

Python: Potencia y sencillez multiplataforma

Lenguaje Todoterreno



Python es un lenguaje potente, seguro, flexible... pero sobre todo sencillo y rápido de aprender, que nos permite crear todo lo que necesitamos en nuestras aplicaciones, de forma rápida y eficaz.

POR JOSÉ MARÍA RUÍZ Y JOSÉ PEDRO ORANTES

Para empezar, debemos saber por qué Python es interesante, por qué es tan famoso y cada vez es más utilizado. Mirando un poco por Internet, se pueden encontrar multitud de aplicaciones que nos muestran un poco las capacidades de este lenguaje de alto nivel. Primero, Python es un lenguaje **orientado a objetos**, esto no significa que lo sea exclusivamente, podemos utilizarlo como queramos, aunque es más natural utilizar su implementación de la OOP. Además, es **libre y gratuito** pudiendo descargar el intérprete y su código fuente. Nos permite programarlo como **script**, lo que posibilita ver el código fuente de aquellas aplicaciones que así

estén desarrolladas, o bien podremos compilarlo obteniendo un *Bytecode* al igual que Java. Es **portable** y nos permite ejecutar nuestros programas en cualquier sistema operativo y/o arquitectura teniendo previamente el intérprete instalado en nuestro ordenador. Realizar un programa bajo este lenguaje, seguramente nos costaría entre la mitad o la cuarta parte del tiempo que tardaríamos en desarrollar el mismo programa en C/C++ o Java esto hace que sea muy **potente**. Veamos una breve comparativa con otros lenguajes:

Hola Mundo en C.

```
main ()
```

```
{
    printf("Hola Mundo");
}
```

Hola Mundo en Java

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

Hola Mundo en Python

```
print "Hola Mundo"
```

Python dispone de otras características que lo convierten en el lenguaje favorito de una comunidad de desarrolladores cada vez más amplia. Por ejemplo, permite la *declaración dinámica de variables*, es decir, no tenemos que declarar las variables y no tenemos que tener en cuenta su tamaño, ya que son completamente dinámicas. Dispone también de un *gestor de memoria* que, de manera similar al recolector de basuras de Java, se encargará de liberar la memoria de objetos no utilizados. Sin embargo, y al igual que Java, no nos permite referirnos a zonas de memoria a bajo nivel, como puede hacerse con C.

Además se puede combinar con otros múltiples lenguajes de programación. Podemos mezclar en nuestras aplicaciones Python y Java (Jython), por ejemplo. O Python con C/C++, lo cual hace que resulte más potente si cabe. Python también cuenta con una amplia biblioteca de módulos que, al estilo de las bibliotecas en C, permiten un desarrollo rápido y eficiente.

La sencillez de Python también ayuda a que los programas escritos en este lenguaje sean muy sintéticos. Como podemos ver en la comparativa anterior, la simplicidad llega a ser asombrosa y solo hemos escrito un simple ejemplo. Si este programa ya supone ahorrarte unas

líneas de código, con una sintaxis tan sencilla y ordenada podemos imaginar que un programa de 1000 líneas en Java, en Python podrían ser alrededor de 250. Muy bien, ¿pero como se usa?

Uso

Para empezar a matar el gusanillo, podemos ir haciendo algunas pruebas interesantes. Vayamos al intérprete Python. Para esto, basta con escribir 'python' en cualquier shell en cualquier versión de Linux que tenga instalado el intérprete (si no lo tienes instalado, hazte con la última versión en [1]). Entonces podemos probar el "Hello World":

```
>>> print 'Hello World'
Hello World
```

Simple, ¿verdad?, ahora podemos probar a utilizar algunas variables:

```
>>> suma = 15 + 16
>>>
>>> print 'el resultado de la
suma es: ', suma
el resultado de la suma es: 31
```

Es recomendable trastear un poco con esto, antes de ponernos a programar algo más complicado, ya que de esta manera es más sencillo hacerse con la sintaxis mucho más rápidamente viendo los resultados de cada prueba.

Veamos ahora alguna propiedad interesante de Python. Los ficheros en Python, no tienen por que llevar extensión ninguna aunque seguramente queremos tenerlos diferenciados del resto de nuestros ficheros. Para ello se utiliza la extensión `.py` o bien `.pyw`. Imaginemos que tenemos un ejemplo `ejemplo.py`, al permitirnos usarlo como script, podemos indicarle la ruta del intérprete en el inicio (`#!/usr/bin/python` en nuestro caso) y dándole permisos de ejecución (en este caso, `chmod +x ejemplo.py`) obtenemos un programa listo para correr `./ejemplo.py`.

```
#!/usr/bin/python
print 'Hello World'<C>
```

Después de toda la introducción técnica va siendo hora de que veamos como es el formato de los programas en Python.

Esto es importante porque, mientras la norma en la mayoría de los lenguajes es dejar al programador la decisión de la manera en que deben ser formateados los archivos fuente, en Python es obligatorio seguir una metodología. En Python todo se hace de una sola manera, es parte de su filosofía: *Solo Hay Una Manera de Hacer Las Cosas*. Comencemos con lo más simple en todo lenguaje, la asignación a una variable:

```
cantidad = 166.386
```

Lo primero que hay que apreciar es que no se usa el `;`. Esto es una característica de las muchas que hacen a Python diferente. Una instrucción acaba con el retorno de carro, aunque el `;` se puede utilizar cuando tenemos dos sentencias están en la misma línea:

```
cantidad = 166.386; pesetas
= 3000
```

Como es un lenguaje dinámico no hay que declarar el tipo de las variables, pero una vez han sido definidas, lo que se hace asignándoles un valor, éstas guardan su tipo y no lo cambiarán a lo largo de la ejecución del programa. Tenemos a nuestra disposición los operadores de siempre (`+`, `-`, `*`, `/`, etc) y una vez que sabemos como manejarlos y como asignar las variables, podemos realizar algo útil, pero lineal. Para que la ejecución no sea lineal necesitamos los bucles y los condicionales.

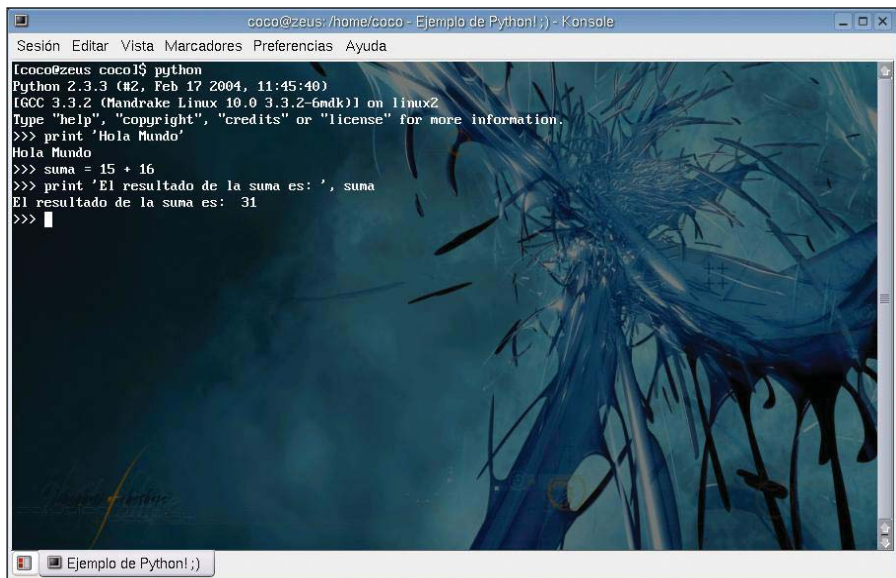


Figura 1: Utilización de Python como una calculadora

En lo que se refiere al tema de los bucles en Python es algo especial. Puede que estemos acostumbrados a los bucles tipo C:

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

Sin embargo, en Python, se toma un enfoque funcional prestado de otros lenguajes como Lisp o Haskell. Se utiliza una función especial llamada `range` que genera una lista de números:

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

generará una ristra de números del 1 al 9. De manera que podemos utilizar una función `range` para crear un bucle `for` en Python de la siguiente manera:

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

Este bucle iteraría con `i` desde 1 a 9, ambos inclusive. La versión del bucle `while` en Python es más normal...

```
while(<I><condición><I>)
```

al igual que `if...`

```
if (<I><condición><I>)
```

¿Por qué no hay puestos cuerpos de ejemplo en esos bucles? Pues porque ahora viene otra novedad. En Python no se usan las famosas `{ y }` de C, Java y otros lenguajes. Se decidió, y no a todo

el mundo le gusta, que se usaría la posición como delimitador. Esto así suena algo extraño, pero si se ve es mucho más sencillo:

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

Comencemos mirando a ese `∴`. Los dos puntos marcan el inicio de un bloque de código Python. Ese bloque aparecerá en bucles, funciones o métodos. Si nos fijamos bien en la sangría de la siguiente línea, vemos una serie de espacios (logrados pulsando la tecla TABULADOR) y una sentencia. Esos espacios son vitales ya que marcan la existencia de un bloque de código. Además, son obligatorios.

Este es uno de hechos más controvertidos de Python, pero también mejora mucho la legibilidad del código. Sólo existe una manera de escribir Python así que todo el código Python se parece y es más fácil de entender. El bloque acaba cuando desaparecen esos espacios:

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

Funciones y Objetos

Tenemos ya las piezas fundamentales para entender las funciones y los objetos. La declaración de una función en Python tiene una sintaxis muy simple:

```
def nombre_funcion (<lista de argumentos>): <CUERPO>
```

Fácil ¿no? Al igual que las variables, a

Listado 1: Una clase sencilla

```
01 >>> class Objeto:
02     def __init__(self, cantidad):
03         self.cantidad =
04             cantidad
05     def getCantidad(self):
06         return self.cantidad
07
08     def setCantidad(self,
09         cantidad):
10         self.cantidad =
11             cantidad
```

Listado 2: Adición y eliminación de elementos de una lista

```
01 >>> b = [ 1 , 2 , 1 ]
02 >>> b.append(3)
03 >>> b.append(4)
04 >>> b
05 [ 1 , 2 , 1 , 3 , 4 ]
06 >>> b.remove(1)
07 >>> b
08 [2, 1, 3, 4]
09 >>> b.pop()
10 4
11 >>> b
12 [2, 1, 3]
13 >>> b.insert(1,57)
14 >>> b
15 [2, 57, 1, 3]
16 >>> b.append(1)
17 >>> b
18 [2, 57, 1, 3, 1]
19 >>> b.count(1)
20 2
21 >>> b.index(57)
22 1
23 >>> b.sort()
24 >>> b
25 [1, 1, 2, 3, 57]
26 >>> b.reverse()
27 >>> b
28 [57, 3, 2, 1, 1]
```

los argumentos no se les asignan tipo. Existen muchas posibilidades en los argumentos pero los veremos más tarde, de momento veamos un ejemplo sencillo:

```
>>> def imprime (texto):
    print texto
>>> imprime("Hola mundo")
Hola mundo
>>>
```

Vuelve a ser sencillo. ¿Y los objetos? Pues también son bastante simples de implementar. Podemos ver un ejemplo en el Listado 1. Con `class` declaramos el nombre de la clase, y los `def` de su interior son los métodos. El método `__init__` es el constructor, donde se asignan los valores iniciales a las variables. `__init__` es un método estándar y predefinido, lo que quiere decir que tendremos que usar ese y no otro para inicializar el objeto.

Todos los métodos, aunque no acepten valores, poseen un parámetro `self`. Éste es otro punto controvertido en Python, `self` es obligatorio, pero no se usa al invocar el método. Se puede ver un ejemplo de clase en el Listado 1. ¿Cómo se crea el objeto? Fácil:

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

Es como llamar a una función. A partir de este momento `a` es una instancia de

Objeto y podemos utilizar sus métodos:

```
>>> print a.getCantidad()
20
>>> a.setCantidad(12)
>>> print a.getCantidad()
12
```

No hay que preocuparse por la administración de la memoria del objeto ya que, cuando `a` no apunte hacia él, el *gestor de memoria* liberará su memoria. Ya tenemos las bases para construir algo interesante. Por supuesto nos dejamos infinidad de cosas en el tintero, pero siempre es mejor comenzar con un pequeño conjunto de herramientas para empezar a usarlas.

Estructuras de datos

Una de las razones por las que los pro-

Listado 3: Ejemplo de un diccionario

```
01 >>> dic = {}
02 >>> dic["Perro"] = "hace guau
03     guau"
04 >>> dic["Gato"] = "hace miau
05     miau"
06 >>> dic["Pollito"] = "hace pio
07     pio"
08 >>> dic
09 {'Perro': 'hace guau guau',
10  'Gato': 'hace miau miau',
11  'Pollito': 'hace pio pio'}
```

gramas scripts de Python resultan tan potentes es que nos permiten manejar estructuras de datos muy versátiles de manera muy sencilla. En Python estas estructuras son las *Listas* y los *Diccionarios* (también llamados Tablas Hash).

Las *listas* nos permiten almacenar una cantidad ilimitada de elementos del mismo tipo. Esto es algo inherente a casi todos los programas, así que Python las incorpora de fábrica. Las listas de Python también vienen provistas de muchas más opciones que sus semejantes en otros lenguajes. Por ejemplo, vamos a definir una lista que guarde una serie de palabras:

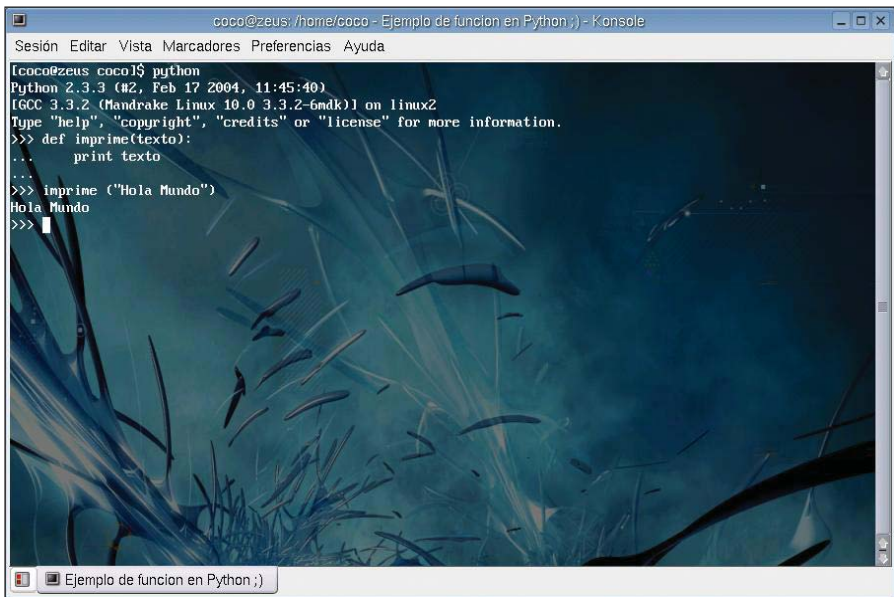
```
>>> a = ["Hola", "Adios",
" Buenas Tardes"]
>>> a
['Hola', 'Adios', ' Buenas
Tardes']
```

Python indexa comenzando desde 0, de manera que 'Hola' es el elemento 0, 'Adios' el 1 y 'Buenas Tardes' el 2, y la longitud de la lista es 3. Podemos comprobarlo de esta forma:

```
>>> a[1]
'Adios'
>>> len(a)
3
```

Podemos añadir elementos a las listas de varias maneras. Nos contentaremos con ver las más normales. Añadamos un elemento y eliminemos otro (ver Listado 2). Las listas también se pueden comportar como una *Pila*, con las operaciones *append* y *pop*. Con *insert* hemos introducido un elemento en la posición especificada (recuerda que siempre comenzamos a contar desde 0). La facilidad con la que Python trata las listas nos permite usarlas para multitud de tareas lo que simplificará mucho nuestro trabajo.

A pesar de su potencia, las listas no pueden hacerlo todo y existe otra estructura que rivaliza con ellas en utilidad, los *Diccionarios*. Mientras las listas nos permiten referenciar a un elemento usando un número, los diccionarios nos permiten hacerlo con cualquier otro tipo de dato. Por ejemplo, con cadenas, bueno más bien casi siempre con cadenas, de



```
coco@zeus:/home/coco - Ejemplo de funcion en Python :) - Konsole
Sesión Editar Vista Marcadores Preferencias Ayuda
[coco@zeus coco]$ python
Python 2.3.3 (#2, Feb 17 2004, 11:45:40)
IGCC 3.3.2 (Mandrake Linux 10.0 3.3.2-6mdk) on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> def imprime(texto):
...     print texto
...
>>> imprime("Hola Mundo")
Hola Mundo
>>>
```

Figura 2: Definición de una función en Python

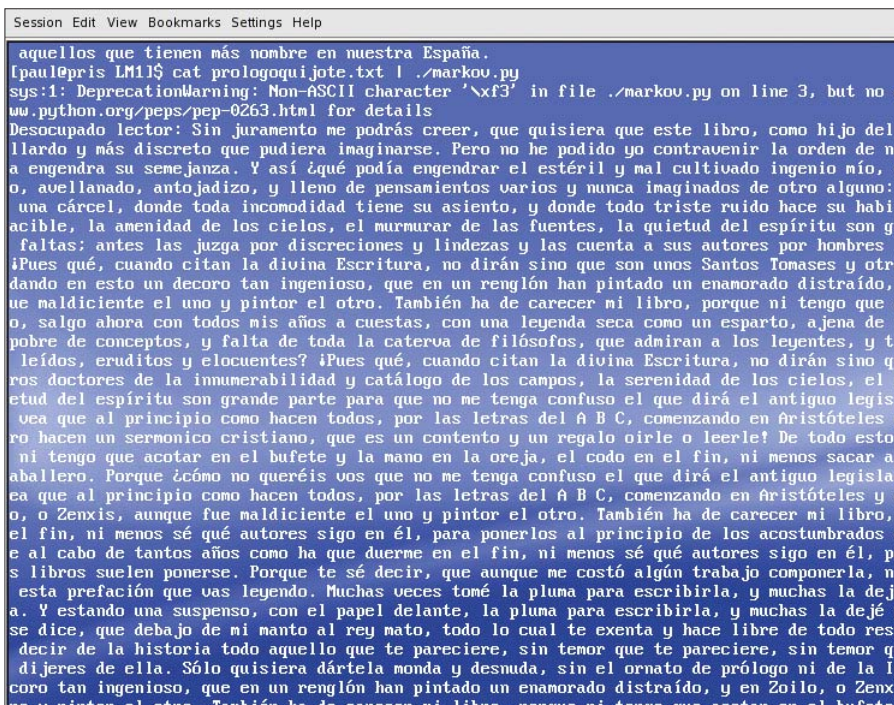
ahí que su nombre sea diccionario (véase el Listado 3). Las listas y los diccionarios se pueden mezclar: diccionarios de listas, listas de diccionarios, diccionarios de listas de diccionarios, etc. Ambas estructuras combinadas poseen una enorme potencia.

Algoritmos + Estructuras de datos = Programas

Ahora nos toca poner todo esto en práctica. Lo normal es hacer un programa sencillo. Pero en lugar de eso vamos a

implementar algo que sea creativo. Este programa es el que se usa en el libro "La práctica de la programación" de Pike y Kernighan para ilustrar como un buen diseño sobrepasa al lenguaje que usemos para ejecutarlo. En el libro se implementa el diseño en C, C++, Java, Perl y AWK. Nosotros lo haremos en Python (ver Listado 4).

El programa acepta un texto como entrada y genera un texto como salida, pero este segundo texto no tiene sentido. Lo que queremos hacer es generar texto



```
Session Edit View Bookmarks Settings Help
aquellos que tienen más nombre en nuestra España.
[paul@pris LM1]$ cat prologoquijote.txt | ./markov.py
sys:1: DeprecationWarning: Non-ASCII character '\xf3' in file ./markov.py on line 3, but no
www.python.org/peps/pep-0263.html for details
Desocupado lector: Sin juramento me podrás creer, que quisiera que este libro, como hijo del
llardo y más discreto que pudiera imaginarse. Pero no he podido yo contravenir la orden de n
a engendra su semejanza. Y así ¿qué podía engendrar el estéril y mal cultivado ingenio mío,
o, avellanado, antojadizo, y lleno de pensamientos varios y nunca imaginados de otro alguno:
una cárcel, donde toda incomodidad tiene su asiento, y donde todo triste ruido hace su habi
acible, la amenidad de los cielos, el murmurar de las fuentes, la quietud del espíritu son g
faltas; antes las juzga por discreciones y lindezas y las cuenta a sus autores por hombres
¿Pues qué, cuando citan la divina Escritura, no dirán sino que son unos Santos Tomases y otr
dando en esto un decoro tan ingenioso, que en un renglón han pintado un enamorado distraído,
ue maldiciente el uno y pintor el otro. También ha de carecer mi libro, porque ni tengo que
o, salgo ahora con todos mis años a cuestras, con una leyenda seca como un esparto, ajena de
pobre de conceptos, y falta de toda la cetera de filósofos, que admiran a los legentes, y t
leídos, eruditos y elocuentes? ¿Pues qué, cuando citan la divina Escritura, no dirán sino q
ros doctores de la innumerabilidad y catálogo de los campos, la serenidad de los cielos, el
tud del espíritu son grande parte para que no me tenga confuso el que dirá el antiguo legis
vea que al principio como hacen todos, por las letras del A B C, comenzando en Aristóteles
ro hacen un sermonico cristiano, que es un contento y un regalo oírle o leerle? De todo esto
ni tengo que acotar en el bufete y la mano en la oreja, el codo en el fin, ni menos sacar a
aballero. Porque ¿cómo no queréis vos que no me tenga confuso el que dirá el antiguo legisla
ea que al principio como hacen todos, por las letras del A B C, comenzando en Aristóteles y
o, o Zenxis, aunque fue maldiciente el uno y pintor el otro. También ha de carecer ni libro,
el fin, ni menos sé qué autores sigo en él, para ponerlos al principio de los acostumbrados
e al cabo de tantos años como ha que duerme en el fin, ni menos sé qué autores sigo en él, p
s libros suelen ponerse. Porque te sé decir, que aunque me costó algún trabajo componerla, n
esta prefación que vas leyendo. Muchas veces tomé la pluma para escribirla, y muchas la dejé
a. Y estando una suspenso, con el papel delante, la pluma para escribirla, y muchas la dejé
se dice, que debajo de mi manto al rey mato, todo lo cual te exenta y hace libre de todo res
decir de la historia todo aquello que te pareciere, sin temor que te pareciere, sin temor q
dijeres de ella. Sólo quisiera dártela munda y desnuda, sin el ornato de prólogo ni de la l
coro tan ingenioso, que en un renglón han pintado un enamorado distraído, y en Zoilo, o Zenz
ro y pintor el otro. También ha de carecer el libro, porque ni tengo que acotar en el bufete
```

Figura 3: El programa de ejemplo aplicado al prólogo de El Quijote

sin sentido pero con estructuras que sí lo tengan. Puede parecer algo muy complicado, pero no lo es tanto si usamos la técnica de *las cadenas de Markov*. La idea es comenzar con las dos primeras palabras, ver si esa combinación se repite en el texto, hacer una lista con las palabras que siguen a alguna de las ocurrencias de esa combinación, elegir una palabra al azar de las que las suceda y reemplazar la primera por la segunda y la segunda por la palabra escogida. De esta manera vamos generando un texto que, aunque carece de sentido, normalmente se corresponde con la estructura de un texto normal aunque disparatado.

Para hacer las pruebas es recomendable conseguir un texto de gran tamaño, en textos pequeños no surtirá tanto efecto nuestro programa. En el proyecto Gutenberg podemos conseguir infinidad de textos clásicos de enorme tamaño en ASCII. Pero somos conocedores de que no todo el mundo entiende el idioma anglosajón, así que en lugar del proyecto Gutenberg podemos coger cualquier texto que queramos modificar, por ejem-

plo alguna noticia de política de un diario digital o alguna parrafada de cualquier weblog de moda. En nuestro caso, hemos escogido parte de la introducción a la obra maestra de la Literatura española, “El Ingenioso Hidalgo, Don quijote de la Mancha”.

Este programa es especialmente interesante y didáctico porque nos permitirá utilizar las estructuras de datos que Python implementa, en particular en los diccionarios y las listas, para generar una tabla donde los índices serán cadenas de texto.

Veamos como funciona. Lo primero es ir introduciendo en el diccionario dos prefijos como índice y las palabras que les siguen en el texto dentro de una lista referenciada por ellos. Eso es un diccionario con dos palabras como índice que contiene una lista:

```
DICCIONARIO[ palabra 1, palabra 2 ] -> LISTA[palabra,...]
```

O sea, si tenemos las palabras “*La gente está ... La gente opina*” en nuestro texto,

crearemos un diccionario de la siguiente forma:

```
>>> dict['La', 'gente'] = ['está']
>>> dict['La', 'gente'].append('opina')
>>> dict['La', 'gente']
['está', 'opina']
```

Vamos haciendo esto de manera sucesiva con todos los conjuntos de dos palabras adyacentes y obtendremos un diccionario en el que muchas entradas referenciarán a una lista de más de un elemento, cuantas más haya mejor.

La magia aparece cuando generamos el texto, puesto que lo que hacemos es comenzar por imprimir las dos primeras palabras y cuando existan varias posibilidades para esa combinación (como con el ejemplo de ‘La’ y ‘gente’), escogemos aleatoriamente entre ellas. Imaginemos que escogemos ‘opina’, entonces escribimos ‘opina’ por la pantalla y buscamos en la entrada de diccionario [‘gente’, ‘opina’] y así sucesivamente, hasta llegar a *no_palabra*. Para entender mejor el funcionamiento del programa, te recomendamos que copies el código fuente y le pases unos ejemplos (por ejemplo con *cat texto.txt | ./markov.py*) para que veas los resultados. Después podemos intentar realizar cambios en el programa, por ejemplo, en lugar de utilizar 2 palabras como índice del diccionario podemos probar con 1 o con 3, y también con el tamaño del texto que se le pase. Con un poco de experimentación es posible conseguir resultados muy interesantes. ■

Listado 4: markov.py genera un texto no-tan-aleatorio

```
01 #!/usr/local/bin/python          23         w1 = w2
02                                24         w2 = palabra
03 #Importamos dos módulos         25
04                                26 # Fin de archivo
05 import random                   27 dict.setdefault((w1, w2), [])
06 #utilizado para escoger un      28         ).append(no_palabra)
   elementos                       29 # GENERAMOS LA SALIDA
07 #aleatorio de una lista         30 w1 = no_palabra
08                                31 w2 = no_palabra
09 import sys                       32
10 #permite acceder a algunas      33 # puedes modificarlo
   clases                           34 max_palabras = 10000
11 #utilizadas por el intérprete   35
12                                36 for i in xrange(max_palabras):
13 no_palabra = "\n"              37     nueva_palabra =
14 w1 = no_palabra                 38     random.choice(dict[(w1, w2)])
15 w2 = no_palabra                39
16                                40     if nueva_palabra ==
17 # GENERAMOS EL DICCIONARIO      41     no_palabra:
18 dict = {}                       42         sys.exit()
19                                43
20 for linea in sys.stdin:         44     print nueva_palabra;
21     for palabra in              45
   linea.split():                  46     w1 = w2
22         dict.setdefault( (w1,   47     w2 = nueva_palabra
```

RECURSOS

[1] Descargas de Python
<http://www.python.org/download/>

LOS AUTORES

José María Ruíz está realizando el Proyecto Fin de Carrera de Ingeniería Técnica en informática de Sistemas y lleva 7 años usando y desarrollando software libre, y desde hace dos en FreeBSD. José Pedro Orantes está cursando 3º de Ingeniería Técnica en Informática de Sistemas y 3º de Ingeniería Técnica en Informática de Gestión, lleva mas de seis años utilizando Linux como escritorio y herramienta de trabajo.