

Python se está convirtiendo en una plataforma

INTÉRPRETES

Gracias al esfuerzo de muchos desarrolladores, ahora podemos disfrutar del poder de Python en sitios donde antes estaba vetada su entrada. **POR JOSÉ MARÍA RUÍZ**

Python no es CPython. No es la primera vez que lo digo, pero es casi un mantra que debemos repetirnos todos los días. El intérprete CPython (ver Recurso 1), a pesar de ser el que usamos la mayoría de nosotros, es sólo una de las implementaciones que se han realizado del lenguaje de programación Python. Es la implementación más importante y la que marca el rumbo de las demás. Pero han aparecido competidoras poderosas, y muchas veces desconocidas, que pueden ir robándole posiciones. Cada una de ellas ha querido centrarse en alguna característica en especial. Las hay que quieren portar Python a plataformas distintas, otras creen que el intérprete de Python podría ser mejorado, otras que el modelo de concurrencia de Python es erróneo,...

Y aún así, todas ellas son Python. Sólo pequeñas incompatibilidades o mejoras específicas nos impiden pasar el código de nuestro programa de unas a otras sin problemas. Vamos a estudiarlas, viendo algunas de las características que las separan de CPython.

Jython

Esta es sin duda la implementación más veterana y mejor conocida de Python. Surgida de la mano de Jim Hugunin a finales

de 1997, buscaba demostrar que era posible mejorar la velocidad de Python si se empleaba la máquina virtual de Java. Su desarrollo se estancó hacia 2005, pero en 2008 Sun Microsystems contrató a varios programadores para resucitarla. ¡E hizo bien! Sin ir más lejos, es uno de los dos lenguajes de script que IBM escogió para su software WebSphere Application Server; existen pues intereses económicos en que Jython avance y no se quede estancado, lo cual es una buena razón para emplearlo.

En el momento de escribir este artículo se encuentra en la versión 2.5.1, liberada el 26 de septiembre de 2009. Por el momento no hay intención de implementar el estándar Python 3.0, y quieren centrarse en dar soporte a la versión 2.X de Python. El objetivo de esta decisión es hacer posible que grandes frameworks Python, como Django o SQLAlchemy, funcionen sin problemas en Jython. ¡Y lo han conseguido! Por desgracia no es posible hacer uso de las librerías Python programadas en C desde Jython, lo que resta algo de compatibilidad a esta opción. Pensemos que Numeric o Pygames hacen uso de librerías C.

Jython es, como ya hemos dicho, la versión más madura de un intérprete alterna-

tivo de Python, así que su integración con JVM es total:

```
from javax.swing import *
ventana = JFrame("Linux Magazine")
etiqueta = JLabel("¡Saludos al lector!", JLabel.CENTER)
ventana.add(etiqueta)
ventana.setDefaultCloseOperation(
    JFrame.EXIT_ON_CLOSE)
ventana.setSize(300, 300)
ventana.show()
```

Importar cualquier librería Java es tan sencillo como llamarla como haríamos con una librería Python. En este pequeño ejemplo creamos una ventana Swing y saludamos (ver Figura 1). Hay que tener en cuenta que el tiempo de arranque de Jython es mayor que el de CPython, puesto que debe arrancarse la JVM, y que la ganancia en velocidad no es tampoco para tirar cohetes, pero nos permite incorporar Python a nuestra aplicación Java sin ningún problema.

Desde principios de 2009 Jython ha sido incorporado como miembro de pleno derecho al IDE NetBeans gracias a la presión de Sun para que así fuese, lo que nos permite

usar este IDE aceptado en la empresa para desarrollar nuestros programas Jython (o incluso CPython).

IronPython

Jim Hugunin también es el artífice de IronPython. Liberó la versión 1.0 en 2006, y curiosamente comenzó su desarrollo para demostrar que los lenguajes script no podrían desarrollarse en la plataforma .Net de Microsoft. ¡Cual fue su sorpresa al comprobar que en realidad era posible! Desde entonces ha estado trabajando en él; el 11 de diciembre de 2009 se liberó la versión 2.6, compatible con la equivalente en CPython. Al igual que Jython, han preferido portar la versión 2.6 de Python a concentrarse en la 3.0, para la cual aún no ven demasiado uso. A diferencia de Jython, IronPython sí implementa *ctypes*, lo que nos permitirá acceder a librerías C desde nuestro intérprete.

En Linux estamos obligados a usar Mono [9] para poder usar IronPython, lo que nos puede acarrear algunos problemas. En particular, he comprobado cómo la versión 2.6 de IronPython falla con mono 2.4.3, por lo que me veo forzado a emplear la versión 2.0.3. Estos fallos se deben a que IronPython hace uso de un nuevo subsistema de .Net llamado DLR: Dynamic Language Runtime. DLR es un conjunto de técnicas y librerías cuyo objetivo es facilitar la implementación de lenguajes dinámicos y mejorar tanto su rendimiento como permitir que puedan interoperar con código en lenguajes estáticos como por ejemplo C#. Aunque Mono está algo atrasado en su implementación, probablemente veamos un soporte completo para DLR en Mono durante 2010.

Podemos ver un ejemplo del uso de IronPython en el Listado 1, donde creamos una ventana usando Windows Forms (ver Figura 2). Como puede observarse, es necesario hacer «alguna triquiñuela» para cargar las librerías de .Net, en particular hacemos uso de un módulo especial llamado *clr* (Common Language Runtime) para poder cargarlas.

Microsoft está apoyando el desarrollo de IronPython, y varias empresas lo están empleando en sus proyectos.

PyPy

PyPy [4] parte de una premisa: No sólo es posible hacer un intérprete de Python usando el propio Python, sino que además se puede hacer que sea mucho más rápido

que el intérprete estándar escrito en C. ¿A que parece una locura? Pues no lo es tanto si tenemos en cuenta dos datos. El primero es que uno de sus desarrolladores es Armin Rigo, el desarrollador del magnífico optimizador Pyco. El segundo, que quieren crear una máquina virtual parecida a la de Java, que aprenda con la ejecución del programa y aplique compilación Just In Time.

La última versión liberada en el momento de escribir este artículo es la 1.1.0, aunque aún no podemos decir que sea una versión que podamos utilizar en producción. Para poder usarla necesitaremos un intérprete de CPython en nuestro sistema, que será el encargado de interpretar el intérprete de PyPy (lioso ¿verdad?), y a pesar de lo que pueda parecer, este intérprete interpretado es igual o más eficiente que CPython!

PyPy se une a IronPython y Jython en su intención de implementar la versión 2.X de Python, pero además PyPy implementa las características que podemos ver en Stackless Python más adelante.

Su uso a día de hoy no pasa de ser experimental, por lo que habrá que esperar un tiempo prudencial para ver cómo se desarrolla su potencial.

Unlanden Swallow

De todos es conocido, y de hecho se publica bastante, que Google usa Python. Pero todo el amor que dispense Google a Python no puede evitar que sus defectos sean evidentes como un faro en mitad de la noche. Y especialmente en un entorno tan extremo como pueden ser las aplicaciones de Google. Por ello creó el proyecto Unlanden Swallow [5], nombre escogido debido al mítico diálogo de los Monty



Figura 1: Hola desde Jython.

Python en «Los Caballeros de la Mesa Cuadrada».

Su objetivo: que en un año tengamos una implementación de Python escrita en C que compile y se ejecute sobre LLVM (la máquina virtual multipropósito, [8]), eliminando si es posible el dichoso GIL y aumentando el rendimiento de forma notable.

Pero una cosa son las intenciones, y otra la realidad. El proyecto ha decidido no tratar de eliminar el GIL durante la iteración del tercer trimestre de 2009, concentrándose en mejorar la gestión del mismo y hacerlo así menos «dañino». En las pruebas realizadas han constatado que el uso de LLVM requiere mucha memoria, por lo que están trabajando en mejorar este aspecto.

Por lo demás, Unlanden Swallow pretende ser un clon perfecto de CPython desde el punto de vista funcional, no debemos esperar ninguna sorpresa, pero desde luego no está aún listo para usarlo en producción. Creo que habrá que esperar a ver su evolución durante 2010, ya que probablemente en la PycCON 2010 sus creadores ofrezcan datos sobre su estabilidad y rendimiento.

Unlanden Swallow también se apunta a la versión 2.X de Python, ignorando la 3.0. A pesar del empuje de la Fundación Python por implantar la versión 3.0, la

Listado 1: Hola Mundo desde IronPython

```
01 import clr
02
03 clr.AddReference("System.Windows.
04 Forms")
05
06 clr.AddReference("System.Drawing"
07 )
08
09 from System.Windows.Forms
10 import Application,
11 Form,Button
12
13 from System.Drawing import
14 Point
15
16 class HelloWorldForm(Form):
17
18     def __init__(self):
19         self.Text = 'Linux Magazine'
20         self.Name = 'Linux Magazine'
21
22         form = HelloWorldForm()
23
24         etiqueta = Button()
25         etiqueta.Text = "Hola lector"
26         etiqueta.Location =
27             Point(50,50)
28         etiqueta.Height = 30
29         etiqueta.Width = 200
30
31         form.Controls.Add(etiqueta)
32
33         Application.Run(form)
```

mayoría de los intérpretes son mucho más conservadores y prefieren mejorar lo que ya funciona.

Stackless Python

Stackless Python es una versión de Python bastante desconocida que incorpora una serie de técnicas especiales para entornos concurrentes. El GIL impide a Python la ejecución de varias hebras si se entra en el código del intérprete (cosa que ocurre constantemente). Christian Tismer, desarrollador principal de Stackless Python, cree que el problema está en la pila de llamadas de C. Por eso el nombre de este intérprete es «Stackless», que se traduce como «sin pila». Todo intérprete necesita una pila de llamadas, pero en Stackless Python esta pila se implementa en el propio Python. Con esta nueva flexibilidad, puede implementar construcciones que sólo se ven en otros lenguajes, como por ejemplo en Erlang. Christian comenzó su desarrollo en 1999, y a día de hoy se emplea en entornos de alto rendimiento, como por ejemplo en los juegos «EVE Online» y «Second Life».

Y bien, ¿dónde están esas extrañas diferencias?

```
01 Python 2.6.4 Stackless 3.1b3
02 060516 (release26-maint,
03 Dec 30 2009, 11:40:27)
04 [GCC 4.2.1 20070719
05 [FreeBSD]] on freebsd8
06 Type "help", "copyright",
07 "credits" or "license" for
08 more information.
09 >>> import stackless
10 >>> def hola(x):
11 ...     print "Hola %s" % (x)
12 ...
13 >>> hola("mundo")
14 Hola mundo
15 >>> stackless.tasklet(hola)
16 ('mundo')
17 <stackless.tasklet object at
18 0x285a2304>
19 >>> stackless.tasklet(hola)
20 ('lector')
21 <stackless.tasklet object at
22 0x285a233c>
23 >>> stackless.tasklet(hola)
24 ('linux')
25 <stackless.tasklet object at
26 0x285a2374>
27 >>> stackless.run()
28 Hola mundo
29 Hola lector
```

```
19 Hola linux
20 >>>
```

Hemos importado el módulo *stackless* y hemos creado tres *tasklets*, que podemos traducir por «tareas». Stackless Python posee un planificador de tareas, como el que emplea Linux para decidir qué programa ejecutar en cada momento, que podemos activar mediante la función *stackless.run()*. Al hacerlo, las tareas se ejecutan de forma independiente.

No es algo realmente sorprendente, lo realmente interesante es lo siguiente:

```
01 >>> def tres_saludos(x):
02 ...     print "1: hola ", x
03 ...     stackless.schedule()
04 ...     print "2: hola ", x
05 ...     stackless.schedule()
06 ...     print "3: hola ", x
07 ...     stackless.schedule()
08 ...
09 >>>
10 >>> stackless.tasklet
11 (tres_saludos)('mundo')
12 <stackless.tasklet object at
13 0x00A45870>
14 >>> stackless.tasklet
15 (tres_saludos)('linux')
16 <stackless.tasklet object at
17 0x00A45A30>
18 >>> stackless.tasklet
19 (tres_saludos)('lector')
20 <stackless.tasklet object at
21 0x00A45AB0>
22 >>>
23 >>> stackless.run()
24 1: hola mundo
25 1: hola linux
26 1: hola lector
27 2: hola mundo
28 2: hola linux
29 2: hola lector
```



Figura 2: Hola desde IronPython.

```
24 3: hola mundo
25 3: hola linux
26 3: hola lector
27 >>>
```

Si nos fijamos en la salida veremos que las tres funciones ejecutan algo de su código y pasan el mando a la siguiente. Esto lo logramos mediante la función *stackless.schedule()*, que indica al planificador que puede ejecutar el código de otra tarea. A esta forma de trabajar se le denomina multitarea cooperativa, puesto que son las funciones las que cooperan unas con otras indicando cuándo es posible dar paso a la siguiente. Este tipo de multitarea es muy básico, pero también es interesante y apropiado en muchas situaciones. Si tenemos tareas, lo normal es buscar un mecanismo que nos permita que se comuniquen entre ellas. Stackless Python provee ese mecanismo mediante los *channels* («canales»); si ejecutamos el código del Listado 2 obtendremos:

```
ARRANCA: receptor
ARRANCA: emisor
hola desde el emisor
PARA: receptor
Solo soy una aburrida tarea
PARA: emisor
```

En este caso no hemos bloqueado de forma cooperativa nuestras tareas, sino que ha sido Stackless Python quien se ha encar-

Listado 2: Canales en Stackless Python

```
01 import stackless
02
03 canal = stackless.channel()
04
05 def receptor():
06     print "ARRANCA: receptor"
07     print canal.receive()
08     print "PARA: receptor"
09
10 def emisor():
11     print "ARRANCA: emisor"
12     canal.send("hola desde el
13     emisor")
14
15 def tarea_aburrida():
16     print "Solo soy una aburrida
17     tarea"
18
19 stackless.tasklet(receptor)()
20 stackless.tasklet(emisor)()
21 stackless.run()
```

Plug into the very best in Open Source technology

Come and experience the most important event for Open Source



The world's No. 1 marketplace for digital business

- Attracted 50,000 visitors at CeBIT 2009
- For the first time prominently placed in Hall 2 at CeBIT 2010
- Numerous opportunities to attend forums, special events and project lounges.
For further information go to www.cebit.de/opensource_e



Deutsche Messe
Hannover · Germany

Deutsche Messe AG · Messengelände · Hannover, Germany · Tel. +49 511 89-0 · incoming@messe.de



gado de ejecutarlas de forma concurrente. Al crear un canal y añadir un mensaje al mismo, la tarea que lo hace se bloquea hasta que dicho mensaje ha sido procesado por otra tarea. Esto nos permite crear sistemas en los que multitud de tareas se intercambian información de forma segura.

Stackless Python implementa, al contrario que sus primos, tanto Python 2.X como Python 3.X, manteniéndose cerca del desarrollo de CPython, entre otras razones porque es un conjunto de modificaciones sobre el mismo.

Cython

Cython [7] surgió a partir del proyecto Pyrex. En lugar de interpretar Python, Cython lo compila a un programa C, que podemos volver a compilar a código máquina. El resultado es un aumento espectacular en el rendimiento de nuestro programa. Veamos un ejemplo rápido antes de profundizar. Supongamos que tenemos un programa realmente simple:

```
def saluda(x):
    print "Hola %s" % (x)
```

Este «programa» debemos guardarlo en un fichero con extensión *pyx*, por ejemplo en *hola.pyx*. Como Cython es un compilador que requiere varios pasos, lo más sencillo es crear un fichero *setup.py* como el siguiente:

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext
ext_modules = [Extension(
    "hola", ["p.pyx"])]
setup(
    name = 'Hola mundo',
    cmdclass = {'compile': build_ext},
    ext_modules = ext_modules
)
```

En él cargamos todas las librerías necesarias y creamos el comando que usaremos, el paso final es:

```
$ python setup.py compile -inplace
running compile
building 'hola' extension
cc -shared -pthread
```

```
build/temp.freebsd-8.0-RELEASE-
i386-2.6/p.o -o /usr/home/
josemaria/miscos-svn/
linuxmagazine/articulos/
2009-12-29-M_Python_mundo_real\
/hola.so
```

Y ahora sólo tenemos que ejecutar Python y cargar nuestra nueva librería, *hola.so*:

```
>>> import hola
>>> dir(hola)
['_builtins_', '__doc__',
'_file_', '__name__',
'_package_', 'saluda']
>>> hola.saluda("mundo")
Hola mundo
>>>
```

Vale, hemos creado una librería en formato binario, un fichero *.so*, ¿pero qué hemos ganado con ello? La verdad es que nada, pero si metemos algo de matemáticas o cálculo intensivo en la ecuación, veremos los resultados. Por ejemplo, el siguiente código sirve para integrar mediante un método numérico:

```
01 from math import sin
02 def f(x):
03     return sin(x**2)
04
05 def integrate_f(a, b, N):
06     s = 0
07     dx = (b-a)/N
08     for i in range(N):
09         s += f(a+i*dx)
10     return s * dx
```

Si lo ejecutamos en Python y Cython, por ejemplo entre 0 y 3.14, veremos pocas diferencias de rendimiento. Lo realmente maravilloso de Cython es que podemos añadir tipos al código fuente, por eso la extensión es *pyx* y no *py*. Cython es mucho más que Python:

```
01 from math import sin
02
03 def f(double x):
04     return sin(x**2)
05
06 def integrate_f(double a,
07                 double b, int N):
08     cdef int i
09     cdef double s, dx
10     s = 0
11     dx = (b-a)/N
12     for i in range(N):
```

```
12     s += f(a+i*dx)
13     return s * dx
```

Si observamos bien, veremos que sólo hemos añadido definiciones de tipos al código fuente. Este cambio permite a Cython compilar el código mucho más eficientemente, y lograremos un aumento de rendimiento espectacular: según las pruebas realizadas por la propia gente de Cython, este código es 24 veces más rápido que la versión sin tipos.

No es de extrañar que sean los científicos y las empresas que buscan rendimiento las que más usan Cython y que aparezca dentro del paquete de cálculo numérico Sage, muy usado en ambientes de investigación.

Conclusión

Existen otros intérpretes y compiladores de Python, pero su alcance es muy minoritario y no poseen una gran comunidad a su alrededor. Podemos encontrar experimentos como compiladores e intérpretes escritos en OCaml o Haskell. Python se está convirtiendo más en un estándar que en una implementación, y las distintas implementaciones han ayudado a impulsar el PEP 3003: se bloquearán los cambios en Python por al menos dos años. Este periodo permitirá a todas las implementaciones y librerías ponerse al día con Python 3.0 sin temor a que sus esfuerzos acaben en la papelera debido a un cambio imprevisto en CPython.

Gracias a las distintas implementaciones, podemos decir que Python va a crecer, ampliando los ámbitos en los que podemos escogerlo como nuestro principal lenguaje de programación sin tener que renunciar a las posibilidades que nos ofrecen grandes plataformas como Java o .Net, o rechazar el rendimiento de C o la concurrencia de Erlang. ■

RECURSOS

- [1] CPython: <http://www.python.org>
- [2] IronPython: <http://www.codeplex.com/IronPython>
- [3] Jython: <http://www.jython.org/>
- [4] PyPy: <http://codespeak.net/pypy/dist/pypy/doc/>
- [5] Unladen Swallow: <http://code.google.com/p/unladen-swallow/>
- [6] Stackless Python: <http://www.stackless.com/>
- [7] Cython: <http://www.cython.org>
- [8] LLVM: <http://llvm.org>
- [9] Mono: <http://www.mono-project.com/>