Introducción a Python

v1.01

Tomás Javier Robles Prado
tjavier@usuarios.retecal.es
http://users.servicios.retecal.es/tjavier

Índice

Introducción

Allá por 1991 cuando un tal Linus Torvalds cacharreaba con su Minix, en Holanda se lanzaba la primera versión de Python. Es curioso que ha sido en Holanda donde se desarrolló también Minix.

El creador del lenguaje es Guido Van Rossum. Actualmente sigue ocupándose del lenguaje con un buen número de colaboradores.

Python es un lenguaje interpretado, orientado a objetos de propósito general. Python permite mantener de forma sencilla interacción con el sistema operativo, y resulta muy adecuado para manipular archivos de texto. Esta característica hace que muchas distribuciones de GNU/Linux utilicen Python para sus herramientas de configuración. También es ampliamente utilizado en la web. Conviene echar un vistazo a Zope.

Actualmente Python tiene una licencia compatible con GPL, lo que significa que puedes modificar y distribuir libremente su código. Aunque esta situación es nueva. Pese a que Python ha sido tradicionalmente libre, la adopción de la GPL es muy reciente. Un paso adelante más.

Python comenzó a desarrolarse en y para ordenadores Mac. Actualmente son en sistemas Linux donde se lleva todo el peso de la programación y el uso en Linux de Python está más extendido que en otras plataformas. Sin embargo Python es multiplataforma y podemos descargar el intérprete para casi cualquier máquina.

El nombre de Python no tiene nada que ver con los reptiles, sino con el espectáculo de la BBC de los Monty Python.

Python como Lenguaje de scripts

Habitualmente sobre todo en entornos *IX se suelen utilizar scripts de shell con cometidos muy diversos, como el arranque del sistema, el arranque del modo gráfico, para automatizar ciertas tareas, sobre todo con archivos de texto, etc. En entornos Microsoft estos scripts equivalen a los antiguos *.bat, aunque la potencia de estos es mucho menor.

Estos scripts están compuestos por una serie de comandos del sistema que se ejecutan secuencialmente. Son sencillos de crear pero tienen una potencia limitada. Un usuario hábil logrará mediante el uso de *grep*, *cut* automatizar algunas tareas sobre archivos de texto, pero pronto se encontrará con limitaciones, se le quedarán pequeños. Veamos un ejemplo:

> *Un usuario tiene un directorio llamado $HOME/pascal/ donde trabaja a diario. Cada día cuando termina su jornada, desea hacer una copia de seguridad de su trabajo en un disquete. Tiene decenas de ficheros fuente y desea que cada vez sólo guarde en el disquete los archivos que han sufrido cambios respecto al día anterior.*

Este es un ejemplo en que los scripts de shell podrían ser insuficientes (al menos yo no conozco forma de hacerlo). Tampoco parece que merezca mucho la pena realizar un programa

en C o Pascal para solucionar este problema, ya que puede resultar tedioso y complicado cuando en realidad el problema es sencillo.

Aquí es donde entran en juego los lenguajes de scripts como Python, Perl o Tcl. Si bien no son los más adecuados para programas grandes (el carecer de control estático de tipos es un arma de doble filo) para programas medianos o pequeños que impliquen el uso de archivos de texto y necesiten interactuar de manera fluida con el SO son los más adecuados.

Los lenguajes de scripts se conocen con el nombre de *middleware* ya que son capaces de gestionar transferencias de datos entre aplicaciones o entre aplicaciones y el SO. Habitualmente se pueden ampliar mediante plugins o módulos (en el caso de Python) e incluso pueden ser empotrados en otras aplicaciones.

Python es un lenguaje interpretado.

En general, se puede decir que un lenguaje es interpretado si sus instrucciones se ejecutan secuencialmente a partir de código fuente. Para ejecutar el código de un lenguaje interpretado, necesitamos un **intérprete** de ese lenguaje. El intérprete irá recibiendo líneas de código que traducirá a lenguaje máquina para que se ejecute. A diferencia de los lenguajes compilados, no se produce un ejecutable (no, los ficheros *.pyc de Python no son ejecutables). De este modo, de una plataforma a otra, sólo habrá que cambiar el intérprete, no el código.

Sintácticamente, en un lenguaje compilado se require que todos los elementos que intervendrán en el programa estén previamente especificados (declarados). Aunque esta es la norma general de los lenguajes compilados, existen excepciones como VB.

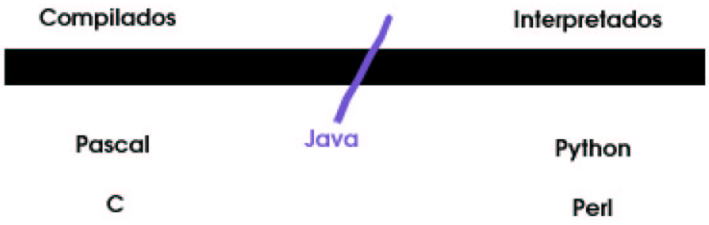
El código interpretado es más lento que el compilado por razones obvias. El código compilado sabe en todo momento la memoria que le hace falta y la que le hará falta a medida que se ejecute. El intérprete, sin embargo, tendrá que realizar estas tareas durante la ejecución, sin ningún tipo de previsión, además de la carga adicional de tener que "traducir" las sentencias del lenguaje, al lenguaje que el intérprete comprende.

Sin embargo, en cuanto a la lentitud podemos realizar una observación general y dos particulares de Python:

Si bien los lenguajes interpretados son más lentos que sus equivalentes compilados, también es cierto que llevan menos tiempo de desarrollo, y que como hemos comentado, se trata de programas cortos, en los que no existen diferencias significativas.

En el caso de Python, si necesitamos velocidad en una rutina, siempre podemos implementar esa rutina en C e importarla a nuestro código. Por otro lado, siempre que importemos con éxito un módulo, el intérprete intentará generar un fichero """"compilado"""", los *.pyc. Se trata de ficheros binarios (bytecode) en el "idioma" del intérprete. Se asemejan al código de bajo nivel que genera un compilador Java. Los *.pyc se cargan con más velocidad, pero los tiempos durante la ejecución son los mismos. En ocasiones se utilizan estos *.pyc para no distribuir los ficheros fuente y dificultar la ingeniería inversa.

Ejemplos de lenguajes compilados son Pascal o C entre muchos otros. Como lenguajes interpretados citaremos Python, Perl o Tcl. A medio camino entre compilados e interpretados se sitúa Java. Los compiladores de Java producen un código de bajo nivel que luego es interpretado por la máquina virtual Java de cada plataforma. Así pues podemos decir que es compilado e interpretado al mismo tiempo.



Python Vs Otros Lenguajes

Veamos algún ejemplo:

JAVA

```
class Hola {
     public static void main(String argument os[]} {
          System.out.println("Hola, mundo");
     }
}
```

PASCAL

```
Program Hola;
Begin
     writeln('Hola, mundo');
End.
```

C

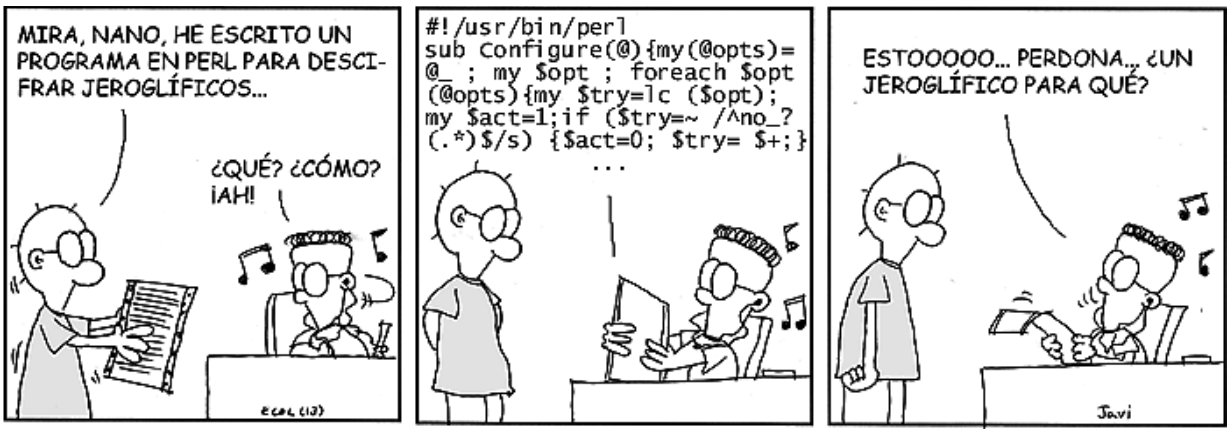
```
void main(){
     printf ("Hola, mundo");
}
```

PYTHON

```
print 'Hola, mundo'
```

Sin duda es el código Python el más sencillo de todos. Los programas en Python suelen ser entre 2 y 4 veces más cortos que sus equivalentes en Java, no sólo por la simplicidad de su sintaxis, sino también por lo avanzado de ciertos tipos (listas, tuplas, diccionarios) que implementa.

¿ Y qué hay de Perl ? Perl es en teoría un lenguaje de características similares a Python. Existe, sobre todo en la comunidad de Python una gran controversia sobre si Python es "mejor" que Perl. La comunidad de Perl, sin embargo no parece muy preocupada en estos temas (una postura bastante más sensata). A simple vista, y sin demasiada experiencia, sólo mirando código, Perl resulta más complejo. Perl se caracteriza por su capacidad de hacer lo mismo de varias formas diferentes. Esto suele confundir a usuarios noveles, e incluso usuarios experimentados tienen problemas para entender código de otras personas (especialmente si se trata de escuelas diferentes). También es cierto que Perl es más eficiente que Python y que su comunidad es más amplia. Sin embargo la de Python es tremendamente activa y amigable. Existe bastante tráfico en las listas de correo (incluso en la española) y contestan cualquier duda amablemente.



Python: Tipos dinámicos y fuertemente tipado.

Python implementa tipos dinámicos. Esto quiere decir que el tipo de un objeto (en Python todo son objetos) se comprueba dinámicamente cada vez que es necesario dicho objeto. A un objeto se le asigna un tipo en el momento en que es asignado. Pero podemos realizar asignaciones de diferentes tipos. Esto significa que un mismo identificador puede referenciar objetos de tipo diferente en distintos puntos del programa. De esta forma es legal

```
>>> a = 7
>>> # a es un entero
...
>>> a = 'pepe'
```

```
>>> # a es una cadena
...
>>> a = [1,7,'pepe']
>>> # a es una lista
...
```

En lenguajes compilados, suele implementarse tipos de datos estáticos. Esto significaría que durante la vida de un programa, una variable no puede cambiar de tipo.

Pese a sus tipos dinámicos, Python es un lenguaje fuertemente tipado. Un lenguaje es fuertemente tipado si sus tipos se mantienen de manera consistente (en Python si no hay una nueva asignación). Esto quiere decir que no podemos sumar una cadena y un entero, por ejemplo:

```
>>> 'pepe' + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

provoca un error. Es necesario realizar una conversión explícita de tipos:

```
 >>> 'pepe' + str(1)
'pepe1'
```

Otros ejemplos de lenguajes fuertemente tipados son Java, Pascal o C. Un lenguaje débilmente tipado es Visual Basic (Basic). En VB se permite concatenar la cadena '12' con el entero 3 y después tratar el conjunto como un entero sin conversión de tipos.

Python como lenguaje orientado a objetos

Python es un lenguaje que no sólo soporta la programación orientada a objetos, sino que la aprovecha hasta sus últimas consecuencias. TODO en Python son objetos. Un entero es un objeto, una cadena es un objeto, una función es un objeto...

Es necesario entender el concepto de objeto de modo adecuado. En general se define objeto como instancia de una clase, sin embargo esta definición no es correcta en Python. No todos los tipos son clases: los tipos internos, como listas y enteros no lo son, ni siquiera otros tipos como los ficheros. Sin embargo, todos los tipos de Python comparten algo de semántica común, descrita adecuadamente mediante la palabra "objeto".

Los objetos en Python tienen individualidad. Podemos asociar diferentes nombres a un mismo objeto, incluso en diferentes ámbitos. Esto se conoce como "alias". Es conveniente tener en cuenta que al generar un "alias" estamos referenciando al mismo objeto, no una copia del mismo. Es importante tener este aspecto en cuenta para evitar sorpresas:

```
>>> a = [1, 'pepe', 1235]
>>> a
```

```
[1, 'pepe', 1235]
>>> b = a
>>> b
[1, 'pepe', 1235]
>>> b[1] = 'paco'
>>> b
[1, 'paco', 1235]
>>> a
[1, 'paco', 1235]
```

En principio se puede pensar que a[1] debería ser 'pepe' pero no es así. Esto sucede porque en la implementación, al realizar la asignación *b = a* lo que en realidad se hace es crear un "alias" o puntero al objeto lista. Aunque el uso de punteros es transparente al usuario, es conveniente tener una idea del modo de funcionamiento interno de Python.

Un lenguaje se dice Orientado a Objetos si hace uso de la herencia. Python implementa herencia múltiple, es decir, una clase puede heredar de varias clases. El modo de buscar métodos o atributos heredados se realiza en profundidad. Esto quiere decir que si tenemos:

```
class nombreClaseDerivada(Base1, Base2, Base3):
    <sentencia-1>
.
    .
    .
    <sentencia-N>
```

tendrá prioridad el quinto ascendiente de la primera clase a la segunda clase.

Un poco de sintaxis

La sintaxis de Python es realmente sencilla. No hay palabras reservadas para iniciar o finalizar bloques ni programas, no es necesario declarar variables...

Podemos usar Python como calculadora:

```
>>> 4 + 8
12
>>> 3 * 9
27
>>> 3846210497830509432759438593476984 56 *
439857943347551473548745328745873 45
169178623925748996783531037531536946475006591736946891660796164
93639320L
>>>
>>> 3 / 2
1
>>> 3.0 / 2
1.5
```

```
>>> 10 % 3
1
```

No existen desbordamientos (los enteros pueden ser arbitrariamente grandes) y la división entera redondea hacia abajo.

Veamos ahora los bucles y sentencias de decisión:

```
>>> while a < 10 :
...     print a
...     a = a + 3
...
1
4
7
```

El bucle while funciona de modo similar al de otros lenguajes.

```
>>> if a != 10 :
...     pass
... else :
...     print a
...
10
```

La sentencia if tiene el comportamiento esperado.

```
>>> for x in range(5):
...     print x
...
0
1
2
3
4

>>> secuencia = [ 1,'pepe',4.2,['juan','paco', 7],2]
>>> for x in secuencia:
...     print x
...
1
pepe
4.2
['juan', 'paco', 7]
2
```

El for es un poco diferente a los for habituales. For en Python exige una secuencia tras la palabra in. Por secuencia entenderemos listas o tuplas. Ya hablaremos de ellas. La instrucción range sirve para generar una secuencia del 0 al valor que fijemos. También es posible pasarle

un intervalo para generar una secuencia entre los valores que pasamos. la variable del for (en este caso x) tomará cada valor de la secuencia. Esto se aprecia mejor en el segundo ejemplo. Vemos la potencia que ofrece Python a la hora de construir tipos.

```
>>> a = 3
>>> while a != 0 :
...     print a
...     a = a - 1
... else :
...     print 'Ya terminamos el bucle'
...
3
2
1
Ya terminamos el bucle
>>> a = 3
>>> while a != 0:
...     print a
...     a = a - 1
...     if a == 0:
...             break
... else:
...     print 'Ya terminamos el bucle'
...
3
2
1
```

Es posible incluir una clausula else al final de un bucle while. Esta se ejecutará al finalizar el while a no ser que hayamos abandonado dicho bucle con una sentencia break.

Si nos fijamos un poco en el código visto hasta ahora, observamos que mantiene un sangrado coherente en cada uno de los bloques. Por ejemplo en Pascal tenemos:

```
while (condición) do
Begin
     sentencia 1;
     ...
     sentencia n;
End;
```

Observamos que existen delimitadores de bloque que entiende la máquina (Begin y End, { y } en C) y delimitadores de bloque que entendemos los humanos (el sangrado). Estos últimos no son obligatorios, sólo sirven para hacer el código más legible. En cierto modo Begin/End y el sangrado representan el concepto de bloque, son redundantes. Python no utiliza palabras reservadas para delimitar bloques, utiliza el sangrado. Es por ello que el código Python presenta siempre una estructura muy ordenada (de otro modo el intérprete daría error). Veamos la confirmación de que los delimitadores no son necesarios:

```
>>> for x in range(10):
...     print 'Número de la secuencia:', x
...     if x % 2 == 0 :
...             print '%d es par' % x
...             if not(x % 4) :
...                     print '%s es múltiplo de 4' % str(x)
...
Número de la secuencia: 0
0 es par
0 es múltiplo de 4
Número de la secuencia: 1
Número de la secuencia: 2
2 es par
Número de la secuencia: 3
Número de la secuencia: 4
4 es par
4 es múltiplo de 4
Número de la secuencia: 5
Número de la secuencia: 6
6 es par
Número de la secuencia: 7
Número de la secuencia: 8
8 es par
8 es múltiplo de 4
Número de la secuencia: 9
```

Como vemos, es el propio Python el que nos obliga a adoptar buenas técnicas de identación. El resultado es un código más claro. Este ejemplo merece algún comentario adicional:

- `if x % 2 == 0 :`
  El operador % es el módulo. La igualdad se representa con == mientras que la asignación con =. Los dos puntos : indican inicio de un bloque. La siguiente instrucción ha de ir un paso más sangrada que la actual. Si no quisiéramos ejecutar ninguna instrucción utilizaríamos `pass`.

- `print '%d es par' % x`
  Un modo de escribir las salidas como en C. Se escribe la cadena con los correspondientes identificadores de tipo, se utiliza el símbolo % y después los identificadores de las variables. No entraremos en detalles.

- `if not(x % 4) :`
  Tanto en Python como en C, 0 es falso y lo demás es cierto. Esto se puede utilizar como en este caso. Si es múltiplo de 4, x%4 es 0 y con el not tendríamos not(falso) que es cierto.

Hemos comentado con anterioridad de los tipos avanzados de datos que implemente Python directamente. Principalmente son tres: listas, tuplas y diccionarios.
Las listas son estructuras de datos que almacenan secuencias de cualquier tipo de datos. Son

ampliamente utilizadas en Python. Son objetos mutables, esto es, podemos modificar cada uno de sus componentes:

```
>>> lista = ['pepe',1,6.5,[1,2],38264274572663375]
>>> lista[0]
'pepe'
>>> lista[3]
[1, 2]
>>> lista[:3]
['pepe', 1, 6.5]
>>> lista[1:3]
[1, 6.5]
>>> lista[1:]
[1, 6.5, [1, 2], 38264274572663375L]
>>> lista.append('nuevo_elemento')
>>> lista
['pepe', 1, 6.5, [1, 2], 38264274572663375L, 'nuevo_elemento']
>>> lista[2] = '3265732'
>>> lista
['pepe', 1, '3265732', [1, 2], 38264274572663375L,
'nuevo_elemento']
```

Las tuplas son similares a las listas, salvo que en este caso no son mutables, no podemos modificar sus elementos :

```
>>> tupla = (1,4324,233.45,'hola')
>>> tupla[1] = 'noooo'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
```

Los diccionarios o memorias asociativas son pares clave-valor donde clave puede ser cualquier objeto no mutable. Las claves han de ser únicas dentro de un diccionario. El mejor modo de comprender su funcionamiento es viendo un ejemplo:

```
>>> dic = { 233 : 'hola, que tal' , 'pepe' : 'gracias por todo'
, (1,2) : 'pues si'}
>>> dic
{233: 'hola, que tal', (1, 2): 'pues si', 'pepe': 'gracias por
todo'}
>>> dic.keys()
[233, (1, 2), 'pepe']
>>> dic.has_key(233)
1
```

Para finalizar echaremos un vistazo a la definición de funciones:

```
>>> def f(x):
...     print x
...
>>> f('pepe')
pepe
>>> f(9)
9
>>> f([1,3,'paco',45.345])
[1, 3, 'paco', 45.344999999999999]
```

Una función se define con la palabra reservada `def` seguida del identificador de la función, y entre paréntesis los parámetros sin indicar su tipo. Si no hay parámetros se ponen los paréntesis vacíos (). Como se observa, se puede llamar a una función pasando parámetros de cualquier tipo. Incluso podemos hacer cosas como:

```
>>> f(f('pepe'))
pepe
None
```

Dado que no hemos especificado valor de retorno para f, la segunda llamada se hace con un argumento vacío (None).

Como ya hemos comentado, todo en Python son objetos, incluso las funciones. Sus atributos y métodos son:

```
>>> dir(f)
['__call__', '__class__', '__delattr__', '__dict__', '__doc__',
'__get__', '__getattribute__', '__hash__', '__init__',
'__name__', '__new__', '__reduce__', '__repr__', '__setattr__',
'__str__', 'func_closure', 'func_code', 'func_defaults',
'func_dict', 'func_doc', 'func_globals', 'func_name']
```

Un atributo especialmente interesante es `__doc__`. En este atributo se almacenan las llamadas cadenas de documentación. Estas cadenas se colocan en la siguiente línea a la definición de la función. Sirven para aclarar qué hace la función y son ampliamente utilizadas por los IDEs (*entornos integrados de desarrollo*) para presentar ayuda contextual. Veamos un ejemplo de `__doc__`:

```
>>> def g():
...     "Esto no hace nada"
...     return 7
...
>>> g()
7
>>> g.__doc__
'Esto no hace nada'
```

Al igual que para el resto de objetos, podemos crear un alias a nuestra función:

```
>>> def f():
...     print 'Hola'
...
>>> f()
Hola
>>> mi_alias = f
>>> mi_alias()
Hola
```

No entraremos en más detalles, pero las funciones en Python admiten argumentos por defecto, por clave e incluso admiten números indeterminados de argumentos. Tampoco hablaremos aquí de las formas lambda ni de las herramientas de programación funcional. Se invita al lector que las consulte en la guía de aprendizaje del lenguaje.

Módulos de interés

Python viene acompañado de una extensa colección de módulos. Entre ellos destacamos los siguientes:

- *calendar*
  Su propio nombre lo indica, funciones de calendario.

- *commands*
  Es el módulo que se utiliza para usar los comandos propios de sistemas Unix. A no ser que sea especialmente necesario, es preferible usar el módulo *os* ya que es independiente de la máquina.

- *curses*
  ¿Necesita comentarios?

- *getopt*
  Para pasar argumentos en línea de comandos

- *math* y *cmath*
  Librería matemática estándar y librería para números complejos.

- *ftplib*
  Biblioteca que de forma sencilla permite la conexión e interacción con un servidor FTP remoto. Muy útil y sencilla.

- *os*
  El módulo de comunicación con el SO estándar. Muy potente y flexible, implementa un interfaz de acceso al sistema operativo independientemente del que sea.

- *pickle* y *cpickle*
  Este módulo es una joya. Permite guardar cualquier estructura de datos, por avanzada

que sea en un fichero, incluso en formato texto.

- *random*
  Incluye numerosas funciones de generación de números aleatorios.

- *strings*
  Funciones de tratamiento y manejo de cadenas.

Para una completa referencia, consultar el índice de módulos de la documentación de Python.

Bibliografía.

La documentación de Python es excelente. Sólo con leer la Guía de aprendizaje (que incluso es amena) se puede tener un cierto dominio del lenguaje. El resto es práctica.

Guido van Rossum. **Guía de Aprendizaje de Python.** Traducida por Marcos Sánchez.
http://pyspanishdoc.sourceforge.net/tut/tut.html

Guido van Rossum. **Referencia de la Biblioteca de Python**
http://pyspanishdoc.sourceforge.net/lib/lib.html

Índice Global de Módulos
http://pyspanishdoc.sourceforge.net/modindex.html

Mark Pilgrim **Dive into Python.** Traducido por Francisco Callejo
http://sourceforge.net/projects/diveintopython/

Javier Malonda **Tira cómica sobre Perl**
http://tira.escomposlinux.org/

Lista de correo Python-es
http://listas.aditel.org/listinfo.py/python-es

Enlaces, código fuente y este documento.
http://users.servicios.retecal.es/tjavier/python/Un_poco_de_Python.html

Acerca de este documento...

v1.0: 25 de Mayo de 2002
v1.01: 27 de Mayo de 2002

Autor: Tomás Javier Robles Prado
Email: tjavier@usuarios.retecal.es
Puede descargar la última versión de este documento en
http://users.servicios.retecal.es/tjavier/python/Un_poco_de_Python.html

Gracias **Chema Cortés** por sus puntualizaciones sobre lenguajes interpretados y compilados así como sobre lenguajes de scripts.