Potência e simplicidade multiplataforma

Pau pra toda obra

O Python é uma linguagem potente, segura e flexível. Mas, ao contrário do que se poderia esperar, é simples e rápida de aprender. Como se não bastasse, praticamente não há nada que você possa fazer em outra linguagem que não possa com Python, de forma rápida e eficaz.

por José María Ruíz e José Pedro Orantes

Para começo de conversa, precisamos saber por que o Python desperta tanto interesse. A linguagem já é uma das mais famosas e é cada vez mais utilizada. Procurando na Internet, podemos encontrar uma quantidade enorme de programas escritos em Python que atestam a qualidade e os recursos dessa linguagem de altíssimo nível. Em primeiro lugar, o Python é uma linguagem *orientada a objetos*. Isso não significa uma prisão: podemos usar o Python de forma procedural (estruturada) e mesmo de forma linear, como os velhos programas em BASIC. Entretanto, depois de perdido o medo, usar sua implementação de OOP acaba ficando natural. Como se não bastasse, o interpretador Python é *livre e gratuito*, o que quer dizer que podemos fuçar em seu código fonte à vontade.

Por falar em código fonte, os programas em Python podem ser tanto interpretados como *scripts* – o que permite ver o código fonte dos programas – como podem ser pseudo-compilados em *Bytecode*, tal qual o Java. O Python também é muito *portátil*: podemos executar nossos programas em qualquer sistema operacional e arquitetura existentes, sem necessidade de modificar o código, desde que haja um interpretador Python instalado no computador.

Por fim, o fator *tempo*: desenvolver um programa em Python seguramente leva a metade do tempo (às vezes, *um quarto* do tempo!) que levaria para desenvolver o mesmo programa em C/C++ ou em Java – sem que o programa seja menos eficiente, muito pelo contrário. Não acredita? Então comparemos um programa "Olá, mundo" em Python com o mesmíssimo programa escrito em outras linguagens:

O programa *Olá mundo* escrito em C (suprimimos da listagem os vários *includes* necessários, o que faria o programa ainda mais complexo):

```
main ()
{
  printf("Olá, Mundo");
}
```

O mesmo *Olá Mundo* em Java:

```
public static void main(String args[])
{
System.out.println("Olá, Mundo");
}
```

E agora, nosso "complicadíssimo" *Olá Mundo* escrito em Python:

```
print "Olá, Mundo!"
```

Pensa que é tudo? O Python tem na manga outras características que o tornam a linguagem favorita de uma comunidade de desenvolvedores cada vez maior. Por exemplo, uma das coisas que a linguagem permite é a *declaração dinâmica de variáveis*: não precisamos declarar **nenhuma** variável de antemão e *não precisamos levar em conta o seu tamanho ou tipo* – o interpretador as aloca na memória de forma automática. O Python dispõe também de um *gerenciador de memória* que, de maneira semelhante ao *garbage collector* (coletor de lixo) do Java, se encarrega de liberar a memória ocupada pelos objetos que não estão mais sendo utilizados. Entretanto – e de forma similar ao Java – não podemos acessar diretamente as áreas de memória com instruções de baixo nível, coisa trivial de se fazer na linguagem C.

Se o Python em algum momento não conseguir resolver algum detalhe do problema, não é preciso refazer todo o programa em outra linguagem. Podemos combinar o Python com outras linguagens de programação e até mais de uma no mesmo programa. Podemos misturar rotinas em Java e Python (maçaroca chamada *Jython*), por exemplo. Ou ainda integrar Python com C/C++, usando o que cada uma tem de mais poderoso onde

for aplicável. O Python também conta com uma ampla biblioteca de módulos que, ao estilo das bibliotecas do C, simplificam e apressam o desenvolvimento de programas mais complexos.

A simplicidade do Python também contribui para que o código fonte dos nossos programas seja bastante sintético. Como pudemos ver na comparação, a simplicidade chega a ser assombrosa – e olhe que só escrevemos um exemplo simplório. Se esse exemplo já deixa patente que é possível economizar algumas linhas de código, podemos supor que, com tal simplicidade estrutural, um programa de 1000 linhas em Java poderia ser escrito com apenas 250 em Python. Muito bem. Mas como se usa essa coisa?

"Pysando" em ovos

Para começar, podemos fazer alguns testes interessantes. Primeiro, vamos conhecer o interpretador do Python. Para chamá-lo, basta abrir um terminal e digitar `python` (obviamente, seguido da tecla ***[Enter]***). É muito difícil que você tenha que instalar alguma coisa, já que praticamente todas as versões do Linux instalam o Python por padrão – até porque muitas delas usam o Python nas entranhas do próprio sistema operacional, em scripts e painéis de configuração. Se, por uma improvável ironia do destino, você não possuir o Python em sua máquina, procure nos CDs de instalação ou no site oficial de sua distribuição. Quando tudo o mais falhar, consulte o site oficial do Python em [1].

Uma vez dentro do interpretador, vamos usar uma variante "pop" do infame exemplo do "Olá, Mundo":

```
>>> print 'Ei, Beavis, estou programando!'
Ei, Beavis, estou programando!
```

Simples, não? A próxima coisa da cartilha de programação é testar algumas variáveis. Vamos a elas:

```
>>> soma = 15 + 16
>>>
>>> print 'O resultado da soma é: ', soma
O resultado da soma é: 31
```

Recomendamos ao leitor que brinque um pouco com o que vimos até aqui antes de alçar vôos mais altos. Usando o interpretador, é mais simples e rápido acostumar-se com a sintaxe dos comandos, já que o resultado é mostrado imediatamente na tela a cada teste.

Observemos agora uma propriedade interessante do Python. Os arquivos do Python – aliás, qualquer arquivo do Linux – não precisam ter nenhuma extensão para funcionar. O Linux diferencia os arquivos pelo seu conteúdo e não pela extensão. Entretanto, usaremos a extensão `.py` para nossos arquivos porque queremos que nós, humanos, os reconheçamos à primeira vista. Em algumas situações usaremos também a extensão `.pyw`.

Imaginemos que um de nossos exemplos seja chamado de `exemplo.py`. Para que possamos usá-lo como script, poderíamos simplesmente chamar o interpretador Python e passar como parâmetro o script a executar, desta maneira:

```
python exemplo.py
```

Mas podemos fazer com que ele se pareça com um programa independente. Para isso, basta indicar, dentro do arquivo que contém nosso código fonte, o caminho para o interpretador Python. No início da primeira linha do arquivo, coloque os caracteres `#!` e o caminho completo até o interpretador Pyhton. Em nosso caso, ficou assim: `#!/usr/bin/python`, na primeira linha do arquivo. Depois, será preciso dar permissão de execução para o arquivo: `chmod +x exemplo.py`. Desse modo, obtemos um programa pronto para rodar sem ajuda externa. Basta digitar num terminal o comando: `./exemplo.py`.

```
#! /usr/bin/python
print 'Oi de novo, Mundo!'
```

Depois dessa introdução toda, é hora de vermos como se parece um programa real em Python. Isso é importante porque, enquanto na maioria das linguagens o programador é livre para formatar os arquivos fonte da maneira como bem entender, em Python é obrigatório seguir uma certa metodologia. Uma das filosofias da linguagem Python é essa: "*Só há uma maneira de se fazer as coisas*" – ao contrário da linguagem Perl, cuja filosofia é, justamente, ter várias maneiras de se fazer a mesma coisa...

Comecemos com a coisa mais simples de se fazer em qualquer linguagem: atribuir um valor a uma variável.

```
quantidade = 16
```

A primeira coisa que se pode observar é que não é preciso terminar cada instrução com um ponto-e-vírgula (`;`) como em outras linguagens. Esta é uma das muitas características que fazem do Python um ambiente bastante diferente. Uma instrução acaba com o caractere de retorno de carro (em outras palavras, com o ***[Enter]***). Podemos usar o `;`, entretanto, para concatenar várias sentenças em uma mesma linha:

```
quantidade = 16; custo = 30
```

Como o Python é uma linguagem dinâmica, não é preciso declarar previamente o tipo da variável – basta usá-la e pronto. Mas uma vez definida, o que se faz na primeira vez em que ela recebe um valor, não é possível alterar mais o tipo da variável durante a execução do programa.

Temos à nossa disposição os operadores matemáticos de sempre (`+`, `-`, `*`, `/`, etc...) e, uma vez que saibamos como usá-los e como atribuir valores a variáveis, podemos começar a escrever

Listagem 1: Uma classe simples

```
>>> class Objeto:
    def __init__ (self, quantidade):
        self.quantidade = quantidade

    def getquantidade(self):
        return self.quantidade

    def setquantidade(self, quantidade):
        self.quantidade = quantidade
```

programas que, embora simples e lineares, já tenham alguma utilidade. Para que possamos fazer com que o programa tome decisões ou se comporte de maneira não-linear, precisamos conhecer os laços e os testes condicionais.

No Python, falar de laços merece uma explicação à parte. Quem já programa na linguagem C está acostumado com laços de contagem como este:

```
for (int a = 1; a < 10; a++)
    printf("%d\n",a);
```

No Python, entretanto, a sintaxe foi emprestada de outras linguagens como o Lisp ou o Haskell. Utilizamos uma função especial chamada `range`, que gera uma lista de números. Por exemplo:

```
range (1,10)
```

criará uma seqüência de números de 1 a 9. podemos usar o `range` para criar um laço similar ao `for` em Python da seguinte maneira:

```
for i in range (1,10)
```

Esse laço criaria iterações de 1 a 9 em `i`, intervalo fechado. O equivalente, em Python, ao comando de laço `while` da linguagem C é mais usual:

```
while (<condição>)
```

O mesmo ocorre com o `if`:

```
if (<condição>)
```

Por que não colocamos condições de exemplo nesses laços? Ahá! Porque o Python tem outra surpresa pra você. Em Python não se usam as famosas chaves `{` e `}` para isolar a condição do comando a repetir – como em C, Java e outras linguagens. Os criadores do Python decidiram – e nem todo mundo gosta disso – que se usaria a *posição* como delimitador. Sim, sabemos, isso soa muitíssimo estranho. Mas se observarmos bem, é na verdade muito mais fácil de entender:

```
>>> quantidade = 2
>>> for i in range (1,10):
    print quantidade*i
```

Comecemos dissecando esse `:`. Os dois pontos marcam o início de um bloco de código Python. Esse bloco aparecerá em laços, funções e métodos. Se observarmos a identação da linha seguinte, veremos uma série de espaços (inseridos com a tecla ***[Tab]***) e uma sentença. Esses espaços são vitais, já que marcam a existência de um bloco de código. Por isso mesmo, são obrigatórios.

Esse é uma das características mais controversas do Python, mas se pensarmos bem veremos que ela melhora muito a legibilidade do código. Só há uma maneira de escrever blocos de código em Python e, portanto, qualquer código em Python se parece com todos os outros programas dessa linguagem – o que torna tudo muito mais fácil de entender. O bloco acaba quando a identação retorna à posição anterior – ou seja, quando os espaços deixam de existir.

```
>>> for i in range (1,10):
    print quantidade*i
    quantidade = quantidade + 1
...
>>>
```

Pyrando na Batatynha

Já temos as peças fundamentais para entender as funções e os objetos. A declaração de uma função em Python tem uma sintaxe bastante simples:

```
def nome_da_função (<lista_de_argumentos>):
    <CÓDIGO>
```

Fácil, não? Da mesma forma que as variáveis, não é preciso definir os tipos para os argumentos. Existem muitas possibilidades no que tange aos argumentos de uma função, mas veremos os detalhes mais tarde. Por ora, vamos nos contentar com o exemplo abaixo, bastante simples:

```
>>> def imprime (texto):
    print texto
>>> imprime("Aí, mano, óia o Mundo!")
Aí, mano, óia o Mundo!
>>>
```

Como sempre, é simples. Mas e os objetos? Esses também são bem simples de implementar. Podemos ver um exemplo na **listagem 1**. Com `class` declaramos o nome da classe. Os comandos `def` em seu interior definem os métodos que pertencem a essa classe. O método `__init__` é o construtor, onde são atribuídos os valores iniciais às variáveis. O `__init__` é um método padrão e predefinido, donde resulta sermos obrigados a usá-lo – e nenhum outro – para inicializar o objeto.

Todos os métodos, embora não aceitem valores, possuem um parâmetro chamado `self`. Esse é outro daqueles pontos controversos do Python. O `self` é obrigatório, mas não é usado para chamar o método. Podemos ver um exemplo de classe na **listagem 1**. Mas como se instancia um objeto? Ora, é fácil: basta fazer o seguinte:

```
>>> a = Objeto(20)
```

Vê? É como chamar uma função. A partir desse momento, `a` será uma instância de `Objeto` e podemos usar seus métodos, como mostrado abaixo:

```
>>> print a.getQuantidade()
20
>>> a.setQuantidade(12)
>>> print a.getQuantidade()
12
```

Não há a menor necessidade de se preocupar com o uso de memória por parte dos objetos pois, quando `a` deixar de ser usado, o *gerenciador de memória* do Python liberará o núcleo usado pelo objeto.

Já temos as bases para construir algo interessante. Obviamente não desejamos uma infinidade de coisas abarrotando nossa bancada, mas sempre é melhor começar com um pequeno conjunto de ferramentas e saber como usá-las.

Listagem 2: Inserção e eliminação de elementos em uma lista

```
>>> b = [ 1 , 2 , 1 ]
>>> b.append(3)
>>> b.append(4)
>>> b
[1 , 2 , 1 , 3 , 4 ]
>>> b.remove(1)
>>> b
[2, 1, 3, 4]
>>> b.pop()
4
>>> b
[2, 1, 3]
>>> b.insert(1,57)
>>> b
[2, 57, 1, 3]
>>> b.append(1)
>>> b
[2, 57, 1, 3, 1]
>>> b.count(1)
2
>>> b.index(57)
1
>>> b.sort()
>>> b
[1, 1, 2, 3, 57]
>>> b.reverse()
>>> b
[57, 3, 2, 1, 1]
```

Pyntando o sete

Uma das razões para que os programas e scripts em Python sejam tão poderosos é a capacidade de manipular estruturas de dados de forma versátil e ainda assim simples. Em Python essas estruturas são as *listas* e os *dicionários* (também chamados de *tabelas hash*).

As *listas* nos permitem armazenar uma quantidade ilimitada de elementos de um mesmo tipo. O uso de listas é algo inerente a quase todos os problemas que podem ser resolvidos com um programa de computador; por isso, o Python já sai de fábrica com a capacidade de lidar com elas. As listas em Python também têm muito mais opções do que suas contrapartes em outras linguagens. Por exemplo, vamos definir uma lista que guarde uma série de palavras:

```
>>> a = ["Olinda", "Piracicaba", "Uruguaiana",➥
 "Vilhena"]
>>> a
['Olinda', 'Piracicaba', 'Uruguaiana',➥
'Vilhena']
```

O Python indexa numericamente os elementos, começando em zero. Isso quer dizer que `Olinda`, o primeiro item, é na verdade o elemento zero da lista, `Piracicaba` é 1, `Uruguaiana` é 2 e `Vilhena` é 3. O tamanho da lista é, portanto, 4. Podemos comprovar tudo isso assim:

```
>>> a[1]
'Piracicaba'
>>> len(a)
4
```

Podemos adicionar elementos às listas de inúmeras maneiras. Veremos, entretanto, apenas as mais usuais, como adicionar um elemento e eliminar outro (veja a **listagem 2**). As listas também podem se comportar como uma *pilha*, com operações típicas como `append` e `pop`. Com `insert` podemos introduzir um elemento na posição especificada (lembre-se de que sempre começamos a contar a partir de zero). A facilidade com que o Python trata as listas nos permite usá-las em uma infinidade de tarefas – o que vai simplificar bastante nosso trabalho.

Listagem 3: Exemplo de um dicionário

```
>>> dic = {}
>>> dic["Cachorro"] = "faz au au"
>>> dic["Gato"] = "faz miau miau"
>>> dic["Pintinho"] = "faz piu piu"
>>> dic
{'Cachorro': 'faz au au',
'Gato': 'faz miau miau',
'Pintinho': 'faz piu piu'}
>>> dic["Cachorro"]
'faz au au'
```

Apesar de seu poder, as listas não podem carregar o piano (*pyano?*) sozinhas. Existe outra estrutura que rivaliza com elas em utilidade: os *dicionários*. Enquanto as listas nos permitem referenciar um elemento usando um número, os dicionários permitem que o façamos com qualquer outro tipo de dado. Por exemplo, com cadeias de caracteres – ou seja, com palavras! Na verdade, usamos os dicionários quase que exclusivamente com palavras – talvez por isso seu nome seja dicionário (veja um exemplo na **listagem 3**).

É possível misturar listas e dicionários: dicionários de listas, listas de dicionários e dicionários de listas de dicionários etc. A combinação de ambas as estruturas de dados dá ao programador um poder de fogo inimaginável.

Vyajando na mayonese

Agora temos que pôr tudo isso em prática. O usual neste tipo de artigo é mostrar ao leitor um programa simples. Entretanto, em vez de subestimar o leitor vamos

estimular sua criatividade. O programa que mostraremos é um dos exemplos do livro "A prática da programação", de Pike e Kernighan, para ilustrar como um algoritmo bem feito sobrepuja a linguagem de programação que usamos para implementá-lo. No livro, o algoritmo é mostrado em C, C++, Java, Perl e AWK. Nós vamos implementá-lo em Python (veja a **listagem 4**).

O programa aceita um texto como entrada e gera um texto como saída, mas esse segundo texto não faz sentido algum. O que queremos é gerar um texto sem sentido, mas com estruturas gramaticais corretas. Pode parecer complicado, mas é na verdade bem simples se usarmos a técnica das *cadeias de Markov*. A idéia é mais ou menos a seguinte:

- 1. Comece pelas duas primeiras palavras do texto de entrada;
- 2. Verifique se essa combinação se repete no restante do texto;
- 3. Faça uma lista com palavras do mesmo tipo das que foram combinadas;
- 4. Escolha, ao acaso, uma palavra qualquer nessa lista;
- 5. Substitua a primeira palavra pela segunda e a segunda pela palavra escolhida na lista.

Dessa maneira, conseguimos criar um texto que, embora sem sentido algum, pareça "de verdade", estruturado corretamente segundo as regras do idioma. A mensagem é disparatada, mas quem é que se importa?

Para fazer os testes é recomendável empregar um texto bastante extenso – nosso programa não causará tanta celeuma em textos pequenos. No site do Projeto Gutenberg podemos conseguir uma infinidade de textos clássicos de grande tamanho e em formato de texto puro ASCII. Sabemos que nem todo mundo é conhecedor da língua de Chaucer, portanto selecionamos um texto que irá proporcionar horas de diversão: Os Lusíadas, de Luís de Camões [3]. Se preferir, pode usar qualquer texto que desejar, até mesmo notícias sobre política de um jornal online ou alguma crônica ferina de um blog da moda.

Esse programa é especialmente interessante e didático porque nos permitirá utilizar as estruturas de dados que o Python implementa, em particular os dicionários e as listas, para gerar uma tabela em que os índices serão palavras.

Vejamos como funciona. A primeira etapa é introduzir, no dicionário, os prefixos como se fossem índices. As palavras que usam esses prefixos seriam os elementos referenciados pelos índices. Eis um dicionário com duas palavras no índice e que contém uma lista:

```
DICIONARIO[ palavra 1,palavra 2] -> ➘
LISTA[palavra,...]
```

Ou seja, se temos as palavras "*Você tem fome... Você tem sede*" em nosso texto, criaremos um dicionário da forma mostrada a seguir:

```
>>> dict['Você', 'tem'] = ['fome']
>>> dict['Você', 'tem'].append('sede')
>>> dict['Você', 'tem'] ['fome','sede']
```

Fazendo isso de maneira sucessiva, com todos os conjuntos de duas palavras adjacentes, obteremos um dicionário em que muitas entradas referenciarão uma lista de mais de um elemento – quanto mais elementos, melhor.

A mágica aparece quando geramos o texto, pois começamos por imprimir na tela as duas primeiras palavras. Quando existir mais de uma possibilidade para essa combinação (como no exemplo com "Você" e "tem"), escolheremos aleatoriamente entre elas (em nosso exemplo havia apenas duas palavras a escolher, "fome" e "sede", mas numa situação real teríamos muitas mais). Imaginemos que o programa escolha a palavra "sede" e a escreva na tela. Depois disso, o programa vai buscar no dicionário pela lista

Listagem 4: O markov.py gera um texto não tão aleatório assim

```
01 #!/usr/local/bin/python
02 # Altere a linha acima de acordo com
03 # o seu sistema. Em muitas distribuições
04 # o Python fica em /usr/bin/python
05
06 # Antes de mais nada
07 # importamos dois módulos:
08
09 import random
10 # Utilizado para escolher um elemento
11 # aleatório de uma lista.
12
13 import sys
14 # Permite acessar algumas das
15 # classes usadas pelo interpretador
16
17 fim_das_palavras = "\n"
18 w1 = fim_das_palavras
19 w2 = fim_das_palavras
20
21 # Geração do dicionário
22 dict = {}
23
24 for linha in sys.stdin:
25     for palavra in linha.split():
26         dict.setdefault( (w1, w2), [] ).append(palavra)
27         w1 = w2
28         w2 = palavra
29
30 # Fim de arquivo
31 # (Caracter EOF)
32 dict.setdefault((w1, w2), [] ).append(fim_das_palavras)
33
34 # Geração da saída
35 w1 = fim_das_palavras
36 w2 = fim_das_palavras
37
38 # Modifique aqui para
39 # alterar o número máximo
40 # de palavras
41 max_palavras = 10000
42
43 for i in xrange(max_palavras):
44     nova_palavra = random.choice(dict[(w1, w2)])
45
46     if nova_palavra == fim_das_palavras:
47         sys.exit()
48
49     print nova_palavra;
50
51     w1 = w2
52     w2 = nova_palavra
```

que é referenciada pela dupla de palavras `['tem','fome']` e assim sucessivamente até chegar ao indicador de `fim_das_palavras`. Para entender melhor o funcionamento do programa, recomendamos que digite o código fonte, rode o programa e insira nele alguns arquivos de texto (por exemplo, `cat texto.txt | ./markov.py`). Os resultados são hilariantes.

Depois de brincar bastante, podemos nos aventurar a modificar um pouco o programa. Uma sugestão é, em vez de uma dupla de palavras, tentar usar como índices uma única palavra ou mesmo três delas. Também tente variar o tamanho do texto (um pequeno, um médio e um grande) para ver a eficiência do algoritmo em relação ao universo de palavras apresentado. Com um pouco de experimentação é possível conseguir resultados bastante interessantes. ■

Informações

[1] Download do Python: www.python.org/download

[2] Jython - Python em Java: www.jython.org

[3] Os Lusíadas no projeto Gutenberg: www.gutenberg.org/etext/3333

[4] Python Brasil: www.pythonbrasil.com.br

[6] Livros sobre Python: wiki.python.org/moin/PythonBooks

Sobre o autor

José María Ruíz está terminando seu Projeto de Conclusão de Curso na Faculdade de Engenharia Técnica em Informática e Sistemas. Já há sete anos usa e desenvolve Software Livre, desde os velhos tempos da telinha preta do MS-DOS até o moderno FreeBSD.

José Pedro Orantes está cursando o 3º ano de Engenharia Técnica em Informática e Sistemas e, simultaneamente, o 3º ano de Engenharia Técnica em Gestão de Informática. Usa o Linux há seis anos no computador de trabalho, tanto para trabalhos de escritório como para desenvolvimento.