Tradução da Versão 0.0.8-d2 de Charles Severance

Python para Informática

Campo Grande - MS

Maio de 2015

Tradução da Versão 0.0.8-d2 de Charles Severance

Python para Informática

Universidade Federal de Mato Grosso do Sul – UFMS

Faculdade de Computação

Campo Grande - MS

Maio de 2015

Lista de ilustrações

Sumário

1 Por que você deve aprender a programar?

Escrever programas (ou programar) é uma atividade muito criativa e recompensadora. Você pode escrever programas por muitas razões, variando desde fazer de sua existência solucionar um problema complexo de análise de dados até se divertir ajudando outra pessoa a resolver um problema. Este livro assume que todo mundo precisa saber como programar e, uma vez que você sabe como programar, você poderá solucionar o que pretende utilizando sua habilidade recém descoberta.

Nós somos rodeados no nosso cotidiano com computadores que vão desde celulares até notebooks. Podemos pensar nestes computadores como nossos "assistentes pessoais" que podem estar ao nosso lado para tomar conta de muitas coisas. O hardware dos computadores atuais é essencialmente desenvolvido para nos perguntar "O que você quer que eu faça em seguida?".

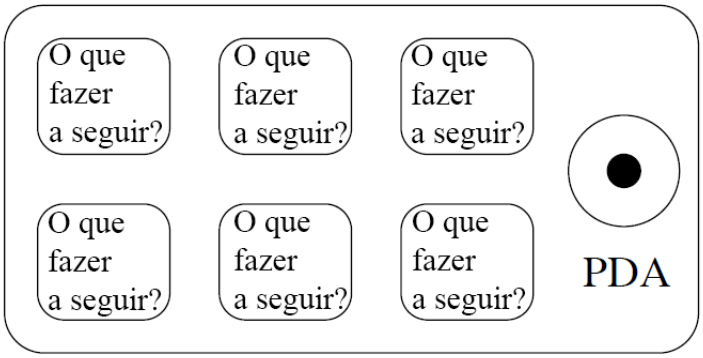


Figura 1: Esquema de solicitação de serviços de hardware.

Programadores adicionam ao hardware um sistema operacional e um conjunto de aplicativos e nós terminamos com um Assitente Pessoal Digital (*Personal Digital Assitant* - PDA) útil e capaz de nos ajudar em muitas coisas diferentes.

Nossos computadores são rápidos, possuem uma grande quantidade de memória e podem ser muito úteis para nós se soubéssemos a linguagem para informar as tarefas que queremos que seja executada "em seguida". Se soubermos esta linguagem, podemos informar aos computadores para executar tarefas que sejam repetitivas. Curiosamente, os tipos de tarefas que os computadores podem realizar melhor são geralmente as coisas que os humanos acham chatas.

Por exemplo, olhe para os três primeiros parágrafos deste capítulo e diga-me a palavra mais utilizada e quantas vezes a mesma foi utilizada. Embora você seja capaz de

ler e entender as palavras em poucos segundos, contá-las é um pouco trabalhoso porque este não é um tipo de trabalho que a mente humana foi desenvolvida para resolver. Para um computador, o oposto é verdade, ler e entender um texto em um papel é difícil para um computador realizar, mas contar as palavras e dizer quantas vezes a palavra mais utilizada apareceu no texto é uma tarefa fácil para um computador:

```
python words.py
Entre file: words.txt
to 16
```

Nosso "assitente pessoal para análise de informações" rapidamente nos disse que a palavra "to" foi utilizada dezesseis vezes nos três primeiros parágrafos deste capítulo.

O fato de computadores serem bons em coisas que as pessoas não são é o motivo pelo qual você deve ter habilidades em falar a "linguagem do computador". Uma vez que você aprenda esta nova linguagem, você pode delegar tarefas a seu parceiro (o computador), deixando mais tempo para que você execute coisas que seja especialmente adequadas a você. Você traz criatividade, intuição e invenção a esta parceria.

1.1 Criatividade e Motivação

Apesar deste livro não ser para programadores profissionais, programação como profissão pode ser um trabalho recompensador, tanto financeiramente quanto pessoalmente. Construir programas úteis, elegantes e inteligentes para outras pessaos usarem é uma atividade bastante criativa. Seu computador ou PDA geralmente contém muitos programas diferentes de diversos grupos de programadores, cada um competindo por sua atenção e interesse. Eles tentam ao máximo atender suas necessidades e proporcionar uma ótima experiência no processo. Em algumas situacões, quando você escolhe um software, os programadores são diretamente compensados pela sua escolha.

Se pensarmos em programas como uma saída para grupos de programadores, talvez a Figura 2 seja uma versão mais sensível de nosso PDA:

Por agora, nossa principal motivação não é ganhar dinheiro ou agradar usuários, mas sim em sermos mais produtivos em manusear dados e informações que encontraremos em nossa vida. No começo, você será ao mesmo tempo o programador e o usuário final dos sesu programas. À medida que você adquira mais habilidades como programador e tiver mais critividade em programação, você pode começar a pensar em desenvolver programas para outros usuários.

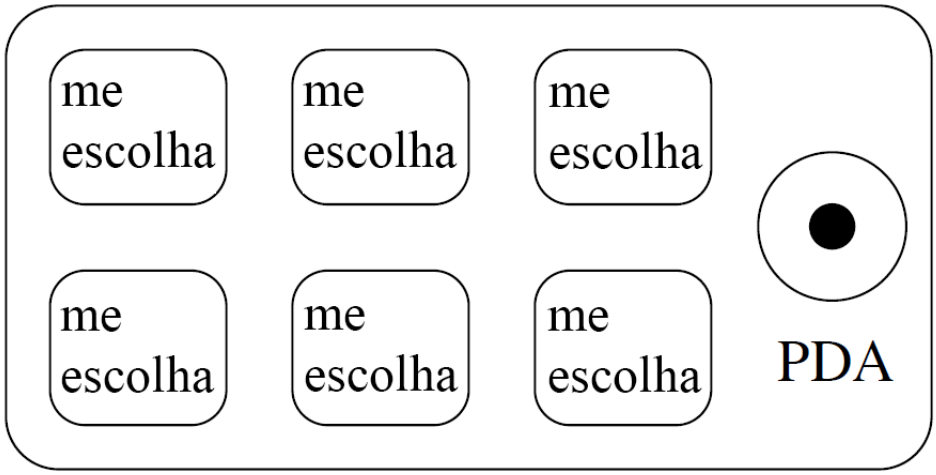


Figura 2: Uma visão de um PDA.

1.2 Arquitetura de Hardware do Computador

Antes de começarmos a aprender a linguagem para dar instruções para computadores para desenvolver software, devemos aprender um pouco como computadores são construídos. Se você desmantelasse seu computador ou telefone celular e olhar por dentro, encontrará as seguintes partes:

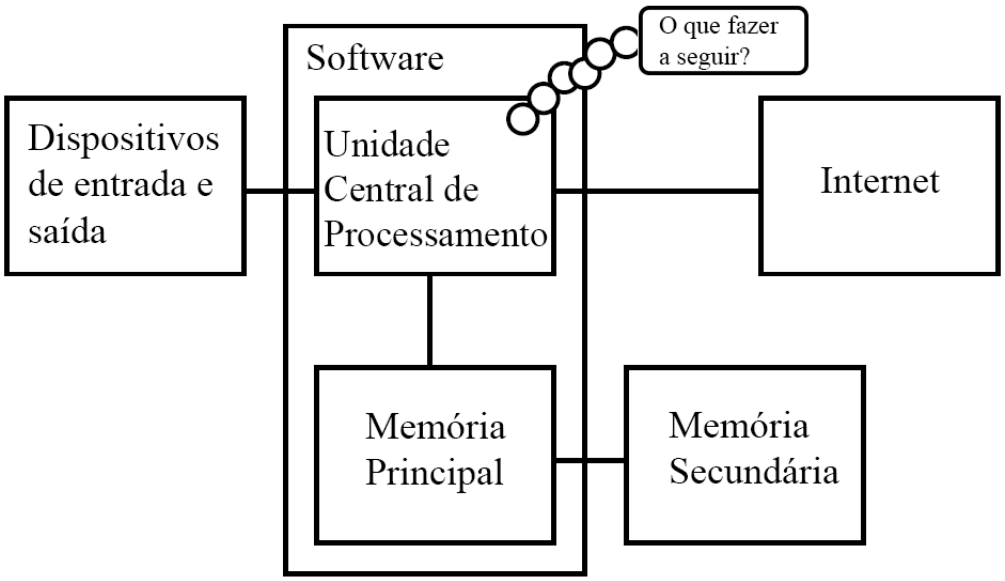


Figura 3: Componentes de um computador.

As definições em alto nível destas são:

- A **Unidade Central de Processamento** (UCP ou CPU) é a parte do computador

desenvolvida para se preocupar com "qual a próxima tarefa?". Se o seu computador for de 3.0 Gigahertz, significa que ele vai te perguntar "qual a próxima tarefa?" três bilhões de vezes por segundo. Você deverá aprender como falar rápido para acompanhar a CPU.

- A **memória principal** é usada para armazenar informações que a CPU precisa rapidamente. A memória principal é quase tão rápida quanto a CPU. Mas as informações armazenadas na memória principal desaparecem quando o computador é desligado.

- A **memória secundária** também é utilizada para armazenar informações, mas é mais devagar que a memória principal. A vantagem da memória secundária é que pode armazenar informações mesmo quando não ha energia para o computador. Exemplos de memória secundária são discos rígidos (HDs) ou memória flash (tipicamente encontrados em pen drives e tocadores portáteis de música).

- Os **dispositivos de entrada e saída** são a tela, teclado, mouse, microfone, altofalante, touchpad, etc. Eles são todas as formas que interagimos com o computador.

- Atualmente, a maioria dos computadores possuem também uma **conexão de rede** para obter informações por meio de uma rede. Podemos pensar numa rede como um local muito lento para armazenar e obter informações que nem sempre estarão disponíveis. Assim, uma rede é uma forma mais lenta e às vezes não confiável de memória secundária.

Apesar de ser melhor deixar os detalhes de como estes componentes funcionam para fabricantes de computadores, ajuda conhecer alguma terminologia de maneira que se possa falar sobre estas partes enquanto se desenvolve programas.

Como programador, seu trabalho é usar e orquestrar cada um destes recursos para solucionar o problema que precisa resolvendo e analisando os dados necessários. Como programador, você estará principalmente "falando" com a CPU e dizendo que tarefa executar em seguida. De vez em quando, você dirá à CPU para usar a memória principal, memória secundária, rede, ou dispositivos de entrada e saída.

Você precisa ser a pessoa que informar para a CPU qual " a próxima" tarefa. Mas seria muito desconfortável encolhê-lo até 5 mm e inseri-lo dentro do computador para poder emitir comandos três bilhoões de vezes por segundo. Ao invés disso, você pode escrever instruções. Chamamos estas instruções armazenadas de **programa**, e o ato de escrevê-las de maneira correta de **programar**.

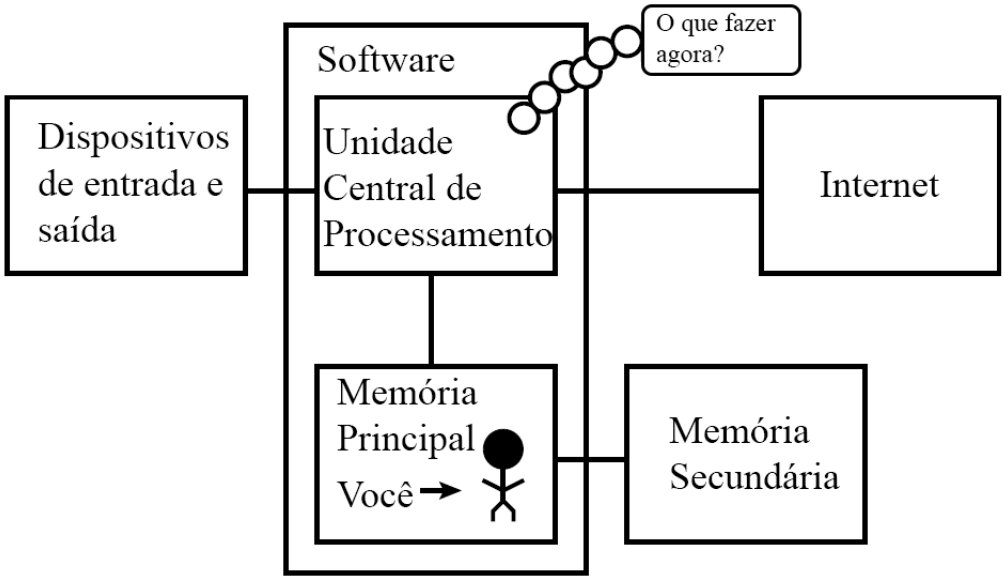


Figura 4: Componentes de um computador, incluindo o programador.

1.3 Entendendo Programação

No resto deste livro, tentaremos torná-lo uma pessoa habilidosa na arte de programar. No final você será um **programador** - talvez não um programador profissional, mas ao menos terá habilidades para olhar para um problema de análise de dados e desenvolver um programa para solucioná-lo.

De certo modo, você precisa de duas habilidades para ser um programador:

- Primeiro você deve conhecer a linguagem de programação (Python) - você deve conhecer o vocabulário e a gramática. Você deve ser capaz de soletrar as palavras nesta nova linguagem corretamente e de saber construir "sentenças" bem formados nessa nova linguagem.
- Segundo você deve "contar uma história". Escrevendo uma história você combina palavras e sentenças para transmitir uma ideia para o leitor. Há habilidade e arte na construção de uma história e a habilidade de escrever histórias é aprimorada cada vez que se escreve e se obtém retorno sobre o que foi escrito. Em programação, nosso programa é a "história" e o problema que se está tentando resolver é a "ideia".

Uma vez que você aprenda uma linguagem de programação tal como Python, você achará bem mais fácil aprender uma segunda linguagem de programação, tais como JavaScript ou C++. Uma nova linguagem de programação possui vocabulário e gramática diferentes, mas uma vez que adquire habilidades para resolver problemas, eles serão os mesmos, independente da linguagem de programação.

Você aprenderá "vocabulário" e "sentenças" do Python muito rapidamente. O que vai levar mais tempo é você ser capaz de escrever um programa coerente para resolver um problema desconhecido. Ensinamos programação muito parecido com como ensinamos a escrever. Começamos lendo e explicando programas e, em seguida, escrevemos programas simples e, por fim, escrevemos programas cada vez mais complexos ao longo do tempo. Em algum momento você "pega o jeito", identifica sozinho padrões entre os programas e naturalmente aprenderá como se deparar com um problema e escrever um programa que resolva-o. E assim que chega neste ponto, programar se torna um processo bastante prazeroso e criativo.

Começamos com o vocabulário e a estrutura de programas em Python. Seja paciente quando lembrar dos programas simples da época que estava começando.

1.4 Palavras e sentenças

Ao contrário das línguas faladas, o vocabulário do Python é bastante reduzido. Chamamos este "vocabulário" de "palavras reservadas". Estas são as palavras que possuem um significado muito especial em Python. Quando o Python vê estas palavras em um programa, elas possuem um, e somente um significado em Python. Mais tarde, ao escrever programas você criará suas próprias palavras reservadas, chamadas de **variáveis**. Você terá uma enrome liberadade para escolher nomes para suas variáveis, mas não poderá usar nenhuma palavra reservada do Python como nome de variável.

Quando treianamos um cachorro, utilizados palavras especias como, "senta", "fica" e "busca". Além disso, quando você fala com um cachorro e não utiliza uma destas palavras reservadas, ele fica olhando para você como uma expressão de dúvida até que você diga uma palavra reservada. Por exemplo, se você disser "eu gostaria que mais pessoas andassem para melhorar a saúde", o que a maioria dos cachorros vão ouvir é "blá blá blá ande blá blá blá". Isto ocorre porque "ande" é uma palavra reservada na linguagem canina. Muitos sugerem que a linguagem entre pessoas e gatos não possuem palavras reservadas[1].

As palavras reservadas da linguagem Python incluem as seguintes:

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

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

O seja, ao contrário dos cachorros, Python já é treinada. Quando você diz “tente”, Python tentará toda vez que você disser esta palavra sem falhas.

Aprenderemos estas palavras reeservadas e como as mesmas são utilizadas, mas neste momento focaremos o equivalente a dizer (em uma linguagem de humano para cachorros) “fale”em Python. O bom de dizer ao Python para falar é que podemos também informar o que dizer passando uma mensagem entre aspas:

```
print ‘Hello World!’
```

Acabamos de escrever nossa primeira sentença em Python sinteticamente correta. Nossa sentença começa com a palavra reservda **print**, seguida de uma string de texto de nossa escolha entre aspas.

1.5 Conversando com Python

Agora que sabemos uma palavra e uma sentença simples em Python, devemos saber como começar uma conversa com Python para testar nossas habilidades na nova linguagem.

Antes de começar a conversar com Python, você primeiro deve instalar o software Python e saber como iniciá-lo em seu computador. São muitos detalhes para colocar neste capítulo, portanto sugiro que consulte *www.pythonlearn.com*, onde estão instruções detalhadas de como configurar e inciar Python nos sitemas Macintosh e Windows. Em algum ponto, você estará em um terminal ou janela de comandos, digitará **python** e o interpretador Python iniciará a execução em modo interativo: aparecerá alguma coisa como:

```
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 o modo como o interpretador Python pergunta “O que você quer que eu execute em seguida?”. Python está pronto para ter uma conversa com você. Tudo o que você precisa saber é como falar a linguagem Python que poderá ter uma conversa.

Digamos que você não saiba nem mesmo as palavras e sentenças mais simples da linguagem Python. Você pode querer usar a linha que os astronautas usam quando eles pousam em um planeta distante e tentam falar com os habitantes do planeta:

```
> > > I come in peace, please take me to your leader
   File " <stdin> ", line 1
     I come in peace, please take me to your leader
          ^
SyntaxError: invalid syntax
> > >
```

Isto não está indo tão bem. A menos que você pense em algo rapidamente, os habitantes do planeta estão suscetíveis a esfaqueá-lo com suas lanças, colocá-lo num espeto, assá-lo e comê-lo no jantar.

Por sorte, você trouxe uma cópia deste livro em suas viagens, folheou até está página e tentou novamente:

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

Isto parece muito melhor, então você tenta se comunicar mais:

```
> > > print You must be the legendary god that comes from the sky'
You must be the legendary god that comes from the sky
> > > print 'We have been waiting for you for a long time'
We have been waiting for you for a long time
> > > print 'Our legend says you will be very tasty with mustard'
Our legend says you will be very tasty with mustard
> > > print We will have a feast tonight unless you say
    File "<stdin>", line 1
     print 'We will have a feast tonight unless you say
                                                       ^
SyntaxError: EOL while scanning string literal
> > >
```

A conversa estava indo muito bem por um instante, mas você cometeu um erro ínfimo usando Python, e Python reclamou novamente.

Neste ponto, você deve ter percebido que ao mesmo tempo que Python é inclivelmente complexo, poderoso e rigoroso em relação à sintaxe utilizada para se comunicar com ele, Python não é inteligente. Você está tendo uma conversa consigo mesmo, mas usando uma sintaxe apropriada.

Neste sentido, quando você usa um programa escrito por outra pessoa, a conversa é entre você e outros programadores com Python atuando como intermediário. Python é uma forma para os criadores de programas expressarem como a conversa deveria proceder.

E em mais alguns capítulos, você será um destes programadores usando Python para conversar com os usuários dos seus programas.

Antes de deixarmos nossa primeira conversa com o interpretador Python, você provavelmente deve saber o modo apropriado de dizer "até logo" ao interagir com habitantes do Planeta Python:

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

> > > if you don't mind, I need to leave
   File "<stdin>", line 1
     if you don't mind, I need to leave
              ^
SyntaxError: invalid syntax

> > > quit()
```

Você perceberá que o erro é diferente das duas outras tentativas incorretas. O segundo erro é diferente porque **if** é uma palavra reservada, o Python identificou a palavra reservada e informou que estamos querendo dizer alguma coisa, mas a sintaxe da sentença está incorreta.

A forma apropriada de dizer "até logo" para o Python is to enter **quit()**no prompt >>> interativo. Provavelmente demorou um tempo para que você percebesse que ter um livro à mão poderia ser útil.

1.6 Terminologia: interpretador e compilador

Python é uma linguagem de **alto nível** cuja intenção é ser relativamente direta para pessoas lerem e escreverem e para computadores lerem e processarem. Outras linguagens de alto nível incluem: Java, C++, PHP, Ruby, Basic, Perl, javaScript e muitas outras. O hardware da Unidade Central de Processamento (UCP) não entende estas linguagens de alto nível.

A UCP conhece a linguagem chamada **linguagem de máquina**. Linguagem de máquina é muito simples e francamenete é muito cansativa para escrever porque é representada apenas por zeros e uns:

```
01010001110100100101010000001111
11100110000011101010010101101101
...
```

Linguagem de máquina parece ser superficialmente simples já que só existem zeros e uns, mas sua sintaxe é ainda mais complexa e mais intrigante do que Python. Assim, pouquíssimos programadores já escreveram em linguagem de máquina. Ao invés disso, construímos diversos tradutores para permitir que programadores escrevam em linguagens de alto nível, tais como Python ou JavaScript, e esses tradutores convertem os programas para linguagem de máquina para executarem pela UCP.

Como a linguagem de máquina está associada ao hardware do computador, linguagem de máquina não é **portável** em diferentes tipos de hardware. Programas escritos em linguagens de alto nível podem ser utilizados em diferentes computadores, utilizando um nterpretador diferente ou recompilando o código para criar uma versão do programa em linguagem de máquina para o novo computador.

Estes tradutores de linguagem de programação se encaixam em duas categorias principais: (1) interpretadores e (2) compiladores.

Um **interpretador** lê o código fonte dos programas escritos pelos programadores, analisa-o, e interpreta as instruções em tempo real. Python é um interpretador e quando rodamos Python interativamente, podemos digitar uma linha de Python (uma sentença), Python a processa imediatamente e fica pronta para digitarmos uma nova linha em Python.

Algumas linhas de Python dizem ao Python que você quer lembrar um valor posteriormente. Precisamos escolher um nome para lembrar o valor posteriormente e podemos usar este nome simbólico para obter o valor posteriormente. Usamos o termo **variável** para se referir aos nomes utilizados para armazenar dados.

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

Neste exemplo, solicitamos ao Python para lembrar o valor seis e usamos o nome **x** de maneira que possamos obter o valor posteriormente. Verificamos que Python realmente lembrou o valor usando **print**. Em seguida, solicitamos ao Python obter o valor de **x**, multiplicá-lo por sete e armazenar o novo valor em **y**. Em seguida, solicitamos ao Python para imprimir o valor atual de **y**.

Apesar de estarmos digitando estes comandos em Python uma linha por vez, Python está tratando-os como uma sequência ordenada de sentenças com as sentenças

posteriores podendo obter os dados criados nas sentenças anteriores. Estamos escrevendo nosso primeiro parágrafo com quatro sentenças em uma ordem lógica e significativa.

É natural para um **interpretador** ser capaz de ter uma conversa interativa como mostrado acima. Um **compilador** precisa que o programa esteja em um arquivo, então roda um processo para transformar o código em alto nível para linguagem de máquina e, em seguida, o compilador armazena os dados em liguagem de máquina em um arquivo para ser executado posteriormente.

Se você utilizar o sistema operacional Windows, estes executáveis terão a extensão ".exe" ou ".dll', que referem a "executável" ou "biblioteca carregada dinaminacamente (dynamic loaded library)", respectivamente. No Linux ou Macintosh, não há extensão que identifica arquivos executáveis.

Se você abrir um arquivo executável em um editor de texto, ele parece completamente louco e ilegível:

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

Não é facil escrever ou ler códigos em linguagem de máquina, portanto é interessante ter um **interpretador** ou **compilador** que permite escrevermos código em linguagens de alto nível, como Python ou C.

Neste ponto da discussão sobre compiladores e interpretadores, você deve conhecer um pouco mais sobre o interpretador Python. Em que linguagem o código está escrito? É construído em uma linguagem compilada? Quando digitamos "python", o que exatamente acontece?

O interpretador Python é escrito em uma linguagem de alto nível conhecida como "C". Você pode olhar o código fonte atual do interpretador Python e a maneira de trabalhar o código fonte acessando http://www.python.org . Então, Python é um programa e é compilado em um linguagem de máquina e quando você instala Python no seu computador (ou o vendedor instalar para você), você copiou um código em linguagem de máquina do programa traduzido Python para seu sistema. No Windows, o código de máquina executável para Python está em um arquivo com um nome parecido com:

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

Existem muito mais do que você precisava saber para ser um programador Python, mas algumas vezes não vale a pena responder a essas pequenas questões incômodas logo no início.

1.7 Escrevendo um programa

Digitar comandos no interpretador Python é uma boa maneira de conhecer as caracyerísticas do Python, mas não é recomendado para resolver problemas mais complexos.

Quado queremos escrever um programa, usamos um editor de texto para escrever instruções em Python em um programa, denominado de **script**. Por convenção, Scripts em Python possuem nomes terminados com ".py".

Para executar um script, é necessário informar ao interpretador Python o nome do arquivo. Em uma janela de comandos do Windows ou Unix, você deveria python hello.py como a seguir:

```
csev$ cat hello.py
print 'Hello world!"
csev$ python hello.py
Hello world!
csev$
```

O "csev$' é o prompt do sistema operacional, e o comando "cat hello.py" está nos mostrando que o arquivo "hello.py' é um programa que contém um linha para imprimir uma string.

Nós chamamos o interpretador Python e informa-o para ler o código fonte do arquivo "hello.py' ao invés de nos solicitar quatro linha interativamente.

Você vai perceber que não há necessidade de ter **quit()** ao final do arquivo que contém o programa em Python. Quando o Python está lendo o código fonte de um arquivo, ele sabe que tem que parar ao alcançar o final do arquivo.

1.8 O que é um programa?

A definição de um **programa** de maneira mais simples é uma sequência de sentenças em Python que foram criadas para fazer alguma coisa. Mesmo o script **hello.py** é um programa. É um programa de uma única linha e geramlmente não muito útil, mas na definição estrita, é um programa em Python.

Pode ser mais fácil de entender o que é um programa pensando sobre um problema que pode ter um programa para resolvê-lo, e então olhar para o programa que deveria

resolver o problema.

Suponha que você está fazendo pesquisas em Computação Social nos *posts* do Facebook e está interessado na palavra usada mais frequentemente em uma série de *posts*. Você poderia imprimir um fluxo de *posts* do Facebook e debruçar-se sobre o texto procurando pela palavra mais comum, mas isto pode levar muito tempo e ser muito propenso a erros. Você seria inteligente ao escrever um programa em Python para resolver a tarefa rapidamente e com acurácia e, então você poderia passar o fim de semana fazendo algo divertido.

Por exemplo, olhe para o seguinte texto sobre um palhaço e um carro. Olhe para o texto e descubra a palavra mais comum e quantas vezes ela ocorreu.

```
the clown ran after the car and the car ran into the tent
and the tent fell down on the clown and the car
```

Então imagine que você esteja fazendo esta tarefa olhando milhões de linhas de texto. Francamente, seria mais rápido você aprender Python e escrever um programa em Python para contar as palavras do que procurar as palavras manualmente.

A notícia melhor ainda é que eu trouxe um programa simples para achar a palavra mais comum em um arquivo texto. Eu escrevi, testei e estou lhe fornecendo para que você possa economiza 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 saber Python para usar este programa. Você precisaria ir ao Capítulo 10 deste livro para entender completamente as técnicas em Python para fazer este programa. Você é o usuário final, você simplesmente usa o programa e se impressiona com sua inteligência e como salvou muito esforço manual de você. Você simplesmente digita o código em um arquivo chamado **words.py** e o executa ou faz o download do código fonte em http://pythonlearn.com/code/ e executa-o.

Este é um bom exemplo de como o Python e a linguagem Python estão atuando entre você (usuário final) e eu (o programador). Python é uma forma de para trocarmos sequências de instruções úteis (ou seja, programas) em uma linguagem comum que pode ser usada por qualquer um que instale Python em seu computador. Então, nenhum de nós está falando com Python, mas estamos nos comunicando por meio do Python.

1.9 Os blocos de programas

Nos próximo capítulos, aprenderemos mais sobre vocabulário, estrutura das sentenças, estrutura dos parágrafos, e estrutura de histórias do Python. Aprenderemos sobre as potentes capacidades do Python e como integrar esta capcidades para criação de programas úteis.

Existem alguns padrões conceituais de baixo nível que usamos para construção de programas. Estas construções não são apenas para programas em Python, eles são parte de todas as linguagens de programação, desde linguagem de máquina até linguagens de alto nível.

- **entrada:** obtém dados do "mundo externo". Isto pode ser feito lendo dados de um arquivo, ou até mesmo por meio de um sensor como microfone ou GPS. Em nossos primeiros programas, a entrada será realizada por meio do usuário digitando dados no teclado.
- **saída:** mostra os resultados dos programas na tela, ou armazena-os em um arquivo talvez escreve-os em um dispositivo como um alto-falante para tocar uma música ou falar um texto.
- **execução sequencial:** executa as sentenças uma após a outra na ordem em que aparecem no script.
- **execução condicional:** verifica algumas condições e executa ou pula uma sequência de sentenças.
- **execução de repetição:** executa um conjunto de sentenças repetidamente, geralmente com alguma variação.
- **reuso:** escrita de um conjunto de instruções uma vez, nomeando-as, e então reusar estas instruções quando preciso em todo o programa.

Parece ser muito simples para ser verdade e, é claro, nem sempre é tão simples. É como dizer que andar é simplesmente "coloque um pé na frente do outro". A "arte" de escrever um programa é integrar este elementos básicos muitas vezes para produzir algo que seja útil para os usuários.

O programa de contagem de palavras acima utiliza todos estes padões, exceto um.

1.10 O que possivelmente pode dar errado?

Como dissemos nas nossas conversas anteriores com Python, devemos nos comunicar de maneira acurada quando escrevemos código em Python. O menor desvio ou erro fará com que o Python desista de olhar seu programa.

Programadores iniciantes frequentemente consideram que Python não deixa espaço para erros e consideram isso como evidência de que o Python seja desprezível, detestável e cruel. Apesar de Python parecer gostar de todo mundo, Pythom os conhece pessoalmente e mantém um rancor contra eles. Em razão deste rancor, Python pega nossos programas perfeitamente escritos e rejeita-os como "impróprio' apenas para nos tormentar.

```
> > > primt ‘Hello world!
   File "<stdin>", line 1
     primt ’Hello world!’

SyntaxError: invalid syntax
> > > primt ‘Hello world!’
   File "<stdin>", line 1
     primt ‘Hello World’
                        ^

SyntaxError: invalid syntax
> > > I hate you Python!
   File "<stdin>", line 1
     I hate you Python!
          ^
SyntaxError: invalid syntax
> > > if you come out of there, I would teach you a lesson
   File "<stdin>", line 1
if you come out of there, I would teach you a lesson
          ^
SyntaxError: invalid syntax
> > >
```

Não há nada a ganhar argumentando com o Python. Ele é uma ferramente, não tem emoção e está feliz e pronto para servi-lo sempre que necessário. Suas mensagens de erros parecem severas, mas são apenas chamados do Python por ajuda. Ele olhou para o que você digitou e simplesmente não entendu o que você informou.

Python é muito mais que um cachorro te amando incondicionalmente, entendendo poucas palavras, olhando para para você como um olhar doce (> > >) e esperando que você diga alguma coisa que ele entenda. Quando Python diz "SytaxError: invalid syntax",

ele está apenas abanando o rabo e dizendo "Você parece ter dito alguma coisa, mas eu não entendo o que você quis dizer, porém, continue falando comigo (> > >)."

Na medida em que seus programas se tornem sofisticados, você encontrará três tipos gerais de erros:

- **Erros de sintaxe:** estes são os primeiros erros que cometerá e os mais fáceis de corrigir. Um erro de sintaxe significa que você violou as regras de "gramática" do Python. Python faz o possível para apontar para a linha e o caracter que ele percebeu que está confuso. A única coisa complicada em erros de sintaxe é que às vezes os erros que precisam ser corrigidos estão antes do local que o Python identificou. Assim, a linha e o caracter que o Python indicar em um erro de sintaxe é apenas um ponto inicial de investigação.

- **Erros de lógica:** um erro de lógica ocorre quando o programa está com a sintaxe correta, mas há um erro na ordem das sentenças e talvez um erro na relação entre as sentenças. Um bom exemplo de erro de lógica pode ser, "tome um gole de sua garrafa de água, coloque-a na sua mochila, vá até a biblioteca e coloque novamente a tampa na sua garrafa".

- **Erros de semântica:** um erro de semântica ocorre quando sua descrição de passos está sintaticamente correta e na ordem correta, mas há simplesmente um erro no programa. O programa está perfeitamente correto mas não faz o que deveria fazer. Um exemplo simples é se você estiver dando coordenadas sobre um restaurante a uma pessoa e diz, "... quando você chegar no cruzamento com o posto de gasolina, você vira à esquerda, anda uma milha e o restaurante é uma construção vermelha à sua esquerda.". Seu amigo está muito atrasado e te telefona dizendo que estão em uma fazenda e andando em volta de um celeiro e nenhum sinal de restaurante. Então você diz: "Você virou à esquerda ou à direita no posto de gasolina?", e ele diz "Eu segui suas instruções perfeitamente, eu as tenho anotadas, e elas dizem vire à direita e ande uma milha até o posto de gasolina.". Então você diz, "Me desculpe, pois minhas instruções estava sintaticamente corretas, mas infelizmente continham um pequeno e indetectável erro de semântica.".

Mais uma vez, nos três tipos de erros, Python está apenas tentando fazer exatamene o que você solicitou.

1.11 A jornada de aprendizado

Assim que você avança em direção ao final deste livro, não se assuste se os conceitos não se fixam perfeitamente na primeira vez. Quando você estava aprendendo a falar,

não era problema para você que nos primeiros anos você apenas emitia apenas alguns sons. E também estava tudo bem se levava seis meses para você mudar de um simples vocabulário para sentenças simples e se levava mais 5 ou 6 anos para mudar de sentenças para parágrafos, e alguns anos a mais para ser capaz de escrever sozinho uma pequena história completa.

Queremos que aprenda Python muito mais rápido, então ensinamos a você tudo ao mesmo tempo nos próximos capítulos. Mas é como aprender uma nova língua que leva tempo para absorver e entender antes que a mesma se torne natural. Pode gerar algumas confusões sempre que vemos e revemos tópicos para tentar mostra a você o retrato todo enquanto definimos pequenos fragmentos que formam o retrato. Apesar do livro ser escrito linearmente, e se você está fazendo um curso o mesmo progredir linearente, não hesite em ser não linear na maneira como aborda o material. Olhe para frente e para trás e leia com um pouco de cuidado. Olhando materiais mais avançados sem entender completamente os detalhes, você pode obter melhor entendimento do "por que?" programar. Revisando materiais e até mesmo refazendo exercícios anteriores, você perceberá que aprendeu muita coisa, mesmo que o material que está olhando pareça ser impenetrável.

Geralmente, quando você está aprendendo sua primeira linguagem de programação, existem alguns momentos "Ah-hah!" fantásticos que você olha distante para uma pedra com um martelo e uma talhadeira, e afasta-se e vê que você de fato está construindo uma bela escultura.

Se alguma coisa parece particularmente difícil, em geral não há sentido em perder o sono olhando para ela por muito tempo. Respire, tire um cochilo, faça um lanche, explique para alguma pessoa (ou quem sabe para seu cachorro) que você está tendo um problema, e então retorne com a cabeça fresca. Eu garanto que assim que você aprender conceitos de programação no livro, você olhará para trás e verá que foi tudo muito fácil e elegante, e que simplesmente levou algum tempo para você absorver isto.

1.12 Glossário

bug: um erro em um programa.

unidade central de processamento: o coração de todo computador. É o que roda o software que escrevemos; também chamada de UCP ou CPU ou "o processador".

compile: traduzir um programa escrito em linguagem de alto nível para uma linguagem de baixo nível uma vez, preparando para uma execução posterior;

linguagem de alto nível: uma linguagem de programação como Python, que é desenvolvida para ser fácil para as pessoas escreverem e lerem.

modo interactivo: uma forma de usar o interpretador Python digitando coman-

dos e expressões no prompt.

interpretar: executar um programa em linguagem de alto nível traduzindo-o linha por linha em tempo real.

linguagem de baixo nível: uma linguagem de programação projetada para ser fácil para um computador executar; também chamda de "código de máquina" ou "linguagem assembly".

memória principal: armazena programas e dados. A memória principal perde as informações quando a energia é desligada.

parse: examinar um programa e analisar sua estrutura sintática.

portabilidade: um propriedade de um programa que pode executar em mais de um tipo de computador.

comando print: um instrução que faz com que o interpretador Python mostre um valor na tela.

resolução de problema: o processo de formular um problema, encontrar uma solução e expressar a solução.

programa: um conjunto de instruções que especifica um cálculo.

prompt: quando um programa mostra uma mensagem e pausa para o usuário digitar alguma entrada para o programa.

memória secundária: armazena programas e dados, mantendo as informações mesmo que a energia seja desligada. Geralmente mais lenta que a memória principal. Exemplos de memória secundária incluem discos rígidos e memórias flash em pen drives.

semântica: o significado de um programna.

código fonte: um programa em liguagem de alto nível.

1.13 Exercícios

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

a) Executar todos os cálculos e lógica de um programa

b) Recuperar páginas na Internet

c) Armazenar informações por um período longo, mesmo que ocorra queda de energia.

d) Obter entrada do usuário

Exercício 1.2 O que é um programa?

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

Exercício 1.4 Qual dos seguintes itens possui "código de máquina"?

a) O interpretador Python

b) O teclado

c) Um arquivo fonte Python

d) um documento de processador de texto

Exercício 1.5 O que está errado no seguinte código:

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

Exercício 1.6 Onde em um computador uma variável como "x" é armazenada depois da seguinte linha em Python ser executada?

```
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 vai imprimir:

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

a) 43

b) 44

c) x + 1

d) Erro porque $x = x + 1$ é impossível matematicamente

Exercício 1.8 Explique cada um dos seguintes conceitos usando uma capacidade humana: (1) Unidade Central de Processamento, (2) Memória principal, (3) Memória secundária, (4) Dispositivo de entrada, (5) Dispositivo de saída. Por exemplo, "O que é equivalente nas pessoas à Unidade Central de Processamento"?

Exercício 1.9 Como você corrige um "Erro de Sintaxe (Syntax Error)"?

2 Variáveis, expressões e sentenças

2.1 Valores e tipos

Um **valor** é um dos elementos básicos no qual um programa trabalha, como uma letra ou um número. Os valores vistos até agora foram `1`, `2` e `'Olá, Mundo!'`

Estes valores são de **tipos** diferentes: `1` e `2` são inteiros e `'Olá, Mundo'` é uma **string**, chamada assim por conter uma "sequência" de letras. Você (e o interpretador) podem identificar strings porque elas estão entre aspas simples.

O comando `print` também funciona para inteiros. Nós usamos o comando `python` para iniciar o interpretador.

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

Se você não tiver certeza de que tipo é o valor, o interpretador pode te informar.

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

Não é de se surpreender que strings pertençam ao tipo `str` e inteiros pertençam ao tipo `int`. Já é menos óbvio que o número com casas decimais pertençam ao tipo chamado `float`, porque estes números são representados em um formato chamado **ponto flutuante**.

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

O que dizer de valores como `'17'` e `'3.2'`? Eles se parecem números, mas estão entre aspas como strings. Logo, eles são strings.

```
> > > type (‘17’)
<type ‘str’>
> > > type (‘3.2’)
<type ‘str’>
```

Quando você digita um número muito grande, você pode ficar tentado colocar vírgulas entre grupos de três dígitos, como em `1,000,000`. Isso não é um inteiro em Python, mas é permitido.

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

Bem, isso não é o que se esperava! Python interpreta `1,000,000` como um comando separador de inteiros, colocando espaços entre eles.

Este é o primeiro exemplo que temos visto de erro de semântica: o código executa sem produzir uma mensagem de erro, mas não faz aquilo que é “certo”.

2.2 Variáveis

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

Um **comando de atribuição** cria novas variáveis e lhes atribui valores:

```
> > > mensagem = ‘E agora totalmente diferente’
> > > n = 17
> > > pi = 3.141592653589931
```

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

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

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

O tipo de uma variável é o tipo do valor a que ela se refere:

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

2.3 Nome de variáveis e palavras-chave

Programadores geralmente escolhem nomes para suas variáveis que são significativos, eles colocam o nome de acordo com o uso da variável.

Nomes de variáveis podem ser longos, podem conter números e letras, mas precisam começar com uma letra. É possível começar com letras maiúsculas, mas é uma boa ideia começar com letras minúsculas (você vai ver o porquê mais tarde).

O caracter (_) pode aparecer no nome e é frequentemente usado em nomes com várias palavras como `meu_nome` ou `teste_de_variavel`.

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

```
> > > 78trombones = 'Grande parada'
SyntaxError: invalid sintax
> > > more@ = 1000000
SyntaxError: invalid sintax
> > > class = 'Advanced Theoretical Zymurgy'
SintaxError: invalid sintax
```

`76trombones` é inválido porque não começa com uma letra, `mais@` é inválido pois contém um caracter inválido (`@`). Mas o que há de errado com `class`?

Isto ocorre porque `class` é uma **palavra-chave** de Python. O interpretador usa esses tipos de palavras para reconhecer a estrutura do programa, e elas não podem ser usadas como nomes de variáveis.

Python possui 31 palavra-chaves [1] para serem utilizadas:

```
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
```

[1] Em Python 3.0, exec não é uma palavra-chave, mas nonlocal é.

É bom que você tenha esta lista em mãos. Se o interpretador reclamar sobre um dos nomes das variáveis e você não sabe o motivo, veja se o nome está nesta lista.

2.4 Sentenças

Uma **sentença** é uma unidade de código que o interpretador Python pode executar. Nós temos visto dois tipos de sentenças: impressão e atribuiçao.

Quanto você digita uma sentença no modo interativo, o interpretador executa e exibe o resultado, se houver.

Um script geralmente contém uma sequência de sentenças. Se existir mais de uma sentença, os resultados aparecem cada vez que as declarações são executadas.

Por exemplo, o script

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

produz a seguinte saída

```
1
2
```

A declaração de atribuição não produz saída.

2.5 Operadores e operandos

Operadores são símbolos especiais que representam operações como soma e multiplicação. Os valores a que os operadores são aplicados denominam-se **operandos**.

Os operadores `+`, `-`, `*`, `/` e `**` são de adição, subtração, multiplicação, divisão e exponenciação, respectivamente, como mostrado nos exemplos a seguir.

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

O operador de divisão pode não fazer o que é esperado.

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

O valor de `minuto` é `59`, e o convencional é que quando se divide `59` por `60` o resultado seja `0,98333`, não `0`. O motivo por esta discrepância é que Python calcula o

piso da divisão [2].

Quando os dois operandos são inteiros, o resultado é um inteiro; a divisão piso descarta a parte fracionária, então no exemplo ele arredonda para zero.

Se os operandos são números com ponto flutuante, Python realiza uma divisão com ponto flutuante, e o resultado é um número `float`.

```
> > > minuto/60.0
0.9833333333333
```

2.6 Expressões

Uma **expressão** é uma combinação de valores, variáveis e operadores. Um valor por si mesmo já é considerado uma expressão, e por isso é uma variável, então os exemplos seguintes são todos expressões inválidas (assumindo que à variável `x` foi atribuído um valor).

```
17
x
x+17
```

Se você digitar uma expressão no modo interativo, o interpretador **avalia** e exibe o resultado:

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

Mas em um código, a expressão por si só não faz nada! Isso é um erro comum para os iniciantes.

Exercício 2.1 Digite a seguinte declaração no interpretador de Python e veja o que ele faz:

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

[2] Em Python 3.0, o resultado desta divisão é um float. Em Python 3.0 o novo operador // executa a divisão inteira

2.7 Ordem das operações

Quando mais de um operador aparece em uma expressão, a ordem de avaliação depende da **regra de precedência**. Para operadores matemáticos, Python segue a matemática convencional. A sigla **PEMDAS** é um bom modo de se lembrar das regras:

- **P**arênteses têm a maior precedência e podem ser usados para forçar a expressão a ser avaliada do jeito que você quer. Como as expressões entre parênteses são avaliadas em primeiro lugar, `2 *(3-1)` é `4`, e `(1 + 1)**(5-2)` é `8`. Você também pode usá-los para deixar a expressão mais fácil de ser compreendida, como em `(minuto * 100)/60`, mesmo que não mude o resultado.

- **E**xponenciação é a próxima em precedência, então `2**1+1` é `3`, não `4`, e `3*1**3` é `3`, não `27`.

- **M**ultiplicação e **D**ivisão possuem a mesma precedência, que é maior que a **A**dição e **S**ubtração, que também possuem a mesma precedência. 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 à direita. Então na expressão `5-3-1` é `1`, não `3`, porque `5-3` ocorre primeiro e o `1` é subtraído do `2`.

Em caso de dúvida, sempre utilize parênteses nas expressões para ter certeza de que as operações serão realizadas na ordem que você deseja.

2.8 Operador de módulo

O **operador de módulo** funciona em números inteiros e produz o resto da divisão do primeiro operando dividido pelo segundo. Em Python, o operador de módulo é representado pelo símbolo de porcentagem (`%`). A síntaxe é a mesma dos outros operadores:

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

Então `7` dividido por `3` é `2` com `1` de resto.

O operador de módulo torna-se surpreendentemente útil. Por exemplo, você pode verificar se `x` é divisível por `y` se `x % y` é zero.

Você também pode extrair o digíto mais a direita de um inteiro ou todos os dígitos de um número inteiro. Por exemplo, `x % 10` retorna o digíto mais a direita de `x` (na base `10`). Similarmente `x % 100` retorna os dois últimos dígitos.

2.9 Operações com strings

O operador `+` funciona com strings, mas não é uma adição no sentido matemático. Ele realiza uma **concatenação**, o que significa que ele une o final de uma string com o início da outra. Por exemplo:

```
> > > primeiro = 10
> > > segundo = 15
> > > print primeiro+segundo
25
> > > primeiro = '100'
> > > segundo = '150'
> > > print primeiro + segundo
100150
```

A saída do programa é `100150`.

2.10 Perguntando ao usuário a entrada

Algumas vezes nós gostaríamos de obter o valor da variável por meio do que o usuário digitar em seu teclado. O Python provê uma função de construção chamada `raw_input` que obtém o valor do teclado[3]. Quando esta função é chamada, o programa para e espera o usuário digitar algo. Quando o usuário pressiona "Return" ou o "Enter", o programa continuará e o `raw_input` retornará o que o usuário digitou como uma string.

```
> > > input = raw_input()
Qualquer coisa
> > > print input
Qualquer coisa
```

Antes de receber a variável fornecida pelo usuário, é uma boa idéia imprimir uma mensagem dizendo o que o usuário deve fornecer. Você pode passar uma string para

[3] Em Python 3.0 esta função é chamada *input*

`raw_input` para ser visualizada pelo usuário antes de pausar e receber a variável.

```
> > > nome = raw_input(‘Qual é seu nome?\n’)
Qual é seu nome?
Chuck
> > > print nome
Chuck
```

A sequência `\n` no final da sentença representa uma **nova linha**, que é um caracter especial que causa uma quebra de linha. É por isso que a entrada do usuário aparece em uma nova linha.

Se você espera que o usuário digite um inteiro, você deve converter o valor retornado para `int` utilizando a função `int()`:

```
> > > prompt = ‘Qual é a velocidade de uma onça?\n’
> > > velocidade = raw_input(prompt)
Qual é a velocidade de uma onça?
17
> > > int(velocidade)
17
> > > int(velocidade) + 5
22
```

Mas se o usuário digita algo diferente do que você espera, como uma string por exemplo, você receberá um erro:

```
> > > velocidade = raw_input(prompt)
Qual é a velocidade de uma onça?
Uma onça do pantanal ou da floresta amazônica?
> > > int(velocidade)
ValueError: invalid literal for int()
```

Veremos como solucionar este tipo de erro adiante.

2.11 Comentários

Assim que os programas se tornam maiores e mais complexos, eles também ficam mais difíceis de ler. Linguagens formais são densas e normalmente é dificil olhar um pedaço de código e descobrir o que ele faz ou porquê. Por esta razão, é uma uma boa ideia adicionar anotações em seu programa para explicar, em uma linguagem natural, o

que o programa faz. Essas anotações são chamadas de **comentários**, e eles iniciam com o símbolo #:

```
# Compute a porcentagem do tempo decorrido de uma hora
porcentagem = (minuto*100)/60
```

Neste caso, apenas o comentário aparece na linha. Você pode colocar comentários ao final de uma linha de comando:

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

Tudo que aparece a partir do símbolo # até o final da linha é ignorado, não tendo efeito em seu programa. Comentários são muito úteis quando informam características não óbvias de seu código. É razoável assumir que o leitor pode descobrir o que o código faz; é muito mais útil explicar o porque.

Esse comentário é redundante e inútil para o código:

```
v = 5 # v recebe 5
```

Esse comentário contém informações úteis que não estão no código:

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

Bons nomes para variáveis podem reduzir a necessidade de comentários, mas nomes longos podem transformar expressões complexas difíceis de se ler.

2.12 Escolhendo variáveis com nomes mnemônicos

Seguindo as regras simples de nomeação de variáveis e evitando palavras reservadas, você possuirá uma grande quantidade de escolhas para nomear suas variáveis. No início essa escolha pode ser confusa tanto quando você lê o programa como quando você escreve seus próprios programas. Por exemplo, os próximos três programas são idênticos, em relação à sua função, mas muito diferentes quando os lê e tenta entendê-los.

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

horas = 35.0
rate = 12.50
pay = hours * rate
printf pay

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

O interpretador do Python vê todos os três programas exatamente iguais, mas os olhos humanos vêem e os entendem de forma diferente. Pessoas facilmente entenderão o **significado** do segundo programa porque o programador escolheu nomes de variáveis que representam o que será armazenado nelas.

Nós chamamos esses nomes bem pensados de "nomes mnemônicos". A palavra *mnemônico*[4] siginifica "ajuda à memória". Nós escolhemos variáveis com nomes mnemônicos para nos ajudar a lembrar o porque criamos estas variáveis em primeiro lugar.

Apesar de ser uma ótima ideia usar variáveis com nomes mnemônicos, variáveis com nomes mnemônicos podem se tornar uma pedra no sapato de programadores novatos no sentido de desenvolverem suas habilidades para criar e entender códigos. Isso se deve ao fato dos programadores mais novos ainda não memorizarem as palavras reservadas pelo sistema (são apenas 31) e, por vezes, variáveis que possuem nomes muito descritivos parecem fazer parte da linguagem e não apenas nomes de variáveis bem escolhidos.

Observe o exemplo de código em Python que faz um laço de repetição sobre alguns dados. Vamos estudar estruturas de repetição mais adiante, mas por enquanto tente entender o que isso significa:

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

O que está acontecendo aqui? Qual dessas palavras (`for`, `palavras`, `in`, etc.) são palavras reservadas e quais são apenas nomes de variáveis? O Python entende fundamentalmente o siginificado das palavras? Programadores iniciantes têm problemas em separar quais partes do código *devem* ser exatamente como no exemplo e quais partes do código são simples escolhas feitas pelo programador. O código a seguir é igual ao código acima:

[4] Veja http://en.wikipedia.org/wiki/Mnemonic para uma descrição estendida da palavra "mnemonic".

```
for fatia in pizza:
   print fatia
```

É mais fácil para iniciantes avaliar esse código e saber quais partes são palavras reservadas definidas pelo Python e quais são simples nomes de variáveis escolhidos pelo programador. É muito claro que o Python não possui entendimento sobre pizza nem sobre fatias, nem mesmo sobre o fato de que uma pizza consiste em uma ou mais fatias.

Mas se nosso programa está realmente lendo e pesquisando palavras na memória, `pizza` e `fatia` são nomes de variáveis não mnemônicas. Escolher o nome das variáveis diverge do verdadeiro propósito do programa.

Após um curto período de tempo você saberá as palavras reservadas mais comuns e começará a vê-las destacadas no programa.

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

As partes do código que são definidas pelo Python (`for, in, print`, e :) estão em negrito e as variáveis escolhidas (`palavra` e `palavras`) não estão em negrito. Muitos editores de texto reconhecem a sintaxe de Python e vão colorir automaticamente as palavras reservadas para dar dicas e manter o nome das suas variáveis e palavras reservadas separadas. Depois de um tempo você começará a ler códigos em Python e rapidamente determinará o que é uma variável e o que é uma palavra reservada.

2.13 Depuração

Neste ponto os erros de sintaxe mais cometidos por você são nomes de variáveis ilegais, como `class` e `yield`, que são palavras chaves, ou `odd~job` e `US\$`, que contêm caracteres inválidos. Se você colocar espaço em um nome de variável, o Python reconhecerá como dois operandos sem um operador:

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

Para erros de sintaxe, a própria mensagem de erro não ajuda muito. As mensagens mais comuns são `SyntaxError: invalid syntax` e `SyntaxError: invalid token`. Nenhuma das mensagens é muito informativa.

Os erros em tempo de execução que você irá encontrar mais frequentemente são: "use before def;" que se refere ao fato de tentar utilizar uma variável antes de atribuir um valor a ela. Isso pode ocorrer se você digitar o nome da variável de maneira errada:

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

Nomes de variáveis são sensíveis ao caso ("case sensitive"), ou seja, `LaTeX` nao é o mesmo que `latex` para o Python.

Nesse ponto, a causa mais provável de um erro semântico é a ordem das operações. Por exemplo, para calcular $\frac{1}{2\pi}$ você deve escrever

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

Mas a divisão acontece primeiro, então você iria calcular o valor de $\pi/2$, que não é a mesma coisa! Não há como o Python saber o que você pretendia escrever, então nesse caso você não recebe uma mensagem de erro; só receberá uma resposta errada.

2.14 Glossário

atribuição: uma sentença que atribui um valor a uma variável.

concatenar: unir o final do primeiro operando com o início do segundo operando.

comentário: informação significativa em um programa para outros programadores (ou qualquer leitor do código-fonte) e não tem efeito sobre a execução do programa.

avaliar: simplificar uma expressão, executando as operações de modo a produzir um único valor.

expressão: uma combinação de variáveis, operadores e valores que representam um único valor resultante.

ponto flutuante: um tipo de dado que representa números com partes fracionárias.

piso da divisão: a operação que divide dois números, desconsiderando a parte fracionária.

inteiro: um tipo de dado que representa números inteiros.

palavra-chave: uma palavra reservada que é usada pelo compilador para analisar um programa; você não pode usar palavras-chaves como `if, def`, e `while` como nomes de variáveis.

mnemônico: um auxiliar de memória. Frequentemente damos nomes mnemônicos para as variáveis para nos ajudar a lembrar o que é armazenado na variável.

operador módulo: um operador, denotado pelo sinal de porcentagem (`%`), que atua sobre inteiros e produz o resto quando um número é dividido por outro.

operando: um dos valores sobre o qual o operandor é aplicado.

operador: um símbolo especial que representa um cálculo simples como adição, multiplicação ou concatenação de strings.

regras de precedência: o conjunto de regras que controla a ordem em que as expressões envolvendo múltiplos operadores e operandos são avaliadas.

sentença: um trecho de código que representa um comando ou ação. Deste forma, as sentenças que vimos até o momento são sentenças de atribuição e impressão.

string: um tipo que representa sequências de caracteres.

tipo: uma categoria de valores. Os tipos vistos são inteiros (tipo `int`), números com 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 faz referência a um valor.

2.15 Exercícios

Exercício 2.2 Escreva um programa que use `raw_input` para perguntar a ele qual seu nome e dê as boas vindas ao usuário.

```
Digite seu nome: Douglas
Olá Douglas
```

Exercício 2.3 Escreva um programa no qual o usuário digite o número de horas trabalhadas e o valor pago por hora e calcule o salário bruto.

```
Horas: 35
Valor/hora: 2.75
Salário Bruto : 96.25
```

Não vamos nos preocupar em fazer com que o nosso salário tenha exatamente dois dígitos depois do ponto decimal, por enquanto. Se você quiser, você pode brincar com a função `round` que já vem embutida em Python para deixar sempre o número com exatamente duas casas decimais.

Exercício 2.4 Assuma as seguintes declarações de atribuição.

```
largura = 17
altura = 12.0
```

Para cada expressão a seguir, escreva o valor da expressão e o tipo (do valor da expressão).

```
1. largura/2
2. largura/2.0
3. altura/3
4. 1 + 2 *5
```

Use o interpretador de Python para verificar suas respostas.

Exercício 2.5 Escreva um programa que toma como entrada uma temperatura em Celsius, converta-a em Fahrenheit e imprima a temperatura convertida.

3 Execução Condicional

3.1 Expressões Booleanas

Uma **expressão booleana** é uma expressão que pode ser verdadeira ou falsa. Os seguintes exemplos utilizam o operador `==`, o qual compara dois operandos e produz `True` se eles são iguais ou `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:

```
x != y      # x não é igual à y
x > y       # x é maior do que y
x < y       # x é menor do que y
x >= y      # x é maior ou igual a y
x <= y      # é menor ou igual a y
x is y      # x é o mesmo que y
x is not y  #  x não é o mesmo que y
```

Embora esses operadores sejam provavelmente familiares para você, os símbolos de Python são diferentes dos símbolos matemáticos. Um erro comum é usar apenas um sinal de igual (`=`) ao invés de um duplo sinal de igual (`==`). Lembre-se que `=` é um operador de atribuição e `==` é um operador de comparação. Não existe tal engano com os operadores `=<` ou `=>`.

3.2 Operadores Lógicos

Existem três **operadores lógicos**: `and`, `or` e `not`. A semântica (significado) desses operadores é similar ao significado em Inglês. Por exemplo:

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

é verdadeiro apenas se `x` for maior que `0` *e* menor que `10`.

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

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

Estritamente falando, os operandos dos operadores lógicos deveriam ser expressões lógicas, mas o Python não é muito estrito. Qualquer número diferente de zero é interpretado como "verdadeiro".

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

Essa flexibilidade pode ser útil, mas existem algumas construções que a tornam confusas. Você pode querer evitá-los (a menos que saiba o que está fazendo).

3.3 Execução condicional

Com o objetivo de escrever programas úteis, nós quase sempre precisamos avaliar condições e mudar o comportamento do programa de acordo com tais avaliações. **Sentenças condicionais** nos dão essa capacidade. A forma mais simples é a sentença `if`:

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

A expressão booleana após a sentença `if` é chamada de **condição**. Nós terminamos a sentença `if` com o caractere dois pontos (:) e a(s) linha(s) após a(s) sentença(s) condicional(is) `if` é(são) identada(s).

Se a condição lógica é verdadeira, então a sentença identada é executada. Se a condição lógica é falsa, então a sentença identada é ignorada.

Sentenças `if` tem a mesma estrutura que as definições de função ou de estruturas `for`. A sentença consiste de uma linha de cabeçalho que termina com o caractere dois

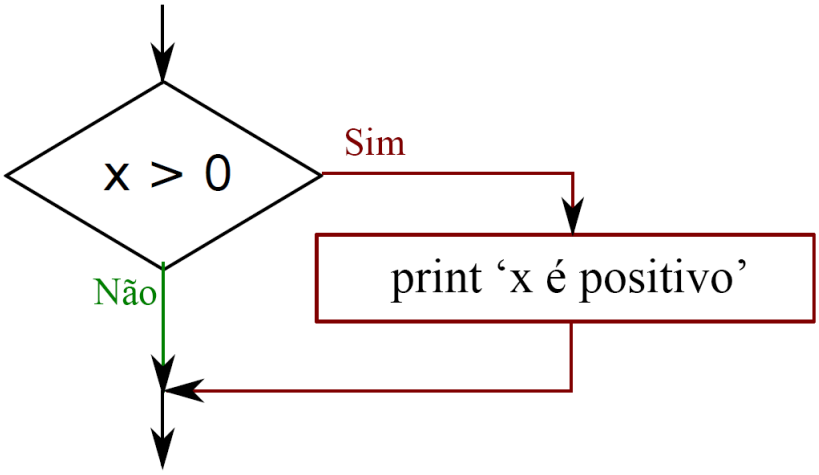


Figura 5: Execução de uma sentença condicional

pontos (:) seguido por um bloco identado. Sentenças como essas são chamadas **sentenças compostas** porque elas se estendem por mais que uma linha.

Não há limite do número de sentenças que podem aparecer no corpo da sentença `if`, mas deve haver pelo menos uma. Ocasionalmente, pode ser necessário ter um corpo sem sentenças (usualmente para marcar o lugar de um código que você não escreveu ainda). Nesse caso, você pode usar a sentença `pass`, que não faz nada.

```
if x < 0 :
   pass # é preciso manipular valores negativos!
```

Se você entrar com uma sentença condicional no interpretador Python, a tela de comando irá mudar de três sinais de > para três pontos para indicar que você está no meio de um bloco de sentenças, como mostrado abaixo:

```
> > > x = 3
> > > if x < 10:
...     print ‘Small’
...
Small
> > >
```

3.4 Execução alternativa

Uma segunda forma da sentença `if` é a execução alternativa, na qual há duas possibilidades e uma condição determina qual será executada. A sintaxe se assemelha a:

```
if x % 2 == 0 :
   print 'x is even'
else:
   print 'x is odd'
```

Se o resto da divisão de `x` por 2 é 0, então sabemos que `x` é par, então o programa exibe uma mensagem para esse fato. Se a condição é falsa, o segundo conjunto de instruções é executado.

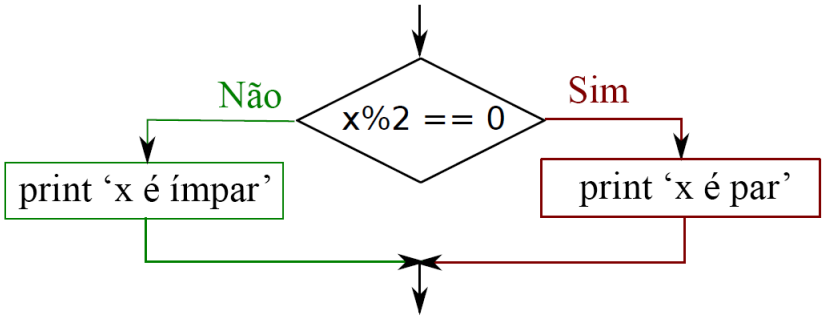


Figura 6: Execução de uma sentença condicional alternativa

Dado que uma condição é sempre verdadeira ou falsa, exatamente uma das alternativas será executada. As alternativas são chamadas **branches**, porque elas são ramos do fluxo da execução.

3.5 Condicionais em Cadeia

Algumas vezes existem mais do que duas possibilidades e, por isso, precisamos de mais do que dois ramos no fluxo de execução. Uma maneira de expressar computacionalmente isso é a condição em cadeia:

```
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 um ramo será executado. Não há limite do número de sentenças `elif`. Se há uma cláusula `else`, ela deve aparecer no final da construção da cadeia. Note, no entanto, que o `else` não é obrigatório.

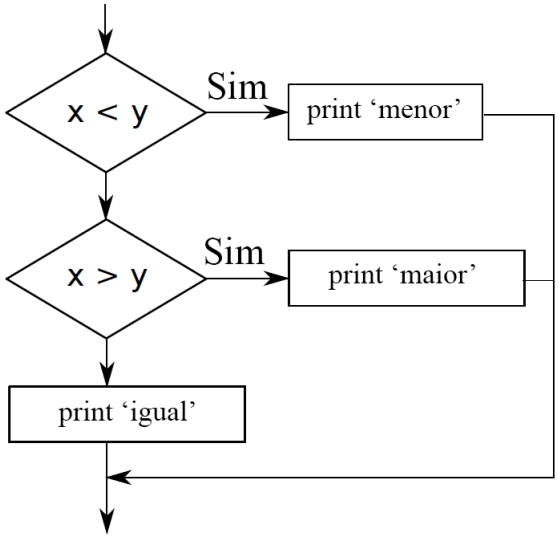


Figura 7: Execução de uma condição em cadeia

```
if choice == 'a':
   print 'Bad guess'
elif choice == 'b':
   print 'Good guess'
elif choice == 'c':
   print 'Close, but not correct'
```

Cada condição é verificada em ordem. Se a primeira é falsa, a próxima é verificada, e assim por diante. Se uma delas for verdadeira, o ramo correspondente é executado, e a sentença termina. Se houver mais de uma condição verdadeira, apenas o primeiro ramo é executado.

3.6 Condicionais Agrupadas

Uma condicional também pode estar contida em outra. Nós poderíamos escrever o seguinte exemplo de tricotomia:

```
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 mais externa contém dois ramos. O primeiro ramo contém uma sentença simples. O segundo ramo contém outra sentença `if`, a qual tem seus dois ramos próprios. Esses dois ramos são ambos sentenças simples, entretanto eles também poderiam

ser sentenças condicionais.

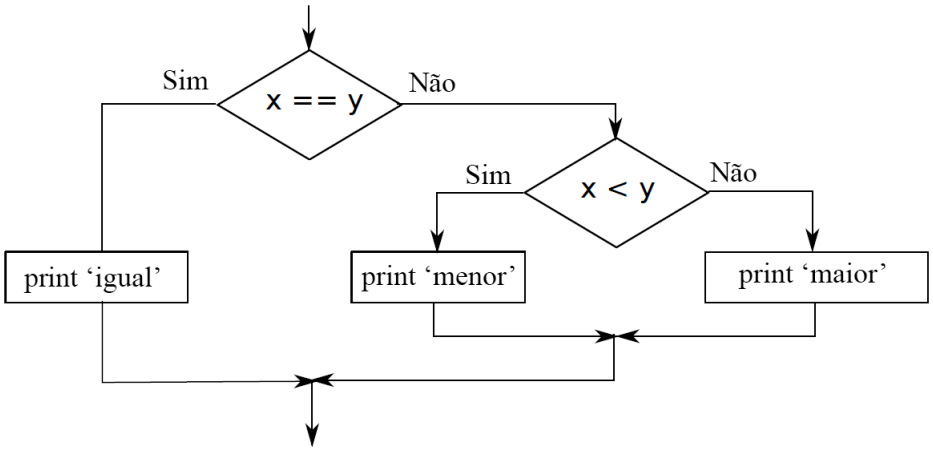


Figura 8: Execução de uma sentença condicional agrupada

Embora a indentação das sentenças faça a estrutura ficar aparente, condicionais agrupadas podem começar a dificultar a leitura. Em geral, é uma boa ideia evitá-las quando puder.

Operadores lógicos, frequentemente, oferecem caminhos para simplificar as condicionais agrupadas. Por exemplo, pode-se reescrever o seguinte código usando uma única condicional:

```
if 0 < x:
   if x < 10:
      print 'x is a positive single-digit number.'
```

A sentença de impressão é executada, apenas se as duas condições forem verdadeiras, assim teríamos o mesmo efeito com o operador `and`:

```
if 0 < x and x < 10:
   print 'x is a positive single-digit number.'
```

3.7 Entendendo exceções usando try e except

Nos capítulos anteriores vimos um segmento de código, no qual tínhamos utilizado as funções `raw_input` e `int` para ler e analisar um número inteiro digitado pelo usuário. Nós também vimos quão traiçoeiro isso poderia ser:

```
> > > speed = raw_input(prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
> > > int(speed)
ValueError: invalid literal for int()
> > >
```

Quando nós estamos executando essas sentenças no interpretador de Python, recebemos uma nova tela de comando do interpretador, e passamos adiante para nossa próxima sentença.

Entretanto, se esse código for colocado em um script de Python e esse erro ocorrer, seu script para imediatamente seu curso com um traceback e não executa a próxima sentença.

Aqui está um exemplo de programa que converte a temperatura de Fahrenheit para Celsius:

```
inp = raw_input('Enter Fahrenheit Temperature:')
fahr = float(inp)
cel = (fahr - 32.0) * 5.0 / 9.0
print cel
```

Se executarmos esse código e colocarmos uma entrada inválida, ele simplesmente falha com uma mensagem de erro pouco amigável:

```
python fahren.py
Enter Fahrenheit Temperature:72
22.2222222222

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

Há uma estrutura codicional de execução no Python que pode lidar com esses tipos de erros (esperados ou não esperados) chamados "try / except". O objetivo do `try` e do `except` é, dado que você sabe que erros podem ocorrer, permitir a adição de sentenças para tratar tais erros. Essas sentenças extras (o bloco de exceção) são ignoradas se não houver erro.

`try` e `except` podem ser entendidos como uma "política de segurança" do Python em uma sequência de sentenças. Nós poderíamos reescrever o conversor de temperatura como segue:

```
inp = raw_input('Enter Fahrenheit Temperature:')
try:
   fahr = float(inp)
   cel = (fahr - 32.0) * 5.0 / 90.
   print cel
except:
   print 'Please enter a number'
```

Python começa executando a sequência de sentenças no bloco do `try`. Se tudo estiver correto, o bloco `except` é ignorado e a execução continua. Se uma exceção (erro) ocorrer no bloco do `try`, a execução segue para as sentenças do bloco `except`.

```
python fahren2.py
Enter Fahrenheit Temperature:72
22.2222222222

python fahren2.py
Enter Fahrenheit Temperature:fred
Please enter a number
```

Lidar com exceções pelo uso do `try` é chamado de **capturar** exceções. Neste exemplo, a cláusula `except` imprime uma mensagem de erro. Em geral, capturar a exceção dá a você a chance de consertar o problema, tentar novamente, ou pelo menos, terminar o programa graciosamente.

3.8 Avaliações de caminhos curtos de expressões lógicas

Quando Python processa uma expressão lógica tal como `x >= 2 e (x/y)>2`, a expressão é avaliada da esquerda para a direita. Devido à definição de `and`, se `x` é menor que 2, a expressão `x >= 2` é `False` e então a expressão inteira é `False` mesmo que `(x/y) > 2` seja avaliada como `True`.

Quando Python detecta que não há nada a ganhar por avaliar o resto das expressão lógica, ele para de avaliar e não computa o valor do resto da expressão. Quando a avaliação de uma expressão lógica para porque todo o valor já é conhecido, ocorre o fato chamado de **encurtar** o caminho da avaliação, ou **short-circuiting**.

Enquanto isso pode parecer um bom método, o comportamento do caminho curto leva à técnica chamada de **padrão guardião**. Considere a seguinte sequência de código

no interpretador de 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` era 0, o que causou um erro em tempo de execução. Mas o segundo exemplo não falhou porque a primeira parte da expressão `x >= 2` foi avaliada como `False` então o `(x/y)` nunca foi executado, devido à regra do caminho curto.

Nós podemos construir a expressão lógica para, estrategicamente, colocar uma avaliação de guarda, logo antes da avaliação que possa 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 >= 2` é `False` então a avaliação para no `and`. Na segunda expressão `x >= 2` é True mas `y != 0` é `False`, então a operação `(x/y)` não é executada.

Na terceira expressão, o `y!=0` está depois do cálculo de `(x/y)`, então a expressão falha com um erro.

Na segunda expressão, dizemos que `y!=0` atua como um guarda para assegurar que executemos `(x/y)` somente se `y` não for zero.

3.9 Depuração

O traceback do Python é exibido quando ocorre um erro contendo uma grande quantidade de informações. E isso pode ser um fator de complicação. Entretanto, as partes fundamentais são:

- O tipo de erro ocorrido; e
- Onde o erro ocorreu.

Erros de sintaxe são normalmente fáceis de se encontrar. Erros de espaços em branco podem ser difíceis porque espaços e tabs são invisíveis e estamos acostumados a ignorá-los.

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

Nesse exemplo, o problema é que a segunda linha foi indentada com um espaço. Mas a mensagem de erro aponta para o `y`, erroneamente. Em geral, mensagens de erro indicam onde o problema foi descoberto, mas o erro em questão ocorreu, de fato, antes desse código, algumas vezes na linha anterior. O mesmo é verdade para erros em tempo de execução. Suponha que você esteja tentando computar um sinal/ruído em decibéis. A fórmula é $SNR_{db} = 10log_{10}(P_{sinal}/P_{barulho})$. Em Python você deve escrever:

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

Mas quando você o executa, você consegue uma mensagem de erro[1]:

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

A mensagem de erro indica a linha 5, mas não há nada de errado nessa linha. Para achar o erro real, pode ser útil imprimir o valor de `ratio`, que no caso é 0. O problema está na linha 4, porque dividindo-se dois inteiros, o resultado será também um inteiro. A solução é para representar um valor da potência e do barulho, ambos em ponto flutuante.

Em geral, mensagens de erro dizem para você onde o problema foi descoberto, mas esse, geralmente, não é o lugar onde o mesmo ocorreu.

3.10 Glossário

corpo: sequência de sentenças dentro de uma sentença composta.

expressão booleana: Uma expressão cujo valor é `True` ou `Falso`.

ramo: Uma das alternativas de sequências de sentenças em uma sentença condicional.

condicional em cadeia: Uma sentença condicional com uma série de ramos alternativos.

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

sentença condicional: Uma sentença que controla o fluxo de execução depen-

[1] Em Python 3.0, você não recebe mais uma mensagem de erro, o operador de divisão realiza uma divisão em float mesmo com operandos inteiros.

dendo de uma condição.

condição: A expressão booleana em uma sentença condicional que determina qual ramo será executado.

sentença composta: Uma sentença que contém um cabeçalho e um corpo. O cabeçalho termina com dois pontos (:). O corpo é identado relativamente ao cabeçalho.

padrão guardião: Expressão lógica com comparações adicionais a fim de garantir a corretude sobre o comportamento dos caminhos curtos.

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

condicionais agrupadas: Uma sentença condicional que aparece em um dos ramos de outra sentença condicional.

traceback: Lista de funções que estão executando, impressa quando uma exceção ocorre.

caminho curto: Quando o Python está em um percurso de avaliação da expressão lógica e para de avaliar porque o Python sabe o resultado final para a expressão sem precisar avaliar o resto da expressão.

3.11 Exercícios

Exercício 3.1 Reescreva seu cálculo de pagamento para dar ao empregado 1,5 vezes o valor da hora para quem trabalhou acima de 40 horas.

```
Entre com as horas: 45
Entre com o valor da hora: 10
Pagamento: 475.0
```

Exercício 3.2 Reescreva seu programa de pagamento usando `try` e `except` tal que seu programa trate a entrada não numérica, imprimindo uma mensagem e finalizando a execução do mesmo neste caso. O trecho a seguir mostra duas execuções do programa:

```
Digite as horas: 20
Digite o valor da hora: nine
Erro! Por favor, digite uma entrada numérica.

Entre com as horas: fourty
Erro! Por favor, digite uma entrada numérica.
```

Exercício 3.3 Escreva um programa que solicite um valor entre 0.0 e 1.0. Se o valor está fora do intervalo imprima um erro. Se o valor está entre 0.0 e 1.0, imprima um conceito de acordo com a tabela a seguir:

```
Score    Grade
>= 0.9    A
>= 0.8    B
>= 0.7    C
>= 0.6    D
< 0.6    F

Digite uma pontuação: 0.95
A

Digite uma pontuação: perfect
Pontuação Ruim

Digite uma pontuação: 10.0
Pontuação Ruim

Digite uma pontuação: 0.75
C

Digite uma pontuação: 0.5
F
```

Execute o programa repetidamente como mostrado acima para testar as saídas para diferentes entradas.

4 Funções

4.1 Chamada de funções

No contexto de programação, uma função é uma sequência de comandos que realiza um cálculo. Quando você define uma função, é preciso especificar o nome e a sequência de comandos. Depois, você poderá "chamar" a função pelo nome. Nós já vimos um exemplo uma **chamada de função**:

```
> > > > type(32)
<type 'int'>
```

O nome da função é `type`. A expressão em parênteses é chamada de **argumento** da função. O argumento é um valor ou variável que estamos passando para a entrada da função. O resultado, para a função `type`, é o tipo do argumento.

É comum dizer que a função "recebe" o argumento e "retorna" um resultado. Esse resultado é chamado **valor de retorno**.

4.2 Funções Embutidas

Python fornece diversas funções embutidas das quais podemos usar sem antes definí-las. Os criadores do Python escreveram um conjunto de funções para resolver problemas comuns e as incluiram no Python, para que possamos usar.

As funções `max` e `min` são funções que nos devolvem o maior e o menor valor de um lista, respectivamente.

```
> > > > max('Hello world')
'w'
> > > > min('Hello world')
' '
> > > >
```

A função `max` nos diz qual "o maior caractere" em uma string (o que retorna a letra "w" para o exemplo anterior) e a função `min` mostra qual o menor caractere, que é o espaço, considerando a frase "Hello world".

Outra função embutida muito comum é a função `len`, a qual nos diz quantos símbolos existem no argumento. Se o argumento para `len` é uma string, ele retorna o

número de caracteres presentes na string.

```
> > > > len("Hello world")
11
> > > >
```

É prudente considerar o nome das funções embutidas do Phyton como palavras reservadas. Assim, o uso de uma variável de nome `MAX`, por exemplo, deve ser evitado.

4.3 Função de conversão de tipo

Python contém funções para converter valores de um tipo para outro. A função `int` recebe um valor e o converte para um número inteiro, se possível, caso contrário retorna uma mensagem de erro.

```
> > > > int (‘32’)
32
> > > > int (‘Hello’)
ValueError: invalid literal for int(): Hello
```

`int` pode converter valores com ponto flutuante para inteiros, mas isso não arredonda o valor; só remove a parte fracionária do valor.

```
> > > > int (3.99999)
3
> > > > int (-2.3)
-2
```

`float` converte inteiros e strings para números com ponto flutuante:

```
> > > > float (32)
32.0
> > > > float(‘3.14159’)
3.14159
```

Finalmente, `str` converte seu argumento para uma string:

```
> > > > str(32)
‘32’
> > > > str(3.14159)
‘3.14159’
```

4.4 Números aleatórios

Dadas as mesmas entradas, a maioria dos programas de computador geram a mesma saída toda vez que executados, então eles são ditos **determinísticos**. O Determinismo de um programa é geralmente uma característica boa, uma vez que esperamos que um mesmo cálculo retorne sempre um mesmo resultado. Em algumas aplicações, entretanto, queremos que o computador seja imprevisível. Jogos é um exemplo óbvio, mas ainda existem outros.

Fazer um programa verdadeiramente não determinístico não é uma coisa fácil, mas existem maneiras de fazê-los parecer com não determinísticos. Uma dessas maneiras é usar **algoritmos** que gerem números **pseudoaleatórios**. Números pseudoaleatórios não são realmente aleatórios, porque eles são gerados por um cálculo determinístico, mas pela avaliação superficial de números pseudoaleatórios é impossível distinguí-los de números aleatórios.

O módulo `random` fornece funções que geram números pseudoaleatórios (os quais serão chamados simplesmente de "aleatórios" daqui em diante).

A função `random` retorna valores reais aleatórios entre 0.0 e 1.0 (incluindo o 0.0 mas não o 1.0). Cada vez que `random` é chamada, você recebe um número distinto. Para ver uma amostra, execute esse laço:

```
import random

for i in range (10):
   x = random.random()
   print x
```

Esse programa produz a seguinte lista de 10 números aleatórios entre 0.0 e 1.0, não incluindo o 1.0.

```
0.301927091705
0.513787075867
0.319470430881
0.285145917252
0.839069045123
0.322027080731
0.550722110248
0.366591677812
0.396981483964
0.838116437404
```

Exercício 4.1 Execute este programa no seu sistema e veja quais números foram gerados. Execute o programa mais de uma vez e veja quais números você obteve.

A função `random` é uma de muitas funções que fornecem números aleatórios. A função `randint` recebe como parâmetros valores correspondentes à `menor` e `maior` e retorna um inteiro neste intervalo (incluindo ambos).

```
> > > random.randint(5, 10)
5
> > > random.randint(5, 10)
9
```

Para escolher aleatoriamente um elemento de uma sequência, você pode utilizar a função `choice`:

```
> > > t = [1, 2, 3]
> > > random.choice(t)
2
> > > random.choice(t)
3
```

O módulo `random` só fornece funções para gerar valores aleatórios de uma distribuição contínua incluindo Gaussiana e exponencial.

4.5 Funções matemáticas

Python possui um módulo matemático que fornece muitas funções matemáticas familiares. Antes de podermos usar tal módulo, é preciso importá-lo:

```
> > > import math
```

Esta sentença cria um **objeto módulo** nomeado `math`. Se você imprimir o objeto módulo, você obterá alguma informação sobre ele:

```
> > > print math
<modue 'math' from '/usr/lib/python2.5/lib-dynload/mah.so'>
```

O objeto módulo contém definições de funções e variáveis. Para acessar uma dessas funções, você tem que especificar o nome do módulo e o nome da função separado por um ponto. Esse formato é chamado de **notação de ponto**.

```
> > > razao = potencia_sinal / potencia_ruido
> > > decibeis = 10*math.log10(razao)

> > > radianos = 0.7
> > > altura = math.sin(radianos)
```

O primeiro exemplo calcula o logaritmo na base 10 da razão entre sinal e ruído. O módulo matemático (`math`) contém também uma função chamada `log` que calcula o logaritmo na base `e`.

O segundo exemplo determina o valor do seno dado em `radianos`. O nome da variável (`radianos`) é uma dica para informá-lo que a função `sin` e outras funções trigonométicas (`cos`, `tan`, etc.) recebem o argumento em radianos. Para converter de graus para radianos, divide-se por 360 e multiplica-se por 2π:

```
> > > graus = 45
> > > radianos = graus / 360.0 * 2 * math.pi
> > > math.sin(radianos)
0.707106781187
```

A expressão `math.pi` retorna o valor da variável `pi` do módulo matemático. O valor dessa variável é uma aproximação do valor de π, com acurácia de cerca de 15 dígitos.

Se você conhece trigonometria, então você poderá verificar o resultado anterior comparando-o com a raiz quadrada de 2 dividida por 2.

```
> > > math.sqrt(2) / 2.0
0.707106781187
```

4.6 Adicionando novas funções

Até agora, apenas funções que vieram junto com o Python foram utilizadas, mas é possível também adicionar novas funções. A **definição de uma função** especifica o nome desta nova função e a série de passos que ela executará quando for chamada. Uma vez definida a função, nós podemos usá-la em qualquer parte do programa.

Aqui está um exemplo:

```
def print_lyrics():
    print "I'm a lumberjack, and I'm okay."
    print 'I sleep all night and i work all day.'
```

`def` é a palavra-chave que indica que uma função será definida. O nome dessa função é `print_lyrics`. As regras para os nomes das funções são as mesmas dos nomes

de variáveis: letras, números e alguns sinais de pontuação são permitidos, mas o primeiro caractere não pode ser um número. Você não pode usar uma palavra-chave como nome da sua função, e você deve evitar ter variaveis e funções com o mesmo nome.

Os parênteses vazios após o nome da função indicam que ela não recebe argumento. Mais adiante construiremos funções que receberão argumentos em suas entradas.

A primeira linha da definição de uma função é chamada de **cabeçalho**; e o resto é chamado de **corpo**. O cabeçalho tem que terminar com dois pontos e o corpo tem que ser identado. Por convenção, a identação possui sempre quatro espaços. O corpo pode conter qualquer número de declarações.

Frases em declarações de impressão são envolvidas com aspas duplas. Aspas simples e aspas duplas exercem o mesmo papel; a maioria das pessoas usam aspas simples, exceto em casos onde um apóstrofo aparece na frase a ser impressa.

Se você escrever a definição de uma função no modo interativo, será impresso reticências (. . .) para você saber que a definição ainda não está completa:

```
> > > > def print_lyrics():
...     print "I'm a lumberjack, and I'm okay."
...     print 'I sleep all night and I work all day.'
...
```

Para terminar a função, você tem que deixar uma linha em branco (isso não é necessário no código).

Definir uma função cria uma variável com o mesmo nome.

```
> > > print print_lyrics
<function print_lyrics at 0xb7e99e9c>
> > > print type(print_lyrics)
<type 'function'>
```

O valor de `print_lyrics` é um **objeto função**, que tem tipo `'function'`.

A sintaxe para chamar a nova função é a mesma para chamar funções embutidas:

```
> > > > print_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

Uma vez definida a função, você pode usá-la dentro de outra função. Por exemplo, para repitir o refrão anterior, nós podíamos escrever uma função chamada `repeat_lyrics`:

```
def repeat_lyrics():
    print_lyrics()
    print_lyrics()
```

e então chamamos a `repeat_lyrics`:

```
> > > > repeat_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and i work all day.
I'm a lumberjack, and I'm okay.
I sleep all night and i work all day.
```

4.7 Definições e Modo de Utilização

Reunindo os fragmentos de código da seção anterior, o programa por completo é:

```
def print_lyrics():
    print "I'm a lumberjack, and I'm okay."
    print 'I sleep all night and I work all day.'

def repeat_lyrics():
    print_lyrics()
    print_lyrics()

repeat_lyrics()
```

Este programa contém duas definições de funções: `print_lyrics` e `repeat_lyrics`. Definições de funções são executadas como quaisquer outros comandos, mas a intenção é criar objetos de função. Os comandos dentro da função não são executados até que a função seja chamada, e a definição da função não gera nenhuma saída.

Como é de se imaginar, é preciso criar uma função antes de executá-la. Em outras palavras, a definição da função tem que ser realizada antes que uma chamada a mesma seja feita.

Exercício 4.2 Mova a última linha do programa anterior para o topo, de modo que a chamada às funções sejam realizadas antes das definições. Execute o programa e veja que mensagem de erro é retornada.

Exercício 4.3 Mova a chamada de função de volta para baixo e mova a definição de `print_lyrics` após a definição de `repeat_lyrics`. O que acontece quando você executa este programa?

4.8 Fluxo de execução

A fim de assegurar que uma função seja definida antes do seu primeiro uso, você tem que saber a ordem em que os comandos são executados, o que é chamado de **fluxo de execução**.

A execução sempre começa com a primeira instrução do programa. As instruções são executadas uma de cada vez, ordenadas de cima para baixo.

As definições de função não alteram o fluxo de execução do programa, mas lembre-se que comandos dentro da função não são executados até que a função seja chamada.

Uma chamada de função é como um desvio no fluxo de execução. Em vez de ir para a próxima instrução, o fluxo segue para o corpo da função, executa todas as declarações da função, e depois volta para continuar de onde parou.

Isso soa bastante simples, até você lembrar que uma função pode chamar outra. Enquanto executa os comandos dentro de uma função, o programa pode ter que executar os comandos de uma outra função. E ao executar essa nova função, o programa pode ter de executar ainda outra função!

Felizmente, Python é bom em manter o controle de onde está, assim, cada vez que uma função é concluída, o programa retoma de onde parou na função que a chamou. Quando se chega ao fim do programa, ele termina.

Entretanto, quando você lê um programa, nem sempre quer fazê-lo de cima para baixo. Às vezes, faz mais sentido uma leitura seguindo o fluxo de execução.

4.9 Parâmetros e Argumentos

Algumas das funções embutidas que temos visto exigem argumentos. Por exemplo, quando você chama `math.sin`, é preciso passar um número como argumento. Algumas funções recebem mais de um argumento: `math.pow` necessita de dois, a base e o expoente.

Dentro da função, os argumentos são atribuídos a variáveis chamadas **parâmetros**. Eis um exemplo de uma função definida pelo usuário que recebe um argumento:

```
def print_twice(bruce):
    print bruce
    print bruce
```

Esta função atribui o argumento a um parâmetro chamado `bruce`. Quando a função é chamada, é impresso o valor do parâmetro (seja ele qual for) duas vezes.

Esta função funciona com qualquer valor que possa ser impresso.

```
> > > print_twice('Spam')
Spam
Spam
> > > print_twice(17)
17
17
> > > print_twice(math.pi)
3.14159265359
3.14159265359
```

As mesmas regras de composição que se aplicam às funções nativas também se aplicam às funções definidas pelo usuário, então nós podemos usar qualquer tipo de expressão como um argumento para `print_twice`:

```
> > > print_twice('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
> > > print_twice(math.cos(math.pi))
-1.0
-1.0
```

O argumento é avaliado antes que a função seja chamada, por isso, nos exemplos as expressões `'Spam' * 4` e `math.cos(math.pi)` só são avaliadas uma vez.

Você também pode usar uma variável como argumento:

```
> > > michael = 'Eric, the half a bee.'
> > > print_twice(michael)
Eric, the half a bee.
Eric, the half a bee.
```

4.10 Funções Produtivas e Funções Void

Algumas das funções que estamos usando, tais como as funções matemáticas, produzem resultados; por falta de um nome melhor, a chamamos de **funções produtivas**. Outras funções, como `print_twice`, executam uma ação, mas não retornam um valor. Elas são chamadas de **funções void (vazias)**.

Quando uma função produtiva é chamada, quase sempre qse deseja fazer algo com o resultado por ela produzido; por exemplo, atribuir tal resultado a uma variável ou usá-lo como parte de uma expressão:

```
x = math.cos(radians)
golden = (math.sqrt(5) + 1) / 2
```

Quando você chama uma função no modo interativo, Python exibe o resultado:

```
> > > math.sqrt(5)
2.2360679774997898
```

Mas em um script, se você chamar uma função não void e não armazenar o resultado desta em uma variável, o valor de retorno será desprezado!

```
math.sqrt(5)
```

Este script calcula a raiz quadrada de 5, mas se você não armazena o resultado em uma variável ou exibe o resultado, não há muita utilidade.

As funções `void` podem exibir algo na tela ou terem algum outro efeito, mas não têm um valor de retorno. Se você tentar atribuir o resultado a uma variável, obterá um valor especial chamado `None`.

```
> > > result = print_twice('Bing')
Bing
Bing
> > > print result
None
```

O valor `None` não é o mesmo que a string de 'None'. É um valor especial que possui seu próprio tipo:

```
> > > print type(None)
<type 'NoneType'>
```

Para retornar um valor de uma função, usamos o comando `return` na nossa função. Por exemplo, poderíamos criar uma função muito simples chamada `addtwo` que adiciona dois números e retorna um resultado.

```
def addtwo(a, b):
    added = a + b
    return added

x = addtwo(3, 5)
print x
```

Quando esse script é executado, a instrução `print` irá imprimir "8" porque a função `addtwo` foi chamada com os números 3 e 5 como argumentos. Dentro da função os parâmetros `a` e `b` receberam os valores 3 e 5, respectivamente. A função calcula a soma

dos dois números e os coloca na variável local chamada `added` e utiliza a instrução `return` para enviar o valor calculado de volta para o local no código onde a chamada foi realizada. Este valor é atribuído à variável `x` e, em seguida, impresso.

4.11 Por que utilizar funções?

Pode não ser evidente por que vale a pena dividir um programa em funções. Entretanto, há várias razões:

- Criar uma nova função lhe dá a oportunidade de nomear um grupo de instruções, o que torna o programa mais fácil de ler, entender e depurar.
- As funções podem fazer com que um programa tenha uma quantidade menor de linhas por eliminar linhas de código repetitivas. Se mais tarde você precisar realizar alguma mudança, basta fazer isso uma única vez e em um só lugar.
- Dividir um programa longo em funções permite que você resolva as partes de um problema e depois, junte todas estas soluções.
- Funções bem projetadas são frequentemente utilizadas em muitos programas. Uma vez que você escreve e depura uma função, você pode reutilizá-la.

Durante todo o resto do livro, muitas vezes, vamos utilizar a definição de função para explicar um conceito. Parte da habilidade de criar e utilizar funções exige que você entenda corretamente uma ideia como, por exemplo: "encontrar o menor valor em uma lista de valores". Mais tarde vamos mostrar o código que encontra o menor valor em uma lista de valores e apresentaremos a você como uma função chamada `min` que leva uma lista de valores como seu argumento e retorna o menor valor na lista.

4.12 Debugging

Se você está usando um editor de textos para escrever seus códigos, você poderá ter alguns problemas com a quantidade de espaços e tabulações. A melhor maneira de evitar esses problemas é usar somente espaços e não usar tabulações. A maioria dos editores de textos que contém suporte para Phyton substituem o tab por quatro espaços, mas existem editores que não fazem isso.

Tabs e espaços geralmente são invisíveis, o que torna mais difícil corrigir os erros. Procure um editor de textos que administre a identação para você.

Não esqueça de salvar os programas antes de executá-los. Alguns ambientes de desenvolvimento fazem isso automaticamente, mas outros não. Dessa forma, o programa que você está vendo no editor de textos pode não ser o mesmo que está sendo executado.

Corrigir erros pode demorar se continuar executando o programa errado por diversas vezes.

Tenha certeza de que o código que você está vendo é o mesmo que está sendo executado. Se você não tiver certeza disso, coloque um `print 'Hello'` no início do programa e execute-o novamente. Se você não ver `'Hello'` no início da execução, então o programa que está sendo executado não é o correto!

4.13 Glossário

Algoritmo: Um processo geral para resolver um tipo de problema.

Argumento: Um valor dado à função quando ela é chamada. Esse valor é atribuído ao parâmetro correspondente da função.

Corpo: Sequência de passos dentro da definição de uma função.

Composição: Usar uma expressão como parte de uma expressão maior, ou uma sentença como parte de uma sentença maior.

Determinístico: Relativo a um programa que faz a mesma coisa toda vez que é executado, dadas as mesmas entradas.

Notação de ponto: Sintaxe utilizada para chamar uma função de um outro módulo. Tal chamada é feita pela especificação do nome do módulo seguido por um ponto e o nome do módulo.

Fluxo de execução: Ordem em que uma sequência de passos é executada.

Função produtiva: Função que retorna um valor.

Função: Uma sequência de passos nomeada. As funções podem ou não ter argumentos e podem ou não produzir resultados.

Chamada de função: Declaração que executa uma função. Consiste do nome da função seguido de uma lista de argumentos.

Definição da função: Declaração que cria uma nova função, especificando o nome, parâmetro e as declarações da função.

Objeto de função: Valor criado por uma instrução `import` que permite acesso aos dados e códigos definidos em um módulo.

Cabeçalho: Primeira linha de uma definição de função.

Declaração `import`: Declaração de leitura de um arquivo de módulo e criação de um objeto de módulo.

Objeto de módulo: Valor criado pela declaração `import`, o que torna possível acessar funções e variáveis definidas no módulo.

Parâmetro: Nome usado dentro de uma função para se referir ao valor passado como argumento para a função.

Pseudo aleatório: Refere-se a uma sequência de números que aparentemente é aleatória, mas é gerada de maneira determinística.

Retorno de valor: Resultado de uma função. Se a chamada à função ocorre em uma expressão, então o valor a ser retornado é o valor da expressão.

Função `void`: Uma função que não retorna valor.

4.14 Exercícios

Exercício 4.4 Qual o propósito da palavra-chave `def` em Python?
a) É uma gíria que significa "o seguinte código é muito legal"
b) Indica o começo de uma função
c) Indica que a sequência de código identada seguinte deve ser armazenada para posterior utilização.
d) b e c são ambas verdadeiras
e) n.d.a

Exercício 4.5 O que o programa a seguir imprime ?

```
def fred():
    print "Zap"
def jane():
    print "ABC"

jane()
fred()
jane()
```

a)Zap ABC jane fred jane
b)Zap ABC Zap
c)ABC Zap jane
d)ABC Zap ABC
e)Zap Zap Zap

Exercício 4.6 Reescreva o programa de computação de pagamento com um tempo e meio das horas extras e crie uma função chamada `computepagamento` que recebe dois parâmetros (o número de horas e o valor pago por hora).

```
Entre com as horas: 45
Entre com o valor da hora: 10
Pagamento: 475.0
```

Exercício 4.7 Reescreva o programa de notas do capítulo anterior usando uma função chamada `computegrau` que recebe uma pontuação como parâmetro e retorna um grau, sendo este último uma string.

```
Score  Grade
> 0.9  A
> 0.8  B
> 0.7  C
> 0.6  D
<= 0.6  F


Execução do Programa:

Entre com a pontuação: 0.95
A

Entre com a pontuação: perfeito
Pontuação ruim

Entre com a pontuação: 10.0
Pontuação ruim

Entre com a pontuação: 0.75
C

Entre com a pontuação: 0.5
F
```

Execute o programa várias vezes para testar diferentes valores de entrada.

5 Iteração

5.1 Atualizando variáveis

Um padrão comum em instruções de atribuição é uma declaração de atribuição que atualiza uma variável - onde o novo valor da variável depende do valor anterior.

```
x = x + 1
```

A instrução anterior significa "obter o valor atual de x, adicionar um, e depois atualizar `x` com o novo valor".

Se você tentar atualizar uma variável que não existe, você recebe um erro, porque Python avalia o lado direito antes de atribuir um valor para `x`:

```
> > > x = x + 1
NameError: name ‘x’ is not defined
```

Antes que você possa atualizar uma variável, você tem que inicializá-la, geralmente com uma simples atribuição:

```
> > > x = 0
> > > x = x + 1
```

Atualizar uma variável, adicionando 1, é chamado de um **incremento**; subtraindo 1 é chamado um **decremento**.

5.2 A instrução while

Computadores são muitas vezes utilizados para automatizar tarefas repetitivas. Repetir tarefas idênticas ou similares sem cometer erros é algo que os computadores fazem bem e as pessoas fazem mal. Por iteração ser tão comum, Python fornece vários recursos de linguagem para torná-lo mais fácil.

Uma forma de iteração em Python é a sentença `while`. Aqui vai um programa simples que faz a contagem regressiva de cinco até zero e, em seguida, diz: “Fim!”.

```
n = 5
while n> 0:
    print n
    n = n-1
print 'Fim!'
```

Você quase pode ler o comando `while` como se fosse em Português. Significa: "Enquanto `n` for maior que 0, exibe o valor de `n` e, em seguida, reduz o valor de `n` em 1. Quando você chega em 0 para o valor de `x`, saia do comando while e exiba a palavra "Fim!".

Mais formalmente, segue o fluxo de execução para um comando `while`:

1. Avaliar a condição, produzindo `True` ou `False`.

2. Se a condição é falsa, saia do comando `while` e continue a execução da próxima instrução.

3. Se a condição é verdadeira, execute o corpo e, em seguida, volte para o passo 1.

Este tipo de fluxo é chamado de **loop** porque após a terceiro passo volta-se ao início (passo 1). Cada vez que executamos o corpo do laço, chamamos isso de uma **iteração**. Para o **loop** acima, diríamos, "Tinha cinco iterações" o que significa que o corpo do loop foi executado cinco vezes.

O corpo do **loop** deveria alterar o valor de uma ou mais variáveis, de modo que, eventualmente, a condição se torna falsa e o loop termina. Chamamos a variável que muda cada vez que o loop é executado e controla quando o **loop** é finalizado de **variável de iteração**. Se não houver nenhuma variável iteração, o **loop** repetirá para sempre, resultando em um **loop infinito**.

5.3 **Loops** infinitos

Uma fonte inesgotável de diversão para os programadores é a observação de que as instruções no shampoo, "Lave, enxágüe, repita" é um loop infinito, porque não há nenhuma **variável de iteração** para dizer-lhe quantas vezes executar o **loop**.

No caso do exemplo de loop while dado anteriormente, podemos provar que o loop termina porque sabemos que o valor de `n` é finito, e podemos ver que o valor de `n` menor a cada iteração do loop, então, eventualmente, temos de chegar a 0. Outras vezes um loop é obviamente infinito porque não tem nenhuma variável de iteração.

5.4 "Loops infinitos" e o comando break

Às vezes, você não sabe que é hora de acabar um loop até chegar ao meio do corpo do loop. Nesse caso, você pode escrever um loop infinito de propósito e, em seguida, usar a declaração `break` para saltar para fora do loop.

Este loop é, obviamente, um **loop infinito** porque a expressão lógica na sentença `while` é simplesmente a constante lógica `True`:

```
n = 10
while True:
    print n,
    n = n - 1
print 'Done!'
```

Se você cometer o erro e executar este código, você vai aprender rapidamente como parar um processo Python que não para em seu sistema ou descobrir onde é o botão de desligar em seu computador. Este programa executará para sempre ou até que a bateria se esgote porque a expressão lógica no topo do loop é sempre verdadeira, em virtude do fato de que a expressão é o valor constante `True`.

Enquanto este é um loop infinito disfuncional, ainda podemos usar este padrão para construir laços úteis, enquanto nós adicionarmos cuidadosamente código ao corpo do loop para explicitamente sair do loop usando break, quando chegarmos à condição de saída.

Por exemplo, suponha que você queira receber uma entrada do usuário até que ele digite "done".

Você poderia escrever:

```
while True:
    line = raw_input ('> ')
    if line == 'done':
        break
    print line
print 'Done!'
```

A condição do loop é `True`, o que é sempre verdade, de modo que o loop é executado repetidamente até que seja executado o comando `break`.

A cada iteração é mostrado o símbolo > solicitando a entrada do usuário. Se o usuário digitar `done`, a instrução `break` faz com que o loop seja interrompido. Caso contrário, o programa repete o que quer que o usuário digite e volta ao topo do loop. Aqui está uma execução simples:

```
> Hello There
Hello There
> finished
finished
> done
Done!
```

Esta forma de escrever `while` é comum, pois você pode verificar a condição em qualquer lugar do loop (e não apenas no início do loop) e você pode expressar a condição de parada afirmativamente ("parar quando isso acontece") ao invés de negativamente ("continuar até que isso aconteça.").

5.5 Finalizando iterações com continue

Às vezes, você está em uma iteração de um loop e quer terminar a iteração atual e saltar imediatamente para a próxima iteração. Nesse caso, você pode usar a função `continue` para passar para a próxima iteração sem terminar o corpo do loop para iteração atual.

Aqui está um exemplo de um loop que copia a sua entrada até que o usuário digite "done", e trato as linhas que começam com o carácter # como linhas que não devem ser impressas (como comentários em Python).

```
while True:
    line = raw_input ('>')
    if line [0] == '#':
        continue
    if line == 'done':
        break
    print line
print "Done!"
```

Aqui está um exemplo de execução deste novo programa com `continue` acrescentado.

```
> hello there
hello there
> # don't print this
> print this!
print this! > done
Done!
```

Todas as linhas são impressas, exceto aquela que começa com o caracte #, porque quando o `continue` é executado, ele termina a iteração atual e segue para o início do

`while` para iniciar a próxima iteração, ignorando, assim, a sentença `print`.

5.6 Laços usando for

Às vezes queremos fazer um loop sobre um **conjunto** de elementos, como uma lista de palavras, as linhas em um arquivo ou uma lista de números. Quando temos uma lista de elementos para o loop, podemos construir um loop definido usando a instrução `for`. Chamamos a instrução `while` de um loop indefinido porque ele simplesmente repete o loop até que alguma condição se torne `False` enquanto que o laço `for` é executado sobre um conjunto conhecido de itens. Assim, são executadas tantas iterações quantos forem os itens no conjunto.

A sintaxe de um loop `for` é semelhante à de um loop `while` em que existe uma instrução `for` e um corpo de loop:

```
friends = [‘Joseph’, ‘Glenn’, ‘Sally’]
for friend in friends:
    print ‘Happy New Year:’, friend
print ‘Done!’
```

Em Python, a variável `friends` é uma lista[1] de três strings e o loop `for` percorre a lista e executa o corpo uma vez para cada uma das três strings da lista resultando na saída:

```
Happy New Year: Joseph
Happy New Year: Glenn
Happy New Year: Sally
Done!
```

Traduzir o loop `for` para o Português, não é tão direto como o `while`, mas se você pensar em `friends` como um **conjunto**, se parece com: “Execute as instruções no corpo do loop `for` uma vez para cada `friend` no conjunto nomeado `friends`.”. Olhando para o loop `for`, **for** e **in** são palavras-chaves reservadas do Python, e `friend` e `friends` são variáveis.

```
for friend in friends:
    print ‘Happy New Year’, friend
```

Em particular, `friend` é a **variável de iteração** do loop, a variável `friend` muda para cada iteração do loop e controla quando o laço `for` termina. A **variável de iteração** passa sucessivamente pelas três strings armazenadas na variável `friends`.

[1] Nós examinaremos listas em mais detalhes no próximo capítulo

5.7 Padrões de Loop

Muitas vezes usamos um loop `for` ou `while` para percorrer uma lista de itens ou o conteúdo de um arquivo e estamos à procura de algo como o maior ou menor valor nos dados que estamos percorrendo.

Estes laços são geralmente construídos por:

- Inicialização de uma ou mais variáveis antes do início do loop;
- Realização de alguma computação dentro do corpo do loop, possivelmente alterando as variáveis no corpo do loop;
- Avaliação das variáveis resultantes quando o loop termina.

Usaremos uma lista de números para demonstrar os conceitos e construção destes padrões de loop.

5.7.1 Loops de contagem e soma

Por exemplo, para contar o número de itens em uma lista, escreveríamos o seguinte loop `for`:

```
count = 0
for intervar in [3, 41, 12, 9, 74, 15]:
    count = count + 1
print ‘Count: ’, count
```

Atribuímos o valor zero à variável `count` antes do início do loop, então nós escrevemos um loop `for` para percorrer a lista de números. Nossa variável de **iteração** é chamada `itervar` e apesar de não usamos `itervar` no loop, ela controla o loop e faz com que o corpo do loop seja executado uma vez para cada um dos valores da lista.

No corpo do loop, adiciona-se 1 ao valor atual de `count` para cada um dos valores na lista. Enquanto o loop está em execução, o valor de `count` é o número de valores que temos visto “até o momento”.

Uma vez que o loop termina, o valor da variável `count` é o número total de itens. Assim, o número total de itens nos é dado ao final do loop. Construímos o loop então temos o que queremos quando o laço termina.

Outro loop semelhante que calcula o total de um conjunto de números é o seguinte:

```
total = 0
for intervar in [3, 41, 12, 9, 74, 15]:
    total = total + intervar
print 'Total: ', total
```

Neste loop utilizamos a **variável de iteração**. Ao invés de simplesmente adicionar 1 à variável `count` como no loop anterior, adicionamos o número atual (3, 41, 12, etc)à variável `total` durante cada iteração do loop. Se você pensar sobre a variável `total`, ela contém a "soma total dos valores até o momento". Então, antes do loop iniciar, `total` é zero, porque ainda não visitamos quaisquer valores. Durante o loop, `total` é o total contínuo, e no final do loop, `total` é a soma total de todos os valores na lista.

À medida que o loop executa, `total` acumula a soma dos elementos; uma variável usada desta forma é, às vezes, chamada de **acumulador**.

Nem o loop de contagem nem o loop somador são particularmente úteis na prática porque há funções embutidas `len()` e `sum()`, que calculam o número de itens em uma lista e a soma de todos os itens na lista, respectivamente.

5.7.2 Loops de máximos e mínimos

Para encontrar o maior valor em uma lista ou sequência, construímos o seguinte loop:

```
largest = None
print 'Before: ', largest
for itervar in [3, 41, 12, 9, 74, 15]:
    if largest is None or itervar > largest:
        largest = itervar
    print 'Loop: ', itervar, largest
print 'Largest: ', largest
```

Quando o programa é executado, a saída é a seguinte:

```
Before : None
Loop: 3 3
Loop: 41 41
Loop: 12 41
Loop: 9 41
Loop: 74 74
Loop: 15 74
Largest: 74
```

A variável `largest` mostra perfeitamente o conceito de "maior valor visto até agora". Antes do loop, atribuímos a `largest` a constante `None`. `None` é um valor especial

constante que podemos armazenar em uma variável para marcar a variável como "vazia".

Antes do laço começar, o maior valor visto até o momento é `None`, uma vez que ainda não foram avaliados quaisquer valores. Enquanto o laço é executado, se `largest` é `None`, então tomamos o primeiro valor como o maior visto até agora. Você pode ver na primeira iteração quando o valor de `itervar` é 3, uma vez que `largest` é `None`, imediatamente definimos `largest` como 3.

Após a primeira iteração, `largest` não tem mais o valor `None`, de modo que a segunda parte da expressão lógica verifica se `itervar > largest` e, caso verdadeiro, troca o valor da variável `largest` para o conteúdo armazenado em `itervar`. Quando vemos um valor "ainda maior", tomamos esse novo valor para `largest`. Você pode ver na saída do programa que `largest` progride de 3 para 41 e depois para 74.

No final do loop, verificamos todos os valores e a variável `largest` agora contém o maior valor da lista.

Para calcular o menor número, o código é muito semelhante, com uma pequena alteração na expressão de comparação.

```
smallest = None
print 'Before: ', smallest
for itervar in [3, 41, 12, 9, 74, 15]:
    if smallest is None or itervar < smallest:
        smallest = itervar
    print 'Loop: ', itervar, smallest
print 'Smallest: ',smallest
```

Novamente, `smallest` é o "menor até agora", antes, durante e após a execução desse loop. Quando o laço termina, `smallest` contém o valor mínimo na lista.

Novamente, como na contagem e soma, as funções embutidas `max()` e `min()` faz esses loops desnecessários.

O que se segue é uma versão simples da função embutida de Python `min()`:

```
def min(values):
    smallest = None
    for value in values:
        if smallest is None or value < smallest:
            smallest = value
    return smallest
```

Na versão da função do código do "menor", removemos todas as sentenças `print`, de modo que é equivalente à função `min` que já está incorporada ao Python.

5.8 Depuração

À medida que você começar a escrever programas maiores, você pode encontrar-se gastando mais tempo com depuração. Mais código significa mais chances de cometer um erro e mais lugares para bugs se esconderem.

Uma maneira de reduzir o tempo de depuração é "depurar por seção". Por exemplo, se houver 100 linhas em seu programa e você deseja checá-los um de cada vez, seriam necessários 100 passos.

Ao invés disso, tente dividir o problema ao meio. Tome a instrução central do programa, ou perto disso. Adicione uma instrução de impressão (ou algo que tem um efeito verificável) e execute o programa.

Se a verificação do ponto médio está incorreta, o problema deve estar na primeira metade do programa. Se ele estiver correto, o problema é na segunda metade.

Toda vez que você executar uma verificação como esta, você reduz pela metade o número de linhas que deve ser verificada. Depois de seis etapas (o que é muito menos do que 100), você teria uma ou duas linhas de código restantes, pelo menos em teoria. Na prática, nem sempre é claro o que é o "meio do programa" e nem é sempre possível verificá-lo. Não faz sentido contar linhas e encontrar o exato ponto médio. Em vez disso, pense sobre pontos do programa onde pode haver erros e lugares onde é fácil inserir uma instrução de impressão.

5.9 Glossário

acumulador: variável usada em um loop para adicionar ou acumular um resultado.

contador: variável usada em um loop para contar o número de vezes que alguma coisa acontece. Nós inicializamos um contador com zero e então incrementamos o contador a cada vez que queremos "contar" alguma coisa.

decremento: atualização que diminui o valor de uma variável.

inicializar: atribuição que dá um valor inicial a uma variável que será atualizada posteriormente.

incremento: atualização que aumenta o valor de uma variável (em geral, em um).

loop infinito: loop em que a condição de término nunca é satisfeita ou para o qual

não há uma condição de término.

iteração: execução repetida de um conjunto de sentenças usando ou uma chamada de função recursiva ou um loop.

5.10 Exercícios

Exercício 5.1 Escreva um programa que leia números repetidamente até que o usuário informe "done". Uma vez que "done" é informado, imprima a soma, quantidade e média dos números. Se o usuário digita alguma coisa diferente de um número, detecte o erro usando `try` e `except` e imprima uma mensagem de erro e pule para o próximo número.

```
Digite um numero: 4
Digite um numero: 5
Digite um numero: bad data
Invalid input
Digite um numero: 7
Digite um numero: done
16 3 5.33333333333
```

Exercício 5.2 Escreva um programa que pede por uma lista de números e imprima o maior e o menor dos números. Sua entrada deve respeitar as condições impostas no Exercício 5.1.

6 Strings

6.1 Uma string é uma sequência

Uma string é uma sequência de caracteres. Você pode acessá-los um de cada vez usando o colchete:

```
> > > fruta = 'banana'
> > > letra = fruta[1]
```

A segunda declaração extrai o caractere na posição de índice 1 da variável `fruta` e o atribui para a variável `letra`.

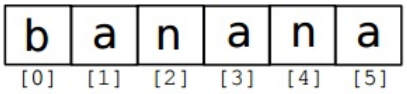
A expressão entre colchetes é chamada de **índice**. O índice indica qual caractere na sequência você deseja. Porém, o resultado pode ser diferente do que você espera:

```
> > > print letra
a
```

Para a maioria das pessoas, a primeira letra de `'banana'` é o `b`, não o `a`. Porém, em Python, o índice é um deslocamento a partir do início da string, e a primeira letra tem o índice zero.

```
> > > letra = fruta[0]
> > > print letra
b
```

Então `b` é a letra de posição zero de `'banana'`, `a` é a letra de posição 1, e `n` é a letra de posição 2.



Você pode usar qualquer expressão, incluindo variáveis e operadores, como um índice. Porém, o valor do índice precisa ser um inteiro. Caso contrário você se deparará com esta situação:

```
> > > letra = fruta[1.5]
TypeError: string indices must be integers
```

6.2 Obtendo o comprimento de uma string usando len

`len` é a uma função pré-definida que retorna o número de caracteres em uma string:

```
> > > fruta = 'banana'
> > > len(fruta)
6
```

Para acessar a última letra de uma string, você deve estar pensando que pode tentar algo como:

```
> > > comprimento = len(fruta)
> > > ultima = fruta[comprimento]
IndexError: string index out of range
```

A razão do erro `IndexError` é porque não existe letra em `'banana'` com o índice 6. Como começamos a contar do zero, as seis letras estão enumeradas de 0 a 5. Para conseguir o último caractere você precisa subtrair 1 de `comprimento`:

```
> > > ultima = fruta[comprimento-1]
> > > print ultima
a
```

Você também pode usar índices negativos, que são contados de trás para frente. A expressão `fruta[-1]` lhe dará a última letra, `fruta[-2]` a penúltima, e assim por diante.

6.3 Percorrendo uma string com um loop

Muitos processos envolvem analisar um caractere da string por vez. Começando de uma posição pré-definida é selecionado um caractere e nele realizado algum ação e esse processo continua até o final definido. Este padrão é chamado de **percurso**. Uma forma de escrever um percurso sobre uma string é usando uma estrutura de repetição `while`:

```
indice = 0
while indice < len(fruta)
    letra = fruta[index]
    print letra
    indice = indice + 1
```

Essa estrutura de repetição percorre a string exibindo uma de suas letras em cada linha. A condição dessa estrutura de repetição é `indice < len(fruta)`, então quando o índice for igual ao comprimento da string, a condição será falsa, e a estrutura de repetição

terá chegado ao seu fim. O último caractere acessado foi o de índice `len(fruta)-1`, que é o último caractere da string.

Exercício 6.1 Escreva uma estrutura de repetição usando o laço `while` que comece no último caractere da string e funcione de trás para frente até o primeiro caractere, exibindo cada letra em uma linha separada.
Outro modo de se escrever um percurso sobre uma string é usando uma estrutura de repetição `for`:

```
for caracter in fruta:
    print caracter
```

Toda vez que o loop é executado, o próximo caractere da string é atribuído à variável `caracter`. A estrutura de repetição continua até não haver mais caracteres em `fruta`.

6.4 Fragmentando Strings

Um segmento de uma string é chamado de **fragmento**. Selecionar um fragmento é como selecionar um caractere:

```
> > > s = 'Monty Python'
> > > print s[0:5]
Monty
> > > print s[6:13]
Python
```

O operador `[n:m]` retorna a parte da string do "n-ésimo" caractere até o "m-ésimo" caractere, incluindo o primeiro mas excluindo o último.

Se o primeiro índice (antes dos dois pontos) for omitido, o fragmento começa no início da string. Se omitir o segundo, o fragmento vai até o final da string:

```
> > > fruta = 'banana'
> > > fruta[:3]
'ban'
> > > fruta[3:]
'ana'
```

Se o primeiro índice é igual ou maior que o segundo, o resultado é uma **string vazia**, representada por duas aspas:

```
> > > fruta = 'banana'
> > > fruta[3:3]
''
```

Uma string vazia não contém caracteres e tem comprimento 0. Excluindo este fator, ela é como qualquer outra string.

Exercício 6.2 Sabendo que `fruta` é uma string, o que `fruta[:]` significa?

6.5 Strings não podem ser mudadas

O uso do operador [ ] do lado esquerdo de uma atribuição, com o intuito de mudar um caracter de uma string não é permitido. Por exemplo:

```
> > > cumprimento = 'Olá, mundo!'
> > > cumprimento[0] = 'J'
TypeError: object does not support item assignment
```

O "object" nesse caso é a string e o "item" é o caractere que você tentou atribuir. Por agora, um **object** é um valor, porém iremos redefiní-lo mais à frente. Um **item** é um dos valores em uma sequência.

A razão desse erro é que as strings são **imutáveis**, isso significa que você não pode realizar mudanças em uma string já existente. O melhor a se fazer é criar uma nova string com as alterações que eram desejadas na original:

```
> > > cumprimento = 'Olá, mundo!'
> > > novo_cumprimento = 'J' + cumprimento[1:]
> > > print novo_cumprimento
Jla, mundo!
```

Esse exemplo concatena uma nova primeira letra em um fragmento de `cumprimento` e atribui a nova string a uma nova variável. A string original não sofre nenhuma alteração.

6.6 Estruturas de repetição e contadores

O programa a seguir conta quantas vezes a letra `a` aparece na string:

```
palavra = 'banana'
contador = 0
for letra in palavra:
    if letra == 'a':
        contador = contador + 1
print contador
```

Esse programa apresenta outro padrão usado na computação chamado de **contador**. A variável `contador` é inicializada em 0, e incrementada toda vez que um `a` é encontrado na string `palavra`. Quando a estrutura de repetição acabar, `contador` irá conter o resultado - o número total de letras `a` existentes na variável `palavra`.

Exercício 6.2 Crie uma função chamada `count`, que recebe uma palavra `p` e um caracter `c` como entrada. Como saída, mostre o número de ocorrências do caracter `c` na palavra `p`.

6.7 O operador in

A palavra `in` é um operador booleano que recebe duas strings e retorna `True` se a primeira string aparace como sub-string da segunda:

```
> > > 'a' in 'banana'
True
> > > 'semente' in 'banana'
False
```

6.8 Comparação entre strings

O operador de comparação visto anteriormente, funciona com strings. Para ver se duas strings são iguais:

```
if palavra == 'banana'
    print 'São bananas.'
```

Outros operadores de comparação são úteis para colocar as palavras em ordem alfabética:

```
if palavra < 'banana':
    print 'Sua palavra,' + palavra + ', vem antes de banana.'
elif palavra > 'banana':
    print 'Sua palavra,' + palavra + ', vem depois de banana.'
else:
    print 'São bananas.'
```

Python não trabalha com letras maiúsculas e minúsculas do mesmo modo que pessoas. Todas as letras maiúsculas vem antes das minúsculas:

```
Sua palavra, Abacaxi, vem antes de banana.
```

Um modo de trabalhar com esse problema é convertendo suas strings para um formato padrão, por exemplo, todas em minúsculas, antes de realizar a comparação.

6.9 Métodos de *strings*

Strings em Python são exemplos de **objetos**. Um objeto contém dados (a string em si) e **métodos**, que são funções embutidas sobre o objeto e que estão disponíveis para o objeto.

Python possui uma função chamada `dir` que lista os métodos disponíveis para um objeto. A função `type` mostra o tipo de objeto.

```
> > > stuff = 'Olá mundo'
> > > type(stuff)
<type 'str'>
> > > dir(stuff)
{'capitalize', 'center', 'count', 'decode', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'index',
'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace',
'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
'partition', 'replace', 'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate',
'upper', 'zfill'}
> > > help(str.capitalize)
Help on method_descriptor:

capitalize(...)
    S.capitalize() -> string

    Return a copy of the string S with only its first character
    capitalized.
> > >
```

A função `dir` lista os métodos, e você pode usar o `help` para conseguir uma explicação superficial sobre algum método. Para conseguir explicações mais detalhadas sobre métodos de string acesse o site `docs.python.org/library/string.html`.

Chamar um **método** é similar a chamar uma função - é preciso enviar um argu-

mento para que ele retorne um valor - mas a sintaxe é diferente. Chama-se um método juntando o nome do método com o nome da variável através de um ponto que separa os dois.

Por exemplo, o método `upper` recebe uma string e retorna uma nova string com todas as suas letras maiúsculas. Porém, não é usada a sintaxe de função `upper(palavra)`, e sim a de método `palavra.upper()`.

```
> > > palavra = 'banana'
> > > nova_palavra = palavra.upper()
> > > print nova_palavra
BANANA
```

Essa forma de notação especifica o nome do método, `upper`, e o nome da string sobre a qual se aplicará o método, `palavra`. Os parênteses em branco indicam que esse método não recebe nenhum argumento.

Quando se realiza uma chamada ao método dizemos que este está sendo `invocado`; nesse caso, estamos invocando `upper` em `palavra`.

Por exemplo, há um método chamado `find` que procura a posição de uma string dentro de outra:

```
> > > palavra = 'banana'
> > > indice = palavra.find('a')
> > > print index
1
```

Nesse exemplo, invocamos `find` em `palavra` e passamos a letra que estamos procurando como parâmetro.

O método `find` pode encontrar uma substring, e não apenas caracteres:

```
> > > palavra.find('na')
2
```

Também pode receber como segundo argumento um índice, que indica de onde a busca deve começar:

```
> > > palavra.find('na',3)
4
```

Uma tarefa comum é a de remover espaços em branco (barras de espaço, tabs, ou novas linhas) do início e do fim de uma string. Para isso, usamos o método `strip`:

```
> > > linha = ‘ Aqui vamos nós ’
> > > linha.strip()
‘Aqui vamos nós’
```

Alguns métodos como **startswith** retornam um valor booleano.

```
> > > linha = ‘Tenha um bom dia’
> > > linha.startswith(‘Tenha’)
True
> > > linha.startswith(‘t’)
False
```

Você irá notar que em certos casos, para usar o método `startswith` do modo que você deseja, terá que usar o `lower` para transformar tudo em minúsculo antes de comparar as strings, como no caso a seguir:

```
> > > linha = ‘Tenha um bom dia’
> > > linha.startswith(‘t’)
False
> > > linha.lower()
‘tenha um bom dia’
> > > linha.lower().startswith(‘t’)
True
```

No último exemplo, nós invocamos o método `lower` e depois usamos `startswith` para checar se a string em minúsculo começa com a letra “p”. Se formos cuidadosos com a ordem, podemos chamar múltiplos métodos em uma única expressão.

Exercício 6.4 Existe um método chamado `count` que é similar à função do último exercício. Leia a explicação desse método em `docs.python.org/library/string.html` e escreva uma invocação que conte quantas vezes a letra `a` aparece em `'banana'`.

6.10 Analisando uma string

Constantemente, queremos encontrar uma sub-string dentro de uma outra string. Por exemplo, se nos for apresentada uma sequência de linhas formatadas desse modo:

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

E quisermos tirar apenas a segunda parte do endereço, (`uct.ac.za`), das linhas. Podemos usar o método `find` e fragmentá-la.

Primeiro, iremos encontrar a posição do caractere ‘@’ na string. Então, encontraremos o primeiro espaço em branco depois dele. Logo em seguida, fragmentaremos a string

para extrair apenas a parte que queremos.

```
> > > data = 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008'
> > > atpos = data.find('@')
> > > print atpos
21
> > > sppos = data.find(' ',atpos)
> > > print sppos
31
> > > host = data[atpos+1:sppos]
> > > print host
uct.ac.za
> > >
```

Usamos uma versão do método `find` de modo que ele apenas começou a procurar o que desejávamos a partir de uma posição específica. Quando o fragmentamos, extraímos os caracteres de "uma posição depois do '@' até o caracter antes do espaço em branco".

A explicação do método `find` encontra-se no link[1].

6.11 Operador de formatação

O **operador de formatação** `%` permite contruir strings, substituindo partes das strings por dados armazenados em variáveis. Quando aplicado a inteiros, `%` é o operador de módulo. Porém, quando o primeiro operando é uma string, `%` é o operador de formatação.

O primeiro operando é o de formato de string, e pode conter uma ou mais sequências de formatação que especificam o formato do segundo operando. O resultado é uma string.

Por exemplo, se o operador de formatação é '`%`d', isso significa que o segundo operando precisa ser formatado como um inteiro (`d` é de "decimal"):

```
> > > camelos = 42
> > > '%d' % camelos
'42'
```

O resultado foi a string '42', que não pode ser confundida com o inteiro de valor 42.

A sequência de formatação pode aparecer em qualquer lugar da string, assim você pode inserir um valor em uma sentença:

[1] docs.python.org/library/string.html

```
> > > camelos = 42
> > > 'Eu tenho %d camelos' % camelos
'Eu tenho 42 camelos.'
```

Se houver mais de uma sequência de formatação em uma string, o segundo argumento precisa ser um registro. Toda sequência de formatação corresponde a um elemento no registro, seguindo uma ordem.

Os exemplos a seguir usam '`%d`' para o formato de inteiros, '`%g`' para o formato de números com ponto flutuante, e '`%s`' para formatar uma string:

```
> > > 'Em %d anos eu terei %g %s.' % (3,1.0,'camelos')
'Em 3 anos eu terei 1.0 camelos.'
```

O número de elementos em um registro precisa corresponder ao número de sequências de formatações em uma string. E também, os tipos de elementos devem corresponder com as sequências de formatações:

```
> > > '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
> > > '%d' % 'dólares'
TypeError: illegal argument type for built-in operation
```

No primeiro exemplo, não existem elementos o suficiente, e no segundo, o tipo está errado.

O operador de formatação é bem poderoso, mas também pode ser difícil de ser usado. Você pode aprender mais sobre eles no link[2].

6.12 Depuração

Uma habilidade que você deve cultivar para melhorar em programação é sempre se perguntar, "O que pode dar errado aqui?", ou talvez, "Que loucura o meu usuário pode fazer para estragar o meu programa?".

Por exemplo, veja o programa em que demonstramos a estrutura de repetição `while` no capítulo anterior:

[2] docs.python.org/lib/typesseq-strings.html

```
while True:
    linha = raw_input('> ')
    if linha[0] == '#' :
        continue
    if linha == 'Fim':
        break
    print linha

print 'Fim!'
```

Veja o que acontece se o usuário enviar uma linha em branco como entrada:

```
> Ola
Ola
> # nao mostre isso
> imprima isso!
imprima isso!
>
Traceback (most recent call last):
    File "copytildone.py", line 3, in <module>
        if line[0] == '#' :
```

Esse código funciona muito bem até receber uma linha em branco como entrada. Assim, não terá nenhum caractere na posição zero para rastrear. Há duas soluções para isso.

Uma possível solução seria simplesmente usar o método `startswith` que retorna `False` se a string é vazia.

```
if line.startswith('#'):
```

Outro modo seria escrever um outro `if` e fazer com que a segunda expressão lógica seja avaliada apenas quando há pelo menos um caractere na string:

```
if len(linha) > 0 and linha[0] == '#':
```

6.13 Glossário

contador: variável usada para contar o número de ocorrências de algum elemento em um conjunto, geralmente inicializada em zero e depois incrementada em um.

string vazia: string que não possui nenhum caractere e tem comprimento zero, representada por duas aspas.

operador de formatação: operador `%`, que recebe uma cadeia de formatação e um registro e então gera uma string que inclui os elementos que o registro está especificando formatados como indicados na cadeia de formatação.

sequência de formatação: sequência de caracteres em uma cadeia de formatação, como `%d`, que especifica o formato em que o valor deve ser apresentado.

cadeia de formatação: string usada com o operador de formatação, que contém sequências de formatação.

flag: variável booleana usada para indicar quando uma condição é verdadeira.

invocação: sentença que chama um método.

imutável: propriedade de uma sequência que impede que caracteres sejam alterados.

índice: valor inteiro usado para selecionar um item em uma sequência, tal como um caractere em uma string.

item: um dos valores em uma sequência.

método: função associada a um objeto, chamada usando o operador ponto(.).

objeto: algo a que uma variável pode se referir. Por enquanto, os termos objeto e valor podem ser utilizados sem distinção.

busca: percurso que termina quando se encontra o que está procurando.

sequência: grupo ordenado, isso é, grupo de valores onde cada um é identificado através de um índice inteiro.

fragmento: parte de uma string especificada através de um intervalo de índices.

percurso: ação de percorrer os itens em uma sequência, realizando uma ação similar com cada um deles.

6.14 Exercícios

Exercicio 6.5 Dado o código a seguir que armazena uma string:

```
str = 'X-DSPAM-Confidence: 0.8475'
```

Use o método `find` e fragmente a string para extrair a porção da string depois dos dois pontos e então use a função `float` para converter a string extraída em um número com ponto flutuante.

Exercicio 6.6 Leia o documento sobre os métodos de string no endereço[3]. Você pode querer experimentar algum deles para ter certeza de que funcionam. `strip` e `replace` são bastante úteis.

A documentação disponível no site usa uma sintaxe que pode confundí-lo. Por exemplo, em `find(sub[,start[, end]])`, os colchetes indicam argumentos opcionais. Então `sub` é requerido, porém `start` é opcional, e se você incluir o `start`, então o `end` é opcional.

[3] docs.python.org/lib/string-methods.html

7 Arquivos

7.1 Persistência

Até agora, nós aprendemos como escrever programas e comunicar nossas intenções à **CPU (Unidade Central de Processamento)** usando condições, funções e iterações. Nós aprendemos como criar e usar estrutura de dados na **memória principal**. A CPU e a memória são onde nosso 'software' trabalha e executa. É onde todo o "pensamento" acontece. Veja o esquema da Figura 9.

Mas se você se lembra das discussões da arquitetura de hardware, uma vez que a energia é desligada, qualquer coisa armazenada tanto na CPU como na memória principal é apagada. Então, até agora, os nossos programas têm sido apenas exercícios divertidos transitórios para aprender Python.

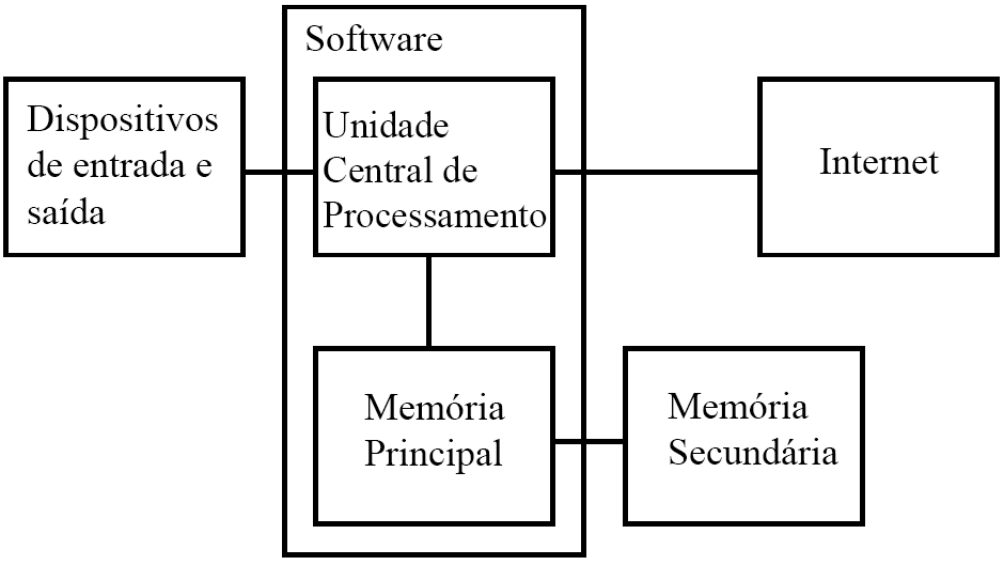


Figura 9: Esquema de um computador genérico.

Nesse capítulo, nós vamos começar a trabalhar com a **Memória Secundária (ou arquivos)**. Memória secundária não é apagada quando a energia é desligada. Ou no caso de um Pen Drive USB, os dados podem ser escritos pelo nossos programas, removido do sistema e transportados para outro sistema.

Iremos primeiramente focar na leitura e escrita de arquivos de texto como os que criamos em um editor de texto. Depois iremos ver como trabalhar com arquivos de banco de dados que são arquivos binários, projetados especificamente para ler e escrever através

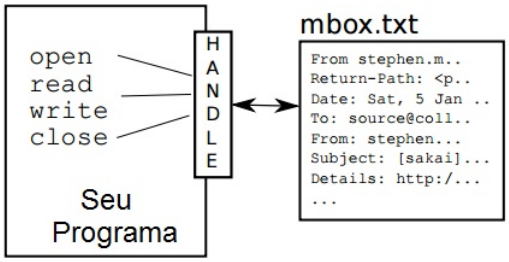
do software do banco de dados.

7.2 Abrindo arquivos

Quando desejamos ler ou escrever um arquivo, primeiro devemos **abrir** o arquivo. Abrindo o arquivo ocorre a comunicação com o seu sistema operacional, pois é ele que sabe onde os dados de cada arquivo são armazenados. Quando se abre um arquivo, pede-se ao sistema operacional para procurar o arquivo pelo nome e ter certeza que o arquivo existe. Nesse exemplo, abrimos o arquivo `mbox.txt` que deveria estar armazenado na mesma pasta em que você está quando inicia o Python. Você pode realizar o download deste arquivo atráves de `www.py4inf.com/code/mbox.txt`

```
> > > fhand = open(‘mbox.txt’)
> > > print fhand
<open file ‘mbox.txt’, mode ‘r’ at 0x1005088b0>
```

Se a função `open` teve sucesso e o arquivo for aberto, o sistema operacional retornará para nós o **file handle**. O `file handle` não são os dados atuais contidos no arquivo, e sim um “handle” que podemos usar para ler os dados. Você está dando um “handle” se o arquivo solicitado existe e se você tem permissões para ler o arquivo.



Se o arquivo não existir, a abertura irá falhar com um “traceback” e você não terá o “handle” para acessar o conteúdo do arquivo:

```
> > > fhand = open(‘stuff.txt’)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: ‘stuff.txt’
```

Depois, iremos usar o `try` e `except` para lidar da melhor forma com a situação em que tentamos abrir um arquivo que não existe.

7.3 Arquivos de texto e linhas

Um arquivo de texto pode ser pensado como uma sequência de linhas, assim como uma string em Python pode ser pensada como uma sequência de caracteres. Por exemplo, este é um simples arquivo de texto que registra a atividade de e-mail de uma equipe de desenvolvimento de um projeto de código aberto:

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
Return-Path: <postmaster@collab.sakaiproject.org>
Date: Sat, 5 Jan 2008 09:12:18 -0500
To: source@collab.sakaiproject.org
From: stephen.marquard@uct.ac.za
Subject: [sakai] svn commit: r39772 - content/branches/
Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev39772
...
```

O arquivo completo de interações de e-mail está disponível através do link[1] e a versão resumida do arquivo está disponível através do link[2]. Esses arquivos estão em formato padrão para um arquivo que contém várias mensagens de e-mail. As linhas que começam com "From" separa as mensagens e as linhas que começam com "From:" são parte das mensagens. Para mais informaçãoes, veja em `en.wikipedia.org/wiki/Mbox`.

Para quebrar o arquivo em linhas, existe um caracter especial que representa o fim da linha, chamado de caracter de nova linha.

Em Python, representamos o caracter de nova linha como \n. Apesar do caracter de nova linha ser representado por dois caracteres, na verdade é apenas um caracter. Quando olhamos para a variável digitando "stuff" no interpretador, isso nos mostra o \n na string, mas quando usamos a função `print` para mostrar a string, vemos a string quebrada em duas linhas pelo caracter de nova linha.

[1] www.py4inf.com/code/mbox.txt
[2] www.py4inf.com/code/mbox-short.txt

```
> > > stuff = 'Hello\nWorld!'
> > > stuff
'Hello\nWorld!'
> > > print stuff
Hello
World!
> > > stuff = 'X\nY'
> > > print stuff
X
Y
> > > len(stuff)
3
```

Você também pode ver que o comprimento da string 'X\nY' são três caracteres porque o caracter de nova linha é um caracter simples.

Assim, quando olhamos para as linhas em um arquivo, precisamos imaginar que lá tem um caracter especial invisível ao fim de cada linha que marca o fim da linha, chamado de **nova linha**.

Portanto, o caracter de nova linha separa os caracteres no arquivo em linhas.

7.4 Lendo arquivos

Já que o **file handle** não contém os dados do arquivo, é fácil construir uma estrutura de repetição 'for' para ler e contar cada uma das linhas em um arquivo:

```
fhand = open('mbox.txt')
count = 0
for line in fhand:
    count = count + 1
print 'Line Count:', count

python open.py
Line Count: 132045
```

Podemos usar o `file handle` como uma sequência em uma estrutura de repetição `for`. O `for` conta o número de linhas no arquivo e imprime o resultado. A tradução aproximada para a estrutura de repetição `for` em Inglês é, "para cada linha no arquivo representado pelo `file handle`, adicionar um a variável `count`".

A razão pela qual a função `open` não lê o arquivo inteiro é que o arquivo pode ser bastante grande, com muitos gigabytes de dados. A sentença `open` gasta a mesma quantidade de tempo independentemente do tamanho do arquivo. A estrutura de repetição

`for` faz realmente com que os dados do arquivo sejam lidos.

Quando o arquivo é lido usando a repetição `for` dessa maneira, Python cuida de dividir os dados do arquivo em linhas separadas utilizando o caractere de nova linha. Python lê cada linha através do \n e inclui a nova linha como caractere na variável `line` para cada iteração do `for`.

Por que o `for` lê os dados uma linha por vez, pode-se eficientemente ler e contar as linhas de arquivos muito grandes sem correr o risco de ficar sem memória principal para armazenar os dados. O programa acima pode contar as linhas de arquivos com qualquer tamanho usando pouca memória, desde que cada linha seja lida, contada, e em seguida descartada.

Se você sabe que o arquivo é relativamente pequeno, comparado com o tamanho da memória principal de sua máquina, você pode ler o arquivo inteiro em uma string usando o método `read` para o `file handle`.

```
> > > fhand = open('mbox-short.txt')
> > > inp = fhand.read()
> > > print len(inp)
94626
> > > print inp[:20]
Frin stephen.marquar
```

Nesse exemplo, todos os 94626 caracteres do arquivo `mbox-short.txt` são lidos diretamente na variável `inp`. Neste exemplo foi usado um "corte" de string para imprimir apenas os vinte primeiros caracteres da string dos dados armazenado em `inp`.

Quando o arquivo é lido dessa maneira, todos os caracteres, incluindo todas as linhas e todos os caracteres de nova linha tornam-se uma grande string armazenada na variável `inp`. Lembre-se que esta forma de utilização da função `open` só deve ser usado se os dados do arquivo couberem confortavelmente na memória principal do computador.

Se o arquivo for muito grande para caber na memória principal, você deve escrever seu programa para ler o arquivo em partes usando a estrutura de repetição `for` ou `while`.

7.5 Pesquisando em um arquivo

Quando voce está procurando por dados em um arquivo, é muito comum ler o arquivo ignorando a maioria das linhas, e processar apenas as linhas que se encaixam em um critério particular. Nós podemos combinar tal comportamento com métodos que atuam sobre string, para construirmos um simples mecanismo de busca.

Por exemplo, se desejamos ler um arquivo e apenas imprimir as linhas que começam

com o prefixo "From:", podemos usar o método `startswith` para selecionar aquelas que se encaixam no critério desejado.

```
fhand = open('mbox-short.txt')
for line in fhand:
    if line.startswith('From:'):
        print line
```

Quando o programa é executado, esta saída é apresentada:

```
From: stephen.marquard@uct.ac.za

From: louis@media.berkeley.edu

From: zqian@umich.edu

From: rjlowe@iupui.edu
...
```

A saída parece ótima, uma vez que as linhas apresentadas começam com "From:", mas porque vemos as linhas em branco? Isso se dá, pelo caracter invisível de nova linha. Cada uma das linhas terminam com tal caractere. Assim, o comando `print` imprime a string armazenada na variável `line`, na qual é incluído o \n, então o comando adiciona outra nova linha, resultando no espaçamento duplo que vemos.

Nós poderiamos usar a quebra de linhas para imprimir toda string exceto o último caracter, mas uma abordagem mais simples é usar o método `rstrip`, que retira o espaço em branco do lado direito de uma string, como se segue:

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if line.startswith('From:') :
        print line
```

Quando este programa é executado, a seguinte saída é apresentada:

```
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: cwen@iupui.edu
```

Conforme o processamento de seu arquivo se torna mais complicado, você pode querer estruturar seus laços de buscas usando `continue`. A ideia básica de uma estrutura de busca é que voce está procurando por linhas "interessantes" e, efetivamente, pulando linhas "desinteresantes". E então, quando encontramos uma linha interessante, fazemos algo com esta linha.

Podemos estrutrar a estrutura de repetição para seguir o padrão de pular linhas desinteressantes como segue:

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    # Skip 'uninteresting lines'
    if not line.startswith('From:') :
        continue
    # Process our 'interesting' line
       print line
```

A saída do programa é a mesma. Em inglês, as linhas desinteressantes são aquelas que não começam com "From:", as quais pulamos(ou descartamos) usando `continue`. Para as linhas interessantes (linhas que começam com "From:") nós as processamos.

Podemos usar o método de string `find` para simular uma busca de um editor de texto, que encontra a string procurada em qualquer lugar de uma linha. Uma vez que `find` procura pela ocorrência de uma string dentro de outra string e retorna a posição da string ou o número -1, representando que a string não foi encontrada, podemos escrever a seguinte estrutura de repetição para mostrar qual linha contém a string "@uct.ac.za".

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if line.find('@uct.ac.za') == -1 :
        continue
    print line
```

O qual produz esta saída:

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
X-Authentication-Warning: set sender to stephen.marquard@uct.ac.za using -f
From: stephen.marquard@uct.ac.za
Author: stephen.marquard@uct.ac.za
From david.horwitz@uct.ac.za Fri Jan 4 07:02:32 2008
X-Authentication-Warning: set sender to david.horwitz@uct.ac.za using -f
From: david.horwitz@uct.ac.za
Author: david.horwitz@uct.ac.za
```

7.6 Permitindo ao usuário escolher o nome do arquivo

Nós realmente não queremos editar nosso código Python toda vez que queremos processar um arquivo diferente. Seria mais viável perguntar ao usuário qual arquivo devemos processar ao começo de cada execução, de forma que os usuários podem usar nossos programas sobre diferentes arquivos sem mudar o código Python. Isto é bem simples utilizando o comando `raw_input`, do seguinte modo:

```
fname = raw_input('Enter the file name: ')
fhand = open(fname)
count = 0
for line in fhand:
    if line.startswith('Subject:') :
        count = count + 1
print 'There were', count, 'subject lines in', fname
```

Lemos o nome do arquivo desejado pelo usuário e guardamos na variável chamada `fname`, então abrimos este arquivo. Agora podemos usar este programa repetidamente em diferentes arquivos.

```
python search6.py
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt

python search6.py
Enter the file name: mbox-short.txt
There were 27 subject lines in mbox-short.txt
```

Antes de espiar a próxima seção, dê uma olhada no programa acima e se pergunte, "O que poderia dar errado?" ou "O que nosso amigável usuário poderia fazer que que acarretaria em uma saída não tão brilhante do nosso pequeno programa, fazendo-nos parecer não tão legal aos olhos do dele?"

7.7 Usando try, except e open

Eu te disse para não espiar. Esta é sua última chance.

E se nosso usuário digitar algo que não é um nome de arquivo?

```
python search6.py
Enter the file name: missing.txt
Traceback (most recent call last):
  File "search6.py", line 2, in <module>
   fhand = open(fname)
IOError: [Errno 2] No such file or directory: 'missing.txt'

python search6.py
Enter the file name: na na boo boo
Traceback (most recent call last):
  File "search6.py", line 2, in <module>
   fhand = open(fname)
IOError: [Errno 2] No such file or directory: 'na na boo boo'
```

Não ria, usuários vão, eventualmente, fazer tudo que estiver ao seu alcance para parar seus programas - seja com má intenção ou não. Na verdade, uma importante parte de qualquer time de desenvolvimento de software é uma pessoa, ou grupo, chamado **Garantia de Qualidade** (GQ), cujo trabalho é fazer as coisas mais loucas possíveis na tentativa de parar o software que foi desenvolvido.

O time de GQ é responsável por encontrar as falhas no programa antes de serem entregues aos usuários, que pagaram pelo software ou nos pagarão para o desenvolvimento de um software. Logo, o time de GQ é o melhor amigo do programador.

Agora que vimos a falha no programa, podemos, elegantemente, consertá-lo usando a estrutura `try/except`. Precisamos assumir que o comando `open` pode falhar e adicionar um código de recuperação quando o comando é falho, do seguinte modo:

```
fname = raw_input('Enter the file name: ')
try:
    fhand = open(fname)
except:
    print 'File cannot be opened:', fname
    exit()

count = 0
for line in fhand:
    if line.startswith('Subject:') :
        count = count + 1
print 'There were', count, 'subject lines in', fname
```

A função `exit` encerra o programa. É uma função que chamamos que nunca retorna valor. Agora quando o usuário (ou o time de QG) digitar nomes errados, nós os "pegamos" e nos recuperamos graciosamente.

```
python search7.py
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt

python search7.py
Enter the file name: na na boo boo
File cannot be opened: na na boo boo
```

Proteger a chamada `open` é um bom exemplo de uso correto de `try` e `except` em um programa Python. Nós usamos o termo "Pythonic" quando estamos fazendo algo no "Jeito Python". Nós podemos dizer que o exemplo acima é um jeito Pythonic de abrir um arquivo.

Quando você se tornar mais hábil em Python, você pode conversar com outro programador Python para decidir qual das soluções equivalentes para um problema é mais Pythonica. O objetivo de ser mais Pythonico captura a noção que programar é parte engenharia e parte arte. Não estamos sempre interessados em apenas fazer um trabalho, também queremos que nossa solução seja elegante e seja apreciada como tal.

7.8 Escrevendo Arquivos

Para escrever um arquivo, você deve abrir com o modo 'w' como um segundo parâmetro:

```
> > > fout = open('output.txt', 'w')
> > > print fout
<open file 'output.txt', mode 'w' at 0xb7eb2410>
```

Se o arquivo já existir, abrí-lo no modo de escrita apaga todos os dados antigos, então tenha cuidado! Se o arquivo não existe, um novo é criado.

O método `write` manipula objetos e os coloca como dados em um arquivo.

```
> > > line1 = 'This here's the wattle,\n'
> > > fout.write(line1)
```

Novamente, o objeto do arquivo (`fout`) nos dá o controle sobre o arquivo. Então, se você chamar a função `write` novamente, são adicionados novos dados ao final do arquivo.

Devemos nos certificar de gerir as extremidades das linhas enquanto escrevemos no arquivo, inserindo explicitamente o caractere de nova linha quando desejamos finalizar uma linha. A sentença `print` acrescenta automaticamente o caractere de nova linha, porém o método `write` não o faz.

```
> > > line2 = 'the emblem of our land.\n'
> > > fout.write(line2)
```

Quando você terminar de escrever, você precisa fechar o arquivo para ter certeza que o último bit do dado foi, fisicamente, escrito em disco, para não perdê-lo se a energia é desligada.

```
> > > fout.close()
```

Poderíamos fechar o arquivo usado para leitura também, mas podemos ser um pouco desleixados se estamos abrindo apenas alguns arquivos, visto que o Python garante que todos os arquivos abertos são fechados ao término do programa. Quando estamos escrevendo arquivos, queremos explicitamente fechar os arquivos, de modo a não deixar nada ao acaso.

7.9 Depuração

Quando você está fazendo a leitura, ou está escrevendo arquivos, você pode se deparar com problemas com espaços em branco. Esses erros podem ser difíceis de depurar, já que espaços, tabulações e caracteres de nova linha são, normalmente, invisíveis:

```
> > > s = '1 2\t 3\n 4'
> > > print s
1 2 3
4
```

A função embutida `repr` pode ajudar. Ela recebe qualquer objeto como argumento e retorna uma string representando-o. Para strings, ele representa espaço em branco como sequências de \:

```
> > > print repr(s)
'1 2\t 3\n 4'
```

Isto pode ser útil para depuração.

Um outro problema que você pode se deparar é a variação de caractere para indicação de fim de linha (nova linha) nos diferentes sistemas. Alguns sistemas usam o

caractere nova linha representado como \n. Outros usam \r. Alguns usam ambos. Se você mover arquivos entre sistemas diferentes essa inconsistência deve gerar erros.

Para a maioria dos sistemas, existem aplicaçoes para converter de um sistema para outro.

Você pode encontrá-los (e ler mais sobre tal problema) no link[3]. Ou, é claro, você pode escrever seu próprio conversor.

7.10 Glossário

catch: previne uma exceção de terminar um programa usando os comandos `try` e `except`.

nova linha: caractere especial usado em arquivos e strings para indicar o final de uma linha.

Pythonic: técnica que trabalha elegantemente em Python. "Usar `try` e `except` é uma maneira *Pythonic* para finalizar um programa quando um arquivo inexistente é informado.

Garantia de Qualidade: pessoa ou grupo focado em assegurar a qualidade global de um produto de software. Está frequentemente envolvido em testar um produto e identificar problemas antes que o produto seja entregue ao usuário.

Arquivo texto: sequência de caracteres armazenada em local permanente como um disco rígido.

7.11 Exercícios

Exercício 7.1 Escreva um programa para ler um arquivo e imprimir o conteúdo do arquivo (linha por linha), com todos os caracteres em maiúsculos. Executando o programa, a saída deve ser como segue:

```
python shout.py
Enter a file name: mbox-short.txt
FROM STEPHEN.MARQUARD@UCT.AC.ZA SAT JAN 5 09:14:16 2008
RETURN-PATH: <POSTMASTER@COLLAB.SAKAIPROJECT.ORG>
RECEIVED: FROM MURDER (MAIL.UMICH.EDU [141.211.14.90])
BY FRANKENSTEIN.MAIL.UMICH.EDU (CYRUS V2.3.8) WITH LMTPA;
SAT, 05 JAN 2008 09:14:16 -0500
```

[3] wikipedia.org/wiki/Newline

Você pode fazer o download do arquivo `mbox-short.txt` no link[4], conforme rodapé.

Exercício 7.2 Escreva um programa para perguntar por um nome de arquivo, e então, ler o arquivo e procurar por linhas da forma:

```
X-DSPAM-Confidence: 0.8475
```

Quando você encontra uma linha que começa com "X-DSPAM-Confidence:" separe a linha para extrair dela o número com ponto flutuante. Conte essas linhas e determine o valor total médio dos valores de confiança de spam a partir destas linhas. Quando você alcançar o fim do arquivo, imprima a confiança média de spam.

```
Enter the file name: mbox.txt
Average spam confidence: 0.894128046745

Enter the file name: mbox-short.txt
Average spam confidence: 0.750718518519
```

Teste seu programa sobre os arquivos `mbox.txt` e `mbox-short.txt`.

Exercício 7.3 Às vezes quando programadores se chateiam ou querem ter um mínimo de diversão, eles adicionam um **Ovo de Páscoa** inofensivo aos seus programas (`en.wikipedia.org/wiki/Easter_egg_(media)`). Modifique o programa que pergunta ao usuário um nome de arquivo e então imprime uma mensagem engraçada quando o usuário digita o nome de arquivo "na na boo boo". O programa deve se comportar normalmente para todos os outros arquivos que existem ou que não existem. Aqui está um exemplo da execução do programa:

```
python egg.py
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt

python egg.py
Enter the file name: missing.tyxt
File cannot be opened: missing.tyxt

python egg.py
Enter the file name: na na boo boo
NA NA BOO BOO TO YOU - You have been punk'd!
```

Nós não estamos encorajando você a colocar Ovos de Páscoa em seus programas - este é apenas um exercício.

[4] www.py4inf.com/code/mbox-short.txt

8 Listas

8.1 Uma lista é uma sequência

Assim como uma string, uma **lista** é uma sequência de valores. Em uma string, os valores são caracteres; em uma lista, eles podem ser de qualquer tipo. Os valores em uma lista são chamados de **elementos** ou **itens**.

Há muitas maneiras de se criar uma lista; a maneira mais simples é colocar os elementos entre colchetes.

```
[10, 20, 30, 40]
[‘sapo crocante’, ‘bexiga de carneiro’, ‘vômito de cotovia’]
```

O primeiro exemplo é uma lista de quatro inteiros. O segundo é uma lista de três strings. Os elementos de uma lista não precisam ser do mesmo tipo. A lista a seguir contém elementos dos do tipo string, float e inteiro e (oh!) outra lista:

```
[‘spam’, 2.0, 5, [10,20]]
```

Uma lista anexada a outra lista é uma lista **aninhada**.

Uma lista que não contém elementos é chamada de lista vazia; você pode criar uma lista vazia colocando colchetes sem quaisquer elementos, `[]`.

Como é de se esperar, você pode atribuir os valores de uma lista a variáveis:

```
> > > queijos = [‘Cheddar’, ‘Edam’, ‘Gouda’]
> > > numeros = [17, 123]
> > > vazio = []
> > > print queijos, numeros, vazio
[‘Cheddar’, ‘Edam’, ‘Gouda’] [17, 123] []
```

8.2 Listas são mutáveis

A sintaxe para acessar os elementos de uma lista é a mesma para acessar os caracteres de uma string - com colchetes. O valor dentro dos colchetes especifica o índice.

Lembrando que os índices iniciam a partir do 0:

```
> > > print queijos[0]
Cheddar
```

Diferente de strings, listas são mutáveis porque você consegue mudar a ordem dos itens em uma lista ou alterar um item. Quando os colchetes aparecem do lado esquerdo de uma atribuição, ele identifica o elemento da lista que será trocado.

```
> > > numeros = [17, 123]
> > > numeros[1] = 5
> > > print numeros
[17, 5]
```

O elemento do índice 1 da lista `numeros`, que anteriormente era 123, passa agora a ser 5.

Você pode imaginar a lista como uma relação entre os índices e os elementos. Essa relação é chamada de **mapping**; cada índice diz respeito a uma orientação para um dos elementos da lista.

Os índices de uma lista trabalham da mesma maneira que os índices de uma string:

- Qualquer inteiro pode ser usado como índice.
- Se você tenta ler ou escrever um elemento que não existe, você obtém um erro do tipo `IndexError`.
- Se um índice possui um valor negativo, o acesso ao elemento da lista é feito a partir do final da lista.

Pode-se usar o operador `in` em listas também.

```
> > > queijos = ['Cheddar', 'Edam', 'Gouda']
> > > 'Edam' in queijos
True
> > > 'Brie' in queijos
False
```

8.3 Leitura de uma lista

A maneira mais comum de ler os elementos de uma lista é com uma estrutura de repetição `for`. A sintaxe é a mesma da usada em strings:

```
for queijo in queijos:
    print queijo
```

Isso funciona bem se você apenas precisa ler os elementos da lista. Mas se você quer escrever ou atualizar os elementos, você precisará dos índices. O jeito mais comum para fazer isso é usando as funções `range` e `len`:

```
for i in range(len(numeros)):
    numeros[i] = numeros[i] * 2
```

Essa estrutura de repetição lê a lista e atualiza cada elemento. `len` retorna o número de elementos na lista. `range` retorna uma lista de índices de 0 à n-1, onde n é o valor retornado por `len`. Durante a estrutura `for`, `i` possui o valor do índice do próximo elemento. O comando de atribuição usa o `i` para ler o valor antigo do elemento e atribuir um novo valor.

A estrutura `for` de uma lista vazia nunca é executada:

```
for x in vazio:
    print 'Isso nunca acontecerá.'
```

No entanto, uma lista pode conter outra lista, a lista aninhada é contada como um único elemento. O tamanho dessa lista é quatro:

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

8.4 Operações em listas

O operador + concatena listas:

```
> > > a = [1, 2, 3]
> > > b = [4, 5, 6]
> > > c = a + b
> > > print c
[1, 2, 3, 4, 5, 6].
```

Similarmente, o operador * repete uma lista um dado número de vezes:

```
> > > [0] * 4
[0, 0, 0, 0]
> > > [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

O primeiro exemplo repete a lista `[0]` quatro vezes. O segundo exemplo repete a lista `[1, 2, 3]` três vezes.

8.5 Fragmentando listas

O operador slice também funciona em listas:

```
> > > t = [‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’]
> > > t[1:3]
[‘b’, ‘c’]
> > > t[:4]
[‘a’, ‘b’, ‘c’, ‘d’]
> > > t[3:]
[‘d’, ‘e’, ‘f’]
```

Se o primeiro índice for omitido, a lista é fatiada a partir do começo. Se você omitir o segundo índice, a lista é fatiada até o final. Assim, se ambos os índices forem omitidos a fatia cortada é uma cópia de toda a lista.

```
> > > t[:]
[‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’]
```

Como as listas são mutáveis, por vezes é útil fazer uma cópia antes de fazer operações que dobram e fragmentam listas. O operador slice do lado esquerdo de uma atribuição pode atualizar múltiplos elementos.

O operador de fragmentação, quando aparece do lado esquerdo de uma operação de atribuição, é capaz de atualizar vários elementos da lista de uma só vez.

```
> > > t = [‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’]
> > > t[1:3] = [‘x’, ‘y’]
> > > print t
[‘a’, ‘x’, ‘y’, ‘d’, ‘e’, ‘f’]
```

8.6 Métodos em listas

Python fornece métodos para operar em listas. Por exemplo, `append` adiciona um novo elemento ao final de uma lista:

```
> > > t = [‘a’, ‘b’, ‘c’]
> > > t.append(‘d’)
> > > print t
[‘a’, ‘b’, ‘c’, ‘d’]
```

`extend` recebe uma lista como um argumento e adiciona todos os elementos desta lista ao final da outra lista:

```
> > > t1 = [‘a’, ‘b’, ‘c’]
> > > t2 = [‘d’, ‘e’]
> > > t1.extend(t2)
> > > print t1
[‘a’, ‘b’, ‘c’, ‘d’, ‘e’]
```

Neste caso, `t2` não é alterado.

`sort` ordena os elementos da lista do menor para o maior:

```
> > > t = [‘d’, ‘c’, ‘e’, ‘b’, ‘a’]
> > > t.sort()
> > > print t
[‘a’, ‘b’, ‘c’, ‘d’, ‘e’]
```

A maioria dos métodos em listas são do tipo `void`; eles modificam a lista e não retornam nenhum valor (`None`). Se você acidentalmente digitar `t = t.sort()`, você ficará desapontado com o resultado.

8.7 Deletando elementos

Há muitas maneiras para apagar elementos de uma lista. Se você sabe o índice do elemento que você quer eliminar, você pode usar o método `pop`:

```
> > > t = [‘a’, ‘b’, ‘c’]
> > > x = t.pop(1)
> > > print t
[‘a’, ‘c’]
> > > print x
b
```

`pop` modifica a lista e retorna o elemento que foi removido. Se você não informar o índice, ele apaga e retorna o último elemento.

Se você não precisar do valor a ser removido, você pode usar o operador `del`:

```
> > > t = [‘a’, ‘b’, ‘c’]
> > > del t[1]
> > > print t
[‘a’, ‘c’]
```

Se você sabe o elemento que quer remover (mas não sabe o índice), você pode usar o `remove`:

```
> > > t = [‘a’, ‘b’, ‘c’]
> > > t.remove(‘b’)
> > > print t
[‘a’, ‘c’]
```

O valor retornado por `remove` é `None`.

Para remover mais de um elemento, você pode usar `del` justamente com o operador slice:

```
> > > t = [‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’]
> > > del t[1:5]
> > > print t
[‘a’, ‘f’]
```

Como sempre, a fatia seleciona todos os elementos até o elemento anterior ao segundo índice.

8.8 Listas e funções

Existem algumas funções internas que podem ser usadas em listas. Tais funções evitam a utilização de estruturas de repetição:

```
> > > nums = [3, 41, 12, 9, 74, 15]
> > > print len(nums)
6
> > > print max(nums)
74
> > > print min(nums)
3
> > > print sum(nums)
154
> > > print sum(nums)/len(nums)
25
```

A função `sum()` apenas funciona quando os elementos da lista são números. As outras funções (`max()`, `len()`, etc.) funcionam com listas de strings e outros tipos que podem ser comparados.

É possível reescrever o programa anterior que calcula a média de uma lista de números digitados pelo usuário.

Primeiro, o programa para calcular uma média sem uma lista:

```
total = 0
cont = 0
while ( True ) :
     inp = raw_input('Digite um número: ')
     if inp == 'pronto' : break
     valor = float(inp)
     total = total + valor
     cont = cont + 1

media = total / cont
print 'Média:', media
```

Neste programa, nós temos a variável `cont` para contar a quantidade de números que o usuário digita e a variável `total` que soma os números digitados.

Nós podemos simplesmente armazenar os números que o usuário digitou e utilizar funções internas para calcular o `total` e `cont` no final.

```
numvet = list()
while ( True ) :
     inp = raw_input('Digite um número: ')
     if inp == 'pronto' : break
     valor = float(inp)
     numvet.append(valor)

media = sum(numvet) / len(numvet)
print 'Média:', media
```

Nós criamos uma lista vazia antes do `while` começar, e enquanto o usuário digita um número, adiciona-se o número na lista. No final do programa, nós apenas calculamos a soma dos números da lista e dividimos pela quantidade de números na lista para imprimir a média. Fazemos isso utilizando as funções `sum` e `len`.

8.9 Listas e Strings

Uma string é uma sequência de caracteres e uma lista é uma sequência de valores, mas uma lista de caracteres não é o mesmo que uma string. Para converter uma string para uma lista de caracteres, você pode usar `list`:

```
> > > s = 'spam'
> > > t = list(s)
> > > print t
['s', 'p', 'a', 'm']
```

Como `list` é o nome da função interna, você deve evitar usá-la como variável. Evite usar o caractere `l` porque se assemelha muito com o dígito `1`. Utilize, neste caso, o caractere `t`.

A função `list` quebra uma string em letras individuais. Se você quer quebrar uma string em palavras, você pode utilizar o método `split`:

```
> > > s = 'ansiado para os fiordes'
> > > t = s.split()
> > > print t
['ansiado', 'para', 'os', 'fiordes']
> > > print t[2]
os
```

Após usar o `split` para quebrar uma string em uma lista de palavras, você pode usar o operador de índice (os colchetes) para trabalhar com uma palavra específica.

Você pode usar o `split` com um argumento opcional chamado de **delimitador**, que corresponde a um caractere a ser utilizado para para dividir as palavras. O exemplo a seguir usa o hífen como delimitador:

```
> > > s = 'spam-spam-spam'
> > > delimitador = '-'
> > > s.split(delimitador)
['spam', 'spam', 'spam']
```

`join` é o inverso de `split`. Concatena os elementos de uma lista de strings. Como `join` é um método de string, então você deve chamá-lo para o delimitador e passar a lista como parâmetro:

```
> > > t = ['ansioso', 'para', 'a', 'viagem']
> > > delimitador = ' '
> > > delimitador.join(t)
'ansioso para a viagem'
```

Nesse caso, o delimitador é um espaço então `join` coloca um espaço entre as palavras. Para concatenar strings sem espaços, você pode usar uma string vazia, `' '`, como delimitador.

8.10 Analisando linhas

Usualmente, quando estamos lendo um arquivo, queremos fazer algo a mais com as linhas que apenas imprimir a linha inteira. Frequentemente, queremos achar as "li-

nhas interessantes" e analisar a linha para encontrar a parte interessante da linha. E se quiséssemos imprimir o dia da semana das linhas que começam com a palavra "From"?

```
From stephen.marquart@uct.ac.za Sab Jan 5 09:14:16 2008
```

O método `split` é muito eficaz quando se trata desse tipo de problema. Podemos escrever um pequeno programa que avalia as linhas que começam com a palavra "From" e, então, inicia um novo `split` nessas linhas e imprime a terceira palavra da linha:

```
fhand = open('mbox-short.txt')
for linha in fhand:
     linha = linha.rstrip()
     if not linha.startswith('From ') : continue
     palavras = linha.split()
     print palavras[2]
```

Note que a forma contraída da sentença `if` foi utilizada. Assim, colocamos o `continue` na mesma linha que o `if`. Essa forma contraída da função `if` funciona da mesma maneira como se o `continue` estivesse na linha seguinte e identado.

O programa produz as seguintes saídas:

```
Sab
Sex
Sex
Sex
    ...
```

Mais adiante, aprenderemos técnicas cada vez mais sofisticadas para escolher as linhas para trabalhar e como extrair destas linhas a parte exata da informação que estamos procurando.

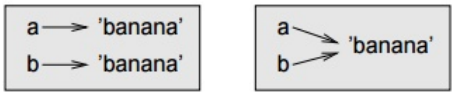
8.11 Objetos e Valores

Se executarmos as instruções de atribuição:

```
> > > a = 'banana'
> > > b = 'banana'
```

Sabemos que ambos, `a` e `b`, referem-se a uma string, mas não sabemos se referem-se à mesma string. Há duas possibilidades:

No primeiro caso, `a` e `b` referem-se a dois objetos diferentes que possuem o mesmo valor. No segundo caso, eles se referem ao mesmo objeto.



Para verificar se duas variáveis referem-se ao mesmo objeto, pode-se usar o operador `is`.

```
> > > a = 'banana'
> > > b = 'banana'
> > > a is b
True
```

Nesse exemplo, Python criou apenas um objeto string e ambos, `a` e `b`, se referem a ele.

Mas quando se cria duas listas, há dois objetos:

```
> > > a = [1, 2, 3]
> > > b = [1, 2, 3]
> > > a is b
False
```

Nesse caso, diríamos que as duas listas são **equivalentes**, pois elas possuem os mesmos elementos, mas não **idênticas**, pois não são o mesmo objeto. Se dois objetos são idênticos, eles também são equivalentes, mas se são equivalentes, não são necessariamente idênticos.

Até agora, usamos "objeto" e "valor" de forma permutável, mas é mais preciso dizer que um objeto possui um valor. Se executarmos `a = [1,2,3]`, `a` se refere a um objeto lista cujo valor é uma sequência particular de elementos. Se outra lista tem os mesmos elementos, ela tem o mesmo valor.

8.12 Pseudônimo

Se `a` se refere a um objeto e atribui-se `b = a`, então ambas as variáveis se referem ao mesmo objeto:

```
> > > a = [1, 2, 3]
> > > b = a
> > > b is a
True
```

A associação de uma variável com um objeto se denomina uma **referência**. Nesse exemplo, há duas referências ao mesmo objeto.

Um objeto com mais de uma referência tem mais de um nome, então dizemos que o objeto tem um **pseudônimo**.

Se o objeto que possui o pseudônimo é mutável, mudanças em um pseudônimo afetam o outro:

```
> > > b[0] = 17
> > > print a
[17, 2, 3]
```

Apesar desse comportamento ser útil, ele é propenso a erros e, por isso, é mais seguro evitá-lo quando se trabalha com objetos mutáveis.

Para objetos imutáveis, como strings, pseudônimos não são um grande problema. Nesse exemplo:

```
a = 'banana'
b = 'banana'
```

Quase nunca faz diferença se `a` e `b` se referem a mesma string ou não.

8.13 Listas como Argumentos

Quando se passa uma lista para uma função, a função recebe uma referência à lista. Se a função modifica os parâmetros de uma lista, quem a chama percebe a mudança. Por exemplo, `delete_head` remove o primeiro elemento de uma lista:

```
def delete_head(t):
   del t[0]
```

Aqui está uma chamada à função `delete_head`:

```
> > > letras = ['a', 'b', 'c']
> > > delete_head(letras)
> > > print letras
['b', 'c']
```

O parâmetro `t` e a variável `letras` são pseudônimos para o mesmo objeto.

É importante distinguir operações que modificam listas e operações que criam novas listas. Por exemplo, o método `append` modifica uma lista, mas o operador `+` cria uma nova lista:

```
> > > t1 = [1, 2]
> > > t2 = t1.append(3)
> > > print t1
[1, 2, 3]
> > > print t2
None
```

```
> > > t3 = t1 + [3]
> > > print t3
[1, 2, 3]
> > > t2 is t3
False
```

Esta diferença é importante quando se escreve funções que devem modificar listas. Por exemplo, essa função `não` deleta o começo de uma lista:

```
def bad_delete_head(t):
   t = t[1:] # !WRONG!
```

O operador slice cria uma nova lista e a atribuição faz o `t` se referir a ela, mas nada disso tem efeito na lista que foi passada como um argumento.

Uma alternativa é escrever uma função que cria e retorna uma nova lista. Por exemplo, `tail` retorna todos os elementos da lista, exceto o primeiro:

```
def tail(t):
   return t[1:]
```

Essa função deixa a lista original sem modificações.

```
> > > letras = ['a', 'b', 'c']
> > > resto = tail(letras)
> > > print resto
['b', 'c']
```

Exercício 8.1 Escreva uma função chamada `chop` que receba uma lista e a modifique, removendo o primeiro e o último elemento, e retorne `None`. Em seguida, escreva uma função chamada `middle` que receba uma lista e retorne uma nova lista que contem todos exceto o primeiro e o último elemento.

8.14 Depuração

O uso negligente ou descuidado de listas (e outros objetos mutáveis) pode levar a longas horas em busca de um erro. Aqui estão algumas armadilhas e formas de evitá-las:

1. Não esqueça de que a maioria dos métodos de listas modifica o elemento e retorna `None`. Isso é o oposto do método de string, que retorna uma nova string e não altera a original.

   Se você está acostumado a escrever códigos de string assim:

   ```
   palavra = palavra.strip()
   ```

   É tentador escrever código de lista assim:

   ```
   t = t.sort() # !WRONG!
   ```

   Como `sort` retorna `None`, a próxima operação com `t` provavelmente fracassará.

   Antes de usar métodos de listas e operadores, você deve ler a documentação atentamente e testá-los no modo iterativo. Os métodos e operadores que as listas compartilham com outras sequências (como strings) são documentados em ([1]). Os métodos e operadores que se aplicam apenas à sequências mutáveis estão documentados em `docs.python.org/lib/typesseq-mutable.html`.

2. Escolha um idioma e se atenha a ele.

   Parte dos problemas com lista é que há muitas formas de se fazer as coisas. Por exemplo, para remover um elemento de uma lista, pode-se usar `pop, remove, del`, ou até uma atribuição slice.

   Para adicionar um elemento, pode-se usar o método `append` ou o operador `+`. Mas não se esqueça de que esses estão certos:

   ```
   t.append(x)
   t = t + [x]
   ```

   E esses estão errados:

   ```
   t.append([x]) - !WRONG!
   t = t.append(x) - !WRONG!
   t + [x] - !WRONG!
   t = t + x - !WRONG!
   ```

[1] docs.python.org/lib/typesseq.html

Experimente cada um desses exemplos no modo iterativo para ter certeza de que entende o que eles fazem. Note que apenas o último causa um `runtime error`; os outros três são legítimos (não causam erros de compilação), mas errados.

3. Faça cópias para evitar pseudônimos.

   Se você quer usar um método como o `sort`, que modifica o argumento, mas precisa manter a lista original, pode fazer uma cópia.

   ```
   orig = t[:]
   t.sort()
   ```

   Nesse exemplo, poderia usar também a função embutida `sorted`, que retorna uma lista nova e organizada e não mexe na original. Mas nesse caso, evite usar `sorted` como um nome de variável!

4. Listas, `split` e arquivos

   Quando lemos e analisamos arquivos, há tantas oportunidades para encontrarmos entradas que possam arruinar nosso programa que é uma boa ideia revisitar o padrão **guardião** quando se trata de escrever programas que leem um arquivo e procuram por uma "agulha no palheiro".

   Vamos revisitar o programa que busca o dia da semana nas linhas `From` do nosso arquivo:

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

   Já que estamos separando essa linha em palavras, podemos dispensar o uso de `startswith` e apenas olhar a primeira palavra de cada linha para determinar se estamos interessados na linha. Podemos usar `continue` para pular linhas que não têm "From" como a primeira palavra:

   ```
   fhand = open('mbox-short.txt')
   for line in fhand:
      palavras = line.split()
      if palavras[0] != 'From' : continue
      print palavras[2]
   ```

   Isso parece bem mais simples e nem precisamos de `rstrip` para remover o final do arquivo. Mas é melhor?

```
python search8.py
Sab
Traceback (most recent call last):
  File "search8.py", line 5, in <module>
   if palavras[0] != 'From' : continue
IndexError: list index out of range
```

Funciona mais ou menos e podemos ver o dia da primeira linha (Sab) mas então o programa falha com um `traceback error`. O que ocorreu de errado? Que dado estragado fez com que o nosso elegante, inteligente e muito Pythonico programa falhasse?

Você poderia ficar olhando para o código por um longo tempo e se perplexar ou pedir ajuda a alguém, mas a abordagem mais rápida e inteligente é adicionar um `print` para te responder o porquê da falha. O melhor lugar para colocá-lo é logo antes da linha onde o programa falhou e imprimir o dado que parece estar causando a falha.

Essa abordagem pode gerar muitas linhas de saída mas você terá imediatamente alguma pista de qual é o problema. Então adicionamos um `print` da variável `palavras` logo antes da linha cinco. Até adicionamos um prefixo "Debug:" para que possamos separar nossa saída normal da nossa saída de erros.

```
for line in fhand:
   palavras = line.split()
   print 'Debug:', palavras
   if palavras[0] != 'From' : continue
   print palavras[2]
```

Quando se executa o programa, as várias linhas impressas como saída fazem com que a tela role, mas no final, vemos nossa impressão "Debug" e o traceback e, então sabemos o que aconteceu logo antes do erro.

```
Debug: ['X-DSPAM-Confidence:', '0.8475']
Debug: ['X-DSPAM-Probability:', '0.0000']
Debug: []
Traceback (most recent call last):
  File "search9.py", line 6, in <module>
   if palavras[0] != 'From' : continue
IndexError: list index out of range
```

Cada linha de impressão do comando `print 'Debug:', palavras`, está imprimindo a lista de palavras que recebemos quando dividimos (`split`) a linha em palavras. Quando o programa falha, a lista de palavras está vazia (`[]`). Se abrirmos o arquivo em um editor de texto e olharmos, nesse ponto ele é assim:

```
X-DSPAM-Result: Innocent
X-DSPAM-Processed: Sab Jan 5 09:14:16 2008
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000

Detalhes: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
```

Os erros ocorrem quando o programa encontra uma linha em branco! Claro, há "zero palavras" em uma linha em branco. Por que não pensamos nisso enquanto estávamos escrevendo o código?! Quando o código procura pela primeira palavra (palavra[0]) para verificar se identifica-se com "From", recebemos um erro `‘‘index out of range’’`.

É claro que esse é o lugar perfeito para se adicionar o código **guardião** para evitar verificar a primeira palavra se a primeira palavra não está ali. Há muitos jeitos de proteger esse código, nós escolheremos checar o número de palavras antes de verificar a primeira palavra:

```
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
   palavras = line.split()
   # print 'Debug:', palavras
   if len(palavras) == 0 : continue
   if palavras[0] != 'From' : continue
   print palavras[2]
```

Primeiro deixamos a instrução `print 'Debug:', palavras` comentada ao invés de removê-la, caso a modificação falhe e ela seja necessária novamente. Então, adicionamos um guardião que checa se há zero palavras e, se esse é o caso, usa `continue` para pular para a próxima linha do arquivo.

Podemos pensar nos dois `continue` como nos ajudando a refinar os conjuntos de linhas que são "interessantes" para nós e quais queremos processar um pouco mais. Uma linha sem palavras é "desinteressante", então pulamos para a próxima linha. O mesmo acontece com uma linha que não começa com "From".

O programa modificado é executado com sucesso, então talvez esteja correto. Nosso guardião certifica-se de que palavras[0] nunca falhe, mas talvez não seja o suficiente. Quando estamos programando, devemos sempre nos perguntar, "O que pode dar errado?".

Exercício 8.2 Descubra qual linha do programa anterior ainda não está guardada propriamente. Tente construir um caso de teste que faça o programa falhar e o modifique para que a linha esteja guardada apropriadamente e teste para ter certeza de que ele seja capaz de avaliar seu novo caso de teste.

Exercício 8.3 Reescreva o código guardião no exemplo acima sem dois `if`. No lugar, use uma expressão lógica composta usando o operador lógico `and` com apenas um `if`.

8.15 Glossário

delimitador: um caractere ou string usada para indicar onde uma string deve ser dividida.

elemento: um dos valores em uma lista (ou outra sequência), também chamado de item.

equivalente: duas variáveis que armazenam o mesmo valor.

idêntico: ser o mesmo objeto (implica equivalência).

índice: valor inteiro que indica um elemento em uma lista.

lista: sequência de valores.

lista aninhada: lista que contém como elemento outra lista.

objeto: elemento ao qual uma variável pode se referir. Possui um tipo e um valor.

percurso em lista: acesso sequencial aos elementos de uma lista.

pseudônimo: circunstância na qual duas ou mais variáveis se referem ao mesmo objeto.

referência: associação entre uma variável e seu valor.

8.16 Exercícios

Exercício 8.4 Faça o download do arquivo de `www.py4inf.com/code/romeo.txt`

Escreva um programa que abra o arquivo `romeo.txt` e leia-o linha por linha. Para cada linha, separe-a em uma lista de palavras usando a função `split`.

Para cada palavra, cheque para ver se a palavra já está em uma lista. Se não estiver na lista, adicione-a.

Quando o programa estiver completo, organize e imprima as seguintes palavras em ordem alfabética.

```
Informe o arquivo: romeo.txt
{'Levantar', 'Mas', 'Ele', 'Julieta', 'Quem', 'já', 'e', 'quebra', 'leste',
'invejoso', 'justo', 'dor', 'ser', 'matar', 'luz', 'Lua', 'pálida', 'doente',
'macio', 'sol', 'o', 'através', 'que', 'janela', 'com', 'lá'}
```

Exercício 8.5 Escreva um programa que leia os dados de uma caixa de e-mail e quando encontrar a linha que começa com "From", divida a linha em palavras usando a função `split`. Estamos interessados em quem mandou a mensagem, que é a segunda palavra na linha From.

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

Você deve analisar a linha `From` e imprimir a segunda palavra de cada linha `From` e então contar o número de linhas que começam com `From` e imprimir a contagem no final.

Esse é um exemplo de saída com algumas linhas removidas:

```
Insira o nome de um arquivo: mbox-short.txt
stephen.marquard@uct.ac.za
louis@media.berkeley.edu
zqian@umich.edu

...algumas linhas removidas...

ray@media.berkeley.edu
cwen@iupui.edu
cwen@iupui.edu
cwen@iupui.edu
27 linhas no arquivo possuem From como a primeira palavra
```

Exercício 8.6 Reescreva o programa que pede ao usuário uma lista de números e imprime o maior e o menor dos números lidos. A leitura deve terminar quando o usuário digitar "pronto". Escreva o programa para armazenar os números que o usuário digita em uma lista e use as funções `max()` e `min()` para calcular o maior e o menor número quando a estrutura de repetição da leitura terminar.

```
Insira um número: 6
Insira um número: 2
Insira um número: 9
Insira um número: 3
Insira um número: 5
Insira um número: pronto
Maior: 9.0
Menor: 2.0
```

9 Dicionários

Um **dicionário** é como uma lista, porém mais generalizado. Em uma lista, as posições (índices) têm que ser números inteiros; em um dicionário os índices podem ser (quase) de qualquer tipo.

Você pode pensar em um dicionário como um mapeamento entre um conjunto de índices (que são chamados de **chaves**) e um conjunto de valores. Cada chave é mapeada para um valor. A associação de uma chave e um valor é chamado de **par chave-valor** ou às vezes um **item**.

Como um exemplo, vamos implementar um dicionário que mapeia palavras do Inglês para o Espanhol, assim as chaves e os valores são todos do tipo string.

A função `dict` cria um novo dicionário sem itens, ou seja, cria um dicionário vazio. Como `dict` é o nome de uma função da linguagem, você deve evitar usá-la como nome de variável.

```
> > > eng2sp = dict()
> > > print eng2sp
{}
```

As chaves, `{}`, representam um dicionário vazio. Para adicionar itens a ele, você pode usar colchetes:

```
> > > eng2sp[‘one’] = ‘uno’
```

Essa linha cria um item que mapeia da chave `‘one’` para o valor `‘uno’`. Se imprimirmos o dicionário novamente, vemos um par chave-valor com dois pontos entre a chave e o valor:

```
> > > print eng2sp
{‘one’: ‘uno’}
```

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

```
> > > eng2sp =  ‘one’: ‘uno’ , ‘two’: ‘dos’, ‘three’: ‘tres’
```

Agora, se você imprimir `eng2sp`, você pode se surpreender:

```
> > > print eng2sp
{ 'one': 'uno', 'three': 'tres', 'two': 'dos' }
```

A ordem dos pares chave-valor não é a mesma dada no momento da criação do dicionário. Na verdade, se você digitar o mesmo exemplo no seu computador, você terá um resultado diferente. Em geral, a ordem dos itens num dicionário é imprevisível.

Mas isso não é um problema porque os elementos de um dicionário nunca são indexados com índices inteiros. Ao invés disso, você usa as chaves (`keys`) para ver os valores correspondentes.

```
> > > print eng2sp['two']
'dos'
```

A chave 'two' sempre mapeia para o valor 'dos'. Assim, a ordem dos itens não importa.

Se a chave não está no dicionário, você tem uma exceção:

```
> > > print eng2sp['four']
KeyError: 'four'
```

A função `len` também opera em dicionários, retornando o número de pares chave-valor:

```
> > > len(eng2sp)
3
```

O operador `in` para dicionários indica se um determinado valor aparece como uma chave no dicionário e não como um valor.

```
> > > 'one' in eng2sp
True
> > > 'uno' in eng2sp
False
```

Para ver se algo aparece como um valor num dicionário, deve-se usar o método `values`, que retorna os valores de um dicionário armazenados como uma lista e, em seguida, usar o operador `in`:

```
> > > vals = eng2sp.values()
> > > 'uno' in vals
True
```

O operador `in` implementa diferentes algoritmos para listas e dicionários. Para listas, é utilizado um algoritmo de busca linear. Quando a lista cresce, o tempo de busca cresce proporcional ao tamanho da lista. Para dicionários, Pyhton usa um algoritmo chamado **hash table** que tem uma notável propriedade; o operador `in` demora a mesma quantidade de tempo independente do número de itens existentes no dicionário. Não explicarei por que funções hash são tão mágicas, mas você pode ler mais sobre isso em `wikipedia.org/wiki/Hash_table`.

Exercício 9.1 Escreva um programa que leia palavras em `words.txt` e as armazene como chaves em um dicionário. Não importa o que os valores são. Então você pode usar o operador `in` como um meio rápido de verificar se uma string está no dicionário.

9.1 Dicionário como um conjunto de contadores

Suponha que você tenha recebido uma string e quer contar quantas vezes cada letra aparece nesta string. Há várias maneiras de se fazer isso:

1. Você poderia criar 26 variáveis, uma para cada letra do alfabeto. Então, poderia examinar a string e, para cada caractere, incrementar o contador correspondente, provavelmente usando uma condicional em cadeia.

2. Você poderia criar uma lista com 26 elementos. Então, você poderia converter cada caractere para um número (usando a função já implementada `ord`), usar o número como um índice dentro da lista, e incrementar o contador apropriado.

3. Você poderia criar um dicionário com caracteres como chaves e contadores como valores correspondentes. A primeira vez que você visse um caractere, você adicionaria um item para o dicionário. Depois disso, você incrementaria o valor de um item existente.

Cada uma dessas opções executa a mesma quantidade de computação, mas cada uma delas implementa a computação de um modo difente.

Uma **implementação** é uma maneira de realizar um cálculo; algumas implementações são melhores que as outras. Por exemplo, uma vantagem da implementação do dicionário é que não precisamos saber antes do tempo quais letras aparecem na string e temos apenas que deixar espaço para as letras que aparecem.

A seguir temos como o código que implementa a opção 3 se parece:

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

Nós estamos efetivamente calculando um **histograma**, que é um termo estatístico para um conjunto de contadores (ou frequências).

A estrutura `for` examina a string. A cada execução do `for`, se o caractere `c` não está no dicionário, criamos um novo item com chave `c` e um valor inicial `1` (desde que vimos essa letra uma vez). Se `c` já está no dicionário nós incrementamos `d[c]`.

A seguir 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, e assim por diante.

Dicionários têm um método chamado `get` que recebe uma chave e um valor padrão. Se a chave aparece no dicionário, `get` retorna o valor correspondente; caso contrário ela retorna o valor padrão. Por exemplo:

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

Podemos usar `get` para escrever nossa estrutura de histograma mais concisa. Porque o método `get` automaticamente cuida do caso em que a chave não está no dicionário, podemos reduzir quatro linhas de código para uma única linha e eliminar a declaração `if`.

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

O uso do método `get` para simplificar essa estrutura de repetição de contagem acaba sendo uma prática muito comum em Python e, por isso, vamos usá-lo muitas vezes

até o final do livro. Então, você deve gastar um tempo para comparar a estrutura de repetição usando a declaração `if` e o operador `in` com a estrutura de repetição usando o método `get`. Eles fazem exatamente a mesma coisa, mas um é mais resumido.

9.2 Dicionários e arquivos

Um dos usos comuns de um dicionário é para contar a ocorrência das palavras em um arquivo com algo escrito. Vamos começar com um arquivo muito simples de palavras obtidas de um texto de Romeu e Julieta[1].

Para o primeiro conjunto de exemplos, usaremos uma versão encurtada e simplificada do texto sem pontuação. Depois vamos trabalhar com o texto da cena com pontuação incluída.

```
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 para ler as linhas do arquivo, quebrar cada linha em uma lista de palavras e, em seguida, percorrer cada uma das palavras na linha, e contar cada palavra usando um dicionário.

Você verá que temos duas estruturas `for`. A estrutura de repetição externa está lendo as linhas do arquivo e a interna está repetindo, em cada uma das palavras, de uma linha em particular. Esse é um exemplo de um padrão chamado de **estrutura de repetição aninhada** porque uma das estruturas é a estrutura de *fora* e a outra é a estrutura de *dentro*.

Devido à estrutura de repetição interna executar todas as iterações cada vez que a estrutura de repetição externa faz uma única iteração, pensamos que a estrutura de repetição interna itera "mais rapidamente" e a externa mais devagar.

A combinação das duas estruturas aninhadas garante que vamos contar todas as palavras em todas as linhas do arquivo de entrada.

[1] Agradecimentos a http://shakespeare.mit.edu/Tragedy/romeoandjuliet/romeo_juliet.2.2.html

```
fname = raw_input ( 'Digite o nome do arquivo: ' )
try:
    fhand = open(fname)
except:
    print 'Arquivo não 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 executamos o programa, vemos ao final, a impressão de todos os contadores sem ordenação (o arquivo `romeo.txt` pode ser encontrado no link[2]).

```
python count1.py
Entre com 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 pouco inconveniente buscar no dicionário a palavra mais comum e o seu contador, então precisamos adicionar um pouco mais ao código Python para que tenhamos uma saída mais útil.

9.3 Estruturas de repetição e dicionários

Se você usa um dicionário como a sequência em uma estrutura de repetição `for`, ele examina as chaves do dicionário. A estrutura de repetição a seguir imprime cada chave e seu valor correspondente.

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

[2] www.py4inf.com/code/romeo.txt

A saída é mostrada a seguir:

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

Novamente, as chaves não estão em uma ordem pré-definida.

Podemos usar este padrão para implementar estruturas de repetição sobre dicionários que atuam como contadores. Por exemplo, se você quer encontrar todas as entradas em um dicionário com um valor acima de 10, podíamos escrever o seguinte código:

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

A estrutura `for` itera através das *chaves* do dicionário, então devemos usar o operador indíce para recuperar o *valor* correspondente para cada chave. A saída é mostrada a seguir:

```
jan 100
annie 42
```

Apenas as entradas com um valor acima de 10 são mostradas.

Se você quer imprimir as chaves em ordem alfabética, primeiro você cria uma lista de chaves em um dicionário usando o método `keys` disponível em objetos do tipo dicionário, e, em seguida, ordena essa lista e percorra a lista ordenada, visualizando a impressão dos pares chave/valor na ordem de classificação, como segue:

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

A saída ficaria como:

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

Primeiro você vê a lista das chaves sem ordenação que obtivemos pelo método `keys`. Em seguida vemos os pares chave/valor em ordem impressos dentro da estrutura `for`.

9.4 Análise avançada de texto

No exemplo anterior, removemos os sinais de pontuação do arquivo `romeo.txt`. O texto original tem várias 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,
```

Como a função `split` do Python procura por espaços e trata palavras como símbolos separados por espaços, trataríamos as palavras "soft!" e "soft" como palavras diferentes e criaríamos uma entrada de dicionário separada para cada palavra.

Considerando que o arquivo tem letras maiúsculas, trataríamos "who" e "Who" como palavras diferentes com contadores diferentes.

Podemos acabar com esses problemas usando os métodos sobre string `lower`, `punctuation`, e `translate`. O método `translate` é o mais sutil dos métodos. Aqui está a documentação para `translate`:

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

Remova todos os caracteres de s que estão em `deletechars` *(caso existam), e então traduza os caracteres usando* `table`*, que deve ser uma string de 256 caracteres retornando a tradução para cada caractere válido, indexado pelo original. Se* `table` *for* `None`*, então somente o passo de remoção do caractere é executado.*

Não vamos especificar `table`, mas vamos usar o parâmetro `deletechars` para remover todas as pontuações. Segue o comando que faz com que o Python nos mostre a lista de caracteres que são considerados como caracteres de pontuação:

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

Segue o novo código para realizar a contagem de palavras do texto `romeo.txt`, agora pontuado.

```
import string                    # Novo código

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

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

print counts
```

Usamos `translate` para remover todas as pontuações e `lower` para transformar todos os caracteres da linha em minúsculos. De resto, o código não foi alterado. Note que para Python 2.5 e anteriores, `translate` não aceita `None` como o primeiro parâmetro então use esse código para a chamada do `translate`:

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

Parte do aprendizado do "Art of Python" ou "Thinking Pythonically" está concretizando que Python muitas vezes tem capacidades internas para muitos problemas de análise de dados comum. Com o tempo, você terá visto exemplos de códigos o suficiente e terá lido a documentação o bastante para saber onde procurar se algúem já escreveu algo que torna 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, ...}
```

Olhando para esta saída, ela ainda está desajeitada. Podemos usar Python para nos dar exatamente o que estamos procurando, mas para fazer isso, precisamos aprender sobre **tuplas** em Python. Vamos voltar a este exemplo uma vez que aprendermos sobre tuplas.

9.5 Depuração

Quando você trabalha com grandes conjuntos de dados pode se tornar complicado realizar a depuração por meio de impressões e busca visual. Aqui estão algumas sugestões para depurar grandes conjuntos de dados:

Reduza a entrada: Se possível, reduza o tamanho do conjunto de dados. Por exemplo, se o programa lê um arquivo de texto, comece apenas com as 10 primeiras linhas, ou com o menor exemplo que você pode achar. Você pode ou editar os arquivos, ou (melhor) modificar o programa para ler apenas as primeiras `n` linhas.

Se houver um erro, você pode reduzir `n` para o menor valor em que o erro acontece e, então, incrementar gradualmente o valor de `n`, até que você encontre e corrija os erros.

Confira os resumos e os tipos: Ao invés de imprimir e conferir o conjunto de dados inteiro, considere resumos de impressão dos dados: por exemplo, o número de itens em um dicionário ou o total de uma lista de números.

Um caso comum de erro em tempo de execução é um valor que não é do tipo certo. Para depurar esse tipo de erro, é suficiente só imprimir o tipo de um valor.

Escreva auto-verificações: Às vezes você pode escrever o código para verificar os erros automaticamente. Por exemplo, se você está calculando a média de uma lista de números, você pode conferir se o resultado não é menor que o menor elemento nem maior que o maior elemento. Isso é chamado de uma "verificação de sanidade" porque detecta os resultados que são "completamente não lógicos".

Outro tipo de verificação compara os resultados de dois cálculos diferentes para ver se eles são consistentes. Isso é chamado de "verificação de consistência".

Imprima adequadamente a saída: Formatar bem a saída da depuração pode tornar mais fácil a tarefa de encontrar o erro.

De novo, o tempo que você gasta construindo andaimes pode reduzir o tempo que você demora para depurar.

9.6 Glossário

dicionário: estrutura de mapeamento de um conjunto de chaves para seus valores correspondentes.

hash table: o algoritmo usado para implementar dicionários em Python.

funções hash: função usada por uma hash table para computar a localização de uma chave.

histograma: conjunto de contadores.

implementação: modo de realizar um cálculo.

item: outro nome para um par chave-valor.

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

par chave-valor: representação do mapeamento de uma chave para um valor.

estruturas de repetição aninhadas: uma ou mais estruturas de repetição "dentro" de outra estrutura de repetição. A estrutura de repetição interna é executada até a conclusão cada vez que a estrutura de repetição externa é executada.

valor: objeto que aparece em um dicionário como a segunda parte de um par chave-valor. Isso é mais específico que nosso uso anterior da palavra "valor".

9.7 Exercícios

Exercício 9.2 Escreva um programa que categorize cada mensagem de e-mail pelo dia da semana que a submissão foi feita. Para fazer isso, procure por linhas que começam com "From", e então procure pela terceira palavra e assim continue executando

o contador para cada um dos dias da semana. No fim do programa imprima o conteúdo de seu dicionário (a ordem não importa).

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

Execução do exemplo:
python dow.py
Digite o nome do arquivo: mbox-short.txt
{ 'Fri' : 20, 'Thu' : 6, 'Sat' : 1 }
```

Exercício 9.3 Escreva um programa que leia um log de e-mails e construa um histograma usando um dicionário para contar quantas mensagens vem de cada endereço de e-mail. Imprima o dicionário.

```
Digite o nome do arquivo: 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 9.4 Adicione linhas ao código anterior a fim de descobrir quem tem mais mensagens no arquivo.

Após todos os dados serem lidos e o dicionário ser criado, olhe no dicionário usando uma estrutura de repetição máxima (olhe a Seção 5.7.2), para descobrir quem tem mais mensagens e imprima quantas mensagens a pessoa tem.

```
Digite o nome do arquivo: mbox-short.txt
cwen@iupui.edu 5

Digite o nome do arquivo: mbox.txt
zqian@umich.edu 195
```

Exercício 9.5 Escreva um programa que registre o nome de domínio (ao invés do endereço) de onde a mensagem foi enviada ao invés de onde o e-mail veio (ou seja, o endereço de e-mail inteiro). No final do programa imprima os conteúdos do dicionário.

```
python schoolcount.py
Digite o nome do arquivo: 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}
```

10 Tuplas

10.1 Tuplas são imutáveis

Uma tupla[1] é uma sequência de valores muito parecida com uma lista. Os valores armazenados em uma tupla podem ser de qualquer tipo, e são indexados por números inteiros. A diferença importante é que tuplas são imutáveis. Tuplas também são **comparáveis** e podem ser buscadas por **hash** de forma que podemos classificar listas de tuplas e usarmos as tuplas como chaves em dicionários Python.

Sintaticamente, uma tupla é uma lista de valores separados por vírgula:

```
> > > t = ‘a’, ‘b’, ‘c’, ‘d’, ‘e’
```

Entretanto, isso não é necessário, é comum valores de tuplas entre parênteses para nos ajudar a identificá-las rapidamente dentro do código Python.

```
> > > t = (‘a’, ‘b’, ‘c’, ‘d’, ‘e’)
```

Para criar uma tupla de um único elemento, deve-se acrescentar uma vírgula no final:

```
> > > t1 = (‘a’,)
> > > type(t1)
<type ‘tuple’>
```

Sem a vírgula, o Python interpreta o (`‘a’`) como string

```
> > > t2 = (‘a’)
> > > type(t2)
<type ‘str’>
```

Outro jeito de criar uma tupla é usando a função `tuple`. Sem nenhum argumento, ele cria uma tupla vazia:

```
> > > t = tuple()
> > > print t
()
```

[1] Fato engraçado: A palavra “tupla” vem dos nomes dados a sequência de números de tamanhos variados: única, dupla, tripla, quádrupla, quíntupla, sêxtupla, setupla, etc.

Se o argumento for uma sequência (string, lista ou tupla), o resultado da chamada para `tuple` é uma tupla com os elementos da sequência:

```
> > > t = tuple('margarida')
> > > print t
('m', 'a', 'r', 'g', 'a', 'r', 'i', 'd', 'a',)
```

Como `tuple` é o nome de um construtor, você deve evitar usá-lo como nome de variável.

A maioria dos operadores de listas também funciona em tuplas. O operador colchetes indexa um elemento:

```
> > > t = ('a', 'b', 'c', 'd', 'e')
> > > print t[0]
'a'
```

E o operador de slice(:) seleciona um intervalo de elementos:

```
> > > print t[1:3]
('b','c')
```

Mas se você tentar modificar um dos elementos da tupla, você terá um erro:

```
> > > t[0] = 'A'
TypeError: object doesn't support item assignment
```

Você não pode modificar os elementos de uma tupla, mas você pode substituir uma tupla por outra:

```
> > > t = ('A',) + t[1:]
> > > print t
('A','b','c','d','e')
```

10.2 Comparando tuplas

O operador de comparação trabalha com tuplas e outras sequências; Python começa pela comparação do primeiro elemento de cada sequência. Se eles forem iguais, a compara cão segue para o próximo elemento, e assim por diante, até que encontre um primeiro elemento que difere. Elementos de subsequência não são considerados (nem mesmo se forem muito grandes).

```
> > > (0, 1, 2) < (0, 3, 4)
True
> > > (0, 1, 2000000) < (0, 3, 4)
True
```

A função `sort` funciona do mesmo jeito. Ordena primeiramente pelo primeiro elemento, mas em caso de empate, ela ordena pelo segundo elemento, e assim por diante.

Essa característica leva a um padrão chamado de **DSU**

Decorate: adiciona uma ou mais chaves de ordenação em posições anteriores aos elementos da lista.

Sort: ordene a lista de tuplas usando a função `sort` do Python, e

Undecorate: extraia os elementos ordenados da sequência.

Por exemplo, suponha que há uma lista de palavras e você quer ordená-las por tamanho, da maior para a menor:

```
txt = 'but soft what light in yonder window breaks'
words = txt.split()
t = list()
for word in words:
    t.append((len(word), word))

t.sort(reverse=True)

res = list()
for length, word in t:
    res.append(word)

print res
```

A primeira estrutura de repetição constrói uma lista de tuplas, onde cada tupla é uma palavra precedida pelo seu tamanho. `sort` compara o primeiro elemento, tamanho, primeiro, e só considera o segundo elemento (a própria palavra) em caso de empate. A palavra chave `reverse=True` diz à função `sort` para executar em ordem decrescente.

A segunda estrutura de repetição percorre a lista de tuplas e constrói uma lista de palavras em ordem decrescente de tamanho. Entre as cinco palavras, elas são ordenadas em uma ordem alfabética *reversa*. Então "what" aparece antes de "soft" na lista. A saída do programa é a seguinte:

```
['yonder', 'window', 'breaks', 'light', 'what', 'soft', 'but', 'in']
```

É claro que as linhas perdem um pouco do seu impacto poético quando transformados em uma lista Python em ordem decrescente de tamanho.

10.3 Atribuição de tuplas

Uma das características sintáticas da linguagem Python é a possibilidade de ter uma tupla no lado esquerdo de um comando de atribuição. Isso permiti atribuir mais de uma variável por vez quando o lado esquerdo é uma sequência.

Nesse exemplo temos dois elementos lista (que são sequências), isso permite atribuir o primeiro e o segundo elementos da sequência para as variáveis `x` e `y` em uma única sentença.

```
> > > m = ['have', 'fun']
> > > x, y = m
> > > x
'have'
> > > y
'fun'
> > >
```

Isso não é mágica. Python traduz grosseiramente para a seguinte sintaxe:[2]

```
> > > m = ['have', 'fun']
> > > x = m[0]
> > > y = m[1]
> > > x
'have'
> > > y
'fun'
> > >
```

Quando usamos uma tupla no lado esquerdo da sentença de atribuição, omitimos os parênteses, mas a seguinte é um sintaxe igualmente válida:

[2] Python não traduz a sintaxe literalmente. Por exemplo, se você tentar isto com um dicionário, você terá um resultado inesperado.

```
> > > m = [‘have’,’fun’]
> > > (x, y) = m
> > > x
‘have’
> > > y
‘fun’
> > >
```

Uma aplicação particularmente inteligente de atribuição de tuplas é permitir que troquemos o valor de duas variáveis em uma única sentença:

```
> > > a, b = b, a
```

Ambos os lados desta sentença são tuplas, mas o lado esquerdo é uma tupla de variáveis; o lado direito é uma tupla de expressões. Cada valor no lado direito é atribuído a sua respectiva variável no lado esquerdo. Todas as expressões no lado direito são avaliadas antes que as atribuições sejam realizadas.

O número de variáveis no lado esquerdo e direito tem de ser o mesmo:

```
> > > a, b = 1, 2, 3
ValueError: too many values to unpack
```

De modo geral, o lado direito pode ser de qualquer tipo de seqüência (string, lista ou tupla). Por exemplo, para dividir um endereço de e-mail em um nome de usuário e um domínio, você escreveria:

```
> > > addr = ‘monty@python.org’
> > > uname, domain = addr.split(‘@’)
```

O valor de retorno do `split` é uma lista com dois elementos; o primeiro elemento é atribuido para `uname`, o segundo para `domain`:

```
> > > print uname
monty
> > > print domain
python.org
```

10.4 Dicionários e tuplas

Dicionários possui um método chamado `items` que retorna uma lista de tuplas, onde cada tupla é um par chave-valor[3].

[3] Este comportamento é levemente diferente em Python 3.0

```
> > > d = {‘a’:10,‘b’:1,‘c’:22}
> > > t = d.items()
> > > print t

[(‘a’, 10), (‘c’, 22), (‘b’, 1)]
```

Como você já deve esperar de um dicionário, os itens não estão em uma ordem particular.

No entanto, uma vez que a lista de tuplas é uma lista, e tuplas são comparáveis, podemos agora ordenar a lista de tuplas. Converter um dicionário para uma lista de tuplas é uma forma de apresentar o conteúdo de um dicionário ordenado pela chave.

```
> > > d = {‘a’:10, ‘b’:1, ‘c’:22}
> > > t = d.items()
> > > t
[(‘a’, 10), (‘c’, 22), (‘b’, 1)]
> > > t.sort()
> > > t
[(‘a’, 10), (‘b’, 1), (‘c’, 22)]
```

A nova lista é ordenada em ordem alfabética crescente pelo valor da chave.

10.5 Atribuição múltipla com dicionários

Combinando `items`, atribuição de tuplas e `for`, é possível construir um padrão de código bom para percorrer as chaves e valores de um dicionário em uma única estrutura de repetição:

```
for key, val in d.items():
   print val, key
```

Esta estrutura de repetição tem duas variáveis de iteração pois `items` retorna uma lista de tuplas e `key`, `val` é uma atribuição de tupla que repete sucessivamente por cada um dos pares de chave/valor no dicionário.

Para cada iteração através da estrutura de repetição, ambos, `key` e `val`, são avançados para o próximo par chave/valor no dicionário (ainda na ordem hash).

A saída do loop é:

```
10 a
22 c
1 b
```

Mais uma vez, a ordem apresentada é a ordem dada pala chave de hash (isto é, nenhuma ordem particular).

Se combinarmos essas duas técnicas, podemos imprimir o conteúdo de um dicionário ordenado pelo valor armazenado em cada par chave/valor.

Para fazer isso, primeiro crie uma lista de tuplas onde cada tupla corresponde a um par (valor, chave). O método `items` nos dá uma lista de tuplas (chave, valor), mas desta vez queremos classificar por valor, e não por chave. Uma vez que construímos a lista com as tuplas valor/chave, é uma simples questão de classificar a lista e imprimir a nova, lista ordenada.

```
> > > d = {'a':10, 'b':1, 'c':22}
> > > l = list()
> > > for key, val in d.items() :
...     l.append( (val, key) )
...
> > > l
[(10, 'a'), (22, 'c'), (1, 'b')]
> > > l.sort(reverse=True)
> > > l
[(22, 'c'), (10, 'a'), (1, 'b')]
> > >
```

Ao construir cuidadosamente a lista de tuplas para ter o valor como o primeiro elemento de cada tupla, podemos classificar a lista de tuplas e obter conteúdos do dicionário ordenados por valor.

10.6 As palavras mais comuns

Voltando ao nosso exemplo de execução do texto de *Romeu e Julieta* Act 2, Cena 2, podemos aumentar o nosso programa para usar esta técnica para imprimir as dez palavras mais comuns no texto da seguinte forma:

```
import string
fhand = open('romeo-full.txt')
counts = dict()
for line in fhand:
    line = line.translate(None, string.punctuation)
    line = line.lower()
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1

# Sort the dictionary by value
lst = list()
for key, val in counts.items():
    lst.append( (val, key) )

lst.sort(reverse=True)

for key, val in lst[:10] :
    print key, val
```

A primeira parte do programa que lê o arquivo e constroi o dicionário que mapeia cada palavra para a contagem de palavras no documento não é alterado. Mas, em vez de simplesmente imprimir `counts` e terminar o programa, construímos uma lista de tuplas `(val, key)` e, em seguida, ordenamos a lista em ordem decrescente.

Uma vez que o valor vem em primeiro lugar na lista, ele vai ser usado para as comparações e, se houver mais do que uma tupla com o mesmo valor, o segundo elemento (a chave) será avaliado, assim nas tuplas de mesmo valor, estas serão ordenadas pela chave.

No final, escrevemos uma estrutura de repetição `for` que faz uma iteração de atribuição múltipla e imprime as dez palavras mais comuns por iterar através de uma fatia da lista (`lst[: 10]`).

Então, agora a saída finalmente se parece com o que queremos para a nossa análise de freqüência de palavras.

```
61 i
42 and
40 romeo
34 to
34 the
32 thou
32 juliet
30 that
29 my
24 thee
```

O fato de que esta análise e separação de dados complexos pode ser feita com 19 linhas fáceis de entender, é uma das razões do porque o Python é uma boa escolha como uma linguagem para explorar informações.

10.7 Usando tuplas como chaves em dicionários

Como tuplas são armazenadas em forme de hash e listas não, se quisermos criar uma chave **composta** para usar em um dicionário, devemos usar uma tupla como a chave.

Uma chave composta seria apropriada, se quiséssemos criar uma lista telefônica que mapeia, a partir do último e do primeiro nome, para números de telefone. Assumindo que definimos as variáveis `last`, `first` e `number`, poderíamos escrever uma declaração de atribuição de dicionário da seguinte forma:

```
directory[last,first] = number
```

A expressão entre colchetes é uma tupla. Nós poderíamos usar atribuição de tupla em uma estrutura de repetição `for` para percorrer este dicionário.

```
for last, first in directory:
   print first, last, directory[last,first]
```

Esta estrutura de repetição percorre as chaves em `directory`, que são tuplas. Ele atribui os elementos de cada tupla para `last` e `first`, e imprime o nome correspondente do número de telefone.

10.8 Sequências: strings, listas and tuplas

Tenho focado em listas de tuplas, mas quase todos os exemplos neste capítulo também trabalham com listas de listas, tuplas de tuplas, e tuplas de listas. Para evitar enumerar as combinações possíveis, às vezes é mais fácil falar sobre sequências de

sequências.

Em muitos contextos, os diferentes tipos de sequências (strings, listas e tuplas) podem ser usados como sinônimos. Então, como e por que você escolhe um e não outro?

Começando com o óbvio, as strings são mais limitadas do que outras sequências, porque os elementos têm de ser caracteres. Elas também são imutáveis. Se você precisa da capacidade de alterar os caracteres em uma string (ao invés de criar uma nova string), você pode querer usar uma lista de caracteres, e não uma string.

As listas são mais comuns do que as tuplas, principalmente porque eles são mutáveis. Mas há alguns casos em que você pode preferir tuplas:

1. Em alguns contextos, como em uma instrução `return`, é sintaticamente mais simples criar uma tupla do que uma lista. Em outros contextos, você pode preferir uma lista.

2. Se você quer usar uma sequência como uma chave de dicionário, você tem que usar um tipo imutável como uma tupla ou string.

3. Se você está passando uma sequência como um argumento para uma função, a utilização de tuplas reduz o potencial de um comportamento inesperado devido a *aliasing* (apelido para variáveis).

Como as tuplas são imutáveis, elas não fornecem métodos como `sort` e `reverse`, que modificam listas existentes. No entanto, Python fornece as funções embutidas `sorted` e `reversed`, que recebe qualquer sequência como parâmetro e retornam uma nova lista com os mesmos elementos em uma ordem diferente.

10.9 Depuração

Listas, dicionários e tuplas são conhecidas genericamente como **estruturas de dados**; neste capítulo começamos a ver estruturas de dados compostas, como listas de tuplas, e dicionários que contém tuplas como chaves e listas como valores. Estruturas de dados compostas são úteis, mas são propensas ao que eu chamo de **erros de forma**, isto é, erros causados quando a estrutura de dados tem um tipo, tamanho ou composição errada ou talvez quando você escreve algum código e esquece a forma dos seus dados.

Por exemplo, se você está esperando uma lista com um inteiro e lhe é dado apenas um inteiro (e não em uma lista), seu código não irá funcionar.

Quando você estiver verificando erros no programa, e especialmente se você está trabalhando em um erro grave, há quatro coisas para se tentar:

reading: Examine seu código, leia-o para si mesmo e verifique se ele diz o que você

realmente quer dizer.

running: Experimente fazer mudanças e executar diferentes versões. Frequentemente se você mostra a coisa certa no lugar certo no programa, o problema torna-se óbvio, mas às vezes você tem de gastar mais tempo para construir casos de validação.

ruminating: Gaste um tempo pensando! Que tipo de erro é: sintaxe, em tempo de execução, semântica? Que informação você obtém das mensagens de erro, ou da saída do programa? Que tipo de erro podia causar o problema que você está vendo? O que você mudou por último, antes do problema aparecer?

retreating: Em algum ponto, a melhor coisa a fazer é voltar, desfazer mudanças recentes, até você voltar em um programa que funciona e que você compreenda. Então você pode começar a reconstruir.

Programadores iniciantes, às vezes, ficam presos a uma dessas atividades e se esquecem das outras. Cada atividade vem com meu próprio modo de falha.

Por exemplo, ler o seu código deve ajudar se o problema é um erro de digitação, mas não ajuda se o problema é um erro conceitual. Se você não entende o que seu programa faz, você pode lê-lo 100 vezes e nunca enxergar o erro, porque o erro está em sua cabeça.

Executar experimentos pode ajudar, especialmente se você executar testes pequenos e simples. Mas se você executar experimentos sem pensar ou ler seu código, você pode falhar em um padrão que eu chamo de "programação do passeio aleatório", que é o processo de fazer mudanças aleatórias até que o programa faça a coisa certa. Desnecessário dizer que programação do passeio aleatório pode tomar um longo tempo.

Você precisa tomar um tempo para pensar. Depuração é como uma ciência experimental. Você deveria ter pelo menos uma hipótese sobre qual é o problema. Se há duas ou mais possibilidades, tente pensar em um teste que eliminaria um deles.

Fazer uma pausa ajuda com o pensamento. O mesmo acontece com a fala. Se você explicar o problema para outra pessoa (ou a si mesmo), às vezes você encontrará a resposta antes de terminar de fazer a pergunta.

Mas, mesmo assim, as melhores técnicas de depuração falharão se houverem muitos erros, ou se o código que você está tentando corrigir é muito grande e complicado. Às vezes, a melhor opção é recuar, simplificar o programa até que você consiga algo que funcione e que você entenda.

Programadores iniciantes são muitas vezes relutantes a recuar porque eles não podem aceitar excluir uma linha de código (mesmo que seja errado). Se isso te faz sentir melhor, copie seu programa em outro arquivo antes de começar a descartá-lo para baixo.

Em seguida, você pode colar as peças de volta, um pouco de cada vez.

Encontrar um erro grave requer leitura, execução, reflexão, e às vezes recuar. Se você ficar preso a uma dessas atividades, tente as outras.

10.10 Glossário

comparável: tipo cujo valor pode ser checado para ver se é maior, menor ou igual a outro valor do mesmo tipo. Tipos que são comparáveis podem ser colocados em uma lista e ordenados.

estrutura de dados: coleção de valores relacionados, frequentemente organizados em listas, dicionários, tuplas, etc.

DSU: abreviação de "decorate-sort-undecorate". Um padrão que envolve construir uma lista de tuplas, ordená-las e extrair parte do resultado.

hashable: tipo que possui uma função *hash* associada. Tipos imutáveis como inteiros, floats e strings são *hashtable*; tipos mutáveis como listas e dicionários não são.

scatter: operação de tratar uma sequência como uma lista de argumentos.

forma (de uma estrutura de dados): resumo do tipo, tamanho e composição de uma estrutura de dados.

singleton: lista (ou outra sequência) com um único elemento.

tupla: sequência imutável de elementos.

atribuição de tupla: atribuição com uma sequência do lado direito e uma tupla de variáveis no lado esquerdo. O lado direito é avaliado e então seus elementos são atribuídos às variáveis no lado esquerdo.

10.11 Exercícios

Exercício 10.1 Revise o programa anterior como segue: Leia e analise as linhas "From" e extraia os endereços de cada linha. Conte o número de mensagens de cada pessoa usando um dicionário.

Após todos os dados terem sido lidos, imprima a pessoa com mais submissões

(commits) através da criação de uma lista de tuplas (contador, e-mail) do dicionário e então ordena a lista em ordem decrescente e imprima a pessoa que tem mais submissões.

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

Digite o nome do arquivo: mbox-short.txt
cwen@iupui.edu 5

Digite o nome do arquivo: mbox.txt
zqian@umich.edu 195
```

Exercício 10.2 Este programa deve contar a distribuição das horas por dia para cada uma das mensagens. Você pode extrair a hora da linha "From" por encontrar a string de tempo e então dividí-la em partes usando o caracter dois pontos (:). Uma vez que você acumulou o contador para cada hora, imprima o contador, um por linha, ordenado pelas horas, como mostrado abaixo.

```
Exemplo de linha:
python timeofday.py
Digite o nome do arquivo: mbox-short.txt
04 3
06 1
07 1
09 2
10 3
11 6
14 1
15 2
16 4
17 2
18 1
19 1
```

Exercício 10.3 Escreva um programa que leia um arquivo e imprima as letras em ordem decrescente de frequência. Seu programa deve converter todas as entrada para minúsculo e somente contar as letras de a-z. Seu programa não deve contar espaços, dígitos, pontuação ou qualquer outros que não sejam as letras de a-z. Encontre exemplos de textos de diferentes linguagens e veja como a frequência das letras variam entre as linguagens. Compare seu resultado com as tabelas em `wikipedia.org/wiki/Letter_frequencies`.