Problemas com a língua espanhola? Pergunte ao Pyton e à R.A.E.

Com a palavra: Señor Python

Depois de uma sessão de bate papo, achamos muito natural escrever "vc" em vez de "você" ou "kra" em lugar de "vossa senhoria". Quando temos que redigir uma carta formal, entretanto, começam as dúvidas. "Viagem é com G ou com J? A palavra 'hêxito' existe?" Herrar é umano, seus bando de anaufabeto! Vamos parár de estuporar a lingoa com a ajuda do Phytom antes que seje tarde...

por José María Ruíz e José Pedro Orantes

Desde que a Internet surgiu escreve-se mais. Supõe-se que os computadores e seus fabulosos corretores ortográficos e gramaticais não nos deixariam assacinar a lingüa. Na prática, a teoria é outra: esses adjutórios nem sempre estão presentes em todos os cantos do sistema operacional. Nos mensageiros instantâneos e no correio eletrônico, por exemplo, usamos muitos jargões, termos técnicos e palavras em inglês – o que nos obriga, via de regra, a desligar os corretores.

Tanto o GNOME como o KDE dispõem de programas próprios para consultar dicionários. São bastante úteis quando, em algum texto, aparece uma palavra difícil ou jargão desconhecido, em qualquer língua. Os programas, então, fazem consultas, sejam locais ou via Internet, a prestigiosos dicionários como o *Webster* ou o famosíssimo (e desconcertante) *Jargon file*.

Entretanto, nem todo mundo quer usar o GNOME ou o KDE (um dos autores deste artigo, pelo menos, não usa). Parece-nos, por isso, um tanto fora de propósito instalar todo um ambiente gráfico apenas e tão-somente para ter acesso a um programa de dicionário. Mas não tema! Podemos fazê-lo sem demasiada dificuldade usando o nosso querido Python e a maravilhosa biblioteca gráfica Tkinter.

Falando com o servidor da *R.A.E.*

Já que os autores deste artigo são espanhóis, além do fato de que este idioma é cada vez mais importante no Brasil devido ao Mercosul, nossos exemplos estarão centrados na língua espanhola. A *R.A.E.* (*Real Academia de la lengua Española*) possui uma página (www.rae.es) na qual é possível consultar um dicionário ou resolver dúvidas lingüísticas. Ter que acessar a página a cada apuro idiomático dá um pouco de trabalho (e é mais uma janela aberta), portanto precisamos de uma alternativa.

Apesar da sopa de letrinhas a que estamos expostos nestes nossos dias (*PHP*, *J2EE*, *.NET*), no fundo todo acesso a um servidor web se dá por meio do protocolo *HTTP*. Esse protocolo não possui estados e serve, basicamente, para solicitar arquivos. Na teoria, o protocolo implementa um grande número de mensagens, mas na prática só dois "comandos" HTTP, se é que se pode chamar assim, são usados: *GET* e *POST*. Se formos olhar no documento que especifica o protocolo, encontraremos muitos mais – existe um que se chama *PUT* e serve para enviar arquivos ao servidor, mas tente encontrar algum software que o tenha implementado... Portanto, só GET e POST é que, na vida real, existem. O GET serve para pedir um arquivo através de uma URL. Já o POST serve para enviar informações ao servidor HTTP.

Se olharmos por esse prisma, um servidor HTTP vê apenas duas coisas: arquivos que pedimos e dados que enviamos. Aliás, há uma curiosidade aí: o POST serve unicamente para enviar dados, mas o GET tem dupla função, podendo solicitar arquivos e, surpreendentemente, também enviar dados ao servidor.

Tanto o GET quanto o POST se prestam, então, ao envio dados ao servidor. Ambos têm vantagens e inconvenientes. Geralmente se usa o GET porque os parâmetros que queremos passar vão "embutidos" na URL. Assim, é possível, por exemplo, guardar uma página nos favoritos já com alguns dados incluídos. Com POST isso não é possível, já que os parâmetros são passados em uma conexão independente e, portanto, não estão presentes na URL. A documentação que rege os padrões diz que o método GET deve ser usado para passar parâmetros de alguma página ao servidor – em um site de comércio eletrônico, por exemplo, o número de catálogo do produto desejado pode ser passado na URL. Já com relação ao POST, os manuais insistem que deve ser usado para passar blocos de dados ao servidor para posterior armazenamento – os dados de um formulário de inscrição de um site, por exemplo.

Se entrarmos em algumas páginas da web e prestarmos atenção às URLs que vão sendo visitadas, podemos notar que várias delas aceitam opções e parâmetros e conseguimos até reconhecer cada um deles. Por exemplo, se a URL é www.lojadopedro.com.br/catalogo.php?produto=437, sabemos que essa loja virtual possui um parâmetro chamado *produto* ao qual foi atribuído o valor `437` (que corresponde, por exemplo, a um secador de gelo). Se alterarmos diretamente na URL o número para `438`, poderemos ir para o próximo item do catálogo (um desentortador de bananas) sem precisar clicar em nada.

Informação importante! Podemos manipular nós mesmos essas opções e parâmetros na própria URL, o que nos permitirá "falar" com o servidor sem a necessidade de usar o navegador como intermediário. É possível, portanto, criar programas que esnobem completamente o navegador e acessem diretamente o servidor HTTP. Esse é, então, o objetivo deste artigo. Temos que localizar esses parâmetros no servidor HTTP da *R.A.E.*, mais precisamente na página de busca de palavras.

Nesse ponto vamos brincar de detetives. Primeiro, acessamos a página da R.A.E e clicamos em *Diccionario de la lengua española*. Nossa primeira pista é bastante estranha: a URL não muda! Não importa o link clicado, o endereço é sempre www.rae.es. Isso significa que nosso navegador está solicitando sempre o mesmo *objeto* ao servidor HTTP – o que causa ainda mais estranheza, já que o conteúdo muda! Isso pode significar várias coisas. Uma análise mais atenta do código fonte em HTML é o suficiente para matar a charada: o site está usando quadros de texto, os famigerados *FRAMES*.

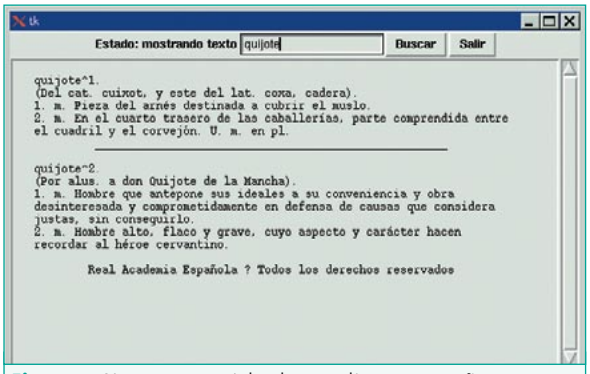

Figura 1: Nosso programinha de consulta em execução.

O *W3C* [1], órgão que dita os padrões da web, desaconselha com veemência o uso da tag `FRAMESET`. A razão é simples: ela mascara a URL e engana o usuário. Por exemplo, estou dizendo que estamos na página de busca do dicionário da *R.A.E.*, mas a URL mostrada na barra de endereços do navegador é, ainda, a da página inicial (www.rae.es). Não podemos indicar a página do dicionário a um amigo, simplesmente porque não sabemos que página é essa: só temos o endereço principal do site. Os FRAMES (que a terra os coma!) destroem completamente a idéia de que um endereço qualquer aponta para um único objeto na Internet.

A página de busca é composta por dois malditos FRAMES (ou subpáginas): *cabecera* e *portada* (*cabeçalho* e *conteúdo*, respectivamente – os nomes, em espanhol, são os que aparecem no código fonte da página). A *cabecera* não nos interessa. A *portada*, que contém o conteúdo, compõe-se por sua vez de dois outros FRAMES: *elección* e *presentación*. É este último que apresenta os resultados da busca. Mas como ele sabe qual palavra queremos consultar?

Cada FRAME contém várias páginas e os FRAMES se sucederão para mostrá-las. Apenas uma delas é de especial interesse para nós: buscon.rae.es/drael/SrvltGUIBusUsual. Essa página aceita um parâmetro chamado `LEMA` e devolve outra página com a definição da palavra atribuída a `LEMA` – ou um aviso de falha, se ela não existir no dicionário. O método usado é POST, como podemos ver nas informações da página (clique com o botão direito na página, escolha a opção *Visualizar informações da página* e, depois, a aba *Formulários*). Os parâmetros passados são LEMA e o método é POST. Muitos aplicativos web, em particular os que usam servlets em Java, podem ocasionalmente buscar seus parâmetros tanto em POST como em GET. Que sorte, esse é o nosso caso!

Como passar parâmetros? É muito simples, vamos usar GET. O formato normalmente é o mesmo: `?<PARAMETRO1>=<VALOR1>&<PARAMETRO2>=<VALOR2>`. Por exemplo: buscon.rae.es/drael/SrvltGUIBusUsual?LEMA=hola. Agora temos que "bolar" um jeito de obter o conteúdo dessa URL.

O bom e velho e velho Lynx

Poderíamos ter projetado um sistema super complexo para trazer a página HTML, eliminar as marcas (*tags*), formatá-la e demais amenidades. Mas a vagabundagem, poderosa aliada do bom programador, nos faz buscar uma maneira mais fácil de fazer isso. De preferência, algo que já

esteja pronto. Se for o trabalho de outra pessoa – pelo qual não vamos pagar nada – melhor ainda.

A maneira mais simples é, sem dúvida, seguir o exemplo do *xdrae* [2]. Esse programinha, escrito em Tcl/Tk, é na verdade um aproveitador descarado que se apóia num pobre operário, o *lynx*. Para quem não conhece, o lynx é um navegador web completo em modo texto. Trocando em miúdos: ele vai até o servidor, baixa a página que queremos ver e a formata de acordo com as marcas HTML, apresentando na tela (em modo texto!) seu conteúdo.

Isso nos ensina duas lições muito importantes sobre o modo UNIX de fazer as coisas. A primeira é que, se o código fonte está disponível livremente, é obrigação moral de todo programador usá-lo. A segunda lição é que, no UNIX, não existe um só programa, pesado e abrangente, que faça de tudo um pouco. No modo UNIX de ser, temos uma infinidade de pequenos programas que fazem uma única coisa, mas o fazem bem feito. Esses pequenos "blocos de construção" podem ser conectados por "canos" (os famosos *pipes*, representados pelo caracter |) para que suas funcionalidades sejam encadeadas. Por exemplo, no comando encadeado `ls -l | wc -l`, o `ls -l` mostra todos os arquivos de uma determinada pasta, um por linha, enquanto o `wc -l` conta o número de linhas. Como resultado, aparece na tela o número de arquivos daquela pasta. Para que tudo isso seja possível, entretanto, a informação que o programa anterior injeta no próximo deve estar em modo texto.

O lynx, normalmente, comporta-se como um navegador comum, tomando posse da tela e permitindo que se navegue por várias páginas seguindo os links apresentados. Mas o programa aceita uma miríade de parâmetros. Um deles, em particular, é genial. O parâmetro `-dump` joga o conteúdo da página acessada, já formatada (em ASCII) e sem marcas HTML, diretamente na saída padrão e, em seguida, encerra a execução do lynx – o que o torna mais facilmente "encaixável" em outros programas.

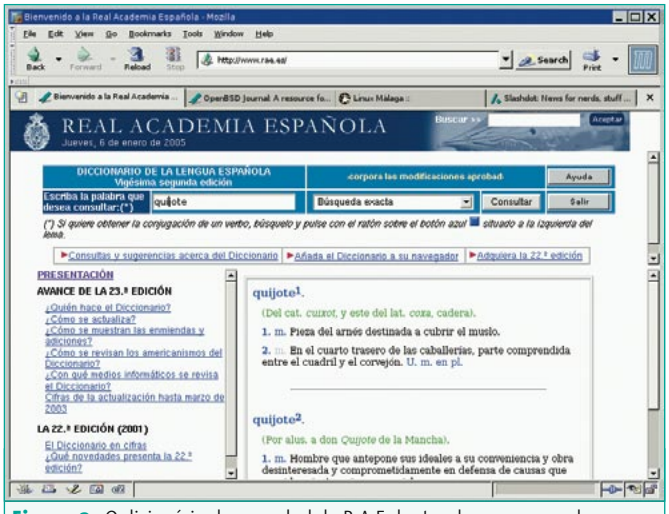

Figura 2: O dicionário de espanhol da R.A.E dentro de um navegador

Mas a página mostrada ainda tem um probleminha: além de seu conteúdo, o lynx monta uma lista no final com todos os links mostrados. Para dar cabo deles há outro parâmetro: `--nolist`. O comando ficaria, então, `lynx -dump --nolist URL`.

popen2

Para poder recolher o que o lynx produz temos que criar uma tubulação entre ele e nosso programa. Para isso, usaremos a função `popen2`, presente na biblioteca de mesmo nome. Essa função executa um programa e cria dois descritores de arquivos, um para enviar o texto ao programa e outro para recebê-lo – respectivamente, a entrada padrão e a saída padrão do programa sendo executado.

A idéia é executar o lynx em comunhão com o popen2. Sua entrada padrão não nos interessa e portanto a fechamos, mas a saída padrão é exatamente o que queremos. Uma vez executada a função popen2, a página produzida pelo lynx pode ser lida como se lêssemos um arquivo em disco.

```
>>> saída, entrada = popen2 ("wc -w")
>>> entrada.write("En un lugar de la Mancha de cuyo nombre no ➥
quiero acordarme")
>>> entrada.close()
>>> print saida.read()
      12
>>>
```

O trecho acima está no primeiríssimo parágrafo da obra literária mais relevante e sublime de todos os tempos: Dom Quixote [3], de Cervantes (só por acaso, espanhol...). O comando `wc -w` devolve o número de palavras que foram lidas da entrada padrão até o fim do arquivo (por isso é que tivemos que fazer um `entrada.close()`: para forçar um fim de arquivo).

Agora só temos que explorar o navegador lynx e ler sua saída, armazenando o valor produzido (no caso, o conteúdo da página visitada) em uma variável.

Script de consulta simples

O *xdrae* usa exatamente esse método, invocando o lynx, que faz a maior parte do trabalho, deixando ao próprio xdrae apenas a representação gráfica dos resultados na tela. Um script simples para que consigamos os mesmos resultados seria o da **listagem 1**.

Quando o Python começou a despontar como grande promessa, fez-se evidente a necessidade de um programa especial para a criação de interfaces gráficas. As pessoas poderiam tentar fazer como no Java e criar sua própria biblioteca gráfica (no caso, *AWT* e *Swing*) mas, em lugar disso, optou-se por usar uma já existente: a anciã *Tcl/Tk*.

Tcl/Tk

A sigla Tcl/Tk soa assaz estranha. Esse nome duplo pode nos trazer à mente o TCP/IP, que é na verdade um conjunto de protocolos que trabalham em simbiose – e, de fato, o paralelo é exatamente esse. Tanto o Tcl/Tk quanto o TCP/IP fazem referência a duas tecnologia que trabalham juntas. No caso do Tcl/Tk, as tecnologias em questão são a linguagem de scripts *Tcl* e a biblioteca de interface gráfica *Tk*. O Tk foi criado depois do Tcl e, talvez por isso, foi implementado como uma extensão desse último.

O Tcl foi criado pelo professor John K. Ousterhout com o objetivo de conseguir uma linguagem de script que fosse fácil de se embutir em um programa em C. A idéia era que o usuário pudesse estender a funcionalidade do aplicativo sem ter que programar na complicada linguagem C. Com o Tcl, o usuário criaria programinhas simples que se "encaixariam" no aplicativo principal.

Pouco tempo depois começou a trabalhar no Tk, uma extensão do Tcl que permitiria a criação de interfaces gráficas de maneira rápida e simples graças ao Tcl.

Mesmo que, numa remota possibilidade, nunca tenhamos escutado falar de Tcl/Tk, sua importância é muito grande. Muitas linguagens usam o Tk como biblioteca gráfica. O conjunto Tcl/Tk é também grandemente difundido como linguagem para prototipação de aplicativos. Mas não só de protótipos vive o Tcl/Tk. Programas inteiros podem ser desenvolvidos com ele, como o famoso clone do MSN Messenger da Microsoft, o *aMSN* [4].

Um "Olá, Mundo!" gráfico

Depois dessa pequena lição de história, que tal pôr a mão na massa? A primeira coisa que precisamos saber é como é a estrutura dos *widgets* no Tk – usaremos a palavra widgets a partir de agora, por ser mais curta, mas queremos deixar claro que em português (e também em espanhol) widget significa *elemento gráfico*. O Tk usa uma árvore para eles, parecida com a árvore de diretórios do seu disco rígido. Entretanto, em lugar da barra habitual (`/`) usamos um ponto (`.`) para separar cada ramo. O aplicativo é criado a partir da raiz `.` e desenvolve-se daí em diante. Os widgets podem conter outros widgets, ou seja, os widgets podem ser "diretórios" de outros widgets.

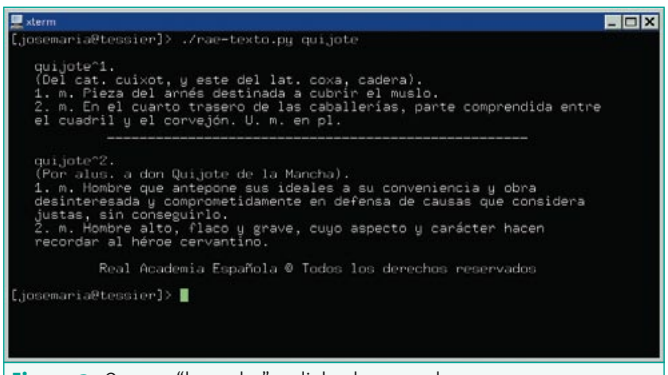

Figura 3: O nosso "buscador" na linha de comando.

Ficou confuso? Um exemplo sempre ajuda. Imagine que você tenha um widget chamado `.painel`. Esse widget pode ter dois outros widgets subordinados a ele, que chamamos de *filhos*. Poderiam ser, por exemplo, `.painel.campo_texto` e `.painel.botaoOK`. Visualmente, veríamos `campo_texto` e `botaoOK` dentro de `.painel`.

Para ver como é na prática, vamos analisar um programa do tipo "olá mundo":

```
>>> import Tkinter
>>> from Tkconstants import *
>>> tk = Tkinter.Tk()
>>>
>>> painel = Tkinter.Frame(tk, relief=RIDGE, borderwidth=2)
>>> painel.pack(fill=BOTH, expand=1)
>>>
>>> etiqueta = Tkinter.Label (painel, text="Olá pessoal da ➥
    Linux Magazine!")
>>> etiqueta.pack(fill=X, expand=1)
>>>
>>> botao = Tkinter.Button (panel,text="Sair", command=tk.destroy)
>>> botao.pack(side=BOTTOM)
>>>
>>> tk.mainloop()
```

O método `pack` é muito importante. Quando você cria um widget dentro de seu "pai", ele não aparece na hora. Para que ele seja corretamente posicionado e mostrado na tela, é preciso "dar um pack" como método do widget – em nosso exemplo, `botao.pack` usa o método `.pack` dentro do widget `botao` para que este último seja exibido. Especificando a propriedade `side=BOTTOM`, posicionamos os widgets uns sobre os outros. `fill=X` serve para que o widget ocupe todo o comprimento na horizontal. Se fosse `fill=Y`, o objeto ocuparia toda a vertical. Já com `fill=XY`, ocuparíamos todo o espaço disponível. `expand=1` faz com que o widget cresça se seu pai crescer; sem ele o widget não modificaria suas dimensões.

O Tk funciona da seguinte maneira: primeiro, construímos a interface gráfica. Cada elemento dessa interface responde a uma série de eventos – por exemplo ser clicado, ou o ponteiro do mouse passar por cima, ou uma ação do

teclado. Cada um desses eventos pode estar vinculado a um procedimento, que é disparado quando o evento ocorrer. Dessa forma, colocamos "por trás" dos widgets a inteligência necessária para que o programa funcione. Por último entramos no *laço principal* (*main loop*).

Figura 4: Nosso "Olá, Mundo!" escrito em Tcl/Tk.

Em Tcl/TK, todas as chamadas a funções de criação de widgets recebem um primeiro argumento. Nele, especificamos quem *carrega* nosso widget (ou, explicando de outra forma, dentro de qual elemento nosso widget está). Em nosso exemplo, o widget `tk` representa a janela principal, dentro da qual há um painel, dentro do qual há dois outros widgets, uma etiqueta e um botão.

O botão tem vinculado dentro de si um comando: `tk.destroy`, que simplesmente "descria" (ou destrói) o widget principal, `tk`, junto com todos os seus filhos – mas poderia ser qualquer coisa. Poderíamos ter incorporado um outro botão *antes* de executar o `tk.mainloop()`:

```
>>> def escreve():
...     print "Olá, cambada!"
...
>>>
>>> botao2 = Tkinter.Button (panel, text="Mensagem", command= escreve )
>>> botao2.pack(side=BOTTOM)
```

Isso abriria outro botão sob o botão `Sair`. Se clicássemos nele, que tem o nome de *Mensagem* na tela (embora internamente seja chamado de `botao2`), apareceria a mensagem *Olá, cambada!* no console de texto. Note que, ainda, não estamos direcionando a mensagem para a interface gráfica.

O projeto da interface gráfica

O Tk tem todos os widgets que existem nas outras bibliotecas gráficas: botões, etiquetas, painéis, imagens, caixas de texto e tudo o que você puder pensar. Para nosso dicionário, precisamos de:

- Um painel
- Uma etiqueta *Estado*, um campo de entrada de texto, um botão *Buscar* e outro *Sair*.
- Um widget de texto com várias linhas para que possamos mostrar a resposta, com uma barra de rolagem associada.
- Para ficar "bonitinho", um título para a janela.

Barras de rolagem + Campos de texto = combinação bastante útil

Um widget de texto é um widget que nos permite mostrar uma grande quantidade de texto. Possui uma infinidade de opções, sendo possível implementar com elas um interpretador de HTML ou criar um texto formatado.

Vamos usar um widget de texto para exibir a resposta do servidor da *R.A.E.*. Temos que solucionar o problema de tamanho: a resposta da *R.A.E.* pode ser muito maior do que o tamanho do nosso campo. Nosso widget de texto não vai ser redimensionável e talvez haja mais texto do que ele pode mostrar.

Para solucionar esse problema, algum iluminado um dia inventou as *barras de rolagem*. Hoje as consideramos "carne de vaca" e não imaginamos que, um dia, elas não existiam. Quando surgiram, foi uma inovação sem precedentes. Nossa idéia é, então, criar uma barra de rolagem e vincular seu movimento e tamanho ao texto mostrado no widget de texto.

No Tk, não podemos vincular o tamanho. Resta-nos, então, o movimento. Os widgets que têm permissão para exibir barras de rolagem possuem um evento chamado `yscrollcommand`, que toma o valor de uma função e o usa para posicionar o texto dentro do widget. Este possui um elemento chamado *viewport*, que é como se fosse uma janela dentro do widget. O usuário só vê o que está nessa "janela". O widget pode conter, por exemplo, os dois volumes de *Don Quijote*, mas no viewport aparecerão apenas uns três ou quatro parágrafos da magnífica obra de Cervantes [5]. Quando vinculamos uma barra de rolagem ao widget de texto, o que fazemos na verdade é vincular o valor de início do viewport à posição da barra.

Listagem 1: Programa buscador.py

```
01 #!/usr/bin/python
02
03 from popen2 import popen2
04 from sys import argv
05
06 def busca_palavra(palavra):
07     saida, entrada = popen2('lynx -dump --nolist "http://
   buscon.rae.es/draeI/SrvltGUIBusUsual?TIPO_HTML=2&LEMA=' +
   palavra + '"')
08     entrada.close()
09     cadeia = saida.read()
10     print(cadeia)
11
12 if __name__ == "__main__":
13     if ( len(argv) == 1 ):
14         print("ERRO: falta um argumento")
15     else:
16         busca_palavra(argv[1])
```

LINUXWORLD™
CONFERENCE & EXPO

WHERE open minds MEET!

África do Sul
Alemanha
Austrália
Canadá
China
EUA
Holanda
Itália
Japão
Korea
México
Reino Unido
Rússia

A LinuxWorld Conference & Expo chega ao Brasil

23-25 de Maio de 2006

Gran Meliá WTC - São Paulo - SP

Mais informações entre em contato:

Tel.: + 55 11 5502-7278 / Fax: +55 11 5505-7872

info@linuxworldbrasil.com.br

www.linuxworldbrasil.com.br

Organizadores: Reed Exhibitions, IDG World Expo
Veículo Oficial: Linux Magazine
Certificador Oficial: Linux Professional Institute
Media Partner: IDG Brasil
Membros de: UBRAFE

Montagem final

E agora, o momento que todos esperávamos. Vamos colocar no caldeirão todas as técnicas e idéias que vimos até aqui e ver se aparece algo que preste. Basicamente o que nosso programa faz é criar uma interface gráfica. Esta, por sua vez, se encarrega de criar e vincular as ações que os widgets requerem. Uma vez criadas, é necessário invocar o método `inicio()`, que executa o laço principal do Tk. Esse laço, ciclicamente, trata de redesenhar a janela e gerenciar os eventos que possivelmente ocorram.

O elemento que tem a missão de buscar na *R.A.E.* uma palavra através do botão *Buscar* é o método `busca_palavra`. Ele executa o lynx com as opções necessárias e com o conteúdo do campo de entrada. Faz isso por intermédio da função popen2. Quando o texto está pronto, ele apaga o conteúdo anterior do widget de texto e insere o texto que acabou de baixar do servidor HTTP da *R.A.E.*. O botão "Sair" fecha o programa.

Cada vez que isso acontece, o programa fica "bloqueado" até ter um resultado. A culpada disso é a própria função popen2, que bloqueia o programa que a chamou até que o lynx termine de executar. Para contornar esse comportamento pegajoso devemos usar as chamadas *threads* (ou múltiplas instâncias de um processo), mas isso está fora do escopo deste artigo. ■

Informações

[1] Página do W3C: www.w3c.org

[2] Página do *xdrae*: xinfo.sourceforge.net/xdrae.html

[3] *Don Quijote de la Mancha*: www.gutenberg.org/etext/2000

[4] Alvaro's MSN Messenger: amsn.sourceforge.net

[5] Miguel de Cervantes Saavedra (em inglês): en.wikipedia.org/wiki/Miguel_de_Cervantes

[6] Miguel de Cervantes Saavedra (em português) pt.wikipedia.org/wiki/Miguel_de_Cervantes

Listagem 2: Programa buscador2.py

```
01 #!/usr/bin/python
02
03 from popen2 import popen2
04 from Tkinter import *
05 from Tkconstants import *
06
07 class GUI:
08     def __init__(self):
09         self.root = Tk()
10         self.cria_painel_busca()
11         self.cria_painel_mostra()
12
13     def inicio(self):
14         mainloop()
15
16     def busca_palavra(self):
17         self.label_estado.config(text="Estado: buscando
   palavra...")
18         palavra = self.entry_palavra.get();
19         saida, entrada = popen2('lynx -dump --nolist
   "http://buscon.rae.es/draeI/SrvltGUIBusUsual?LEMA=' +
   palavra + '"')
20         entrada.close()
21         texto = saida.read()
22         # Apagamos o texto anterior
23        self.label_estado.config(text="Estado: mostrando texto")
24         self.text_mostra.delete(1.0,END)
25         self.text_mostra.insert(1.0,texto)
26         saida.close()
27
28     def cria_painel_mostra(self):
29         self.painel_mostra = Frame(self.root)
30         self.painel_mostra.pack()
31
32         # Vinculamos uma barra de rolagem
33         self.scrollbar= Scrollbar(self.painel_mostra)
34         self.scrollbar.pack(side=RIGHT,fill=Y)
35
36         # Caixa de texto para mostrar a definição da
37         # palavra
38         self.text_mostra = Text(self.painel_mostra,
   yscrollcommand = self.scrollbar.set)
39         self.text_mostra.pack()
40
41     def cria_painel_busca(self):
42         self.painel_busca = Frame(self.root)
43         self.painel_busca.pack()
44
45         self.label_estado = Label(self.painel_busca,
   text="Estado: parado")
46         self.label_estado.pack(side=LEFT)
47
48         self.entry_palavra = Entry(self.painel_busca)
49         self.entry_palavra.pack(side=LEFT)
50
51         # Botão para sair do programa
52         self.btn_sair = Button(self.painel_
   busca,text="Sair", command= self.root.destroy)
53         self.btn_sair.pack(side=RIGHT)
54
55         self.btn_busca = Button(self.painel_busca,
   text='Buscar', command=self.busca_palavra)
56         self.btn_busca.pack(side=RIGHT)
57
58 if __name__ == "__main__":
59
60     app = GUI()
61     app.inicio()
```