- Bitácora
- Traducciones
- Tutorial de Python

Programa como un Pythonista: Python Idiomático

Por David Goodger

Traducción al castellano de "Code Like a Pythonista: Idiomatic Python"
por Raúl González Duque el día 8 de Junio de 2008

En este tutorial interactivo, trataremos en profundidad algunos modismos y técnicas esenciales de Python, añadiendo así algunas herramientas útiles a tu caja de herramientas.

Mis credenciales: Soy

- un habitante de Montreal,
- padre de dos grandes niños, marido de una mujer muy especial,
- programador en Python a tiempo completo,
- autor del proyecto Docutils y de reStructuredText,
- un editor de las Propuestas de Mejora de Python (Python Enhancement Proposals o PEPs),
- uno de los organizadores de la PyCon 2007, y coordinador de PyCon 2008,
- un miembro de la Fundación de Software Python,
- Director de la Fundación el año pasado, y su Secretario actual.

En el tutorial que presenté en la PyCon 2006 (llamado Procesamiento de Datos y Texto), me sorprendí de las reacciones que coseché con algunas técnicas que utilicé y que yo pensaba que eran bastante conocidas. Pero muchos de los asistentes no conocían estas herramientas que los programadores experimentados de Python utilizan sin pensar.

Muchos de vosotros habréis visto algunas de estas técnicas y modismos anteriormente. Con un poco de suerte aprenderéis algunas técnicas que no habíais visto antes y puede que también algo nuevo acerca de las que ya conocíais.

El Zen de Python (1)

Estas son las guías generales de Python, pero están abiertas a interpretación. Se requiere de sentido del humor para interpretarlas correctamente.

Si utilizas un lenguaje de programación cuyo nombre se debe a un grupo de cómicos, es conveniente tener sentido del humor.

> Hermoso es mejor que feo.
> Explícito es mejor que implícito.
> Simple es mejor que complejo.
> Complejo es mejor que complicado.
> Plano es mejor que anidado.
> Disperso es mejor que denso.
> La legibilidad cuenta.
> Los casos especiales no son suficientemente especiales como para romper las reglas.
> Aunque lo pragmático gana a la pureza.
> Los errores nunca deberían dejarse pasar silenciosamente.
> A menos que se silencien explícitamente.
>
> ...

El Zen de Python (2)

> Cuando te enfrentes a la ambigüedad, rechaza la tentación de adivinar.
> Debería haber una -- y preferiblemente sólo una -- manera obvia de hacerlo.
> Aunque puede que no sea obvia a primera vista a menos que seas holandés. (NT: Guido van Rossum, creador de Python, es holandés).
> Ahora es mejor que nunca.
> Aunque muchas veces nunca es mejor que *ahora mismo*.
> Si la implementación es difícil de explicar, es una mala idea.
> Si la implementación es sencilla de explicar, puede que sea una buena idea.
> Los espacios de nombres son una gran idea -- ¡tengamos más de esas!
>
> —Tim Peters

Este poema en particular comenzó como una especie de broma, pero contiene grandes verdades acerca de la filosofía detrás de Python. El Zen de Python ha sido formalizado en el PEP 20, en cuyo resumen podemos leer:

> Hace mucho tiempo el entusiasta de Python Tim Peters plasmó de forma concisa los principios de diseño por los que guiarse al escribir en Python según el BDFL (NT: Benevolent Dictator for Life o Dictador Benévolo de por Vida, en este caso Guido van Rossum, el creador de Python) en 20 aforismos, de los cuales sólo 19 han pasado a forma escrita.
>
> —http://www.python.org/dev/peps/pep-0020/

Puedes decidir por ti mismo si te consideras un "Pythoneer" o un "Pythonista". Los términos tienen connotaciones algo distintas.

En caso de duda:

```
import this
```

Pruébalo en el intérprete de Python:

```
>>> import this
```

Aquí tienes otro huevo de pascua:

```
>>> from __future__ import braces
  File "<stdin>", line 1
SyntaxError: not a chance
```

¡Menudo grupo de humoristas! :-)

Estilo de Programación: La legibilidad cuenta

> Los programas deben escribirse para que los lean las personas, y sólo de forma circunstancial para que los ejecuten las máquinas.
>
> —Abelson y Sussman, *Estructura e Interpretación de Programas de Computadora*

Intenta que tus programas sean fáciles de leer y obvios.

PEP 8: Guía de Estilo del Código Python

Una lectura muy interesante:

http://mundogeek.net/traducciones/guia-estilo-python.htm

PEP = Python Enhancement Proposal (Propuesta de Mejora de Python)

Un PEP es un documento de diseño que proporciona información a la comunidad Python, o describe una nueva característica para Python o sus procesos o entorno.

La comunidad Python tiene sus propios estándares en lo que al estilo del código se refiere, listados en el PEP 8. Estos estándares son distintos a los de otras comunidades, como C, C++, C#, Java, VisualBasic, etc.

Debido a que la indentación y los espacios en blanco son tan importantes en Python, la Guía de Estilo del Código Python es prácticamente un estándar. ¡Sería inteligente seguir esta guía! La mayor parte de los proyectos de código abierto y los proyectos caseros siguen la guía de estilo casi al pie de la letra (o eso espero).

Espacios en blanco 1

- 4 espacios por cada nivel de indentación.
- Evita los caracteres de tabulación.
- **Nunca** mezcles caracteres de tabulación y espacios.
- Una línea en blanco entre funciones.
- Dos líneas en blanco entre clases.

Espacios en blanco 2

- Añade un espacio después de las comas "," en diccionarios, listas, tuplas, y listas de parámetros, y después de los dos puntos ":" en los diccionarios, pero no antes.
- Pon espacios alrededor de las asignaciones y las comparaciones (excepto en las listas de parámetros).
- No coloques espacios justo después de abrir un paréntesis o antes de cerrar un paréntesis o justo antes de una lista de parámetros.
- No dejes espacios al inicio o final de las cadenas de documentación.

```
def haz_cuadrados(clave, valor=0):
    """Devuelve un diccionario y una lista..."""
    d = {clave: valor}
```

```
    l = [clave, valor]
    return d, l
```

Nombres

- `minusculas_con_guiones` para funciones, métodos, atributos
- `minusculas_con_guiones` o `TODO_MAYUSCULAS` para las constantes
- `PalabrasEnMayusculas` para las clases
- `camelCase` **sólo** si es necesario adaptarse a convenciones que ya se utilizaban en el código
- Atributos: `interfaz`, `_interno`, `__privado`

  Pero intenta evitar la forma `__privado`. Yo en particular nunca la utilizo. Creeme. Si la usas, en el futuro lo lamentarás.

Líneas largas

Mantén las líneas de código por debajo de los 80 caracteres.

Aprovecha que las sentencias continúan en la siguiente línea de forma implícita dentro de paréntesis/llaves/corchetes:

```
def __init__(self, primero, segundo, tercero,
             cuarto, quinto, sexto):
    salida = (primero + segundo + tercero
              + cuarto + quinto + sexto)
```

Utiliza las barras invertidas como último recurso:

```
ParteIzquierda.muy_larga \
    = parte_derecha.incluso_mas()
```

Las barras invertidas no son un mecanismo robusto; tienen que situarse al final de la línea. Si añadieras un espacio después de la barra invertida, dejaría de funcionar. Además, afean el código.

Cadenas largas

El parseador concatena los literales de cadena adyacentes automáticamente:

```
>>> print 'u' 'n' "o"
```

```
uno
```

Los espacios entre las cadenas no son necesarios, pero ayudan a la legibilidad. Se puede utilizar cualquier tipo de comillas:

```
>>> print 't' r'\/\/' """o"""
t\/\/o
```

La cadena con una "r" como prefijo es una cadena "raw" (cruda). En las cadenas raw las barras invertidas no se consideran caracteres de escape. Son útiles para las expresiones regulares y las rutas del sistema de ficheros de Windows.

Ten en cuenta que las variables cadena **no** se concatenan automaticamente:

```
>>> a = 'tres'
>>> b = 'cuatro'
>>> a b
  File "<stdin>", line 1
    a b
      ^
SyntaxError: invalid syntax
```

Esto se debe a que la concatenación automática se trata de una característica del parseador/compilador de Python, no del intérprete. Es necesario utilizar el operador "+" para concatenar cadenas en tiempo de ejecución.

```
texto = ('Se pueden crear cadenas mas largas '
        'a partir de varias cadenas cortas.')
```

Los paréntesis permiten que la sentencia continúe en la siguiente línea de forma implícita.

Para las cadenas de varias líneas se utilizan comillas triples:

```
"""Triples
comillas
dobles"""

'''\
Triples
comillas
simples\
'''
```

Observa que en el último ejemplo (el de las triples comillas simples), se utilizan barras invertidas para escapar los caracteres de nueva línea. Esto elimina los retornos de carro extra, proporcionando una estructura muy legible, con el texto y las comillas justificadas a la izquierda. Las barras invertidas se deben colocar al final de las respectivas líneas.

Sentencias compuestas

Bien:

```
if foo == 'bla':
    haz_algo()
haz_primero()
haz_segundo()
haz_tercero()
```

Mal:

```
if foo == 'bla': haz_algo()
haz_primero(); haz_segundo(); haz_tercero()
```

El espacio en blanco y la indentación son indicadores visuales muy buenos del flujo del programa. La indentación de la primera versión muestra al lector que algo está ocurriendo, mientras que la falta de indentación del segundo ejemplo hace que el "if" pase más desapercibido.

Utilizar varios estamentos en la misma línea es un pecado capital. En Python, *la legibilidad cuenta*.

Cadenas de documentación y comentarios

Cadenas de documentación = **Cómo usar** el código

Comentarios = **Por qué** (razones) y **cómo funciona el código**

Las cadenas de documentación explican **cómo** usar el código, y están destinadas a los **usuarios** del código. Usos de las cadenas de documentación:

- Explicar el propósito de la función aunque pueda parecer obvia para tí, ya que puede no ser tan obvia para otras personas.
- Describir los parámetros que espera la función, los valores de retorno, y las excepciones que puede lanzar.
- Si el método está fuertemente acoplado con un único llamador, menciónalo (aunque deberías tener cuidado ya que puede cambiar más tarde).

Los comentarios explican el **por qué**, y están dirigidos a **las personas que van a mantener** tu código. Puedes incluir notas para tí mismo, como:

```
# !!! BUG: ...
```

```
# !!! FIX: Esto es una chapuza

# ??? ¿Por qué está esto aquí?
```

Ambos grupos te incluyen a ti, ¡así que asegúrate de escribir buenas cadenas de documentación y comentarios!

Las cadenas de documentación son muy útiles en el modo interactivo (`help()`) y para sistemas de auto documentación.

Tener cadenas de documentación o comentarios incorrectos es mucho peor que no tenerlos en absoluto. ¡Mantenlos actualizados! Cuando hagas cambios, asegurate de que los comentarios y las cadenas de documentación continúan siendo consistentes con el código, y no lo contradicen.

Existe un PEP dedicado por completo a las cadenas de documentación, el PEP 257, "Convenciones sobre cadenas de documentación":

> http://www.python.org/dev/peps/pep-0257/

Pragmatismo antes que pureza

> La consistencia estúpida es el demonio de las mentes pequeñas.
>
> —Ralph Waldo Emerson

Siempre existen excepciones. Del PEP 8:

> Lo que es más importante: aprende a saber cuándo ser inconsistente -- algunas veces la guía de estilo no debería aplicarse. Cuando tengas dudas, usa tu criterio. Mira otros ejemplos y decide qué parece mejor. ¡Y no dudes en preguntar!
>
> Dos buenas razones para romper una regla en particular:
>
> 1. Cuando aplicar la regla produciría que el código fuera menos legible, incluso para personas que están acostumbradas a leer código que sigue las normas.
> 2. Para ser consistente con código heredado que también rompe las normas (puede que por razones históricas) -- aunque esto también puede verse como una oportunidad de arreglar el desaguisado de otra persona (al más puro estilo XP).

... pero es importante saber establecer límites

Popurrí de modismos

Una selección de idioms o modismos pequeños y útiles.

Pasemos ahora a la parte importante del tutorial: montones de modismos.

Vamos a comenzar con algunos más sencillos e iremos aumentando el nivel poco a poco.

Intercambiar valores

En otros lenguajes:

```
temporal = a
a = b
b = temporal
```

En Python:

```
b, a = a, b
```

Puede que hayas visto esto antes. ¿Pero sabías cómo funciona?

- La **coma** es el operador que se utiliza para construir las tuplas.
- Se crea una tupla a la derecha del igual (empaquetamos los valores en una tupla).
- A la izquierda tenemos otra tupla que es el destino (desempaquetamos la tupla).

La parte derecha se **desempaqueta** asignando los valores a los nombres de la parte izquierda.

Más ejemplos de desempaquetado:

```
>>> l =['David', 'Pythonista', '+1-514-555-1234']
>>> nombre, rango, tlf = l
>>> nombre
'David'
>>> rango
'Pythonista'
>>> tlf
'+1-514-555-1234'
```

Son útiles para usar en bucles que iteran sobre datos estructurados:

`l` (L) es la lista que acabamos de crear (los datos de David). Así que `personas` es una lista que contiene dos elementos, siendo cada uno de ellos una lista de 3

elementos.

```
>>> personas = [l, ['Guido', 'BDFL', 'desconocido']]
>>> for (nombre, rango, tlf) in personas:
...     print nombre, tlf
...
David +1-514-555-1234
Guido desconocido
```

Cada elemento de `personas` se desempaqueta en una tupla `(nombre, rango, tlf)`.

Se puede anidar de forma arbitraria (sólo hay que asegurarse de que la estructura de la parte izquierda y la derecha se correspondan):

```
>>> david, (gnombre, grango, gtlf) = personas
>>> gnombre
'Guido'
>>> grango
'BDFL'
>>> gtlf
'desconocido'
>>> david
['David', 'Pythonista', '+1-514-555-1234']
```

Más acerca de las tuplas

Hemos visto que el operador de construcción de tuplas es la **coma**, no los paréntesis. Por ejemplo:

```
>>> 1,
(1,)
```

El intérprete de Python muestra los paréntesis por claridad, te recomiendo que tú también los utilices:

```
>>> (1,)
(1,)
```

¡No te olvides de la coma!

```
>>> (1)
1
```

En una tupla de un solo elemento, es necesario añadir una coma al final; en tuplas de 2 elementos o más, la coma al final es opcional. Para tuplas vacías se puede utilizar un par de paréntesis para atajar:

```
>>> ()
()
```

```
>>> tuple()
()
```

Un fallo muy común es añadir una coma incluso cuando lo que buscas no es una tupla. Es fácil no darse cuenta de que hay una coma en el código:

```
>>> valor = 1,
>>> valor
(1,)
```

Por lo tanto, si ves una tupla en un lugar donde no debería existir, ¡comprueba las comas!

"_" interactivo

Esta es una característica muy útil aunque, sorprendentemente, muy poca gente la conoce.

En el intérprete interactivo, cada vez que evalúas una expresión o llamas a una función, el resultado se almacena en una variable temporal llamada _ (un guión bajo):

```
>>> 1 + 1
2
>>> _
2
```

_ almacena el valor de la última expresión *impresa*.

Cuando el resultado es `None`, no se imprime nada, por lo que _ no cambia. ¡Qué conveniente!

Esta característica sólo funciona en el intérprete interactivo, no dentro de los módulos.

Es especialmente útil cuando trabajas en un problema de forma interactiva, y quieres almacenar el resultado para el próximo paso:

```
>>> import math
>>> math.pi / 3
1.0471975511965976
>>> angulo = _
>>> math.cos(angulo)
0.50000000000000011
>>> _
0.50000000000000011
```

Construyendo cadenas a partir de subcadenas

Comenzaremos con una lista de cadenas:

```
colores = ['rojo', 'azul', 'verde', 'amarillo']
```

Ahora queremos unir todas las cadenas en una cadena más larga. Especialmente cuando tenemos un gran número de subcadenas...

No hagas esto:

```
resultado = ''
for s in colores:
    resultado += s
```

Esto es muy ineficiente.

Tiene un consumo de memoria enorme y da muy mal rendimiento. Lo que hace el código es calcular, almacenar, y descartar cada paso intermedio.

En su lugar podemos hacer esto:

```
resultado = ''.join(colores)
```

El método `join()` de las cadenas realizan la copia en un solo paso.

Si estás trabajando con unas pocas docenas o cientos de cadenas, puede que no haya mucha diferencia. Pero deberías acostumbrarte a construir las cadenas de forma más efectiva, porque con miles de cadenas o con bucles, hay una gran diferencia.

Construyendo cadenas, Variaciones 1

A continuación tenéis algunas técnicas para usar el método `join()`.

Si quieres espacios entre las subcadenas:

```
resultado = ' '.join(colores)
```

O comas y espacios:

```
resultado = ', '.join(colores)
```

Aquí tenemos un caso muy común:

```
colores = ['rojo', 'azul', 'verde', 'amarillo']
print 'Elige', ', '.join(colores[:-1]), \
      'o', colores[-1]
```

Para que la frase tenga más sentido gramáticamente, pondremos comas entre todos los valores a excepción del último par, donde queremos la palabra "o". El particionado de listas es la herramienta adecuada para el trabajo. Con `[:-1]` obtenemos todos los valores a excepción del último, y los unimos utilizando una coma y un espacio.

Por supuesto, este ejemplo no funcionaría para listas de 0 o 1 elementos.

Salida:

```
Elige rojo, azul, verde o amarillo
```

Construyendo cadenas, Variaciones 2

Si necesitas aplicar una función para generar las subcadenas:

```
resultado = ''.join(funcion(i) for i in items)
```

Esto requiere del uso de una *expresión generadora*, que cubriremos en profundidad más tarde.

Si necesitas construir las subcadenas de forma incremental, primero añádelas a una lista:

```
items = []
...
items.append(item)  # varias veces
...
# items ya esta completo
resultado = ''.join(funcion(i) for i in items)
```

Añadimos las partes a una lista de forma que podamos usar el método `join` por cuestiones de eficiencia.

Utiliza `in` cuando sea posible (1)

Bien:

```
for clave in d:
    print clave
```

- Normalmente `in` es más rápido.
- Este patrón funciona con elementos almacenados en cualquier tipo de contenedor (como listas, tuplas, y conjuntos).

- `in` también es un operador (como veremos más tarde).

Mal:

```
for clave in d.keys():
    print clave
```

Esto se limita a objetos que cuenten con un método `keys()`.

Utiliza `in` cuando sea posible (2)

Pero `.keys()` es **necesario** cuando se modifica el diccionario:

```
for clave in d.keys():
    d[str(clave)] = d[clave]
```

`d.keys()` crea una lista estática de las claves del diccionario. De otra forma, se lanzaría una excepción "RuntimeError: dictionary changed size during iteration" (el tamaño del diccionario cambió durante la iteración).

Por motivos de consistencia, utiliza `clave in dict`, no `dict.has_key()`:

```
# haz esto:
if clave in d:
    ...hacer algo con d[clave]

# no esto:
if d.has_key(clave):
    ...hacer algo con d[clave]
```

En este ejemplo `in` es un operador.

El método `get` de los diccionarios

A menudo tenemos que inicializar las entradas de un diccionario antes de utilizarlo:

Esta es la forma naïf de hacerlo:

```
navs = {}
for (portfolio, acciones, posicion) in datos:
    if portfolio not in navs:
        navs[portfolio] = 0
    navs[portfolio] += posicion * precios[acciones]
```

`dict.get(clave, por_defecto)` nos ahorra tener que comprobar si existe la clave:

```
navs = {}
```

```
for (portfolio, acciones, posicion) in data:
    navs[portfolio] = (navs.get(portfolio, 0)
                       + position * precios[acciones])
```

Mucho más sencillo.

El método `setdefault` de los diccionarios (1)

Ahora vamos a inicializar valores mutables para un diccionario. Cada valor del diccionario será una lista. Esta es la forma naïf:

Inicializando valores mutables:

```
acciones = {}
for (portfolio, accion) in datos:
    if portfolio in acciones:
        acciones[portfolio].append(accion)
    else:
        acciones[portfolio] = [accion]
```

El método `dict.setdefault(clave, por_defecto)` es mucho más eficiente:

```
acciones = {}
for (portfolio, accion) in datos:
    acciones.setdefault(portfolio, []).append(
                                          accion)
```

`dict.setdefault()` es equivalente a "devuelve el valor, o asigna el valor y devuélvelo". O "asigna si es necesario, después devuelve". Es especialmente útil si la clave de tu diccionario es costosa de calcular o larga de escribir.

El único problema con `dict.setdefault()` es que el valor por defecto siempre se evalúa, se necesite o no. Esto sólo importa si el valor es costoso de calcular.

Si el valor por defecto **es** costoso de calcular, puede que prefieras usar la clase `defaultdict`, que veremos en breve.

El método `setdefault` de los diccionarios (2)

Aquí vemos que el método `setdefault` de los diccionarios también se puede utilizar como una sentencia independiente:

```
navs = {}
```

```
for (portfolio, accion, posicion) in datos:
    navs.setdefault(portfolio, 0)
    navs[portfolio] += posicion * precios[accion]
```

El método `setdefault` del diccionario devuelve el valor por defecto, pero nosotros lo ignoramos. Lo que estamos haciendo es aprovechar un efecto secundario de `setdefault`, que es el de asignar el valor del diccionario sólo si no existe ya un valor para esa clave.

defaultdict

Nuevo en Python 2.5.

`defaultdict` es una novedad de Python 2.5, parte del módulo `collections`. `defaultdict` es idéntico a los diccionarios normales, a excepción de un par de cosas:

- toma un parámetro extra como primer argumento: una función de factoría por defecto; y
- cuando se accede a una clave del diccionario por primera vez, se llama a la función de factoría por defecto y el resultado se utiliza para inicializar el valor del diccionario para esa clave.

Hay dos formas de obtener `defaultdict`:

- importar el módulo `collections` y referenciarlo a partir del módulo,

  ➔

- o importar `defaultdict` directamente:

  ➔

```
import collections
d = collections.defaultdict(...)
```

```
from collections import defaultdict
d = defaultdict(...)
```

A continuación tenemos el ejemplo que vimos anteriormente, en el que cada valor del diccionario debía inicializarse a una lista vacía, reescrito usando `defaultdict`:

```
from collections import defaultdict

acciones = defaultdict(list)
for (portfolio, accion) in datos:
    acciones[portfolio].append(accion)
```

Ya no hay ninguna complicación. En este caso, la función de factoría por defecto

es `list`, la cual devuelve una lista vacía.

Para obtener un diccionario con valores por defecto de 0 podríamos usar `int` como función de factoría por defecto:

```
navs = defaultdict(int)
for (portfolio, accion, posicion) in datos:
    navs[portfolio] += posicion * precios[accion]
```

De todas formas deberías tener cuidado con `defaultdict`. Puedes encontrarte con excepciones de tipo `KeyError` al intentar acceder a claves de instancias de `defaultdict`. Es necesario usar un condicional "clave in dict" si necesitas comprobar la existencia de una clave en concreto.

Construyendo y dividiendo diccionarios

Aquí tenéis una técnica útil para construir un diccionario a partir de dos listas (o secuencias): una lista de claves, y otra lista de valores.

```
nombre = ['John', 'Eric', 'Terry', 'Michael']
apellido = ['Cleese', 'Idle', 'Gilliam', 'Palin']

pythons = dict(zip(nombre, apellido))

>>> pprint.pprint(pythons)
{'John': 'Cleese',
 'Michael': 'Palin',
 'Eric': 'Idle',
 'Terry': 'Gilliam'}
```

La inversa, por supuesto, es trivial:

```
>>> pythons.keys()
['John', 'Michael', 'Eric', 'Terry']
>>> pythons.values()
['Cleese', 'Palin', 'Idle', 'Gilliam']
```

Observa que el orden del resultado de .keys() y .values() es distinto del de los elementos de las listas utilizadas para construir el diccionario. El orden de entrada es diferente del orden de salida. Esto es debido a que los diccionarios son colecciones no ordenadas. Por otro lado, el orden será siempre consistente (esto es, el orden de las claves se corresponderá con el de los valores), siempre y cuando el diccionario no se haya cambiado entre llamadas.

Comprobar si algo es cierto

```
# haz esto:       # no esto:
if x:             if x == True:
    pass              pass
```

Es elegante y eficiente aprovechar los valores booleanos intrínsecos de los objetos Python.

Comprobar una lista:

```
# haz esto:       # no esto:
if items:         if len(items) != 0:
    pass              pass

                  # y menos esto:
                  if items != []:
                      pass
```

Valores booleanos

Los nombres `True` y `False` son instancias del tipo `bool`, valores Booleanos. Como `None`, sólo hay una instancia de cada uno de ellos.

False	True
`False` (== 0)	`True` (== 1)
`""` (cadena vacía)	cualquier cadena menos `""` (`" "`, `"cualquiera"`)
`0`, `0.0`	cualquier número menos `0` (1, 0.1, -1, 3.14)
`[]`, `()`, `{}`, `set()`	cualquier contenedor no vacío (`[0]`, `(None,)`, `['']`)
`None`	casi cualquier objeto que no sea explícitamente False

Ejemplo del valor booleano de un objeto:

```
>>> class C:
...  pass
...
>>> o = C()
>>> bool(o)
True
>>> bool(C)
True
```

(Ejemplos: ejecuta truth.py.)

Para controlar el valor booleano de instancias de una clase definida por el

usuario, utiliza los métodos especiales `__nonzero__` o `__len__`. Usa `__len__` si tu clase es un contenedor para el que tiene sentido el tamaño:

```
class MiContenedor(object):

    def __init__(self, data):
        self.data = data

    def __len__(self):
        """Devuelve mi tamanyo."""
        return len(self.data)
```

Si tu clase no es un contenedor, usa `__nonzero__`:

```
class MiClase(object):

    def __init__(self, value):
        self.value = value

    def __nonzero__(self):
        """Devuelve mi valor booleano (True o False)."""
        # Esto puede ser tan complejo como queramos:
        return bool(self.value)
```

En Python 3.0, `__nonzero__` se ha renombrado a `__bool__` por consistencia con el tipo `bool`. Para mejorar la compatibilidad, añade lo siguiente a la definición de la clase:

```
__bool__ = __nonzero__
```

Índices y elementos (1)

Aquí tienes una buena forma de ahorrarte escribir demasiado si necesitas una lista de palabras:

```
>>> items = 'cero uno dos tres'.split()
>>> print items
['cero', 'uno', 'dos', 'tres']
```

Supongamos que quieres iterar sobre los elementos de la lista, y que necesitamos tanto el índice como el elemento en sí:

```
                - o -
i = 0
for item in items:      for i in range(len(items)):
    print i, item               print i, items[i]
    i += 1
```

Índices y elementos (2): `enumerate`

La función `enumerate` toma como parámetro una lista y devuelve pares (indice, elemento):

```
>>> print list(enumerate(items))
[(0, 'cero'), (1, 'uno'), (2, 'dos'), (3, 'tres')]
```

Necesitamos usar un `list` para imprimir el resultado porque `enumerate` es una función lazy: genera los elementos uno a uno, un par cada vez, sólo cuando se requieren. Un bucle `for` es un lugar en el que se requiere un elemento cada vez. `enumerate` es un ejemplo de un *generador*, que trataremos más detenidamente más adelante. `print` no utiliza un elemento cada vez -- queremos el resultado completo, así que hay que convertir el generador explícitamente en una lista al imprimirlo.

De esta forma nuestro bucle es mucho más simple:

```
for (indice, item) in enumerate(items):
    print indice, item
```

```
# compara:               # compara:
indice = 0               for i in range(len(items)):
for item in items:           print i, items[i]
    print indice, item
    indice += 1
```

La versión con `enumerate` es más corta y simple que la versión de la izquierda, y también mucho más fácil de leer y entender.

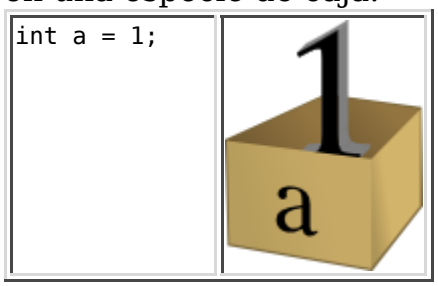
El siguiente ejemplo muestra cómo la función `enumerate` en realidad devuelve un iterador (un generador es algo parecido a un iterador):

```
>>> enumerate(items)
<enumerate object at 0x011EA1C0>
>>> e = enumerate(items)
>>> e.next()
(0, 'cero')
>>> e.next()
(1, 'uno')
>>> e.next()
(2, 'dos')
>>> e.next()
(3, 'tres')
>>> e.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
StopIteration
```

Otros lenguajes tienen "variables"

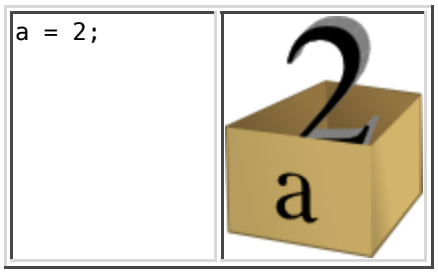
En muchos otros lenguajes, al asignar un valor a una variable se coloca el valor

en una especie de caja.



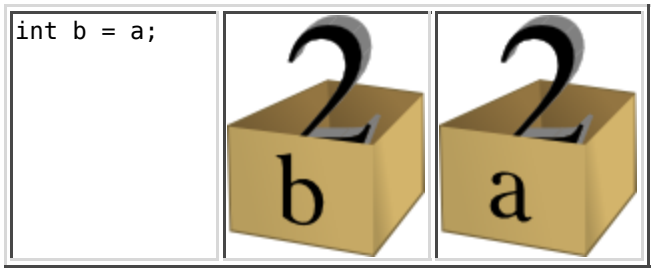
La caja "a" ahora contiene el entero 1.

Al asignar otro valor a la misma variable lo que ocurre es que el valor que contiene la caja se sustituye por el nuevo:



Ahora "a" contiene el entero 2.

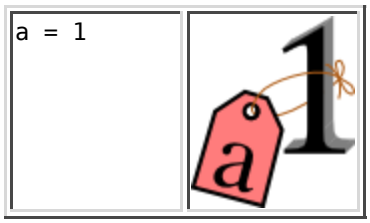
Al asignar una variable a otra se realiza una copia del valor y se coloca en la nueva caja:



"b" es una segunda caja, con una copia del entero 2. La caja "a" tiene una copia totalmente independiente.

Python tiene "nombres"

En Python, un "nombre" o "identificador" es algo parecido a una etiqueta de un paquete de correos adjuntada a un objeto.



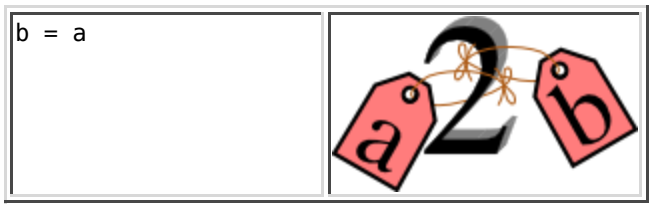
En este caso, el objeto entero 1 tiene una etiqueta con texto "a".

Si reasignarmos "a", lo que hacemos es colocar la etiqueta a otro objeto:

Ahora el nombre "a" está asignado al objeto entero 2.

El objeto entero original 1 ya no tiene la etiqueta "a". Puede que siga existiendo, pero ya no podemos accederlo a través del nombre "a". (Cuando un objeto no tiene más referencias o etiquetas asociadas, se elimina de memoria.)

Si asignamos un nombre a otro, lo único que hacemos es adjuntar otra etiqueta de nombre a un objeto existente:



El nombre "b" es sólo una segunda etiqueta añadida al mismo objeto que "a".

Aunque en Python normalmente también usamos el término "variables" (porque es terminología común), en realidad nos estamos refiriendo a "nombres" o "identificadores". En Python, las "variables" son etiquetas con nombres para los valores, no cajas con nombres.

Aunque no aprendas nada más de este tutorial, espero que al menos hayas entendido cómo funcionan los nombres en Python. Entender este punto te dará buenos dividendos, ayudándote a evitar casos como este:

➔

Valores por defecto para los

parámetros

Este es un fallo muy común entre los novatos. Incluso programadores más experimentados pueden cometer el mismo fallo si no entienden los nombres en Python.

```
def anyadir_erroneo(nuevo_item, una_lista=[]):
    una_lista.append(nuevo_item)
    return una_lista
```

El problema con esto es que el valor por defecto de `una_lista`, una lista vacía, se evalúa en el momento de definir la función. De modo que cada vez que llamas a la función, obtienes **el mismo** valor por defecto. Prueba la función varias veces:

```
>>> print anyadir_erroneo('uno')
['uno']

>>> print anyadir_erroneo('dos')
['uno', 'dos']
```

Las listas son objetos mutables; puedes modificar su contenido. La forma correcta de obtener una lista por defecto (o diccionario, o conjunto) es crearla en tiempo de ejecución, **dentro de la función**:

```
def anyadir_correcto(nuevo_item, una_lista=None):
    if una_lista is None:
        una_lista = []
    una_lista.append(nuevo_item)
    return una_lista
```

% Formateo de cadenas

El operador `%` de Python funciona de forma similar al de la función `sprintf` de C.

Aunque si no sabes C, este dato no te será de mucha utilidad. Basicamente, puedes establecer una plantilla o formato para los valores.

En este ejemplo, la plantilla contiene dos especificaciones de conversión: "%s" significa "inserta una cadena aquí", y "%i" significa "convierte un entero en una cadena e insértalo aquí". "%s" es particularmente útil porque utiliza la función `str()` de Python para convertir cualquier objeto en una cadena.

Los valores interpolados deben ajustarse a la plantilla; en este caso tenemos dos valores, una tupla.

```
nombre = 'David'
mensajes = 3
texto = ('Hola %s, tienes %i mensajes'
```

```
        % (nombre, mensajes))
print texto
```

Salida:

```
Hola David, tienes 3 mensajes
```

Puedes encontrar más detalles en la Referencia de Python, sección 2.3.6.2, "Operaciones de Formateo de Cadenas". ¡Añade a marcadores esta página! Si aún no lo has hecho, ve a python.org, descarga la documentación en HTML (en un fichero .zip o un tarball), e instalala en tu máquina. No hay nada como tener la guía definitiva al alcance de la mano.

Formateo de cadenas avanzado

De lo que la gente no se da cuenta es de que existen otras formas más flexibles de realizar el formateo de cadenas:

Por nombre, con un diccionario:

```
valores = {'nombre': nombre, 'mensajes': mensajes}
print ('Hola %(nombre)s, tienes %(mensajes)i '
       'mensajes' % valores)
```

En este caso especificas los nombres de los valores a interpolar, los cuales se buscan en el diccionario que se pasa como argumento.

¿Notas la redundancia? Los nombres "nombre" y "mensajes" ya están definidos en el espacio de nombres local. Podemos aprovecharnos de esto.

Por nombre, usando el espacio de nombres local:

```
print ('Hola %(nombre)s, tienes %(mensajes)i '
       'mensajes' % locals())
```

La función `locals()` devuelve un diccionario con todos los nombres disponibles en el espacio de nombres local.

Este es un mecanismo muy potente. Con esto, puedes hacer todo el formateo de cadenas que quieras sin tener que preocuparte de ajustar los valores a interpolar con la plantilla.

Pero el poder puede ser peligroso. ("Un gran poder conlleva una gran responsabilidad.") Si utilizas `locals()` con una plantilla proporcionada de forma externa, expones tu espacio de nombres local completamente al llamador. Esto es algo a tener en cuenta.

Para examinar tu espacio de nombres local:

```
>>> from pprint import pprint
>>> pprint(locals())
```

`pprint` es un módulo muy útil. Si no lo conoces todavía, intenta jugar un poco con él. ¡Facilita muchísimo la depuración de las estructuras de datos!

Formateo de cadenas avanzado

El espacio de nombres de los atributos de un objeto es un simple diccionario, `self.__dict__`.

Por nombre, usando el espacio de nombres del objeto:

```
print ("Encontrados %(num_errores)d errores"
       % self.__dict__)
```

Es equivalente a lo siguiente, pero más flexible:

```
print ("Encontrados %d errores"
       % self.num_errores)
```

Nota: Los atributos de clase se encuentran en class.__dict__. Las búsquedas en los espacios de nombre en realidad son búsquedas encadenadas en los diccionarios.

Comprensión de listas

La comprensión de listas ("listcomps" para abreviar) son atajos sintácticos para el siguiente patrón:

La forma tradicional, con un bucle `for` y sentencias `if`:

```
nueva_lista = []
for item in una_lista:
    if condicion(item):
        nueva_lista.append(fn(item))
```

Usando comprensión de listas:

```
nueva_lista = [fn(item) for item in una_lista
               if condicion(item)]
```

La compresión de listas es muy clara y concisa, hasta cierto punto. Puedes tener varios bucles `for` y varias condiciones `if`, pero a partir de dos o tres, o si las condiciones son algo complejas, te sugiero utilizar bucles `for` normales.

Aplicando el Zen de Python, elige la forma más legible.

Por ejemplo, una lista de los cuadrados de 0–9:

```
>>> [n ** 2 for n in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Una lista de los cuadrados de los números impares de 0–9:

```
>>> [n ** 2 for n in range(10) if n % 2]
[1, 9, 25, 49, 81]
```

Expresiones generadoras (1)

Calculemos la suma de los cuadrados de los números hasta el 100:

Como un bucle:

```
total = 0
for num in range(1, 101):
    total += num * num
```

Podemos utilizar la función `sum` para facilitarnos el trabajo, construyendo la siguiente secuencia.

Como comprensión de listas:

```
total = sum([num * num for num in range(1, 101)])
```

Como expresión generadora:

```
total = sum(num * num for num in xrange(1, 101))
```

Las expresiones generadoras ("genexps") son exactamente iguales a la compresión de listas, excepto que las listcomps son *greedy* (avariciosas), mientras que las expresiones generadoras son *lazy* (perezosas). Las listcomps construyen la lista resultado de una sola vez. Las expresiones generadoras calculan un valor cada vez, cuando se necesita. Esto es particularmente útil para secuencias largas donde la lista a calcular no es más que un paso intermedio y no el resultado final.

En este caso, sólo estamos interesados en la suma total; no necesitamos la lista de cuadrados intermedia. Utilizamos `xrange` por la misma razón: genera los valores uno por uno.

Expresiones generadoras (2)

Por ejemplo, si estuvieramos sumando los cuadrados de varios miles de millones de enteros, nos quedaríamos sin memoria si utilizáramos comprensión de listas, pero con las expresiones generadoras no tendríamos ningún problema. ¡Aunque llevaría mucho tiempo, claro!

```
total = sum(num * num
            for num in xrange(1, 1000000000))
```

La diferencia en la sintaxis es que las listcomps utilizan corchetes, mientras que las expresiones generadoras no lo hacen. Las expresiones generadoras necesitan colocarse entre paréntesis a veces, así que deberías acostumbrarte a utilizarlos.

Regla general:

- Utiliza comprensión de listas cuando la lista sea el resultado que nos interesa.
- Utiliza una expresión generadora cuando la lista no sea más que un paso intermedio.

Aquí tenéis un ejemplo que vi recientemente en el trabajo.

➔

Necesitábamos un diccionario que mapeara los números de los meses (tanto en forma de enteros como de cadenas) a códigos de meses para futuros contratos. Podría hacerse en una sola línea de código.

➔

Esto funciona de la siguiente forma:

- La función `dict()` toma una lista de pares clave/valor (tuplas de 2 elementos).
- Tenemos una lista de códigos para los meses (cada código es una única letra, y una cadena no es más que una lista de letras). Usamos enumerate sobre la lista para obtener tanto el mes como el índice.
- Los números de los meses comienzan con 1, pero en Python los índices comienzan en 0, así que el número de mes es el índice más uno.
- Queremos buscar los meses tanto por cadenas como por enteros. Podemos usar las funciones `int()` y `str()` para hacerlo por nosotros, e iterar sobre ellos.

Ejemplo reciente:

```
codigos_meses = dict((fn(i+1), codigo)
    for i, codigo in enumerate('FGHJKMNQUVXZ')
    for fn in (int, str))
```

Valores resultantes en `codigos_meses`:

```
{ 1:  'F',  2:  'G',  3:  'H',  4:  'J', ...
 '1': 'F', '2': 'G', '3': 'H', '4': 'J', ...}
```

Ordenando

Ordenar listas en Python es muy sencillo:

```
a_list.sort()
```

(Ten en cuenta que se ordena la lista original, el método `sort` **no** devuelve una lista o una copia de la lista ordenada.)

Pero, ¿qué ocurre si tenemos una lista de datos a ordenar, pero el orden no es el normal (es decir, ordenar por la primera columna, después la segunda, etc.)? Puede que necesitemos ordenar por la segunda columna y luego por la cuarta.

Podemos usar el método `sort` indicando una función propia para comparar:

```
def mi_comparador(item1, item2):
    return cmp((item1[1], item1[3]),
               (item2[1], item2[3]))

una_lista.sort(mi_comparador)
```

Esto funcionaría, pero sería muy lento para listas largas.

Ordenando con DSU *

DSU = Decorate-Sort-Undecorate (Decorar-Ordenar-Desdecorar)

* Nota: Normalmente no es necesario utilizar DSU. Consulta la siguiente sección, Ordenando con claves para descubrir un nuevo enfoque.

En lugar de crear nuestras propias funciones para comparar, creamos una lista auxiliar que se ordene de forma natural:

```
# Decorar:
a_ordenar = [(item[1], item[3], item)
             for item in una_lista]

# Ordenar:
a_ordenar.sort()

# Desdecorar:
una_lista = [item[-1] for item in a_ordenar]
```

La primera línea crea una lista que contiene tuplas: copias de elementos a ordenar en orden de prioridad, seguidos de los datos completos.

La segunda línea utiliza la ordenación nativa de Python, que es muy rápida y eficiente.

La tercera línea obtiene el **último** valor de la lista ordenada. Recuerda, este último valor son los datos completos. Descartamos los elementos que usamos para ordenar, que ya no necesitamos más.

Se trata entonces de un compromiso entre espacio y complejidad frente a tiempo. Es más sencillo y rápido, pero necesitamos duplicar la lista original.

Ordenando con claves

En Python 2.4 se introdujo un parámetro opcional para el método `sort` de las listas, "key" (clave), para indicar una función que toma un solo parámetro y que se utiliza para calcular una clave que utilizar para comparar los elementos de la lista. Por ejemplo:

```
def mi_clave(item):
    return (item[1], item[3])

a_ordenar.sort(key=mi_clave)
```

La función `mi_clave` se llama una vez para cada elemento de la lista `a_ordenar`.

Puedes crear tus propias funciones para calcular la clave, o utilizar cualquier función de un sólo parámetro que se ajuste a tus necesidades:

- `str.lower` para ordenar alfabéticamente sin diferenciar mayúsculas y minúsculas.
- `len` para ordenar basándose en el tamaño de los elementos (cadenas o contenedores).
- `int` o `float` para ordenar de forma numérica, con cadenas que representan números como "2", "123", "35".

Generadores

Ya hemos visto las expresiones generadoras. Podemos crear expresiones generadoras tan complejas como queramos, como funciones:

```
def mi_generador_range(parada):
    valor = 0
    while valor < parada:
```

```
        yield valor
        valor += 1

for i in mi_generador_range(10):
    haz_algo(i)
```

La palabra clave `yield` convierte una función en un generador. Cuando llamas a una función generadora, en lugar de devolver el valor de retorno inmediatamente Python devuelve un objeto generador, que es un iterador; tiene un método `next`. Los bucles `for` simplemente llaman al método `next` del iterador, hasta que se lanza una excepción `StopIteration`. Puedes lanzar una excepción `StopIteration` de forma explícita, o bien de forma implícita al llegar al final del código del generador como en el ejemplo anterior.

Los generadores pueden ayudar a simplificar el trabajo con secuencias/iteradores, porque no necesitamos construir listas concretas; solo calculan un valor cada vez. La función generadora mantiene el estado.

Esta es la forma en la que funciona un bucle `for` en realidad. Python comprueba la secuencia que se pasa después de la palabra clave `in`. Si es un contenedor simple (como una lista, una tupla, un diccionario, un conjunto, o un contenedor creado por el usuario) Python lo convierte en un iterador. Si ya es un iterador no hace nada.

Entonces Python llama repetidamente al método `next` del iterador, asigna el valor devuelto al contador del bucle (`i` en este caso), y ejecuta el código indentado. El proceso se repite una y otra vez, hasta que se lanza la excepción `StopIteration`, o se ejecuta una sentencia `break`.

Un bucle `for` puede tener una clausula `else`, cuyo código se ejecuta después de que el iterador se quede sin elementos, pero **no** si se sale al ejecutar un `break`. Esta distinción permite usos muy elegantes. Las cláusulas `else` no se suelen utilizar mucho con los bucles `for`, pero pueden ser muy útiles. Algunas veces una cláusula `else` es la mejor forma de expresar la lógica que necesitas.

Por ejemplo, si necesitas comprobar que una condición se cumple para todos los elementos de una secuencia:

```
for item in secuencia:
    if condition(item):
        break
else:
    raise Exception('La condicion no se satisface.')
```

Generadores de ejemplo

Filtrar líneas en blanco de un archivo CSV (o elementos de una lista):

```
def filtrar_filas(iterador_filas):
    for fila in iterador_filas:
        if fila:
            yield fila

archivo_datos = open(ruta, 'rb')
ifilas = filtrar_filas(csv.reader(archivo_datos))
```

Leyendo líneas de un archivo de texto/datos

```
archivodatos = open('archivodatos')
for linea in archivodatos:
    haz_algo(linea)
```

Esto es posible porque los archivos soportan el método `next`, como otros iteradores: listas, tuplas, diccionarios (para sus claves), generadores.

Aunque hay que tener cuidado con una cosa: debido a la forma en la que funciona el buffer de datos, no puedes mezclar los métodos `.next` y `.read*` a menos que utilices Python 2.5+.

EAFP contra LBYL

EAFP = Easier to Ask Forgiveness than Permission (Es más sencillo pedir perdón que pedir permiso)

LBYL = Look Before You Leap (Piensa antes de actuar)

Generalmente se considera mejor utilizar el enfoque EAFP, pero no siempre es así.

- Duck typing (tipado de pato)

  Si camina como un pato, habla como un pato, y tiene el aspecto de un pato: es un pato. (¿Un ganso? Suficientemente parecido.)

- Excepciones

  Si `x` debe ser una cadena para que el código funcione, por qué no escribir

  ```
  str(x)
  ```

> en lugar de intentar algo como
>
> ```
> isinstance(x, str)
> ```

Ejemplo de `try/except` EAFP

Puedes rodear el código que pueda lanzar alguna excepción en un bloque `try/except` para capturar los errores, y probablemente acabaras con una solución que será mucho más general que si hubieras intentado anticipar todas las posibilidades.

```
try:
    return str(x)
except TypeError:
    ...
```

Nota: Especifica siempre las excepciones a capturar. Nunca uses cláusulas `except` sin indicar el tipo de la excepción. Esto podría capturar excepciones no esperadas, haciendo que tu código fuera más dificil de depurar.

Importando

> ```
> from modulo import *
> ```

Es probable que hayas visto esta forma con un "carácter comodín" de las sentencias import. Puede que incluso te guste. **No lo utilices.**

Parafraseando una diálogo muy conocido:

> (Dagobah, jungla, pantanos, y niebla.)
>
> LUKE: ¿Es mejor `from module import *` que los imports explícitos?
>
> YODA: No, no mejor. Más rápido, más fácil, más seductor.
>
> LUKE: Pero ¿cómo sabré por qué los imports explícitos son mejores que la forma con el carácter comodín?
>
> YODA: Saberlo tu podrás cuando tu código intentes leer seis meses después.

Los carácteres comodín en los imports pertenecen al lado oscuro de la fuerza de Python.

¡Nunca!

El uso de `from module import *` conlleva polución del espacio de nombres. Tendrás montones de cosas en el espacio de nombres local que no esperabas tener. Puede que te encuentres con nombres importados sobreescribiendo nombres definidos en local. No podrás distinguir fácilmente de dónde vienen algunos nombres. Aunque pueda ser un atajo conveniente, no debería utilizarse en código en producción.

Moraleja: **¡no uses el carácter comodín para los imports!**

➔

Es mucho mejor:

- referenciar nombres a través de su módulo (identificadores totalmente calificados),

  ➔

- importar un módulo con un nombre largo usando un nombre más corto (alias; recomendado),

  ➔

- o importar explicitamente sólo los nombres que necesitas.

  ➔

¡Alerta de polución del espacio de nombres!

En su lugar,

Referencia los nombres a través de su módulo (identificadores totalmente calificados):

```
import modulo
modulo.nombre
```

O importa un módulo con un nombre largo usando un nombre más corto (alias):

```
import modulo_con_nombre_largo as mod
mod.nombre
```

O importa explicitamente sólo los nombres que necesitas:

```
from modulo import nombre
nombre
```

Aunque este método no se presta a usarse en el intérprete interactivo, donde puedes querer editar y recargar un módulo.

Modulos y Scripts

Para crear un módulo que se pueda importar y también ejecutar como si de un script se tratara:

```
if __name__ == '__main__':
    # codigo del script
```

Al importar un módulo su atributo `__name__` toma como valor el nombre del archivo que define el módulo, sin el ".py". De forma que el código de la sentencia `if` no se ejecutará al importarse el módulo. Cuando se ejecuta como un script el atributo `__name__` tiene como valor "__main__", por lo que, ahora sí, se ejecutará el código del script.

A excepción de algunos pocos casos especiales, no deberías colocar ningún código ejecutable en el primer nivel del código. Colocalo en funciones, clases, métodos, y controla su ejecución usando `if __name__ == '__main__'`.

Estructura de un módulo

```
"""Cadena de documentacion del modulo"""

# imports
# constantes
# clases de tipo excepcion
# funciones para uso publico
# clases
# funciones y clases internas

def main(...):
    ...

if __name__ == '__main__':
    estado = main()
    sys.exit(estado)
```

Esta es la forma en que deberían estructurarse los módulos.

Procesar los argumentos de línea de comandos

Ejemplo: cmdline.py:

```
#!/usr/bin/env python
```

```
"""
Cadena de documentacion del modulo.
"""

import sys
import optparse

def procesar_linea_comandos(argv):
    """
    Devuelve una tupla de dos elementos: (objeto preferencias, lista args).
    `argv` es una lista de argumentos, o `None` para ``sys.argv[1:]``.
    """
    if argv is None:
        argv = sys.argv[1:]

    # inicializamos el parser:
    parser = optparse.OptionParser(
        formatter=optparse.TitledHelpFormatter(width=78),
        add_help_option=None)

    # a continuacion definimos las opciones:
    parser.add_option(      # descripcion personalizada; coloca --help al final
        '-h', '--help', action='help',
        help='Muestra este mensaje de ayuda y termina.')

    prefs, args = parser.parse_args(argv)

    # comprueba el numero de argumentos, verifica los valores, etc.:
    if args:
        parser.error('el programa no soporta ningun parametro de linea de comandos; '
                     'se ignora "%s".' % (args,))

    # cualquier otra cosa que haya que hacer con las preferencias y los argumentos

    return prefs, args

def main(argv=None):
    prefs, args = procesar_linea_comandos(argv)
    # aqui iria el codigo de la aplicacion, como:
    # ejecutar(prefs, args)
    return 0        # exito

if __name__ == '__main__':
    estado = main()
    sys.exit(estado)
```

Paquetes

```
paquete/
    __init__.py
    modulo1.py
    subpaquete/
```

```
__init__.py
modulo2.py
```

- Utilizados para organizar tu proyecto.
- Reduce el número de entradas en la ruta de carga.
- Reduce los conflictos de nombres al importar.

Ejemplo:

```
import paquete.modulo1
from paquetes.subpaquete import modulo2
from paquetes.subpaquete.modulo2 import nombre
```

En Python 2.5 contamos con imports relativos y absolutos a través de future import:

```
from __future__ import absolute_import
```

No he utilizado esta característica demasiado, así que convenientemente cortaremos la explicación aquí.

Simple es mejor que complejo

> Depurar es el doble de complicado que programar. Por lo tanto, si escribes tu código tan inteligentemente como te sea posible, por definición, no serás lo suficientemente inteligente para depurarlo.
>
> —Brian W. Kernighan, co-autor de *El Lenguaje de Programación C* y la "K" en "AWK"

En otras palabras, ¡haz que tus programas sean sencillos!

No reinventes la rueda

Antes de escribir cualquier código,
➔ ➔ ➔ ➔

- Comprueba la librería estándar de Python.
- Comprueba el Índice de Paquetes de Python ("Cheese Shop"):

  http://cheeseshop.python.org/pypi

- Busca en internet. Google es tu amigo.

Referencias

- "Python Objects", Fredrik Lundh, http://www.effbot.org/zone/python-objects.htm
- "How to think like a Pythonista", Mark Hammond, http://python.net/crew/mwh/hacks/objectthink.html
- "Python main() functions", Guido van Rossum, http://www.artima.com/weblogs/viewpost.jsp?thread=4829
- "Python Idioms and Efficiency", http://jaynes.colorado.edu/PythonIdioms.html
- "Python track: python idioms", http://www.cs.caltech.edu/courses/cs11/material/python/misc/python_idioms.html
- "Be Pythonic", Shalabh Chaturvedi, http://shalabh.infogami.com/Be_Pythonic2
- "Python Is Not Java", Phillip J. Eby, http://dirtsimple.org/2004/12/python-is-not-java.html
- "What is Pythonic?", Martijn Faassen, http://faassen.n--tree.net/blog/view/weblog/2005/08/06/0
- "Sorting Mini-HOWTO", Andrew Dalke, http://wiki.python.org/moin/HowTo/Sorting
- "Python Idioms", http://www.gungfu.de/facts/wiki/Main/PythonIdioms
- "Python FAQs", http://www.python.org/doc/faq/