Modismos y Anti-Modismos en Python

Por Moshe Zadka

Traducción al castellano de "Idioms and Anti-Idioms in Python"
por Raúl González Duque el día 9 de Abril de 2008

Resumen

Este documento puede considerarse un compañero del tutorial de Python. Muestra cómo utilizar Python, y, casi incluso más importante, cómo no usar Python.

Construcciones del lenguaje que no deberías usar

Aunque Python tiene relativamente pocas trampas o gotchas comparado con otros lenguajes, sigue teniendo algunas construcciones que sólo son de utilidad en situaciones muy específicas, o que son sencillamente peligrosas.

from modulo import *

En definiciones de funciones

`from modulo import *` no es válido dentro de definiciones de funciones. Aunque muchas versiones de Python no comprueban esta condición, no deja de ser inválido, de la misma forma que tener un buen abogado no le transforma a uno en inocente. Nunca lo utilices de esta forma. Incluso en las versiones en las que se aceptaba, producía que la función se ejecutara mucho más lentamente, porque el compilador no podía estar seguro de qué nombres eran locales y cuáles globales. En Python 2.1 el uso de esta construcción produce warnings y algunas veces también errores.

A nivel de módulo

Aunque `from modulo import *` es perfectamente válido a nivel de módulo, normalmente su uso sigue siendo una mala idea. En primer lugar porque al utilizarlo perdemos una importante propiedad de Python - y es que puedes saber dónde se define cada nombre de primer nivel simplemente usando la función de búsqueda de tu editor. Además te arriesgas a encontrarte con errores en el futuro, si alguno de los módulos incorpora nuevas funciones o clases.

Una de las preguntas más horribles con las que te puedes encontrar en los grupos de noticias es por qué el siguiente código no funciona:

```
view plain  copy to clipboard  print  ?
01. f = open("www")
02. f.read()
```

Por supuesto, funciona perfectamente (asumiendo que tienes un archivo llamado "www"). Pero no funciona si tenemos un `from os import *` en algún lugar del módulo. El módulo `os` tiene una función llamada `open()` que devuelve un entero. Aunque algunas veces pueda resultar de utilidad, sobre escribir las funciones por defecto es uno de los efectos colaterales más molestos.

Recuerda, nunca estás seguro de los nombres que exporta un módulo, así que importa sólo lo que necesites -- `from modulo import nombre1, nombre2`, o manten cada cosa en su módulo y accede a ellos cuando lo necesites — `import modulo;print modulo.nombre`.

Cuándo es adecuado

Hay situaciones en las que el uso de `from modulo import *` es adecuado:

- En el intérprete interactivo. Por ejemplo el escribir `from math import *` transforma a Python en una calculadora científica increíble.
- Al extender un módulo en C con un módulo en Python.
- Cuando el módulo especifica que es seguro usar `from import *`.

exec, execfile() y amigos, sin adornos

El término "sin adornos" se refiere al uso sin indicar un diccionario explicitamente, en cuyo caso estas costrucciones evaluan el código en el entorno *actual*. Esto es peligroso por las mismas razones por las que lo es `from import *` -- puede modificar variables que estés utilizando y estropear el resto del código. Simplemente evita usarlo.

Mal:

```
>>> for name in sys.argv[1:]:
```

```
>>>     exec "%s=1" % name
>>> def func(s, **kw):
>>>     for var, val in kw.items():
>>>         exec "s.%s=val" % var  # invalido!
>>> execfile("handler.py")
>>> handle()
```

Bien:

```
>>> d = {}
>>> for name in sys.argv[1:]:
>>>     d[name] = 1
>>> def func(s, **kw):
>>>     for var, val in kw.items():
>>>         setattr(s, var, val)
>>> d={}
>>> execfile("handle.py", d, d)
>>> handle = d['handle']
>>> handle()
```

from modulo import nombre1, nombre2

Esta es una advertencia más ligera que las anteriores pero aun así es algo que no deberías usar a menos que tengas buenas razones para hacerlo. La razón por la que se trata de una mala idea es porque de repente tienes un objeto que vive en dos espacios de nombres distintos. Cuando el ligado de un espacio de nombres cambia, el otro no lo hará, por lo que habrá una discrepancia entre ambos. Esto ocurre, por ejemplo, cuando se recarga un módulo, o cuando se cambia la definición de una función en tiempo de ejecución.

Mal:

view plain copy to clipboard print ?

```
01. # foo.py
02. a = 1
03.
04. # bar.py
05. from foo import a
06. if algo():
07.     a = 2 # cuidado: foo.a != a
```

Bien:

view plain copy to clipboard print ?

```
01. # foo.py
02. a = 1
03.
04. # bar.py
05. import foo
06. if algo():
07.     foo.a = 2
```

except:

Python cuenta con una cláusula `except:` que sirve para capturar todas las excepciones. Como *todos* los errores en Python producen excepciones, esto provoca que muchos errores de programación parezcan errores en tiempo de ejecución, y dificulta el trabajo de depuración.

En el siguiente código podemos ver un buen ejemplo:

```
view plain  copy to clipboard  print  ?
01. try:
02.     foo = opne("archivo") # "open" esta mal escrito
03. except:
04.     sys.exit("no se pudo abrir el archivo")
```

La segunda línea lanza una excepción de tipo `NameError` el cual se captura por la clausula except. El programa terminará, y no tendrás ni idea de que esto no tiene nada que ver con que se pueda o no leer "archivo".

El siguiente ejemplo está mejor escrito

```
view plain  copy to clipboard  print  ?
01. try:
02.     foo = opne("archivo") # lo cambiaremos a "open" en cuanto ejecutemos
03. except IOError:
04.     sys.exit("no se pudo abrir el archivo")
```

Hay algunas situaciones en las que el uso de la clausula `except:` es adecuado, como en el caso de un framework al ejecutar retrollamadas, no queremos que ninguna retrollamada moleste al framework.

Excepciones

Las excepciones son una característica muy útil de Python. Deberías aprender a lanzarlas cuando ocurra algo inesperado, y capturarlas sólo en los lugares en los que puedas hacer algo por remediarlas.

El siguiente es un anti-modismo muy popular:

```
view plain  copy to clipboard  print  ?
01. def get_status(archivo):
02.     if not os.path.exists(archivo):
03.         print "no se encontro el archivo"
04.         sys.exit(1)
05.     return open(archivo).readline()
```

Supongamos que el archivo se borra justo entre la ejecución de `os.path.exists()` y

la llamada a `open()`. Esto haría que la última línea lanzara una excepción de tipo `IOError`. Lo mismo ocurriría si el archivo existiera pero sólo tuviera permisos de lectura. Como al probar la ejecución del programa no se aprecia ningún error, el resultado de la prueba será satisfactorio, y se mandará el código a producción. Entonces el usuario se encuentra con un `IOError` que no se ha capturado y tiene que lidiar con mensajes extraños de trazado de pila.

Aquí tenemos una forma mejor de hacerlo.

```
view plain  copy to clipboard  print  ?
01. def get_status(file):
02.     try:
03.         return open(file).readline()
04.     except (IOError, OSError):
05.         print "no se encontro el archivo"
06.         sys.exit(1)
```

En esta versión hay dos posibilidades, o bien el archivo se abre y se lee la línea (por lo que funciona incluso en conexiones NFS o SMB poco fiables), o se muestra el mensaje y se aborta la ejecución.

Aun así, `get_status()` asume demasiadas cosas -- que sólo se utilizará en un script que no se ejecutará por mucho tiempo y no, por ejemplo, en un programa que corra durante días en un servidor. Por supuesto al llamar a la función se podría hacer algo como

```
view plain  copy to clipboard  print  ?
01. try:
02.     status = get_status(log)
03. except SystemExit:
04.     status = None
```

asi que intenta usar cuantas menos clausulas `except` mejor en tu código -- normalmente estas consistirán en un `except` que capture todo en `main()`, o en llamadas internas que siempre deberían ejecutarse con éxito.

Por lo tanto la mejor versión sería probablemente

```
view plain  copy to clipboard  print  ?
01. def get_status(file):
02.     return open(file).readline()
```

El código que llama a la función puede lidiar con la excecpión si lo necesita (por ejemplo, si prueba la función con varios archivos en un bucle), o simplemente dejar que la excepción se propague.

La última versión tampoco es muy buena -- debido a detalles de implementación,

el archivo no se cerrará cuando se lance una excepción hasta que el manejador termine, y puede que no ocurra en alguna implementación que no se base en C (como por ejemplo Jython)

view plain copy to clipboard print ?

```
01. def get_status(file):
02.     fp = open(file)
03.     try:
04.         return fp.readline()
05.     finally:
06.         fp.close()
```

Usando las baterías

De vez en cuando la gente intenta reescribir cosas que ya se encuentran en las librerías estándar de Python, y normalmente el resultado es pobre. Normalmente es mucho mejor utilizar la extensa librería que incluye Python por defecto que reinventar la rueda.

Un módulo muy útil que poca gente conoce es `os.path`. Este módulo facilita una aritmética de rutas adecuada a tu sistema operativo, y normalmente será una mejor opción que cualquier que puedas crear.

Compara:

view plain copy to clipboard print ?

```
01. # arg!
02. return dir+"/"+file
03. # mejor
04. return os.path.join(dir, file)
```

Otras funciones útiles en `os.path` son `basename()`, `dirname()` y `splitext()`.

Hay otras muchas funciones incluídas por defecto que la gente parece no conocer por alguna razón: `min()` y `max()`, por ejemplo, pueden hallar los valores mínimos/máximos de cualquier secuencia cuyos elementos sean comparables, pero a pesar de ello mucha gente escribe sus propias versiones de `min()` y `max()`. Otra función muy útil es `reduce()`. Un uso clásico de `reduce()` es el siguiente

view plain copy to clipboard print ?

```
01. import sys, operator
02. nums = map(float, sys.argv[1:])
03. print reduce(operator.add, nums)/len(nums)
```

Este pequeño script imprime la media de todos los números pasados por línea de comandos. La función `reduce()` suma todos los números, y el resto es solo algo de pre y post procesado.

De la misma forma, observa que `float()`, `int()` y `long()` aceptan todos argumentos de tipo cadena, por lo que se pueden utilizar para parsear -- asumiendo que estes dispuesto a enfrentarte a las excepciones `ValueError` que pueden lanzar.

Usando la barra invertida para continuar sentencias

Dado que Python interpreta el caracter de nueva línea como una marca de fin de sentencia, y dado que a menudo es más cómodo dividir las sentencias en varias líneas, mucha gente hace algo como esto:

```
view plain  copy to clipboard  print  ?

01. if foo.bar()['first'][0] == baz.quux(1, 2)[5:9] and \
02.    calculate_number(10, 20) != forbulate(500, 360):
03.       pass
```

Deberías tener en cuenta que esto es peligroso: un caracter de espacio perdido después del `\` causaría que la línea no fuera correcta. En este caso, al menos se trataría de un error de sintaxis, pero si el código fuera:

```
view plain  copy to clipboard  print  ?

01. value = foo.bar()['first'][0]*baz.quux(1, 2)[5:9] \
02.         + calculate_number(10, 20)*forbulate(500, 360)
```

el error sería mucho menos evidente.

Normalmente es mucho mejor usar la continuación implícita que se da dentro de los paréntesis.

Esta versión es a prueba de balas:

```
view plain  copy to clipboard  print  ?

01. value = (foo.bar()['first'][0]*baz.quux(1, 2)[5:9]
02.         + calculate_number(10, 20)*forbulate(500, 360))
```