

La revolución se acerca al mundo de Python

PYTHON 3000

Aurelio www.exe.hu

Guido van Rossum tenía una espinita clavada en el corazón. Sabía desde hacía tiempo que Python poseía una serie de defectos fatales. Python 3000 es el resultado de años de desarrollo corrigiendo estos defectos.

POR JOSÉ MARÍA RUÍZ

Python es un lenguaje que ya carga sobre sus espaldas más de 15 años de desarrollo. Y ha sido un lenguaje exitoso, lo que conlleva el, muchas veces, odioso problema de la compatibilidad hacia atrás. Todos los programadores quieren que sus programas funcionen con las últimas versiones de Python, aunque los escribiesen hace 10 años. Este problema es muy complicado de solucionar, aunque Python ha evolucionado lentamente para ir permitiendo a los programadores adaptarse a los cambios sin muchos inconvenientes.

Pero Guido, el creador del lenguaje, ya sabía desde hace años que Python poseía una serie de problemas que requerían soluciones drásticas. Soluciones drásticas que implican cambios drásticos que harán que muchas programas Python dejen de funcionar. Por tanto, se estableció un proyecto con un nombre un poco futurista: Python 3000 o Python 3k. Su objetivo no era otro que partir de una mentalidad nueva en la que el problema de la compatibilidad entre versiones sea ignorado.

Python 3000, ver Recursos [1] y [2], será el futuro Python 3.0, que supuestamente será liberado en octubre de 2008. Implica un cambio en muchos de los conceptos fundamentales del lenguaje y la forma de programar. Por ello se liberará a la vez que Python 2.6, la siguiente versión del Python «tradicional». Aún así Python

2.6 comenzará a incorporar muchas de las novedades de Python 3.0 de forma opcional, de manera que los programadores puedan ir adaptando sus programas progresivamente en lugar de tener que reescribirlos por completo para estar a la última.

Un Brillante `__future__`

Python lleva años preparándose para este tipo de cambios dramáticos. Por ello, desde hace ya tiempo, las distintas versiones de Python traen consigo una librería un poco especial: `__future__`.

`__future__` («futuro» en español) carga en su interior algunas novedades que se espera vengan integradas en la siguiente versión de Python. Así es posible usar la sentencia (un poco llamativa):

```
from __future__ import *
```

que nos permite «traer cosas del futuro» para usarlas en el presente. Por ejemplo, Python 2.6 traerá consigo la nueva expresión *with* (directamente heredada de Lisp, ¡un hurra por Python!) que simplifica mucho nuestra vida, como ya veremos más adelante.

Por lo tanto, no debe cundir el pánico, podemos dejar nuestros programas exactamente como están ahora mismo y usar, cuando sea liberado, Python 2.6. Si quere-

mos ir haciendo pruebas siempre podemos usar la librería `__future__`.

Python se Pone Serio

¿Acaso Python no era un lenguaje de programación serio? Bueno, lo era y lo es. Pero a pesar de ello adolecía de una serie de problemas o decisiones de diseño que hacían que su uso en la gran empresa se mirase con cierto recelo.

En general, el mundo de Python ha estado bastante alejado de los debates teóricos sobre la programación orientada a objetos. Python también ha ido cargando un gran número de librerías de dudoso uso a día de hoy que han lastrado su librería estándar durante años. Por último, existen aún pequeñas espinas que hacen que el desarrollo con Python no sea tan fluido o simple como en otros lenguajes (y la simplicidad es una meta siempre buscada por la comunidad Python). Vamos a ir desgranando estos cambios en lo que sigue.

2to3

A nadie le gustan los traumas, y Python 3000 puede provocar más de uno. Consientes, como son, de que su comunidad no desea este tipo de giros bruscos en el timón del lenguaje, Guido y compañía han preparado una herramienta llamada *2to3* capaz de traducir código Python 2.x a Python 3000. Es decir, podemos pasar a

esta herramienta un programa Python 2.x y obtendremos un programa que funcionará con Python 3000.

El cambio Más Simple... y Más Complicado

Existe un cambio realmente pequeño y ridículo, pero el responsable de la inmensa mayoría de los problemas que Python 3k generará en los programas que se crearon con versiones anteriores de Python: *print* pasa de ser una expresión a ser una función de una librería. Pero ¿qué significa exactamente esto?

La expresión *print* nos permite escribir en pantalla. Es simple: *print* "Hola mundo". ¿Qué hay de malo en ella? Pues la respuesta también es simple, este diseño hace que sea un quebradero de cabeza para los programadores que crean librerías.

Digamos que estamos creando una librería que quiere sobrescribir funciones de *print*, quiere que *print* se comporte de otra manera. Hasta ahora podíamos cambiar hacia dónde iban los datos que *print* enviaba, redirigiendo las salidas estándar y de error. Pero, ¿qué ocurre si lo que quiero hacer es cambiar su comportamiento? Digamos que deseo poder colocar un decorador que permita que lo que escriba *print* se mande también a una impresora. A día de hoy es imposible hacerlo con Python 2.x.

Por ello *print* pierde su categoría de expresión del lenguaje, deja de formar parte del lenguaje Python, y pasa a ser una función de la librería estándar, como cualquier otra. Ahora podremos reemplazarla, o incluso guardarla:

```
>>> p = print
>>> p("Hola mundo")
Hola mundo
```

¡Ridículo verdad? ¡Pues hasta ahora era imposible hacerlo! Evidentemente *print*, al ser una función, requiere que sus argumentos estén entre dos paréntesis, lo que a muchos no les va a gustar demasiado. Elimina la inmediatez de poder escribir: *print* "Hola mundo".

Pero Python 3000 nos ofrece otro cambio que muchos estaban deseando: las cadenas pasan a tener una función *format()*. Si quisiéramos formatear una cadena en Python 2.x teníamos que usar la expresión %, como en:

```
>>> "Hola %s" % ("mundo")
Hola mundo
```

Python 3000 nos permite hacer esto usando directamente la clase *string* (como hace por ejemplo Ruby):

```
>>> "Hola {}".format("Mundo")
'Hola Mundo'
>>> "Hola %s" % ("Mundo")
'Hola Mundo'
```

aunque Python 3000 no elimina el uso de %, se espera que próximas versiones sí lo eliminen, por lo que ha sido declarado obsoleto. El nuevo formateador de cadenas de Python 3000 es muy potente, pues posee una sintaxis que nos permite expresar muchas más opciones y de forma más clara.

```
>>> "{0}, hace {0}".format("calor")
'calor, hace calor'
```

Como ventaja podemos decir que el uso de los parámetros por posición nos permite hacer uso del mismo parámetro en varios puntos de la cadena.

Tipo Byte

Existe otro cambio que romperá gran cantidad del código de Python 2.x. Hasta ahora existían cadenas Unicode y cadenas de 8-bytes. Durante años han existido bugs que aparecían esporádicamente cuando se mezclaban ambos tipos de cadenas. La solución no consistía en arreglar los bugs, puesto que implicaría introducir una cantidad de parches que harían que el código no se pudiera mantener. El problema estaba en el propio diseño de las cadenas en Python 2.x.

Python 3000 puede partir de cero, por lo que se optó por cambiar la estructura interna de las cadenas para solucionar estos problemas de una vez por todas. Ahora todas las cadenas, se declare o no así, son Unicode. Todo trabajo con cadenas será en este formato, y por tanto Python no dudará al trabajar con distintos tipos de cadenas. Este cambio hace que mucho código que jugaba con los *encodings* de las cadenas (cambiando de *ASCII* a *UNICODE* para poder trabajar con diferentes tipos de cadenas) quede obsoleto.

Se introduce, asimismo, un nuevo tipo de dato *byte* para trabajar con datos a bajo nivel. El API de entrada/salida nos permite trabajar con ficheros binarios usando *bytes* en lugar de cadenas de caracteres como hasta ahora. Para decla-

rar una cadena de tipo *byte* tenemos que emplear:

```
>>> b"holo mundo"
b'holo mundo'
>>>
```

¡Las hebras Han Muerto, Vivan los Procesos!

De todos es sabido que Guido no ve con buenos ojos las hebras. Python 3000 aún posee el famoso GIL, un bloqueo que afecta a mucha parte del código y que hace que Python no sea un lenguaje especialmente eficiente respecto a las hebras. Guido dice una y otra vez que las hebras empeoran el código, y que las pruebas que ha realizado eliminando el GIL no muestran un aumento en rendimiento lo suficientemente grande como para justificar su eliminación (que sería muy complicada y traumática a día de hoy). Por tanto, el mundo de Python, aunque dispone de hebras, nunca ha hecho un gran uso de ellas.

Pero el universo cambia mucho más rápido que las personas, y aquí estamos, con procesadores multinúcleo (un mismo procesador posee varios procesadores en su interior) que no son explotables sin algún sistema que nos permita hacer que nuestro programa haga varias cosas a la vez, aprovechando así todos los procesadores de nuestros sistemas.

Las hebras han tenido siempre un competidor, los procesos. La diferencia entre ambos es que los procesos, en teoría, al ser entidades gestionadas por el sistema operativo, son lentos. Cada proceso es un programa que el sistema operativo tiene que controlar. Las hebras son más livianas, son pequeñas unidades de código que se ejecutan dentro de nuestro programa, ajenas al sistema operativo. Pero a día de hoy, ni los procesos son tan lentos, ni las hebras son ajenas al sistema operativo. El único inconveniente de los procesos ha sido siempre lo engorroso que resulta su uso.

Pero Python 3000 trae consigo una librería que, en ciertos aspectos, es revolucionaria. Nos referimos a *multiprocessing*, una librería que hace que la programación con múltiples procesos se simplifique enormemente. Para ello, *multiprocessing* nos ofrece un interfaz que hace a los procesos comportarse como si fuesen hebras dentro de nuestro programa. Un ejemplo simple sería:

```

from multiprocessing
import Process

def f(nombre):
    print('¡hola {0}!'
          .format(nombre))

if __name__ == '__main__':
    p = Process(target=f,
               args=('Juan',))
    p.start()
    p.join()

```

La clase *Process* oculta totalmente el duro trabajo subyacente. Sólo hemos de pasarle la función que queremos ejecutar, mediante el parámetro *target*, y los argumentos que aceptará en una tupla. Con nuestro proceso listo, sólo tenemos que arrancarlo con el método *start()*. El método *join* nos permite bloquear el programa hasta que el proceso acabe, si es eso lo que deseamos. ¿Sencillo verdad? Lo que ha ocurrido en realidad es que Python ha arrancado otro intérprete, al que ha pasado la función y los argumentos. Este nuevo intérprete, al estar en un sistema multinúcleo y ser un proceso, puede ejecutarse en el otro procesador de nuestro sistema mientras nuestro programa principal sigue en ejecución.

Este ejemplo es sólo una mínima demostración de las capacidades de *multiprocessing*, que posee mecanismos de comunicación entre procesos muy potentes, y que exploraremos en un artículo futuro.

Decoradores de Clases

Los decoradores han sido una de las características especiales de Python. Son funciones que aceptan otras funciones como parámetro, las decoran (hacen algo más) y entonces las ejecutan. De esta manera es muy sencillo, por ejemplo, hacer un decorador que registre el tiempo de ejecución de una función.

Guido ha visto siempre los *decorators* como un mecanismo algo complicado de entender, lo cual está en contra de la filosofía de Python. Aunque debido a la presión, cedió finalmente a que se extendiera su uso. Hasta ahora los decoradores se aplicaban a las funciones o métodos de clases, pero en Python 3000 pueden aplicarse también a las clases, un deseo siempre presente en la comunidad Python, ya que una vez que se acostumbró a ellos quería poder «decorar» el comportamiento de las clases también.

¿Qué ventaja nos da esta nueva característica? En la lista de correo de Python aparecieron algunos ejemplos. Imaginemos que poseemos un sistema que funciona como un despertador: a determinada hora arranca una serie de funciones que realizan una serie de tareas. Las funciones deben ejecutarse, pero generalmente también queremos que realicen más tareas. Por ejemplo, que dejen constancia de si la ejecución ha sido correcta. Es mucho más simple realizar esta tarea empleando clases, puesto que éstas pueden poseer distintos métodos, uno para preparar el sistema, otro para dejar constancia de la ejecución, otro para enviar un email... Por tanto sería interesante poseer un código como el siguiente:

```

>>>import despertador
>>>@despertador.cada_noche
...class Mantenimiento:
...     def prepara() : ...
...     def ejecuta() : ...
...     def log() : ...
...     def alerta() : ...

```

Simplemente situar encima de una de las clases que queremos que se ejecute cada noche el *decorator* adecuado. Si Python es un lenguaje orientado a objetos, no tenía mucho sentido olvidarlos de las clases en los *decorators* ¿no?

Diccionarios

Los diccionarios han sufrido algunos cambios y mejoras que pueden trastocar nuestra idea sobre el funcionamiento de sus métodos. El mayor cambio, a mi parecer, ha sido la introducción de un nuevo concepto, los denominados *view objects* («objetos vista»).

Hasta ahora, ciertos métodos de los diccionarios devolvían estructuras de datos. Por ejemplo, el método *keys()* nos devolvía una lista con las llaves que almacenaba el diccionario. Esa lista la podíamos emplear para recorrer el diccionario. Esta forma de actuar poseía un defecto quizá no muy evidente. Cuando le pido las llaves a un diccionario, éste me devuelve las llaves como una lista, que paso a usar. Pero, ¿qué ocurre si las llaves han cambiado? ¿debo volver a pedírselas al diccionario? Es muy complicado tener un programa coherente si tenemos hebras o múltiples procesos en ejecución, porque el diccionario puede ser modificado antes de que empleemos las llaves. Y recordemos que Python 3000 se toma muy en serio el problema de las hebras y el multiproceso.

Los *view objects* no son entidades estáticas, sino que hacen referencia al diccionario que los creó, de forma que cualquier cambio en el diccionario se verá reflejado en el *view object*. Por ejemplo:

```

01 >>> d = {'Nombre' : 'Juan',
02         'Apellidos': 'García'}
03 >>> vista = d.keys()
04 >>> for llave in vista:
05 ...     print(llave)
06 ...
06 Apellidos
07 Nombre
08 >>> d['Dirección'] = 'Calle
09 Linux Magazine'
10 >>> for llave in vista:
11 ...     print(llave)
12 ...
12 Apellidos
13 Nombre
14 Dirección
15 >>> print(vista)
16 <dict_keys object at
17 0x284c4ae0>
18 >>> list(vista)
19 ['Apellidos', 'Nombre',
20 'Dirección']

```

Así, si tenemos una vista del diccionario con sus llaves, sabemos que jamás tendremos que actualizarla, porque cualquier cambio en él se verá reflejado en la vista. La última sentencia que vemos es interesante: las vistas, como tales, no son imprimibles. Se comportan como un iterador, por lo que hay que emplear la función *list()* para poder generar una lista, esta vez estática, con las llaves del diccionario.

Existe otro cambio, de agradecer, respecto a los diccionarios. Ahora es posible generar un diccionario usando el equivalente a una *comprehension list*, que permite definir una lista como una operación sobre los componentes de un rango u otra lista. Era posible hacer lo siguiente:

```

>>> [x*2 for x in [1,2,3,4]]
[2, 4, 6, 8]
>>> [x.capitalize() for x in
['buenos', 'días']]
['Buenos', 'Días']
>>>

```

Ahora también es posible expresar un diccionario de la misma forma:

```

>>> {x : x*2 for x in
[1,2,3,4]}

```

```
{1: 2, 2: 4, 3: 6, 4: 8}
>>> palabras = {x : "En
mayúsculas es {0}"}
format(x.upper()) for x in
['buenos', 'días'] }
>>> palabras['días']
'En mayúsculas es DÍAS'
>>>
```

Contextos

Los contextos son una de las grandes innovaciones, tomada prestada directamente de Lisp. Un contexto es un trozo de código en el que imperan una serie de variables o condicionantes. El ejemplo típico es trabajar con un fichero. Un fichero debe ser abierto, usado y cerrado. Pero aparecen muchos errores en los programas porque el programador se olvida de cerrar el fichero. Los contextos solucionan este problema de una manera muy simple:

```
>>> with open('/tmp/hola',
'a+') as fichero:
...     print(fichero.
readline())
...     print(fichero.closed)
...
Hola mundo
False
>>> fichero.closed
True
>>>
```

En este ejemplo se abre un fichero dentro de un contexto con la expresión *with*, que se asegura de abrirlo, y cuando acabe el contexto, cerrarlo. De esta forma nunca olvidaremos de cerrar el fichero. El atributo *closed* del fichero nos devuelve *True* si está cerrado y *False* en caso contrario, por lo que podemos comprobar que una vez fuera del contexto el fichero está cerrado.

Los objetos deben adaptarse para poder usarse dentro de un contexto, en particular deben poseer los métodos:

- `__enter__`, que se ejecuta al inicializar el contexto.
- `__exit__`, que se ejecuta al salir del contexto.

Conjuntos

Ahora los conjuntos son individuos de primer nivel en el lenguaje, y podemos definirlos directamente:

```
>>> frutas = {'pera',
'manzana', 'plátano', 'pera'}
>>> 'pera' in frutas
```

```
True
>>> 'naranja' in frutas
False
>>> frutas[0]
Traceback (most recent call
last):
  File "<stdin>", line 1, in
<module>
TypeError: 'set' object does
not support indexing
```

El fallo se debe a que un conjunto no puede ser indexado, pues sus elementos no poseen cantidad ni posición.

El abc

Las *abc* («Abstract Base Class») son una de las características más esperadas, puesto que hacen que el sistema de clases de Python se vuelva más normal. Python emplea un sistema de objetos que se suele denominar Duck Typing («Tipado estilo Pato»), que viene de la famosa frase: «si anda como un pato y hace cuack como un pato, entonces es un pato».

Una clase en Python puede implementar el método `__setitem__`, que nos permite que sea posible emplear la sintaxis *clase[valor] = algo* como si la clase fuese una lista o un diccionario. De hecho, podemos implementar `__setitem__` sin hacer lo propio con su compañero inseparable, `__getitem__`, que nos permite acceder a los datos empleando la sintaxis *clase[valor]*.

Lo que ocurre es que, aunque útil, este sistema posee algunos defectos. En particular es complicado saber cuándo, de verdad, una clase se va a comportar como, digamos, un diccionario. Deberíamos introducirnos en el interior de la clase y ver si tiene todos los métodos necesarios. Esto es engorroso y nos puede llevar algo de tiempo.

La solución a este dilema consiste en las *abc*, que son un primas cercanas a las interfaces en lenguajes como Java. Una *abc* es una clase abstracta que contiene en su interior los métodos para, digamos, un diccionario. Si la clase no los implementa el programa dará un error, Python se asegurará de que la clase que diga implementar el *abc* de un diccionario posea todos los métodos necesarios. Así, simplemente con preguntar a una clase si implementa el *abc* adecuado podremos saber si podremos trabajar con ella. Esto simplifica mucho el diseño de sistemas de clases.

Python 3000 viene de serie con *abc* para los objetos que podemos denominar colecciones (como las listas, diccionarios o con-

juntos), así como para los números y los streams de entrada/salida. Expandiremos la explicación sobre este nuevo concepto en artículos posteriores.

Otros Cambios

El que sigue es uno de los mayores dolores de cabeza con las cadenas en Python. El código fuente de Python está por defecto en ASCII. Por si fuera poco debemos emplear la fórmula:

```
# -*- coding: utf-8 -*-
```

Porque Python no aceptaba el código fuente en UTF-8, a no ser que le digamos lo contrario, algo que ya olía a viejo, puesto que todos los editores de texto actuales, y sus respectivos sistemas operativos, emplean UTF-8. Python 3000 acepta código fuente en UTF-8 como normal, rompiendo con el pasado.

El rendimiento de Python se ha mejorado en 30% más o menos, lo que mejora aún más su competitividad con otros lenguajes de programación. El aumento de rendimiento también ha permitido reimplementar mucho código que estaba escrito en C en Python, facilitando su mantenimiento sin que disminuya el tiempo de ejecución.

Conclusión

Por increíble que parezca, sólo hemos rasgado la superficie. En otra entrega exploraremos la gran cantidad de librerías que se han introducido además de todas las que han dejado de formar parte de la librería estándar. Python 3000 supone un salto cualitativo, que lleva años madurándose, y que tiene como objetivo colocar a Python entre los lenguajes más usados del mundo. Con casi 20 años de experiencia a sus espaldas, la experiencia de Python es un grado importante en una época en la que la novedad es lo que impera.

Python 3000 puede conseguir que el mundo abra sus ojos a unos de los lenguajes más interesantes de las últimas décadas. ■

RECURSOS

- [1] Novedades en Python 3000: <http://docs.python.org/dev/3.0/whatsnew/3.0.html>
- [2] Descarga de Python 3000: <http://www.python.org/download/releases/3.0/>