

Python presenta novedades esta temporada con la aparición de la versión 2.5.

# LA SERPIENTE MUDA SU PIEL

Metadatos, evaluación parcial de funciones, cláusula 'with',... la comunidad Python no ha parado durante los últimos meses para integrar una serie de mejoras en Python que lo acercan cada vez más a Lisp o Haskell.

¡Ahora la serpiente tiene nuevos colmillos y son mucho más peligrosos! **POR JOSÉ MARÍA RUIZ**

¿Pero se puede mejorar Python? Quizás la respuesta a esta pregunta se encuentre en el ensayo del polémico Paul Graham «Beating the averages» (ver Recurso [1]). En él Graham nos dice que hace tiempo descubrió un problema que afectaba a los programadores y que pasó a denominar la «paradoja de Blub». Para él, el programador adapta su cerebro al lenguaje de programación que emplea, de forma que piensa usando los conceptos en los que se basa ese lenguaje.

Por lo tanto, existen diferencias apreciables entre los distintos lenguajes de programación, puesto que se basan en conceptos distintos. El lenguaje C se basa en el manejo a bajo nivel de los

datos, llegando a trabajar a nivel binario. En Java, sin embargo, todo lo que hay es un objeto de algún tipo, aunque posee una librería enorme de funciones con las que manipularlos. Para Paul Graham Lisp es el más potente de los lenguajes, un punto y aparte en la evolución de los mismos, debido a que fue desarrollado a partir de una teoría matemática.

Imaginemos un lenguaje llamado «Blub» y a Juan, un programador que no conoce «Blub». Juan está limitado a los conceptos y propiedades del lenguaje que emplea. Paul se pregunta ¿Qué ocurre si «Blub» tuviese una serie de conceptos que ahorran tiempo y además hace cosas que el lenguaje de Juan simplemente no puede hacer?

En un primer momento Juan directamente ignorará el lenguaje «Blub». Esto ocurre porque el cerebro de Juan no es capaz de imaginar siquiera en esos conceptos, ya que se encuentra fuera del entorno de conceptos de su lenguaje. En cambio, si Juan aprendiese a usar «Blub», cuando volviese a su antiguo lenguaje lo notaría... incómodo, debido a que su cerebro ahora emplea conceptos de «Blub» que su anterior lenguaje de programación ni siquiera permite implementar.

Algunas de las nuevas características de Python son como las de «Blub». No existían o era imposible implementarlas en sus versiones previas. Aunque una vez que las pruebas no puedes vivir sin ellas. Pero ¿cuál ha sido el foco bajo el

cual se han creado estas nuevas características? Pues principalmente han sido cuatro:

- Disminuir las líneas de código necesarias para escribir algo en Python.
- Aumentar el rendimiento del intérprete de Python.
- Mejorar significativamente el sistema de paquetes.
- Añadir algunas oportunas librerías.

Para hacer uso de las nuevas características normalmente hay que añadir `from __future__ import absolute_import` en el fichero donde hagamos uso de ellas. Muchas están aún en pruebas y no pasarán a ser de uso común por el momento. Vamos a repasar las más importantes (listadas en la Tabla [1]) y ver algunos ejemplos.

## Expresiones condicionales

Comencemos por algo que para muchos ya era necesario. Puede que sea la primera vez que Python rompe su regla de «sólo hay una manera de hacer las cosas». Los programadores que vienen del mundo de Perl suelen decir que Python les resulta un lenguaje para masoquistas, porque siempre tienes que hacer las cosas de la misma forma y no existen atajos.

Comenzó una discusión en la comunidad Python, que es la manera que tiene el software libre de generar ideas nuevas. A los programadores les resultaba engorrosa la sintaxis de la estructura `if` en Python:

**Tabla 1: Mejoras importantes en Python 2.5**

Expresiones condicionales
Sistema de módulos mejorado
Evaluación parcial de funciones
Generadores más potentes
Cláusula <code>with</code>
Sistema de excepciones mejorado
Librería <code>hashlib</code>
Librería <code>sqlite3</code>

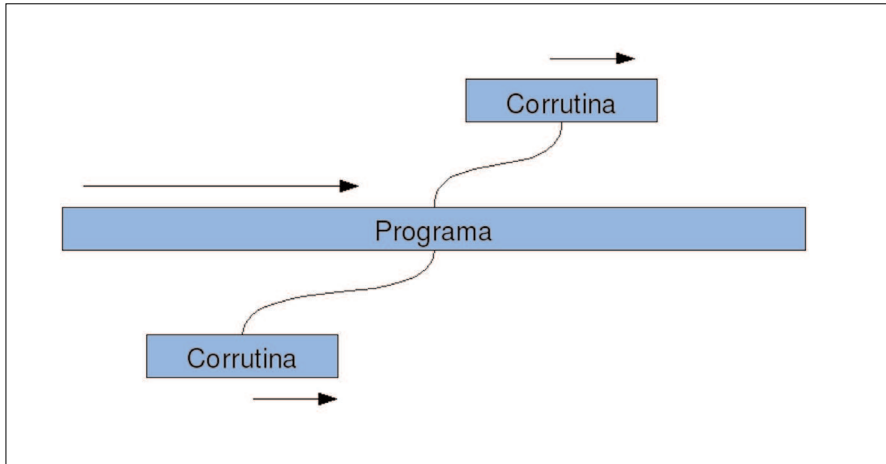


Figura 1: Esquema de corrutinas vs funciones.

```
if sentencia :
    resultado_1
else:
    resultado_2
```

Mientras tanto, la gente de Perl tiene media docena de formas de hacer esto mismo y que usan dependiendo de la circunstancia. Así que para zanjar la discusión, el propio Guido van Rossum, uno de los creadores de Python y gran gurú de la comunidad, propuso:

```
resultado_1 if sentencia
else resultado_2
```

Puede resultar algo raro ver un *if* a no ser que lo pongamos en contexto, digamos que queremos asignar un valor a una variable dependiendo de una condición:

```
if juan_conectado:
    juan.estado = 'conectado'
else:
    juan.estado = 'desconectado'
```

Hemos tenido que escribir cuatro líneas de código para hacerlo muy simple; la solución en Python 2.5 es más compacta:

```
juan.estado = 'conectado'
if juan_conectado else
'desconectado'
```

Pero como en todo en la vida, aquí hay opiniones encontradas. Probablemente hay quien vea en esta nueva sintaxis opcional un acercamiento a Perl, y por tanto al código ilegible; otros, sin embargo, tendrán ahora una sonrisa en

la boca pensando en cuántas líneas de código se podrían haber ahorrado de haberse hecho esto antes.

## Metadatos en paquetes

Los paquetes de Python poseen una serie de metadatos que nos permiten identificar al creador, la versión del paquete, su nombre... Pero digamos que eran más bien para documentación. Existe un problema horroroso cuando provees software: ¡las terroríficas dependencias! Quien haya usado cualquier sistema de paquetes en Linux tarde o temprano ha tenido que vérselas con un programa que no quiere instalarse. O que directamente no funciona, debido a que necesita una librería no presente en el sistema o en una versión que no es la adecuada..

En Python este problema ha estado oculto durante mucho tiempo, no había demasiadas librerías y apenas existían dependencias entre ellas. Pero el éxito te lleva a la fama, y la fama hace que muchos programadores se interesen por hacer software o librerías para tu lenguaje de programación. Así que poco a poco el problema se fue convirtiendo en un escollo para el crecimiento de Python.

¿Solución? Pues la de siempre, crear un sistema de paquetes potente, que bregue con las dependencias. Por el momento se han incorporado a los metadatos de los paquetes de las librerías de Python tres nuevas entradas que podemos ver en la Tabla [2].

Además se ha añadido un nuevo metadato: *download\_url*. Almacena la dirección URL de la cual descargar el código fuente del módulo. Un ejemplo

del uso de estos nuevos metadatos podría ser la siguiente definición de módulo:

```
VERSION = '1.0'
setup(name='MiPaquete',
      version=VERSION,
      requires=['numpy',
               'zlib (>=1.1.4)'],
      obsoletes=
[MiPaqueteViejo]
      download_url=('http://
www.misitioweb.com/mipaquete/Bs
dist/pkg-%s.tar.gz'
                  % VERSION),
      )
```

## Rutas relativas y absolutas

Los módulos de Python normalmente necesitan hacer uso de otros módulos que deberían estar instalados. Si todo está correctamente instalado, deberíamos poder hacer uso de las funciones que proveen esos módulos importándolos. El problema es que hasta ahora Python sólo permitía cargar módulos,

### Listado 1: Nuevas funciones de los generadores

```
01 >>> def counter (maximum):
02 ...     i = 0
03 ...     while i < maximum:
04 ...         val = (yield i)
05 ...         # If value
06 ...         provided, change counter
07 ...         if val is not
08 ...             None:
09 ...                 i = val
10 ...                 else:
11 ...                     i += 1
12 >>> it = counter(10)
13 >>> print it.next()
14 0
15 >>> print it.next()
16 1
17 >>> print it.send(8)
18 8
19 >>> print it.next()
20 9
21 >>> print it.next()
22 Traceback (most recent call
23 last):
24   File ``t.py'', line 15, in ?
25     print it.next()
26 StopIteration
```

pero sólo de forma relativa. Vamos a ver este problema con detenimiento.

Supongamos que nuestro paquete tiene un módulo llamado *string.py*, si dentro de mi paquete pongo `import string` se cargará el módulo *string.py* de mi paquete. Pero ¿qué ocurre si el módulo que quiero es el módulo *string* que es parte de Python? Tenemos un conflicto de nombres, estamos cargando el paquete incorrecto. Lo peor es que con Python 2.4 no existe una manera de cargar el módulo *string* de Python.

En Python 2.5 puedes usar una versión de *import* que busca de forma absoluta, es decir, que diferencia entre un *import* dentro de tu propio paquete respecto a uno de fuera. Para poder identificar los *import* relativos a tu propio paquete debes usar un punto:

```
from .string import
funcion1, funcion2
from . import string
```

De esta forma Python sabe que tiene que buscar en tu propio paquete y no fuera. Incluso se pueden realizar *imports* de sub-sub-sub-módulos:

```
from ...LM import articulo
#Importa proyectos.revistas.LM.
articulo
```

## Evaluación parcial de funciones

La evaluación parcial de funciones es una técnica que ayuda a mejorar la forma en que se diseñan los programas. La teoría subyacente a esta característica es muy potente y se basa en uno de los primeros modelos de computación, el cálculo lambda de Alonzo Church.

Lo mejor que tiene la evaluación parcial de funciones es que nos ahorra escribir gran cantidad de código redundante. Al igual que pasaba con las nuevas expresiones condicionales, esta característica era echada de menos por muchos ex-programadores de Scheme o Common Lisp. Y es que Python es un lenguaje bastante usado en Inteligencia Artificial, por lo que no faltan en sus filas antiguos programadores de Lisp (como por ejemplo, la gente de Reddit, ver Recurso [2], o el propio Peter Norvig, Recurso [3]).

Digamos que tenemos una función llamada *registra(servicio,mensaje)*, que

acepta dos variables *servicio* y *mensaje*. Esta función será muy usada en nuestro programa. Una de las dos variables en realidad sirve para seleccionar algún comportamiento (un servicio en concreto), mientras que la otra sólo sirve para enviarle un dato a la función, el mensaje a mostrar. Podemos definir la función así:

```
def registra(servicio,
mensaje):
    print "[%s] %s" %
(servicio,
mensaje)
```

Cada vez que usemos la función tenemos que decirle el servicio que genera el mensaje. Podemos crearnos nuestra propia función para un servicio que usemos mucho:

```
def registra_pantalla(
mensaje):
    registra("Pantalla",
mensaje)
```

Y lo mismo tendríamos que hacer con cada servicio. Acabaríamos teniendo una pantalla de nuestro editor de textos llena de definiciones de funciones que serían copias casi exactas, diferenciándose en un único valor las unas de las otras. ¿Existe una solución a este problema? Pues sí, aquí es donde entra la evaluación parcial de funciones. Por el momento no forma parte de Python, sino de un módulo que tendremos que cargar:

```
import functools
```

Ya tenemos lo que necesitamos, ahora podemos hacer uso de la función *functools.partial()*, que acepta la función que vamos a «recortar» y los parámetros que

## Listado 2: Ejemplo de uso de with

```
01 from __future__ import
with_statement
02
03 class hola:
04     def __enter__(a):
05         print "entro"
06     def __exit__(a,b,c,d):
07         print "salgo"
08
09 with hola() as h:
10     print "hola"
```

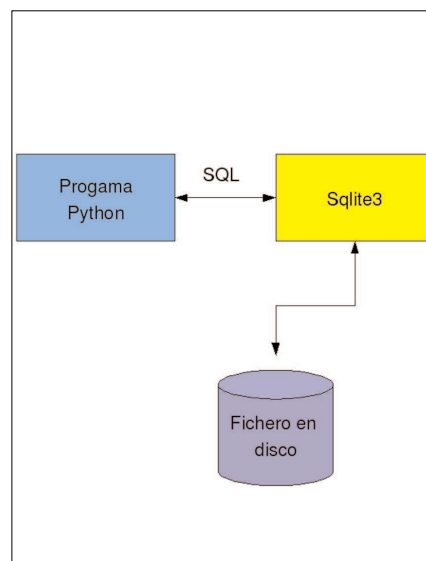


Figura 2: Esquema de uso de SQLite3.

dejaremos constantes, y nos devuelve una función:

```
registra_pantalla =
functools.partial(registra,
servicio="Pantalla")
```

A continuación podemos hacer uso de *registra\_pantalla()* como si la hubiésemos definido nosotros mismos:

```
>>> registra_pantalla(
"¡Está todo lleno de Python!")
[Pantalla] ¡Está todo lleno de
Python!
>>>
```

Y listo, ya hemos generado una función a partir de la evaluación parcial de otra.

## Listado 3: Ejemplo de uso de SQLITE3

```
01 >>> conexion =
sqlite3.connect('/tmp/burbuja'
)
02 >>> c = conexion.cursor()
03 >>> c.execute('create table
profetas
04 ... (fecha
timestamp, nombre varchar)')
05 ...
06 >>> c.execute('insert into
eventos
07 ... values
('2006-10-05','TOCHOVISTA')')
08 ...
09 >>>
```

Este ejemplo es demasiado simple. Veamos cómo podemos aprovechar esto desde el punto de vista del diseño de programas. Es normal encontrarse funciones que difieren unas de otras en una constante. Un ejemplo del uso de la evaluación parcial es la siguiente función, que va a ser un generador de funciones incrementadoras:

```
>>> def suma (a, b):
...     return a + b
...
>>> incrementa_1 = functools.
partial(suma, 1)
>>> incrementa_1(42)
43
>>>
```

Con la evaluación parcial de funciones se pueden apastar funciones como en el caso de *suma*:

```
>>> suma = functools.
partial(suma, 1)
>>> suma(2)
3
>>>
```

Con la evaluación parcial de funciones podemos centrarnos en crear funciones que posteriormente nos servirán para crear tipos más específicos de las mismas.

## Nuevos generadores

Los generadores son parte vital de Python. Sin ellos los bucles serían imposibles. Los generadores se diferencian del resto de funciones en que emplean la declaración *yield*, que bloquea la función donde aparece, devolviendo un valor. Como *yield* es una declaración y no una expresión, no devolvía ningún valor. Veámoslo con un ejemplo:

```
>>> def contador (maximo):
...     i = 0
...     while i < maximo:
...         yield i
...         i += 1
```

Aquí tenemos una función que hace uso de *yield* para devolver un valor cada vez que llamamos a *next()*. Vamos a probarla:

```
>>> a = contador(10)
>>> a
<generator object at 0x81432cc>
>>> a.next()
0
>>> a.next()
1
>>> a.next()
2
```

En este ejemplo hemos creado un generador y después, usando el método *next()* hemos ido generando números a partir de 0. Antes de Python 2.5, *yield* era una declaración y no devolvía nada, mientras que ahora, en Python 2.5, es una expresión que devuelve el valor que «eleva». Así que en Python 2.5 podemos escribir sentencias como *numero = (yield i)*.

Otra novedad importante es la introducción del método *send()* en los generadores. Su comportamiento es algo peculiar. Cuando empleamos este método en un generador, el valor que pasemos será el siguiente que el generador devolverá, pero a partir de ese momento *yield* devolverá *None*. Si el lector se preguntaba el porqué del cambio en la forma de trabajar de *yield* que explicamos en los anteriores párrafos, *send()* es uno de los culpables (ver Listado [1]).

El siguiente *next()* después de *send()* ejecuta el *else* de *contador* debido a que *yield* devuelve *None*.

Otro punto nuevo es que los generadores a partir de ahora son corrutinas (ver Figura [1]). Una corrutina no es exactamente una función porque a diferencia de éstas puede ser interrumpida y llamada numerosas veces. El uso de *yield* en una función la convierte en una corrutina. Para poder tratar con las corrutinas/generadores se han incorporado dos nuevos métodos, ver Tabla [3].

Tanto *throw()* como *close()* son necesarios si queremos poder comunicarnos con

el interior del generador. Es muy importante tener claro que las corrutinas son funciones que están vivas, y con la que sólo nos comunicamos mediante estos métodos. Así *close()* nos permite usar las sentencias *try...finally* dentro del generador para limpiar o cerrar objetos. Imaginemos que usamos un generador para recorrer una tabla de una base de datos, en tal caso podemos cerrar la conexión cuando dentro del generador aparezca la excepción *GeneratorExit*.

## Claúsula «With»

Esta característica sí que es extraña, porque ninguna otra es tan cercana a Lisp. La cláusula *with* es la típica característica que no echas de menos porque ni siquiera sabes que existe. Y en realidad es muy simple y cómoda. Digamos que tenemos que realizar alguna operación en la que hay que acceder a un recurso, lo usamos y finalmente debemos liberarlo. Esto es algo común, no sólo con ficheros, conexiones de red o bases de datos, sino también con datos compartidos y protegidos por variables de bloqueo cuando se programa con hebras. Veámoslo con un ejemplo:

```
>>> with open('/etc/passwd',
'r') as f:
...     for linea in f:
...         print linea
```

En lugar de tener que abrir el fichero, operar con él y cerrarlo, todo se soluciona con una simple sentencia. Además, al tabular el código dentro de *with*, queda claro el código que trabaja con el fichero *f*. Pero lo mejor es que podemos crear nuestros propios *with* de forma que podemos simplificar el uso de nuestras funciones y objetos.

El contexto de *f* es *with*, así que es como si fuese local a *with* y lo podremos reutilizar más veces. Esto puede parecer una nimiedad, pero cuando te acostumbras a usar ciertos nombres para las variables, esto es agua de

**Tabla 2: Nuevas entradas en los metadatos de los paquetes**

<i>requires</i>	módulos que necesita ésta misma para funcionar.
<i>provides</i>	submódulos que se ofrecen.
<i>obsoletes</i>	módulo al que el paquete deja obsoleto.

**Tabla 3: Nuevos métodos de los generadores**

<i>throw()</i>	nos permite soltar una excepción dentro de un generador.
<i>close()</i>	suelta la excepción <i>GeneratorExit</i> dentro del generador.

mayo. Yo por ejemplo suelo usar *f* para los ficheros.

Para ello debemos implementar en el objeto que usaremos en la cabecera de *with* los métodos `__enter__(self)`, que prepara el entorno, y `__exit__(self, type, value, tb)` que lo limpia (ver Listado [2]).

## Nuevo try/except/finally

Parece extraño pero Python permite:

- Capturar excepciones, o (exclusivo)
  - Ejecutar incondicionalmente un bloque de código dentro de un *try*
- Por el contrario, el resto de lenguajes con excepciones permiten ambas opciones. Pero hasta Python 2.5 no ha sido posible en Python hacer uso de ambas simultáneamente. Quizá el actual trabajo de Guido en Google, donde usan tanto C++ como Java, puede haberle inducido a reconsiderar el sistema de excepciones de Python. Esta «novedad» hace que el sistema de excepciones de Python se comporte de forma más normal, un ejemplo puede ser:

```
>>> try:
...     1/0
... except BaseException:
...     print "un error ha ocurrido"
... finally:
...     print "pero nos vamos"
...
un error ha ocurrido
pero nos vamos
```

## Excepciones 2.0

Guido parecía no conformarse con el nuevo sistema de recogida de excepciones, así que ha seguido incorporando características de Java. En este caso se trata de la jerarquía de excepciones. Python posee, a partir de la versión 2.5, una nueva excepción padre para todas las demás excepciones: *BaseException*. La usamos en el último ejemplo porque esta excepción, al ser el padre de todas las demás, permite capturarlas a todas. Los desarrolladores de Python, advirtiendo que la sentencia *except BaseException* aparecería constantemente, han creado un atajo:

```
>>> try: ... 1/0 ... except: ...
print
"un error ha ocurrido" ...
finally: ... print "pero nos vamos"
```

Lo que han hecho es permitir a *except* aparecer en solitario, lo que equivale a decir *except BaseException*. Nunca está de más ahorrarse unos cuantos tecleos ;).

## Nueva HASHLIB

Las funciones de hash criptográfico estaban esparcidas por los módulos de Python. Así, por ejemplo, *MD5* tenía su propio módulo. Debido al nuevo impulso que se ha dado a la librería base, se ha decidido crear un módulo específico para funciones hash. Este módulo, llamado *hashlib* incorpora implementaciones de *MD5* y distintos tipos de *SHA* no disponibles hasta Python 2.5. Además, el rendimiento ha aumentado bastante gracias al uso de *OpenSSL* cuando está disponible en el sistema. Ahora intentad haced por ejemplo:

```
>>> import hashlib
>>> a = hashlib.md5()
>>> a.update("hola")
>>> a.digest()
'M\x18c!\xc1\xa7
7\xf0\xf3T\xb2\
x97\xe8\x91j\xb2@'
```

## SQLITE3

Este es uno de los platos fuertes de Python 2.5. Python es uno de los primeros lenguajes en incorporar de serie... una base de datos. Así, tal como suena. *SQLITE3* es una mini-base de datos que soporta gran parte de SQL y que trabaja sobre ficheros. Quizá estemos ante la muerte de los ficheros de texto o binarios tal como los conocemos. El lector puede imaginarse las posibilidades que se abren. Por ejemplo, si Python ya trae de serie un módulo de manipulación de correo electrónico y uno que permite crear y manipular bases de datos *SQLITE3*, sería posible implementar un servidor de correo electrónico en Python y que almacenase los correos en la base de datos. De hecho Firefox 2.0 va a usar *SQLITE3* internamente para el almacenamiento de favoritos.

Esta base de datos va a dar mucho que hablar.

*SQLITE3* emplea ficheros normales y corrientes como repositorio para los datos. Puedes crear una base de datos y comenzar a usarla inmediatamente. Es, además, muy rápido, debido a que todas las consultas se realizan mediante llamadas a funciones de *SQLITE3* directamente, sin intermediarios (ver Listado [3] y Figura [2]).

## Conclusión

Los cambios introducidos en Python 2.5 buscan ir introduciendo en Python algunas de las características más interesantes de otros lenguajes. En particular, parece ser que la atención de la comunidad está comenzado a inclinarse por los lenguajes funcionales, como Scheme o Lisp.

Se ha realizado un esfuerzo titánico para hacer el sistema de módulos y librerías mucho más potente y coherente con el resto del lenguaje. Y finalmente, se nota que Guido está recibiendo ideas en Google (recordemos que ya lleva allí un año y pico, si no dos).

El salto ha sido evolutivo más que revolucionario, nada realmente importante se ha modificado en el sistema y ninguno de los cambios hará que programas realizados con versiones anteriores dejen de funcionar. Se ha realizado un enorme esfuerzo, aunque oculto debido a que nadie se fija en él, en mejorar tanto el rendimiento como la seguridad de Python, reimplementando muchas funciones de la librería base desde cero.

Desde mi punto de vista la comunidad Python ha sabido mantener uno de los puntos fuertes de su lenguaje: la sencillez y que seguirá siendo una plataforma estable sobre la que construir grandes programas. Esperemos que el nuevo sistema de módulos haga que proliferen tanto como los de Perl. En definitiva, un magnífico salto evolutivo. ■

## RECURSOS

- [1] <http://www.paulgraham.com/avg.html>
- [2] <http://www.reddit.com>
- [3] <http://www.norvig.com>