = 'brontosaurus'
d = dict()
for c in word:
    d[c] = d.get(c,0) + 1
print d
```

O uso do método `get` para simplificar esse laço de contagem é um “idiomatismo” comum em Python e será usado diversas vezes no decorrer do livro. Desse modo, vale a pena dedicar um tempo e comparar o laço usando `if` e o operador `in` com o laço usando o método `get`. Eles fazem exatamente a mesma coisa, mas o segundo é mais sucinto.

4.2 Dicionários e arquivos

Um dos usos comuns de dicionários é na contagem da ocorrência de palavras em arquivos de texto. Começemos com um arquivo muito simples contendo palavras extraídas de *Romeu e Julieta*.

Para os primeiros exemplos, usaremos uma versão mais curta e simplificada do texto, sem pontuações. Em seguida, trabalharemos com o texto da cena com as pontuações incluídas.

```
But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

Escreveremos um programa em Python, que lerá as linhas do arquivo, transformará cada linha em uma lista de palavras e, então, iterará sobre cada palavra na linha contando-a usando um dicionário.

Veremos que temos dois laços `for`. O laço externo lê as linhas do arquivo e o interno percorre cada palavra de uma linha em particular. Este é um exemplo de um padrão chamado **laços aninhados** porque um dos laços é *externo* e o outro é *interno*.

Pelo fato do laço interno executar todas suas iterações para cada uma que o laço externo faz, diz-se que o laço interno itera “mais rapidamente” ao passo que o externo itera mais lentamente.

A combinação dos laços aninhados garante que contaremos todas as palavra de todas as linhas do arquivo de entrada.

```
fname = raw_input('Digite o nome do arquivo: ')
try:
    fhand = open(fname)
except:
    print 'Arquivo nao pode ser aberto:', fname
    exit()

counts = dict()
for line in fhand:
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1

print counts
```

Quando rodamos o programa, vemos o resultado bruto das contagens de modo não sorteado. (o arquivo `romeo.txt` está disponível em www.py4inf.com/code/romeo.txt)

```
python count1.py
Digite o nome do arquivo: romeo.txt
{'and': 3, 'envious': 1, 'already': 1, 'fair': 1,
'is': 3, 'through': 1, 'pale': 1, 'yonder': 1,
'what': 1, 'sun': 2, 'Who': 1, 'But': 1, 'moon': 1,
'window': 1, 'sick': 1, 'east': 1, 'breaks': 1,
```

```
'grief': 1, 'with': 1, 'light': 1, 'It': 1, 'Arise': 1,
'kill': 1, 'the': 3, 'soft': 1, 'Juliet': 1}
```

É um tanto quanto inconveniente procurar visualmente em um dicionário por palavras mais comuns e suas contagens. Desse modo, precisamos adicionar mais código Python para obter um resultado que seja mais útil.

4.3 Laços de repetição e dicionário

Se um dicionário for usado como a sequência em um bloco `for`, esse iterará sobre as chaves do dicionário. Este laço exibe cada chave e o valor correspondente:

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
for key in counts:
    print key, counts[key]
```

Que resulta em:

```
jan 100
chuck 1
annie 42
```

Mais uma vez, as chaves não respeitam nenhum tipo de ordenamento.

Podemos usar este padrão para implementar os diferentes estilos de laço que foram descritos anteriormente. Por exemplo, se quiséssemos encontrar todas as entradas em um dicionário com valor acima de dez, poderíamos escrever o seguinte código:

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
for key in counts:
    if counts[key] > 10 :
        print key, counts[key]
```

O laço `for` itera pelas *chaves* do dicionário, então devemos usar o operador de índice para obter o *valor* correspondente para cada chave. Eis o resultado da execução:

```
jan 100
annie 42
```

Vemos apenas as entradas com valor acima de dez.

Para exibir as chaves em ordem alfabética, deve-se gerar uma lista das chaves do dicionário por meio do método `keys`, disponível em objetos dicionário, e então ordenar essa lista. Em seguida, itera-se pela lista ordenada, procurando cada chave e exibindo os pares chave-valor de modo ordenado, como em:

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
lst = counts.keys()
print lst
lst.sort()
for key in lst:
    print key, counts[key]
```

O que gera a seguinte saída:

```
['jan', 'chuck', 'annie']
annie 42
chuck 1
jan 100
```

Primeiramente, pode-se ver a lista não ordenada das chaves, obtida pelo método `keys`. Em seguida, vemos os pares chave-valor gerados no laço `for`.

4.4 Processamento avançado de texto

No exemplo acima, no qual usamos o arquivo `romeo.txt`, todas as pontuações foram removidas para tornar o texto o mais simples possível. O texto original possui muitas pontuações, como mostrado abaixo.

```
But, soft! what light through yonder window breaks?
It is the east, and Juliet is the sun.
Arise, fair sun, and kill the envious moon,
Who is already sick and pale with grief,
```

Uma vez que a função do Python `split` procura por espaços e trata palavras como tokens separados por espaços, as palavras “soft!” e “soft” seriam tratadas como *diferentes* e seriam criadas entradas separadas no dicionário para cada uma delas.

Além disso, como o arquivos possui letras capitalizadas, as palavras “who” e “Who” seriam tratadas como diferentes e teriam contagens diferentes.

Podemos solucionar ambos os problemas usando os métodos de string `lower`, `punctuation` e `translate`. Dentre esses três o método `translate` é o mais complexo. Eis a documentação para `translate`:

```
string.translate(s, table[, deletechars])
```

Deleta todos os caracteres de s que estão em deletechars (se presente) e traduz os caracteres usando table, que deve ser uma string com o comprimento de 256 caracteres fornecendo a tradução para cada valor de caractere, indexado pela sua posição. Se table é None, então apenas a deleção de caracteres é realizada.

Não iremos especificar o parâmetro `table`, mas iremos usar `deletechars` para deletar todas as pontuações. Iremos utilizar a lista de caracteres que o próprio Python considera como “pontuação”:

```
>>> import string
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

Faremos a seguinte modificação em nosso programa:

```

import string                                     # New Code

fname = raw_input('Digite o nome do arquivo: ')
try:
    fhand = open(fname)
except:
    print 'Arquivo nao pode ser aberto:', fname
    exit()

counts = dict()
for line in fhand:
    line = line.translate(None, string.punctuation)    # New Code
    line = line.lower()                                # New Code
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1

print counts

```

O programa se manteve praticamente o mesmo, com a exceção de que usamos `translate` para remover todas as pontuações e `lower` para tornar a linha em caixa baixa. Note que para Python 2.5 e versões anteriores, `translate` não aceita `None` como primeiro parâmetro. Então, use este código para chamar `translate`:

```
print a.translate(string.maketrans(' ', ' '), string.punctuation)
```

Parte de aprender a “Arte do Python” ou “Pensar pythonicamente” está em perceber que Python geralmente tem capacidades embutidas para analisar muitos dados de problemas comuns. No decorrer do tempo, vê-se exemplos de código e documentação suficientes para se saber onde procurar para ver se alguém já escreveu alguma coisa que faça seu trabalho mais fácil.

A seguir está uma versão abreviada da saída:

```

Digite o nome do arquivo: romeo-full.txt
{'swearst': 1, 'all': 6, 'afeard': 1, 'leave': 2, 'these': 2,
'kinsmen': 2, 'what': 11, 'thinkst': 1, 'love': 24, 'cloak': 1,
a': 24, 'orchard': 2, 'light': 5, 'lovers': 2, 'romeo': 40,
'maiden': 1, 'whiteupturned': 1, 'juliet': 32, 'gentleman': 1,
'it': 22, 'leans': 1, 'canst': 1, 'having': 1, ...}

```

Buscar informações nessa saída ainda é difícil e podemos usar Python para nos fornecer exatamente o que estamos procurando; contudo, para tanto, precisamos aprender sobre as **tuplas** do Python. Retornaremos a esse exemplo uma vez que aprendermos sobre tuplas.

4.5 Depuração

Conforme se trabalha com conjuntos de dados maiores, pode ser difícil de depurá-los por exibição e checagem à mão. Eis algumas sugestões para depuração de conjuntos de dados grandes:

Reduza a entrada: Se possível, reduza o tamanho do conjunto de dados. Por exemplo, se o programa lê um arquivo de texto, comece com apenas 10 linhas, ou com o menor exemplo que pode ser construído. Pode-se ainda editar os próprios arquivos, ou (melhor) modificar o programa de tal modo a ler apenas as n linhas.

Se houver um erro, pode-se reduzir n até o menor valor que manifesta o erro, e, então, aumentá-lo gradualmente conforme se encontra e se corrige os erros.

Verificar sumários e tipos: Ao invés de exibir e verificar o conjunto de dados por completo, considera-se exibir sumarizações dos dados: por exemplo, o número de itens em um dicionário ou o total de uma lista de números.

Valores que não são do tipo correto são uma causa comum de erros de execução. Para depurar esse tipo de erro, geralmente basta exibir o tipo dos valores em questão.

Escreva auto-verificações: Há momentos em que se pode escrever código para verificar erros automaticamente. Por exemplo, se está calculando-se a média de uma lista de números, pode-se verificar se o resultado não é maior que o maior valor na lista nem menor que o menor valor. Isso é chamado de “verificação de sanidade” porque ele detecta resultados que sejam “completamente ilógicos”.

Há outro tipo de teste que compara resultados de duas computações diferentes para ver se esses são consistentes. Tal verificação é chamada de “verificação de consistência”

Exiba saídas de maneira aprazível: Formatar a saída da depuração pode fazer com que seja mais fácil de se detectar erros.

Novamente, tempo gasto construindo arcabouços pode reduzir o tempo gasto com depuração.

4.6 Glossário

busca: Uma operação de dicionário que encontra um valor a partir de uma dada chave.

chave: Um objeto que aparece em um dicionário como a primeira parte de um par chave-valor.

dicionário: Um mapeamento entre um conjunto de chaves e seus valores correspondentes.

função de hash: A função usada por uma tabela de hash para calcular a posição de uma chave.

histograma: Um conjunto de contagens.

implementação: Uma maneira de se realizar uma computação.

item: Outro nome para um par chave-valor.

laços aninhados: Quando há um ou mais laços “dentro” de outro laço. O laço interno é executado completamente para cada execução do laço externo.

par chave-valor: A representação de um mapeamento de uma chave a um valor.

tabela de hash: O algoritmo usado para implementar os dicionários de Python.

valor: Um objeto que aparece em um dicionário como a segunda parte em um par chave-valor. Esse é mais específico do que nosso uso anterior da palavra “valor”.

4.7 Exercícios

Exercício 4.2 Escreva um programa que categorize cada mensagem de e-mail pelo dia da semana que o commit (<https://pt.wikipedia.org/wiki/Commit>) foi feito. Para tanto, procure por linhas que comecem com “From”, então busque pela terceira palavra e mantenha um procedimento de contagem para cada dia da semana. Ao final do programa, exiba o conteúdo do dicionário (ordem não importa).

Amostra de linha:

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
```

Amostra de execução:

```
python dow.py
```

```
Enter a file name: mbox-short.txt
```

```
{'Fri': 20, 'Thu': 6, 'Sat': 1}
```

Exercício 4.3 Escreva um programa que leia um log (https://pt.wikipedia.org/wiki/Log_de_dados) de correio eletrônico, escreva um histograma usando um dicionário para contar quantas mensagens vieram de cada endereço de e-mail e, por fim, exiba o dicionário.

```
Enter file name: mbox-short.txt
```

```
{'gopal.ramasammycook@gmail.com': 1, 'louis@media.berkeley.edu': 3,  
'cwen@iupui.edu': 5, 'antranig@caret.cam.ac.uk': 1,  
'rjlowe@iupui.edu': 2, 'gsilver@umich.edu': 3,
```



```
'david.horwitz@uct.ac.za': 4, 'wagnermr@iupui.edu': 1,  
'zqian@umich.edu': 4, 'stephen.marquard@uct.ac.za': 2,  
'ray@media.berkeley.edu': 1}
```

Exercício 4.4 Insira código no programa acima para descobrir quem tem mais mensagens no arquivo.

Após todos os dados terem sido lidos e o dicionário criado, percorra o dicionário usando um laço de máximo (veja Sessão ??) para encontrar quem tem mais mensagens e exiba quantas mensagens existem para essa pessoa.

```
Enter a file name: mbox-short.txt  
cwen@iupui.edu 5
```

```
Enter a file name: mbox.txt  
zqian@umich.edu 195
```

Exercício 4.5 Este programa leva em consideração o nome do domínio (ao invés do endereço) de onde a mensagem foi mandada e não de quem essa veio (isto é, o endereço de e-mail inteiro). Ao final do programa, exiba o conteúdo do dicionário.

```
python schoolcount.py  
Enter a file name: mbox-short.txt  
{ 'media.berkeley.edu': 4, 'uct.ac.za': 6, 'umich.edu': 7,  
'gmail.com': 1, 'caret.cam.ac.uk': 1, 'iupui.edu': 8}
```

G

Capítulo 5

Banco de Dados e Structured Query Language (SQL)

5.1 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 organizados 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 **índices** assim que o dado é adicionado, isto permite ao computador acessar rapidamente uma entrada em particular.

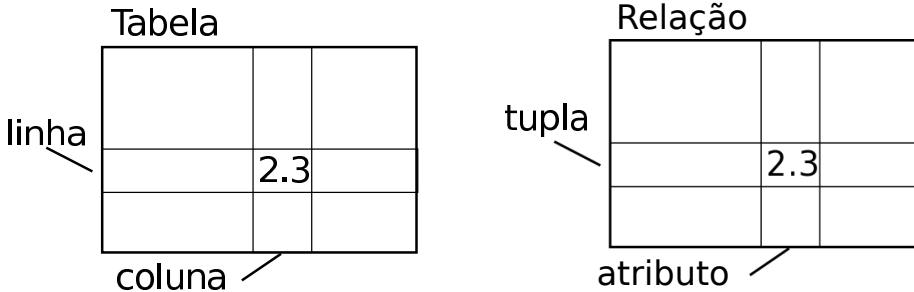
Existem diferentes tipos de sistemas de bancos de dados que são utilizados para diferentes propósitos, alguns destes são: Oracle, MySQL, Microsoft SQL Server, PostgreSQL, e SQLite. Focaremos no uso do SQLite neste livro pois é um banco de dados comum e já está integrado ao Python. O SQLite foi desenvolvido 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.

5.2 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.

5.3 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.

5.4 Criado uma tabela em um banco de dados

Bancos de dados precisam de estruturas mais bem definidas do que listas ou dicionários em Python¹.

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()

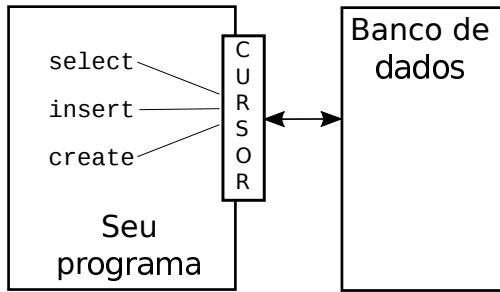
cur.execute('DROP TABLE IF EXISTS Tracks ')
cur.execute('CREATE TABLE Tracks (title TEXT, plays INTEGER)')

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 **cursor** 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.

¹Atualmente 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()`.

²Em 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.

Faixas

título	tocadas
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 execução 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 construída 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.

5.5 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 portátil 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, opcional, 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.

5.6 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 rastreamos 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 itens `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()

cur.close()

```

```

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 declaração `UPDATE` com a cláusula `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 cláusula 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, simplesmente 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.

5.7 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*, inserirmos 20 registros com “*drchuck*” como primeiro parâmetro e assim acabaremos duplicando a *string* muitas vezes no banco de dados.

Esta duplicação de dados viola uma das melhores práticas da **normalização 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 processador 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)
```

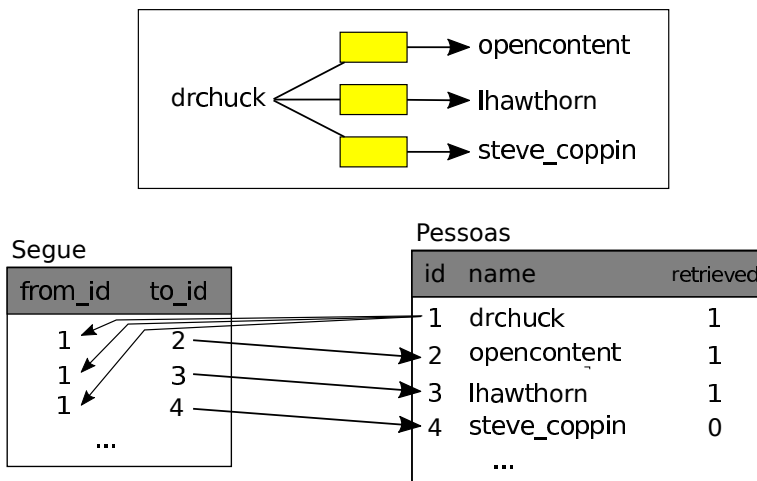
Perceba que nós não estamos mais mantendo uma conta de amigo em cada linha da tabela `People`. Quando selecionamos `INTEGER PRIMARY KEY` como o tipo da nossa coluna `id`, estamos indicando que gostaríamos que o SQLite gerencie esta coluna e defina uma chave numérica única automaticamente para cada linha 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 conveniência no nosso programa, como veremos a seguir. As regras nos impede de cometer enganos e facilita na escrita dos nossos códigos.

Em essência, 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.



5.8 Programando com múltiplas tabelas

Agora nós iremos refazer o programa de rastreamento do Twitter utilizando duas tabelas, as chaves primárias, e as chaves de referências estão descritas anteriormente. Abaixo está o código da nova versão do programa:

```
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()

```

Este programa está começando a ficar um pouco complicado, mas ilustra os padrões que precisamos para utilizar quando estamos usando chaves inteiras para conectar as tabelas. Os padrões básicos são:

1. Criar tabelas com chaves primárias e restrições.
2. Quando temos uma chave lógica para uma pessoa (i.e., conta) e precisamos do valor de `id` para a pessoa, dependendo se a pessoa já está na tabela `People` ou não, também precisaremos de: (1) olhar para a pessoa na tabela `People` e recuperar o valor de `id` da pessoa, ou (2) adicionar a pessoa na tabela `People` e pegar o valor de `id` para a nova linha recém adicionada.
3. Inserir a linha que captura a relação com “segue”.

Vamos tratar cada um dos ítems acima, em partes.

5.8.1 Restrições em uma tabela

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 our tables. When we create our tables: Assim como projetamos a estrutura da tabela, podemos informar ao banco de dados

que gostaríamos de reforçar algumas regras. Estas regras nos ajudam a não cometer enganos e a não inserir dados incorretos nas nossas tabelas. Quando criamos nossas tabelas:

```
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))''')
```

Indicamos que a coluna `name` na tabela `People` deve ser `UNIQUE`. Também indicaremos que a combinação dos dois números em cada linha da tabela `Follows` devem ser únicos. Estas restrições nos mantém longe da possibilidade de cometer enganos como adicionar a mesma relação mais de uma vez.

Podemos obter vantagens destas restrições conforme mostra o seguinte código:

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

Adicionamos a condição `OR IGNORE` a declaração `INSERT` para indicar que se este `INSERT` em particular pode causar uma violação para a regra “`name` deve ser único”, assim o banco de dados tem permissão de ignorar o `INSERT`.

De forma similar, o seguinte código garante que não adicionemos a mesma relação `Follows` duas vezes.

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

Novamente, nós simplesmente dizemos para o banco de dados ignorar nossa tentativa de inserir `INSERT` se isto violar a regra de exclusividade que especificamos para a linha `Follows`.

5.8.2 Restaurar e/ou inserir um registro

Quando solicitamos ao usuário uma conta do Twitter, se a conta existir, precisamos verificar o valor do `id`. Se a conta não existir ainda na tabela `People`, devemos inserir o registro e pegar o valor do `id` da linha inserida.

Isto é um padrão muito comum e é feito duas vezes no programa acima. Este código mostra como verificamos o `id` da conta de um amigo, quando extraímos um `screen_name` de um nó de `user` recuperado do JSON do Twitter.

Ao longo do tempo será cada vez mais provável que a conta já esteja registrada no banco de dados, então primeiro checamos para ver se o registro existe em `People` utilizando uma declaração de `SELECT`.

Se tudo estiver certo³ dentro da seção `try`, recuperamos o registro usando `fetchone()` e depois recuperar o primeiro (e somente o primeiro) elemento da tupla que retornou e a armazenamos em `friend_id`.

³Em geral, quando uma sentença inicia com “se tudo estiver certo” você verá que o código precisa utilizar a condição `try/except`.

Se o `SELECT` falhar, o código de `fetchone()[0]` falhará e o controle irá mudar para a seção `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
```

Se terminar no código do `except`, isto significa que aquela linha não foi encontrada, então devemos inseri-la na linha. Usamos o `INSERT OR IGNORE` somente para evitar erros e depois chamamos `commit()` para forçar que a base de dados seja atualizada. Depois que a escrita esteja completa, nós podemos checar, com `cur.rowcount`, para ver quantas linhas foram afetadas. Uma vez que estamos tentando inserir uma simples linha, se o número de linhas afetadas é alguma coisa diferente de 1, isto é um erro.

Se o `INSERT` for executado com sucesso, nós podemos verificar, através do `cur.lastrowid` para descobrir qual valor o banco de dados associou na coluna `id` na nossa nova linha.

5.8.3 Armazenando a conexão do amigo

Uma vez que sabemos o valor da chave para o usuário do Twitter e o amigo extraído do JSON, simplesmente inserimos os dois números dentro da tabela `Follows` com o seguinte código:

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

Note que deixamos o banco de dados cuidar para nós de realizar a “inserção-dupla” da conexão criando a tabela com a restrição única e depois adicionando `OR IGNORE` a nossa condição de `INSERT`.

Esta é um exemplo de execução deste programa:

```
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
```

Nós iniciamos com a conta `drchuck` e depois deixamos o programa pegar automaticamente as próximas duas contas para recuperar e adicionar a nossa base de dados.

Abaixo estão as primeiras linhas das tabelas `People` e `Follows` depois que esta execução é finalizada:

```
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.
```

Você também pode ver os campos `id`, `name` e `visited` na tabela `People` e os números dos finais das conexões na tabela `Follows`. Na tabela `People`, podemos ver que as primeiras três pessoas foram visitadas e seus dados foram recuperados. Os dados na tabela `Follows` indica que `drchuck` (usuário 1) é amigo de todas as pessoas mostradas nas primeiras cinco linhas. Isto mostra que o primeiro dado que recuperamos e armazenamos foram dos amigos do `drchuck`. Se você mostrou mais linhas da tabela `Follows`, você poderá ver os amigos dos usuários 2 e 3.

5.9 Três tipos de chaves

Agora que iniciamos a construção de um modelo de dados colocando nossos dados dentro de múltiplas tabelas e linhas conectadas nestas tabelas utilizando **chaves**, precisamos olhar para algumas terminologias sobre as chaves. Existem genericamente três tipos de chaves que são utilizadas em um banco de dados.

- Um **chave lógica** é uma chave que o “mundo real” pode usar para checar uma linha. Na nossa modelagem, o campo `name` é uma chave lógica. Ele é o nome que usamos para o usuário e nós na verdade para uma linha de usuário diversas vezes no nosso programa usando o campo `name`. Você perceberá

que faz sentido adicionar a restrição de `UNIQUE` para uma chave lógica. Uma vez que é através de chaves lógicas que consultamos uma linha do mundo exterior, faz um pouco de sentido permitir que múltiplas linhas tenham o mesmo valor em uma tabela.

- Um **chave primária** é usualmente um número que é associado automaticamente por um banco de dados. Geralmente não terá significado fora do programa e só é utilizada para conectar as linhas de diferentes tabelas. Quando queremos verificar uma linha em uma tabela, normalmente buscamos pela linha utilizando a chave primária, é a forma mais rápida de encontrar uma linha. Uma vez que chaves primárias são números inteiros, eles ocupam pouco espaço e podem ser comparados ou ordenados rapidamente. No nosso modelo, o campo `id` é um exemplo de chave primária.
- Uma **chave estrangeira** é normalmente um número que aponta para a chave primária associada a uma linha em outra tabela. Um exemplo de chave estrangeira no nosso modelo é o campo `from_id`.

Nós estamos utilizando uma convenção de sempre chamar uma chave primária de um campo `id` e adicionando o sufixo `_id` para qualquer campo que seja uma chave estrangeira.

5.10 Utilizando o JOIN para recuperar informações

Agora que seguimos as regras de normalização de bancos de dados e temos os dados separados em duas tabelas, associadas através de chaves primárias e chaves estrangeiras, precisamos ser capazes de construir uma chamada de `SELECT` que reagrupa os dados em toda as tabelas.

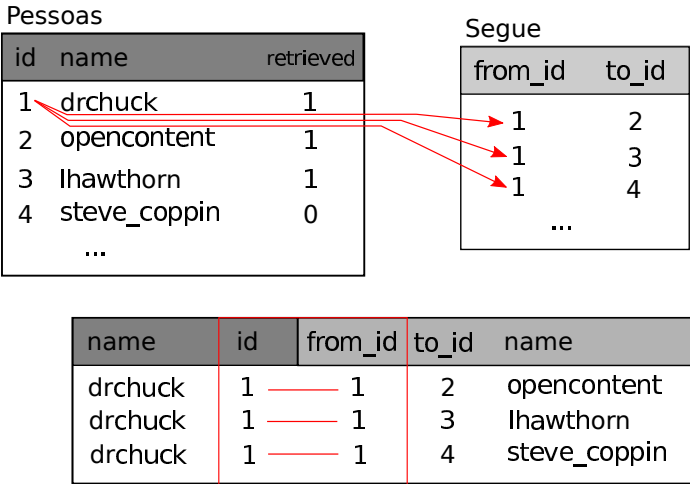
SQL utiliza a cláusula `JOIN` para reconectar estas tabelas. Na cláusula `JOIN` você especifica o campo que serão utilizados para reconectar as linhas entre as tabelas.

Abaixo, um exemplo de um `SELECT` com um `JOIN`:

```
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.

O `JOIN` indica que o campo que utilizamos são selecionados cruzando ambas as tabelas `Follows` e `People`. A cláusula `ON` indica como as duas tabelas devem ser unidas: Junta as linhas da tabela `Follows` às da tabela `People` onde o campo `from_id` em `Follows` tem o mesmo valor do campo `id` na tabela `People`.



O resultado do JOIN cria uma super “metalinha” que tem os dois campos da tabela People que casam com os campos da tabela Follows. Onde existir mais de uma ocorrência entre o campo id e o from_id da tabela People, então o JOIN cria uma “metalinha” para cada par de linhas que correspondem, duplicando os dados conforme for necessário.

O seguinte código demonstra o dado que nós teremos no banco de dados após o executar o programa coletor de dados (acima) diversas vezes.

```
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()
```

Neste programa, primeiro descarregamos a tabela `People` e `Follows` e depois descarregamos um subconjunto de dados das tabelas juntos.

Aqui temos a saída do programa:

```
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.
```

Você vê as colunas das tabelas `People` e `Follows` e por último conjunto de linhas é o resultado do `SELECT` com o `JOIN`.

No último `SELECT`, nós estamos procurando por contas que tem amigos de “conteúdo aberto” (i.e., `People.id=2`).

Em cada uma das “meta-linhas” da última seleção, as primeiras duas colunas são da tabela `Follows` seguidas pelas colunas três até cinco da tabela `People`. Você também pode ver que a segunda coluna (`Follows.to_id`) relaciona a terceira coluna (`People.id`) em cada uma das “meta-linha” que foram juntas.

5.11 Sumário

Este capítulo cobriu os fundamentos para o uso, básico, de banco de dados no Python. É muito mais complicado escrever código para usar um banco de dados para armazenar informações do que dicionários ou arquivos com Python, então se existe poucas razões para utilizar um banco de dados, a menos que sua aplicação realmente precise das capacidades de um banco de dados. As situações onde um banco de dados podem ser muito úteis são: (1) quando sua aplicação precisa realizar pequenas atualizações com um conjunto grande de dados, (2) quando seus

dados são tão grandes que não podem ser armazenados em um dicionário e você precisa acessar estas informações repetidas vezes, ou (3) quando você tem um processo de execução demorada e você quer ser capaz de parar e recomeçar e manter os dados entre as pesquisas.

Você pode construir um banco de dados simples com uma única tabela para atender muitas aplicações, mas a maioria dos problemas vão necessitar de várias tabelas e conexões/relações entre as linhas em diferentes tabelas. Quando você começar a fazer relações entre as tabelas, é importante fazer algum planejamento e seguir as regras de normatização de banco de dados para fazer um melhor uso das capacidades dos bancos de dados. Uma vez que a principal motivação para utilizar um banco de dados é que você tem um grande conjunto de dados para tratar, é importante modelar os dados eficientemente, assim seus programas poderão rodar tão rápido quanto for possível.

5.12 Depuração

É comum quando você está desenvolvendo um programa em Python para se conectar em um banco de dados SQLite, é executar um programa para checar os resultados utilizando o Navegador SQLite. O navegador permitirá que rapidamente verifique se o seu programa está funcionando corretamente.

Você deve ter cuidado, porque o SQLite cuida para que dois programas não façam modificações nos dados ao mesmo tempo. Por exemplo, se você abrir um banco de dados no navegador e faz alterações no banco de dados, enquanto não pressionar o botão “salvar”, o navegador “trava” o arquivo do banco de dados e impede qualquer outro programa de acessar o arquivo. Desta forma seu programa em Python não conseguirá acessar o arquivo se ele estiver travado.

Então, uma solução é garantir que fechou o navegador ou utilizar o menu **Arquivo** para fechar o banco de dados no navegador antes de tentar acessar o banco de dados através do Python, evitando problemas no seu código porque o banco de dados está travado.

5.13 Glossário

atributo: Um dos valores dentro de uma tupla. Comumente chamado de “coluna” ou “campo”.

restrição: Quando ordenamos a um banco de dados para reforçar uma regra em um campo ou em uma linha na tabela. Uma restrição comum é insistir que não pode haver valores duplicados em um campo em particular (i.e., todos os valores tem que ser únicos).

cursor: Um cursor permite execução de um comando SQL em um banco de dados e recuperar informações de um banco de dados. Um cursor é similar a um *socket* ou identificador de arquivos para uma conexão de rede e arquivos, respectivamente.

navegador de banco de dados: um conjunto de *software* que permite se conectar diretamente a um banco de dados e manipulá-lo diretamente, sem escrever um programa.

chave estrangeira: Uma chave numérica que aponta para uma chave primária de uma linha em outra tabela. Chaves estrangeiras estabelecem relações entre linhas armazenadas em diferentes tabelas.

índice: Dados adicionais que um banco de dados mantém, como linhas e inserções dentro de uma tabela para realizar consultas mais rápido.

chave lógica: Uma chave que o “mundo externo” utiliza para consultar uma linha em particular. Por exemplo em uma tabela de contas de usuários, o email de uma pessoa pode ser um bom candidato a chave lógica para a informação de usuário.

normalização: Projetar um modelo de dados para que nenhum dado seja replicado. Armazenamos cada item de dados em um lugar no banco de dados e referenciamos isso em qualquer lugar utilizando chave estrangeira.

chave primária: Uma chave numérica associada a cada linha que é usada para referenciar uma linha em uma tabela de outra tabela. Normalmente o banco de dados é configurado para automaticamente associar chaves primárias assim que linhas são inseridas.

relação: Uma área dentro do banco de dados que contém tuplas e atributos. Tipicamente chamada de “tabela”.

tupla: Uma entrada em um banco de dados que é um conjunto de atributos. Tipicamente chamado de “linha”.

Capítulo 6

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 o seu computador e executam algumas operações em seus arquivos. Arquivos são organizados em diretórios (também chamados de pastas). Scripts Python simples podem fazer o trabalho de tarefas simples que devem ser feitas em 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 ao comando `open` e nos permite escrever um laço de repetição para ler o conteúdo de um arquivo, `socket` nos permite escrever um laço de repetição para ler o conteúdo de uma conexão e `urllib` nos permite abrir um documento web e navegar por meio de um laço de repetição no seu conteúdo.

6.1 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 leitura, o Python o procura 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 string `cwd` que identifica um arquivo é chamado de `path`. Um **caminho relativo** é relativo ao diretório atual (corrente); Um **caminho absoluto** tem início no diretório raiz do sistema de arquivos.

Os caminhos que temos visto até agora são nomes de arquivos simples, por isso são relativos ao diretório atual. Para encontrar o caminho absoluto de 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
```

Da mesma forma, `os.path.isfile` verifica se é um arquivo.

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

```
>>> os.listdir(cwd)
['musicas', 'fotos', 'memo.txt']
```

6.2 Exemplo: Limpando um diretório de fotos

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 cópias 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 e-mail. Eu armazenei essas mensagens em um arquivo de texto no mesmo diretório do arquivo das imagens. Eu criei uma estrutura de diretórios baseada no mês, ano, dia e hora no qual 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 mensagem quebrou e adicionou um bando de dados inúteis no meu servidor ao invés de legenda.

Eu queria passar por estes arquivos e descobrir quais dos arquivos texto eram realmente legendas e quais eram lixo e, em seguida, apagar os arquivos que eram lixo. A primeira coisa a fazer foi conseguir um simples inventário dos arquivos 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
```

O segredo para um código tão pequeno é a utilização da biblioteca `os.walk` do Python. Quando nós chamamos `os.walk` e inicializamos um diretório, ele "caminha" 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 uma lista de sub-diretórios e o terceiro valor é uma lista de arquivos no diretório corrente.

Nós não temos que procurar explicitamente dentro de cada diretório por que nós 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 próxima coisa a se 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 tanto 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 grandes e com o mesmo tamanho (2578 e 2565). Quando observamos alguns desses arquivos maiores manualmente, parece que os arquivos grandes 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>
    ...
```

Espiando o conteúdo destes arquivos, parece que não há informações importantes, então provavelmente podemos eliminá-los.

Mas antes de excluir os arquivos, vamos escrever um programa para procurar por arquivos que possuem mais de uma linha e exibir 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 são exceção em meus dados. Então, fizemos a seguinte mudança no programa para lidar com esses arquivos também.

```
lines = list()
for line in fhand:
    lines.append(line)
if len(lines) == 3 and lines[2].startswith('Sent from my iPhone'):
    continue
if len(lines) > 1:
    print len(lines), thefile
    print lines[:4]
```

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

```
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 era ”ruim” nós usamos `continue` para ignorar os maus arquivos para que pudéssemos refinar nosso código para encontrar mais padrões que eram ruins.

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 possíveis arquivos que queremos apagar e por quê esses arquivos são eleitos a exclusão. O Programa produz a seguinte saída:

```
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 nos certificar que não inserimos um bug em nosso programa ou talvez na nossa lógica, pegando arquivos que

não queríamos. Uma vez satisfeitos de que esta é a lista de arquivos que queremos excluir, faremos 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 iPhone'):
    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 irá produzir nenhuma 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 sequência onde usamos o Python primeiro para navegar através dos diretórios e arquivos procurando padrões. Usamos o Python devagar para ajudar a determinar como faríamos para limpar nosso diretório. 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 que você precisa resolver pode ser bastante simples precisando procurar pelos nomes dos arquivos, ou talvez você precise ler cada arquivo, procurando por padrões dentro dos mesmos, às vezes você precisa ler o conteúdo dos arquivos fazendo alguma mudança em alguns deles, seguindo algum tipo de critério. Todos estes são bastante simples uma vez que você entenda como `os.walk` e outros utilitários `os` podem ser usados.

6.3 Argumentos de linha de comando

Nos capítulos anteriores tivemos uma série de programas que solicitavam por um nome de arquivo usando `raw_input` e então, liam os dados de um arquivo e processavam os dados, como a seguir:

```
nome = raw_input('Informe o arquivo:')
handle = open(nome, 'r')
texto = handle.read()
...
```

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

```
python words.py
Informe o arquivo: mbox-short.txt
...
```

Nós podemos colocar strings adicionais depois do nome do arquivo Python na linha de comando e acessá-los de dentro de um programa Python. Eles são chamados **argumentos de linha de comando**. Aqui está um simples programa que demonstra a leitura de argumentos a partir de uma linha de comando:

```
import sys
print 'Contagem:', len(sys.argv)
print 'Tipo:', type(sys.argv)
for arg in sys.argv:
    print 'Argumento:', arg
```

Os conteúdos de `sys.argv` são uma lista de strings onde a primeira string contém o nome do programa Python e as outras são argumentos na linha de comando após o nome do arquivo Python.

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

```
python argtest.py ola alguem
Contagem: 3
Tipo: <type 'list'>
Argumento: argtest.py
Argumento: ola
Argumento: alguem
```

Há três argumentos que são passados ao nosso programa como uma lista de três elementos. O primeiro elemento da lista é o nome do arquivo (`argtest.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, obtendo o nome do arquivo a partir 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 pegamos o segundo argumento da linha de comando, que contém o nome do arquivo (pulando o nome do programa na entrada [0]). Nós abrimos o arquivo e lemos seu conteúdo, como segue:

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

Usar argumentos de linha de comando como entrada, torna o seu programa Python fácil de se reutilizar, especialmente quando você somente precisa passar uma ou duas strings.

6.4 Pipes

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

Qualquer programa que consiga rodar a partir do shell também pode ser executado a 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 mostra o conteúdo do diretório corrente (no modo detalhado). Você pode rodar `ls` com `os.open`:

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

Um argumento é uma string que contém um comando shell. O valor de retorno é um ponteiro para um arquivo que se comporta exatamente como um arquivo aberto. Você pode ler a saída do processo `ls` uma linha de cada vez com o comando `readline` ou obter tudo de uma vez com o comando `read`:

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

Quando terminar, você fecha o pipe como se fosse um arquivo:

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

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

¹Ao usar pipes para interagir com comandos do sistema operacional como `ls`, é importante saber qual sistema operacional você está usando e executar somente comandos pipe que são suportados pelo seu sistema operacional.

6.5 Glossário

absolute path: Uma string que descreve onde um arquivo ou diretório é armazenado, começando desde o “topo da árvore de diretórios” de modo que ele pode ser usado para acessar o arquivo ou diretório, independentemente do diretório de trabalho corrente.

checksum: Ver também **hashing**. O termo “checksum” vem da necessidade de se verificar se os dados corromperam durante o envio pelo rede ou quando gravados em um meio de backup. Quando os dados são gravados ou enviados, o sistema emissor calcula o checksum e também o envia. Quando o dado foi completamente lido ou recebido, o sistema receptor calcula novamente o checksum com base nos dados recebidos e os compara com o checksum recebido. Se os checksum’s não corresponderem, devemos assumir que os dados estão corrompidos, uma vez que já finalizou a transmissão. checksum

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

current working directory: O diretório corrente no qual você está. Você pode mudar seu diretório de trabalho usando o comando `cd`, disponível na maioria dos sistemas operacionais em sua interface de linha de comando. Quando você abre um arquivo em Python usando apenas o nome do arquivo, sem o caminho, o arquivo deve estar no diretório de trabalho atual, onde está executando o programa.

hashing: Leitura através de uma grande quantidade de dados, produzindo um checksum global para os dados. As melhores funções hash produzem muito poucas “colisões”, que é quando você passa diferentes dados para a função hash e recebe 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**, com exceção de que o pipe só pode ser usado para conectar programas em execução no mesmo computador (ou seja, não através de uma rede).
pipe

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

shell: Uma interface de linha de comando para um sistema operacional. Também chamado em alguns sistemas operacionais de “terminal”. Nesta interface, você digita um comando com parâmetros em uma única linha e pressiona “enter” para executar o comando.

walk: Um termo que usamos para descrever a noção de visitar uma árvore inteira de diretórios e sub-diretórios, até que tenhamos visitado todos eles. Nós chamamos isso de “caminhar pela árvore de diretórios”.

6.6 Exercícios

Exercício 6.1 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 exercício é procurar por essas duplicatas.

1. Escreva um programa que caminhe no diretório e em todos os seus subdiretórios, procurando por todos os arquivos com o sufixo `.mp3` e liste o par de arquivos com o mesmo tamanho. Dica: Use um dicionário onde a chave seja o tamanho do arquivo do `os.path.getsize` e o valor seja o nome do caminho concatenado com o nome do arquivo. Conforme você for encontrando cada arquivo, verifique se já tem um arquivo que tem o mesmo tamanho do arquivo atual. Se assim for, você tem um arquivo duplicado, então imprima o tamanho e os nomes dos dois arquivos (um a partir do hash e o outro a partir do arquivo que você está olhando no momento).
2. Adaptar o programa anterior para procurar arquivos com conteúdo duplicado usando um hash ou um **checksum**. Por exemplo, MD5 (Message-Digest algorithm 5) recebe uma “mensagem” grande e retorna um “checksum” de 128 bits. A probabilidade de que dois arquivos com diferentes conteúdos retornem o mesmo checksum é muito pequena.

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

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

Você deve criar um dicionário onde o checksum é a chave e o nome do arquivo é o valor. Quando você calcular um checksum e ele já existir no dicionário como uma chave, então você terá dois arquivos duplicados. Então imprima o arquivo existente no dicionário e o arquivo que você acabou de ler. Aqui estão algumas saídas de uma execução sob uma pasta com arquivos de imagens.

```
./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 envio a mesma foto mais de uma vez ou faço uma cópia de uma foto de vez em quando sem excluir a original.

Índice Remissivo

índice, 45, 80

, 90

absolute path, 82

algorithm

MD5, 91

and operador, 34

arcabouço, 53

Argumentos, 87

atribuição, 29

atributo, 79

avaliar, 23

body, 42

bool tipo, 33

boolean expression, 42

branch, 36, 42

bug, 16

busca, 53

BY-SA, iv

código de máquina, 17

código fonte, 17

caractere

chave, 45

caractere chave, 45

caractere underscore, 21

CC-BY-SA, iv

celsius, 38

chained conditional, 36

chave, 45, 53

chave estrangeira, 80

chave lógica, 80

chave primária, 80

checksum, 91

close method, 89

comentário, 26, 29

compilar, 16

concatenação, 24, 29

condição, 35, 42

condicional

aninhada, 42

aninhado, 37

encadeada, 42

condicional aninhada, 42

condicional aninhado, 37

condicional encadeada, 42

conditional

chained, 36

connect function, 59

contador, 47

conversão de temperatura, 38

CPU, 16

Creative Commons License, iv

cursor, 80

cursor function, 59

database, 57

indexes, 57

debugando, 28

debugging, 41

declaração

composta, 35

condicional, 34

se, 34

declaração composta, 35

declaração condicional, 34

definir membro, 47

depuração, 53

dicionário, 45, 54

laço de repetição com, 50

directory, 81

current, 90

cwd, 90

- working, 82, 90
- divisibilidade, 24
- division
 - floating-point, 22
 - floor, 22, 42
- duplicate, 91
- elif keyword, 37
- else keyword, 35
- entrada pelo teclado, 25
- erro
 - execução, 41
 - semântico, 20, 29
 - sintaxe, 28
 - tempo de execução, 28
- erro de execução, 41
- erro de sintaxe, 28
- erro em tempo de execução, 28
- erro semântico, 17, 20, 29
- especial valor
 - False, 33
 - True, 33
- estilo, 50
- exceção, 28
 - KeyError, 46
 - OverflowError, 41
 - ValueError, 26
- execução alternativa, 35
- execução condicional, 34
- exists function, 82
- expressão, 22, 23, 29
 - booleana, 33
- Expressão booleana, 33
- expression
 - boolean, 42
- fahrenheit, 38
- False valor especial, 33
- file name, 81
- floating-point division, 22
- floor division, 22, 29, 42
- folder, 81
- frequência, 48
- função
 - dict, 45
 - len, 46
 - raw_input, 25
- função de hash, 54
- função len, 46
- função raw_input, 25
- função dict, 45
- function
 - connect, 59
 - cursor, 59
 - exists, 82
 - getcwd, 81
 - popen, 89
- getcwd function, 81
- guardian pattern, 40, 42
- hardware, 3
 - arquitetura, 3
- hashing, 90
- histograma, 48, 54
- idiomatismo, 48
- implementação, 47, 54
- instrução, 22, 30
 - atribuição, 20
 - condicional, 42
 - pass, 35
 - print, 17
- instrução composta, 42
- instrução condicional, 42
- instrução de atribuição, 20
- instrução pass, 35
- instrução print, 17
- integer, 29
- interpretar, 16
- item
 - dicionário, 54
- KeyError, 46
- keyword
 - elif, 37
 - else, 35
- laço
 - aninhado, 49, 54
- laço de repetição

- com dicionários, 50
- laços aninhados, 49, 54
- linguagem
 - programação, 5
- linguagem de alto nível, 16
- linguagem de baixo nível, 17
- linguagem de programação, 5
- ls (Unix command), 89
- método
 - get, 48
 - keys, 50
 - values, 46
- método get, 48
- método keys, 50
- método values, 46
- MD5 algorithm, 91
- memória principal, 17
- memória secundária, 17
- membro
 - definir, 47
 - dicionário, 46
- mensagem de erro, 20, 28
- method
 - close, 89
 - read, 89
 - readline, 89
- mnemônico, 26, 29
- modo interativo, 7, 16, 22
- modo script, 22
- module
 - os, 81
 - sqlite3, 59
- MP3, 91
- navegador de banco de dados, 80
- normalização, 80
- normalização de banco de dados, 80
- not operador, 34
- nova linha, 25
- o tipo string, 19
- operador, 30
 - and, 34
 - comparação, 33
 - in, 46
 - lógico, 33, 34
 - módulo, 24, 29
 - not, 34
 - or, 34
 - string, 24
- operador aritmético, 22
- operador de comparação, 33
- operador in, 46
- operador lógico, 33, 34
- operador módulo, 29
- operador, aritmético, 22
- operadr módulo, 24
- operando, 22, 30
- or operador, 34
- ordem das operações, 23, 29
- os module, 81
- OverflowError, 41
- palavra chave, 21, 29
- par chave-valor, 45, 54
- parênteses
 - precedência de sobrecarga, 23
- parse, 17
- path, 81
 - absolute, 82, 90
 - relative, 82, 90
- pattern
 - guardian, 40, 42
- PEMDAS, 23
- pipe, 89
- ponto flutuante, 29
- popen function, 89
- portabilidade, 17
- precedência, 30
- programa, 12, 17
- prompt, 17, 25
- Python 3.0, 22, 25
- read method, 89
- readline method, 89
- regras de precedência, 23, 30
- relação, 80
- relative path, 82
- resolução de problema, 17
- resolução de problemas, 5

restrição, 79
Romeu e Julieta, 49, 51

script, 11
se declaração, 34
semântica, 17
sensitividade de case, nomes de
 variáveis, 29
shell, 89, 90
short circuit, 39, 42
sqlite3 module, 59
string, 19, 30
 operação, 24
string entre aspas, 19

tabela de hash, 54
tabela hash, 47
tipo, 19, 30
 bool, 33
 dict, 45
 float, 19
 int, 19
 str, 19
tipo float, 19
tipo int, 19
traceback, 38, 41, 42
travessia, 48, 50
True valor especial, 33
tupla, 80

Unicode, 61
unidade central de processamento, 16
Unix command
 ls, 89
usar depois de definir, 28

valor, 19, 30, 54
ValueError, 26
variável, 20, 30
verificação de consistência, 53
verificação de sanidade, 53

walk, 91
whitespace, 41
working directory, 82

