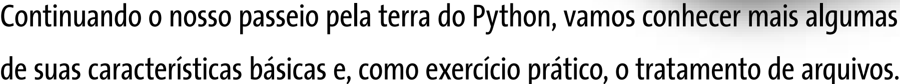
Operações básicas com arquivos

Criando um álbum de imagens



Continuando o nosso passeio pela terra do Python, vamos conhecer mais algumas de suas características básicas e, como exercício prático, o tratamento de arquivos.

por José Maria Ruíz e José Pedro Orantes

www.sxc.hu

Na edição anterior tivemos uma primeira idéia sobre como trabalhar com objetos na linguagem Python. Foi algo muito simples, mas já nos dava condições de organizar nosso código em torno deles. O Python faz uso intensivo de objetos em suas APIs, especialmente no tratamento de exceções e erros – o que nos dá a possibilidade de criar nossas próprias exceções quando algo vai mal. Uma exceção é uma mensagem que podemos capturar quando se executa uma dada função ou método e surge algum tipo de erro.

Em outras linguagens, normalmente controlamos esses erros pelo valor devolvido pela função quando o erro ocorre – na linguagem C esse é o procedimento padrão. Essa técnica é cheia de "pegadinhas" e, como tudo na vida, tem suas vantagens e desvantagens. Entretanto, o Python usa as tais exceções e não o retorno de valores de erro.

Quando uma função gera uma exceção dizemos que ela *levanta uma exceção*. É bastante comum termos que tratar exceções em nossos programas, já que muitas das operações que precisamos fazer podem afetar recursos que não estão disponíveis no momento. Como neste artigo vamos trabalhar com arquivos e conexão à Internet – recursos que, em um determinado instante, podem não estar disponíveis – precisamos ter uma idéia básica de como tratar essas exceções.

Para começar, vamos criar um objeto que administre um recurso que pode não estar disponível. Em nosso primeiro exemplo, o objeto gerencia uma variável (ver **listagem 1**). Poderíamos criar um objeto da classe `obj_variable` e chamar o método `set_variable(23)`. Mas como poderíamos nos certificar de que a variável `var` contém o valor 23 depois da chamada? Pode ser que `var` não fosse 0, porque uma chamada anterior poderia ter alterado seu conteúdo. A única coisa que poderíamos fazer é chamar `reset_variable()` e assim assegurar que nosso valor seja atribuído à variável. Só que, dessa forma, destruiríamos o valor anterior e não saberíamos o que poderia acontecer.

Portanto, precisamos de um mecanismo de comunicação que informe ao usuário que um valor foi atribuído a essa variável. Para isso, as exceções nos serão de grande valia.

Na **listagem 2** aparece uma classe que herda, da classe `Exception` (veremos os mecanismos de herança em outro artigo dedicado aos objetos), uma exceção chamada `Var_Atribuida`. Quando tentamos, na classe `obj_variable`, atribuir um valor à variável `var` e esta não é zero, o Python dispara – ou melhor, *levanta* – a exceção `Var_Atribuida`. Se não controlarmos a porção de código em que se encontra `set_variable()` e uma exceção aparece, o Python encerra a execução do programa.

Listagem 1

```
01 >>> class obj_variable:
02 ...     __init__(this):
03 ...         var = 0
04 ...
05 ...     set_variable(this, valor):
06 ...         if (var == 0):
07 ...             var = valor
08 ...
09 ... reset_variable(this):
10 ...     var = 0
11 ...
>>>
```

A idéia por trás das exceções é que é possível tratá-las e evitar males maiores – em alguns casos, as exceções podem ser usadas para recuperação de erros. Para o tratamento de exceções temos a estrutura *try-except*. Envolvemos com essa estrutura o código cujas exceções queremos tratar (veja **listagem 3**).

A partir de agora, quando dissermos que uma função gera uma exceção, estamos considerando que essa função está envolvida pela estrutura try-except. Não vamos nos aprofundar muito no tema exceções neste artigo; basta saber que são geradas por condições de erro e que podem ser tratadas.

Listagem 3

```
01 >>> try:
02 ...    set_variable(12)
03 ...    set_variable(34)
04 ... except:
05 ...    print "ERRO: Tentativa de atribuir um"
06 ...    print "valor a uma variável já usada!"
07 ...
08 ERRO: Tentativa de atribuir um
09 valor a uma variável já usada!
10 >>>
```

Trabalhando com arquivos

Como já conhecemos os conceitos de objetos em Python, podemos agora falar sobre como acessamos os arquivos em disco:

Para acessar um arquivo, primeiro precisamos criar um objeto `file`. Este objeto faz parte da biblioteca padrão do Python, portanto não precisamos importar nenhuma biblioteca externa adicional.

```
>>> arquivoz = file('texto.txt')
```

Por padrão, `file` abre os arquivos em modo somente leitura. Isso significa que, se o arquivo não existir, será gerado um erro. Para verificar se o arquivo existe, podemos usar a função `exists()` da biblioteca *os.path*.

```
>>> import os.path
>>> os.path.exists('texto.txt')
True
>>> os.path.exists('cabra-peludo-e-feio.txt')
False
```

Portanto, se vamos abrir um arquivo, temos que verificar se ele já existe. Se em lugar de ler um arquivo existente quiséssemos criar um novo, devemos invocar o construtor de `file` com os parâmetros:

```
>>> arquivox = file('texto.txt','w')
```

Esse segundo parâmetro opcional nos permite definir o tipo de acesso que ao arquivo que vamos fazer. Temos várias possibilidades: podemos ler (`r`), escrever (`w`), adicionar dados ao final do arquivo (`a`) e até mesmo obter acesso de leitura e escrita (`r+w`). Dispomos também do modificador `b` para indicar acesso binário. O Python considera, a não ser que indicado em contrário, todos os arquivos como sendo de texto. Veremos todas as combinações possíveis na **listagem 4**.

Se tudo correr bem, com qualquer dessas chamadas teríamos em `arquivox` um objeto que gerencia o arquivo indicado – e podemos manipular o arquivo fazendo operações com o objeto. As operações mais comuns são ler a partir do arquivo e escrever nele. Para isso, o objeto `file` dispõe dos métodos `read()`, `readline()`, `write()` e `writeline()`. Todos operam com cadeias de caracteres: `readline()` e `writeline()` trabalham com linhas de texto (terminadas com um retorno de carro, ou seja, a tecla ***[Enter]***), enquanto `read()` e `write()` o fazem com cadeias de caracteres sem limites.

Na **listagem 5** vemos algumas operações de manipulação de um arquivo. A primeira coisa que temos que fazer é criar o tal arquivo. Usaremos o modo de escrita `w`, que criará um arquivo

Listagem 2

```
01 >>> class Var_Atribuida(Exception):
02 ...    """ Exceção disparada ao se tentar atribuir um valor
03 ...    a uma variável já usada em obj_variable"""
04 ...    pass
05 ...
06 >>>
07 >>> class obj_variable:
08 ...    """ Administra uma variável """
09 ...    def __init__(self):
10 ...       self.var = 0
11 ...
12 ...    def set_variable(self, valor):
13 ...       if (self.var == 0):
14 ...          self.var = valor
15 ...    else:
16 ...       raise Var_Atribuida
17 ...
18 ...    def reset_variable(self):
19 ...       self.var = 0
20 ...
21 >>>
22 >>> a = obj_variable()
23 >>> a.set_variable(12)
24 >>> a.set_variable(34)
```

Listagem 4

```
01 >>> arquivo = file('texto.txt','r')
02 >>> arquivo = file('texto.txt','w')
03 >>> arquivo = file('texto.txt','a')
04 >>> arquivo = file('texto.txt','r+w')
05 >>> arquivo = file('texto.txt','r+b')
06 >>> arquivo = file('texto.txt','rb')
```

caso não exista um ou truncará o existente – trocando em miúdos, apagará o antigo e criará um novo, vazio. Se quiséssemos adicionar texto a um arquivo existente sem apagar seu conteúdo usaríamos `a`. Depois escrevemos nele uma cadeia de caracteres com um retorno de carro na metade da linha e, em seguida, fechamos o arquivo. Esse "***[Enter]***" no meio da linha servirá para nossas experiências com o código. É boa prática de programação fechar todos os arquivos quando não estiverem mais em uso. Em nosso caso, ainda o vamos usar, mas fechamos simplesmente para abri-lo novamente em modo somente leitura (`r`).

Com o arquivo aberto apenas para leitura, lemos 4 bytes e os armazenamos na variável `cadeia`. Quando lemos com `read()`, vamos avançando pelo arquivo. Por essa razão, o `readline()` posterior lerá apenas `mundo\n` em vez de ler `Caro mundo`. Também percebemos que ele pára o processamento no retorno de carro, em lugar de prosseguir. O segundo `readline()` vai direto para a cadeia `Adeus, mundo cruel`.

Mas... o que ocorreria se em uma das leituras encontrássemos o fim do arquivo? Se lêssemos uma cadeia de caracteres com um fim de arquivo nela (*EOF* – End of File), apenas o EOF seria lido e obteríamos uma cadeia nula (*null*). Isso é importante para comprovar que chegamos ao fim do arquivo. Assim, um laço que escreva na tela o conteúdo do arquivo verificaria, em cada iteração, se a cadeia de caracteres devolvida por `readline()` é *null*.

Agora que já sabemos criar arquivos, temos que aprender a apagá-los. Nada mais fácil: existe uma função chamada `remove()`, da biblioteca *os*, que faz isso. Essa função toma como parâmetro o caminho completo até o arquivo (incluindo seu nome) e o apaga. Se em lugar de um arquivo passarmos o nome de um diretório, o Python subirá a exceção `OSError`.

```
>>> import os
>>> os.remove(texto.txt)
>>>
```

Diretórios e sistemas de arquivos

Com esses pouco métodos temos ao nosso alcance a manipulação básica de arquivos. Mas nosso programa deve também ter a capacidade de criar diretórios. Para isso, a função `mkdir()` vem bem a calhar. Ela aceita um nome como parâmetro e cria um diretório com ele. Se quisermos criar uma árvore completa de diretórios (por exemplo, queremos criar o diretório `/material/caneta/esferografica/vermelha`, sendo que o diretório `caneta` ainda não existe, muito menos os seus subdiretórios) temos que usar outra função, chamada `makedirs()`. Ambas as funções pertencem ao módulo `os`, portanto para usá-las temos que as importar antes:

```
>>> import os
>>> os.mkdir('MeusDocumentos')
>>> os.makedirs('MeusFilmes/Comedias')
```

Para apagar esses diretórios usaremos a função `rmdir()` e `removedirs()`. A primeira apaga um único diretório, a segunda apaga um caminho completo. Vamos ver o mecanismo mais de perto:

```
>>> os.rmdir('MeusDocumentos')
>>> os.removedirs('MeusFilmes/Comedias')
```

A função `rmdir()` apagará o diretório o `MeusDocumentos`, que não contém nenhum outro objeto em seu interior (nem diretórios, nem arquivos). Caso o diretório não esteja vazio, a função devolverá um erro. Já `removedirs()` começaria a apagar desde o diretório mais "fundo" do caminho (em nosso caso, `Comedias`) e iria apagando em direção ao mais "raso" (em nosso caso, `MeusFilmes`). Imaginemos, entretanto, que dentro de `MeusFilmes` também haja um diretório chamado `Drama`. A função `removedirs()` apagaria o diretório `Comedia`, mas quando fosse apagar `MeusFilmes` levantaria uma exceção, já que `MeusFilmes` não está vazio – contém o diretório `Drama`. Podemos dizer, então,

Listagem 5

```
01 >>> arquivo = file('/tmp/texto.txt','w')
02 >>> arquivo.write("Caro mundo\nAdeus, mundo cruel")
03 >>> arquivo.close()
04 >>>
05 >>> arquivo = file('/tmp/texto.txt','r')
06 >>> cadeia = arquivo.read(4)
07 >>> cadeia
08 'Caro'
09 >>> cadeia = arquivo.readline()
10 >>> cadeia
11 ' mundo\n'
12 >>> cadeia = arquivo.readline()
13 >>> print cadeia
14 'Adeus, mundo cruel'
15 >>> arquivo.close()
```

Listagem 6: O "catador" de imagens

```
001 #!/usr/local/bin/python
002 # -NOTA-------------
003 # O arquivo que deve ser passado
004 # como argumento deve consistir de
005 # uma lista com uma URL por linha.
006 # --------------
007 class Lista_URLs:
008     """Recebe um arquivo e armazena ➘
suas linhas em uma lista. Possui métodos ➘
para recuperar as linhas guardadas lá."""
008
009     def __init__(self,nome):
010         self.lista= []
011         self.contador = 0
012         self.arquivo = file(nome)
013         self.cadeia = self.arquivo.➘
readline()
014
015         while(self.cadeia != '\n'):
016             self.lista.append(self.➘
cadeia)
017             self.cadeia = self.➘
arquivo.readline()
018         self.arquivo.close()
019
020     def rebobina(self):
021         self.contador = 0
022
023     def seguinte(self):
024         if ( self.contador >= ➘
len(self.lista)):
025             return ''
026         else:
027             self.valor = self.➘
lista[self.contador]
028             self.contador = self.➘
contador + 1
029             return self.valor
030
031     def fim(self):
032         return (self.contador == ➘
len(self.lista))
033
034     def distancia(self):
035         return len(self.lista)
036
037 def cria_diretorio(cadeia):
038     componentes = cadeia.split('.')
039
040     if(os.path.exists➘
(componentes[0])):
041         print "Error: o diretorio ➘
ja existe"
042         sys.exit()
043     else:
044         os.makedirs(componentes[0])
045         os.chdir(componentes[0])
046         print 'Criando diretorio ' + ➘
componentes[0]
047
048 def descarga_urls(lista):
049     lista.rebobina()
050
051     while( not lista.fim() ):
052         url = lista.seguinte()
053
054         componentes = url.split('/')
055         servidor = componentes[2]
056
057         caminho_imagem = '/'
058         for i in range( 3,  len➘
(componentes)):
059             caminho_imagem = ➘
caminho_imagem + '/' + componentes[i]
060
061         print 'Baixando imagem: ' + ➘
url[:-1]
062         conexao = httplib.➘
HTTPConnection(servidor)
063
064         conexao.request("GET", url)
065         resposta = conexao.➘
getresponse()
066
067         conteudo = resposta.read()
068         conexao.close()
069
070         if( resposta.status != ➘
'200'):
071             nome_arq = componentes[l➘
en(componentes) -1]
072             nome_arq = nome_arq[:-1]
073             arquivo = file(nome_arq ➘
,'w')
074             arquivo.write(conteudo)
075             arquivo.close()
076         else:
077             print "Não consegui ➘
baixar " + url
078
079 def gera_index(lista):
080     print 'Gerando índice index.html'
081     arquivo = file('index.html','w')
082
083     arquivo.write('<html>\n')
084     arquivo.write('<head>\n')
085     arquivo.write('<title> Imagens ➘
</title>\n')
086     arquivo.write('</head>\n')
087     arquivo.write('<body>\n')
088     arquivo.write('<h1>Imagens</h1>\n')
089     arquivo.write('<ul>\n')
090
091     lista.rebobina()
092     url = lista.seguinte()
093
094     componentes = url.split('/')
095     imagem = componentes[len➘
(componentes) - 1]
096
097     while( url != ''):
098         arquivo.write('<li>➘
<img src=\"'+ imagem +'\"></img></li>\n')
099         url = lista.seguinte()
100         componentes = url.split('/')
101         imagem = componentes[len➘
(componentes) - 1]
102
103     arquivo.write('</ul>\n')
104     arquivo.write('</body>\n')
105     arquivo.write('</html>\n')
106
107     arquivo.close()
108 #----------------
109 # Main (corpo principal do programa)
110 #----------------
111 if __name__ == '__main__':
112
113     import httplib
114     import os
115     import os.path
116     import sys
117
118     if len(sys.argv) == 2:
119         lista = Lista_URLs(sys.➘
argv[1])
120         cria_diretorio(sys.argv[1])
121         descarga_urls(lista)
122         gera_index(lista)
123
124     elif len(sys.argv) == 0:
125         print 'A sintaxes do ➘
programa é:\n'
126         print sys.argv[0] + ' ➘
arquivo\n'
127         print 'O arquivo deve conter ➘
uma URL por linha.'
129     else:
129         print "ERRO: a sintaxe é " ➘
+ sys.argv[0] + " <arquivo>"
```

que `removedirs()` aplica recursivamente a função `rmdir()`, da direita para a esquerda no caminho informado, e gera as mesmas condições de erro.

Imaginemos agora que necessitamos mudar para outro diretório de trabalho. No momento em que o programa é iniciado, o diretório de trabalho é o mesmo em que o programa foi chamado. Entretanto, nem sempre queremos gravar nossos arquivos ali. Qualquer referência a um arquivo será feita considerando o diretório de trabalho, a não ser que indiquemos o caminho completo para o arquivo. Para poder mudar o diretório de trabalho do programa, o módulo `os` tem uma função bacana, o `chdir()`. Se a invocarmos dentro de nosso programa:

```
>>> os.chdir('/tmp')
```

qualquer referência a um arquivo, depois disso, será direcionada para `/tmp`.

Agora podemos:

- Abrir, fechar e modificar um arquivo;
- Criar e apagar um diretório;
- Trocar o diretório de trabalho.

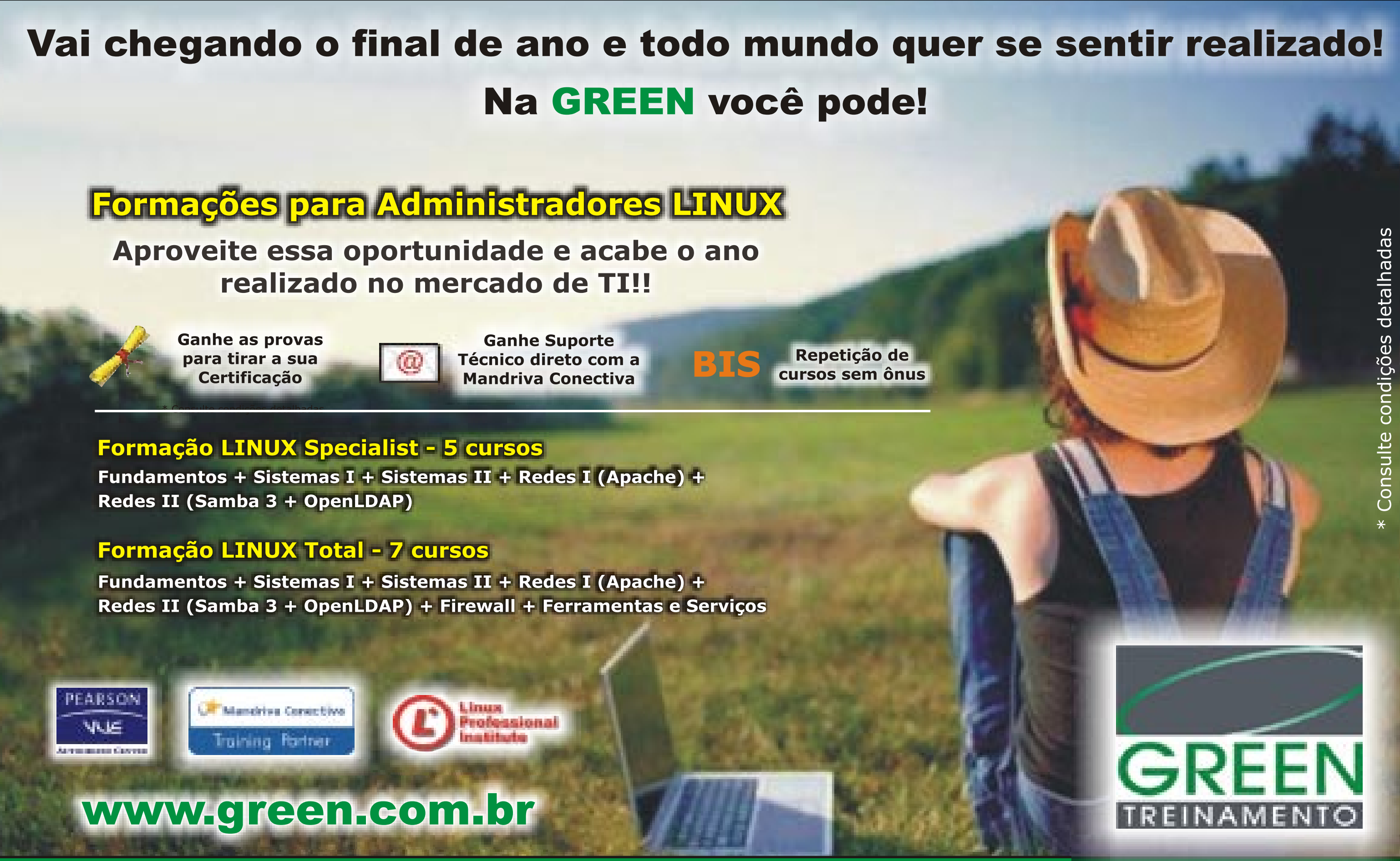
O Python e a web

O Python possui uma quantidade impressionante de bibliotecas para trabalhar com recursos de Internet. O Zope [1], um servidor de aplicações de grande sucesso, foi escrito em Python e faz uso de praticamente todos os recursos de rede disponíveis. Outros exemplos são o Mailman [2] e o Bittorrent [3].

Devido à sua flexibilidade, o Python é usado como linguagem de implementação para um número considerável de programas que usam a rede, bem como aplicativos de computação distribuída. Por isso, não é de se estranhar que o Python seja a linguagem preferida por muitos para implementar muitas das mais ousadas novidades no campo da conectividade.

Mas vamos começar pelo começo. Queremos trazer um recurso da rede para nossa máquina e, para isso, empregaremos uma URL [4] semelhante a www.seudomínio.com.br/imagem.jpg. Mas antes disso precisamos iniciar uma conexão com o servidor.

Para isso, o quente é usar a biblioteca *httplib*, que já vem com o Python. Essa biblioteca nos permite estabelecer uma conexão com um servidor HTTP e mandar comandos a ele nesse protocolo. Os comandos HTTP são simples, codificados em texto puro. De todos eles, o que nos interessa no momento é o comando `GET`. Quando acessamos um servidor HTTP, o que fazemos é, na realidade, solicitar que ele nos envie algum objeto. Esse pedido se faz mediante a emissão do comando `GET objeto`. Por exemplo, se queremos a página `index.html` do site www.python.org, primeiro nos conectamos ao servidor HTTP presente neste ende-



reço. Uma vez conectados, enviamos ao servidor o comando `GET index.html`. Nesse momento o servidor nos devolve, no mesmo canal, o conteúdo do arquivo `index.html`.

Dito assim parece ser coisa trivial, muito fácil, "melzinho na chupeta". Entretanto, é uma tarefa que, em outras linguagens de mais baixo nível, precisaria de uma grande quantidade de código, várias bibliotecas e rotinas de tratamento de erros. Em Python teremos muitíssimo menos dificuldades. Acompanhe a explanação a seguir e consulte a **listagem 6** para ver uma implementação real do que estamos dizendo.

A primeira coisa a fazer é importar a biblioteca `httplib`. Depois, criamos uma conexão ao servidor em questão e solicitamos o envio do arquivo `index.html`. Essa conexão gera uma resposta, formada por várias partes – entre elas um código numérico (como o famoso *404*), um texto que descreve o erro e uma conexão ao arquivo que pedimos. No caso de uma conexão bem-sucedida, o código numérico recebido é `200`, um `OK` e uma conexão com o arquivo. Uma vez com acesso ao arquivo, lemos seu conteúdo com a função `read()` e o armazenamos numa variável chamada `conteudo`. Por fim, podemos encerrar a conexão , fechando o arquivo remoto como se ele estivesse em nosso próprio disco rígido.

Nesse momento, temos a informação que queríamos na variável `conteudo` e já fechamos a conexão. Não é muito difícil, não é verdade?

Parâmetros

Estamos acostumados a poder passar parâmetros aos programas; em UNIX isso é bastante comum. Mas... como podemos obter os parâmetros de execução no Python? De novo, temos que recorrer a uma biblioteca. Mais precisamente, à biblioteca *sys*.

Ela nos proporciona acesso aos argumentos que foram passados ao programa na linha de comando. Todos eles ficam armazenados numa variável chamada `argv`. Essa variável é, na realidade, uma lista. Por isso mesmo, podemos obter os argumentos simplesmente acessando cada uma das posições dessa lista. A posição `0` contém o nome do programa que estamos executando. A partir da posição `1` encontraremos os parâmetros passados. Como é uma lista, podemos obter a quantidade de parâmetros com a função `len()`.

Programa

Com tudo isso na cachola, é hora de pôr a teoria para rebolar. Mãos à obra, então, para escrever um programa que pode ser útil. Nosso programa realizará as seguintes tarefas:

- Aceitar um parâmetro de entrada que informará o nome de um arquivo;
- O programa abrirá esse arquivo e o lerá, linha a linha. Cada linha do arquivo conterá uma URL que aponta para uma imagem;
- Cada URL será introduzida em uma lista para uso posterior;
- Uma vez que tenhamos acabado de ler o arquivo, o fecharemos e entraremos na segunda parte do programa;
- Criaremos um diretório com o nome do arquivo informado;
- Mudaremos o diretório de trabalho para esse diretório;
- Descarregaremos cada uma das URLs dentro do diretório;
- Geraremos um arquivo `index.html` que mostre as imagens.

"Argh! Muito trabalho!" Calma, estimado leitor. É para isso mesmo que os programas servem. Obviamente não faremos todos os testes que seriam necessários, já que o programa cresceria para além do didático. Fica, então, como tarefa para casa a implementação das melhorias necessárias. Pensemos, por ora, no algoritmo do programa. Temos várias partes para desenvolver:

Primeiro testaremos a existência e armazenaremos o nome do arquivo. A seguir, leremos as URLs. O passo seguinte é criar um diretório e mudar para ele. Depois, baixaremos todas as URLs. Por fim, geraremos o arquivo HTML.

Bem, ao trabalho! As URLs serão armazenadas em uma lista. Poderíamos usar objetos, mas essa é uma das coisas maravilhosas do Python: ***NÃO*** somos obrigados a usar objetos. Não queremos com isso dizer que os objetos são ruins, coisa do demo, nada disso. Mas em certas ocasiões os objetos podem ser melindrosos. Por exemplo, poderíamos criar um objeto `Lista_URLs` que aceitasse como parâmetro em seu construtor o nome de um arquivo e depois nos permitisse ir lendo as URLs uma após a outra. Também podemos fazer o mesmo com uma função que carregue todas as URLs em uma variável global. Vamos mostrar como fazer tudo isso com um objeto. Deixamos ao leitor a tarefa de explorar as possibilidades de substituir o objeto por uma variável global e as funções de lista.

Alguns avisos importantes: o programa só funciona com URLs de imagens e o arquivo que contém essas URLs deve terminar com uma linha em branco (ou seja, um ***[Enter]*** depois da última imagem).

Esse programa (que está na íntegra na **listagem 6**) é bastante simples e está bem comentado. Novamente, exortamos aos leitores que, uma vez compreendido o funcionamento do programa, tentem melhorá-lo e introduzir o tratamento de exceções. Boa sorte. ■

Informações

[1] Zope: www.zope.org

[2] Mailman: www.gnu.org/software/mailman

[3] BitTorrent: bittorrent.com

[4] Uniform Resource Locator: en.wikipedia.org/wiki/URL