["Python"] * 40

Cuarenta características de Python que quizás no conoces

Víctor Terrón — http://github.com/vterron

Unknown Unknowns

"There are things we know that we know. There are known unknowns. That is to say there are things that we now know we don't know. But there are also unknown unknowns. There are things we do not know we don't know" [Donald Rumsfeld, 2002]

- En Python hay funcionalidades increíbles, imprescindibles una vez que las conoces, que podríamos no echar en falta jamás porque ni siquiera sabíamos que existían.
- El propósito de esta charla es presentar una serie de aspectos interesantes de Python que en estos años he descubierto que mucha gente, incluso programadores veteranos, desconoce.

Unknown Unknowns

- Algunas de las funcionalidades que vamos a discutir aquí son muy prácticas y otras curiosidades de indiscutiblemente escasa o nula utilidad en nuestro día a día. Pero todos ellos son conceptos sencillos de entender y que merece la pena saber que están ahí, incluso si no los usamos... por ahora.
- Tenemos 1:15 minutos para cada uno de los puntos, así que muchos de ellos sólo vamos a poder verlos muy por encima. Pero al menos habrán dejado de ser *unknown unknowns*.

¡No os limitéis a escuchar!

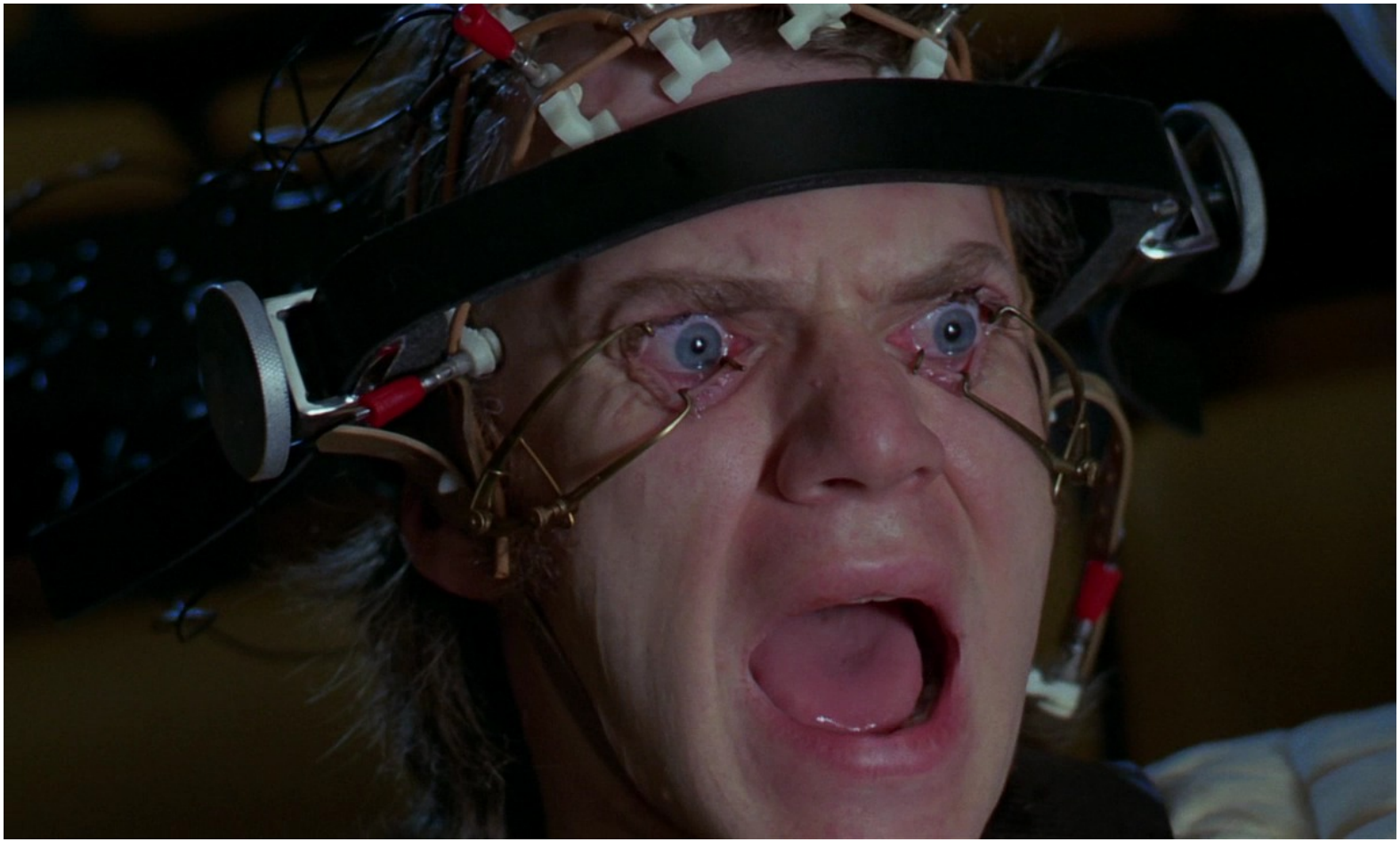
No suele ser divertido escuchar a nadie hablar durante casi una hora. Participad, intervenid, criticad, opinad. ¡Si digo algo que no tiene ningún sentido, corregidme!

El código fuente está disponible en:

`http://github.com/vterron/PyConES-2013`

Erratas, correcciones, enlaces interesantes...
¿enviará alguien algún pull request antes de que termine esta charla?

¿Listos?



1. Intercambiar dos variables

Normalmente, en otros lenguajes de programación, tenemos que usar una variable temporal para almacenar uno de los dos valores.

Por ejemplo, en C

```
int x, y;
int tmp;
tmp = x;
x = y;
y = x;
```

1. Intercambiar dos variables

Python nos permite hacer

$$a, b = b, a$$

```
>>> a = 5
>>> b = 7
>>> a, b = b, a
>>> a
7
>>> b
5
```

1. Intercambiar dos variables

Desde nuestro punto de vista, ambas asignaciones ocurren simultáneamente. La clave está en que tanto a, b como b, a son tuplas.

Las expresiones en Python se evalúan de izquierda a derecha. En una asignación, el lado derecho se evalúa antes que el izquierdo. Por tanto:

- El lado derecho de la asignación es evaluado, creando una tupla de dos elementos en memoria, cuyos elementos son los objetos designados por los identificadores b y a.

1. Intercambiar dos variables

- El lado izquierdo es evaluado: Python ve que estamos asignando una tupla de dos elementos a otra tupla de dos elementos, así que *desempaqueta* (tuple unpack) la tupla y los asigna uno a uno:
  - Al primer elemento de la tupla de la izquierda, a, le asigna el primer elemento de la tupla que se ha creado en memoria a la derecha, el objeto que antes tenía el identificador b. Así, el nuevo a es el antiguo b.
  - Del mismo modo, el nuevo b pasa a ser el antiguo a.

Explicación detallada en Stack Overflow:

`http://stackoverflow.com/a/14836456/184363`

1. Intercambiar dos variables

¿Cómo se declara una tupla de un único elemento?

```
1,
```

o, para más claridad,

```
(1,)
```

Es la coma, no el paréntesis, el constructor de la tupla

1. Intercambiar dos variables

Los paréntesis por sí mismos no crean una tupla: Python evalúa la expresión dentro de los mismos y devuelve el valor resultante:

Esto sólo suma dos números

```
>>> 2 + (1)
3
```

Esto intenta sumar entero y tupla (y fracasa)

```
>>> 2 + (1,)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'tuple'
```

2. Encadenamiento de operadores lógicos

En vez de escribir

```
x => y and y < z
```

A diferencia de C, todos los operadores lógicos tienen la misma prioridad, y pueden ser encadenados de forma arbitraria.

Mucho mejor

```
x <= y < z
```

Las dos expresiones de arriba son equivalentes, aunque en la segunda `x` sólo se evalúa una vez. En ambos casos, `z` no llega a evaluarse si no se cumple que `x <= y` (*lazy evaluation*).

3. 0.1 + 0.2 != 0.3

¿Por qué 0.1 + 0.2 == 0.3 es False?

```
>>> 0.1 + 0.2 == 0.3
False
```

Porque los números flotantes se representan internamente, en cualquier ordenador, como fracciones binarias. Por ejemplo, 1.25 es equivalente a 1/2 + 3/4.

3. 0.1 + 0.2 != 0.3

La mayor parte de las fracciones decimales no puede ser representada exactamente con una fracción binaria. Lo máximo que los números flotantes pueden hacer es aproximar su valor real — con bastantes decimales, pero nunca el exacto.

```
>>> print "%.30f" % 0.1
0.100000000000000055511151231257
>>> print "%.30f" % 0.2
0.200000000000000011102230246252
>>> print "%.30f" % 0.3
0.299999999999999888977697537484
```

Lo mismo ocurre en base decimal con, por ejemplo, el número 1/3. Por más decimales con que se represente, 0.333333... no deja de ser una aproximación.

3. 0.1 + 0.2 != 0.3

Normalmente no somos conscientes de esto porque Python por defecto nos muestra una aproximación del valor real de los números flotantes, redondeándolos a una representación práctica para nosotros.

```
>>> print 0.1
0.1
>>> print 0.2
0.2
>>> print 0.3
0.3
>>> 0.1 + 0.2
0.30000000000000004
```

Floating Point Arithmetic: Issues and Limitations

http://docs.python.org/2/tutorial/floatingpoint.html

3. 0.1 + 0.2 != 0.3

Nunca deberíamos comparar números flotantes directamente

Una opción, simple, es comprobar si ambos valores son lo "*suficientemente iguales*" usando un valor máximo de error absoluto admisible, normalmente llamado epsilon.

```
>>> epsilon = 1e-10
>>> abs((0.1 + 0.2) - 0.3) < epsilon
True
>>> abs((0.1 + 0.2) - 0.3)
5.551115123125783e-17
```

3. 0.1 + 0.2 != 0.3

En caso de no conocer el rango de los números con los que vamos a trabajar, tenemos que compararlos en términos de error relativo ("*son iguales al 99.999%*).

Comparing Floating Point Numbers, 2012 Edition

`http://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/`

3. 0.1 + 0.2 != 0.3

En Python tenemos el módulo decimal

```
>>> import decimal
>>> x = decimal.Decimal("0.1")
>>> y = decimal.Decimal("0.2")
>>> z = decimal.Decimal("0.3")
>>> x + y == z
True
```

decimal — Decimal fixed point and floating point arithmetic

http://docs.python.org/2/library/decimal.html

4. int(True) == 1

El valor numérico de True es 1; el de False, 0

```
>>> int(True)
1
>>> int(False)
0
>>> 37 + True
38
>>> 7 / False
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: float division
```

4. int(True) == 1

De hecho, la clase bool hereda de int

```
>>> issubclass(bool, int)
True
>>> isinstance(True, int)
True
>>> bool.mro()
[<type 'bool'>, <type 'int'>, <type 'object'>]
```

PEP 285: Adding a bool type [2002]

http://www.python.org/dev/peps/pep-0285/

5. import this

El Zen de Python

Los principios filosóficos en los que se cimenta Python

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
[...]
```

PEP 20: The Zen of Python [2004]

http://www.python.org/dev/peps/pep-0020/

5. import this

Algo bastante más desconocido: el contenido del módulo this.py está cifrado con el algoritmo ROT13 — cada letra sustituida por la que está trece posiciones por delante en el alfabeto:

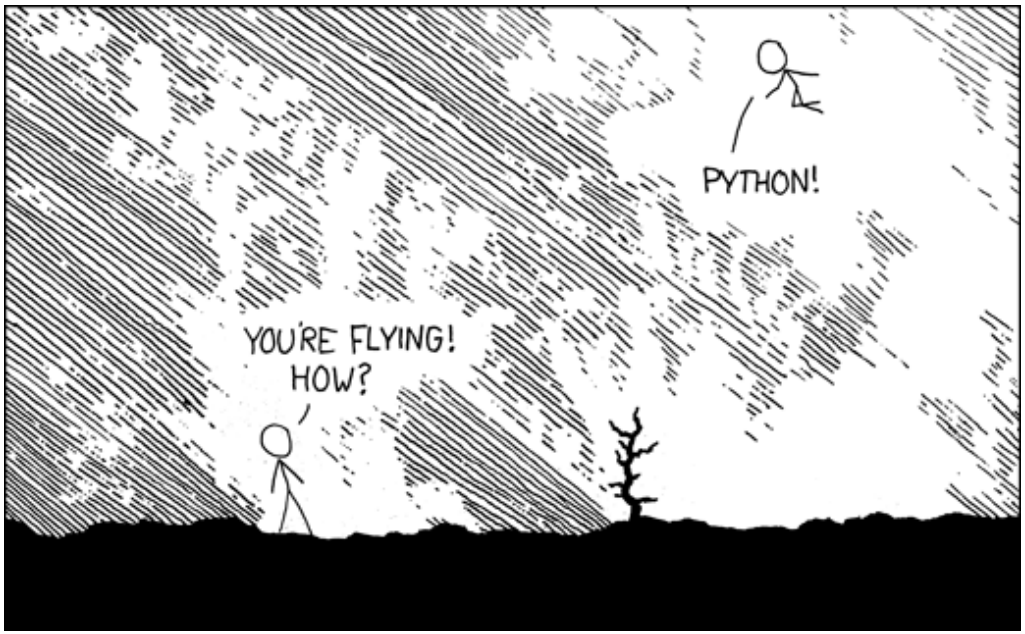
```
>>> import inspect
>>> print inspect.getsource(this)
s = """Gur Mra bs Clguba, ol Gvz Crgref

Ornhgvshy vf orggre guna htyl.
Rkcyvpvg vf orggre guna vzcyvpvg.
Fvzcyr vf orggre guna pbzcyrk.
Pbzcyrk vf orggre guna pbzcyvpngrq.
Syng vf orggre guna arfgrq.
Fcnefr vf orggre guna qrafr.
Ernqnovyvgl pbhagf.
[...]
```

this and The Zen of Python

http://www.wefearchange.org/2010/06/import-this-and-zen-of-python.html

6. El módulo antigravedad

import antigravity



http://xkcd.com/353/

Añadido en Python 3, pero también está disponible en Python 2.7

6. El módulo antigravedad

antigravity.py

```
>>> print inspect.getsource(antigravity)
import webbrowser
webbrowser.open("http://xkcd.com/353/")
```

En Py3K el módulo antigravity incluye la función geohash(), referencia a otra viñeta de XKCD en la que se describe un algoritmo que genera coordenadas en base a la fecha y tu posición actual: http://xkcd.com/426/

The History of Python - import antigravity

http://python-history.blogspot.com.es/2010/06/import-antigravity.html

7. Bloques de código

En Python los bloques de código se definen mediante el sangrado, no utilizando palabras clave, como en Fortran, o llaves, como en C. Ya que hay a quien le desagrada, e incluso odia esta característica, gracias al módulo `__future__` es posible habilitar el uso de las llaves para la delimitación de bloques.

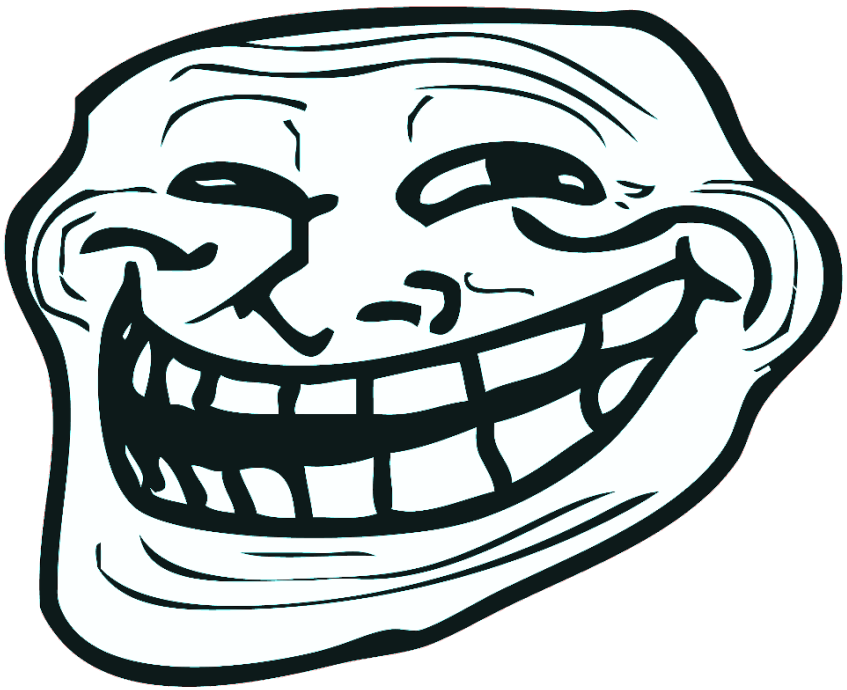
```
from __future__ import braces
```

Python White Space Discussion

`http://c2.com/cgi/wiki?PythonWhiteSpaceDiscussion`

7. Bloques de código

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



7. Bloques de código

Pero, si de verdad quieres usarlos, ¡hay una forma!

```
if foo: #{
    print "It's True"
#}
else: #{
    print "It's False"
#}
```

Is it true that I can't use curly braces in Python?

http://stackoverflow.com/a/1936210/184363

8. Concatenación eficiente de cadenas

Tenemos varias cadenas de texto que queremos unir:

```
>>> palabras = "uno", "dos", "tres"
>>> resultado = ""
>>> for p in palabras:
>>>     resultado += p
>>> resultado
'unodostres'
```

Las cadenas de texto en Python son inmutables, por lo que cada vez que asignamos una cadena a una variable un nuevo objeto es creado en memoria. En este código estamos calculando, almacenando y desechando cada paso intermedio. Eso es inaceptablemente lento.

8. Concatenación eficiente de cadenas

La forma Pythónica de concatenar cadenas es así:

resultado = "".join(palabras)

El método str.join(iterable) devuelve una cadena que es la concatenación de las cadenas en iterable, utilizando como separador entre los elementos la cadena que llama al método.

```
>>> palabras = "uno", "dos", "tres"
>>> "-".join(palabras)
'uno-dos-tres'
```

Y lo hace de una única pasada — rápido y eficiente.

9. printf en Python

Los objetos str y unicode tienen el operador %, que nos permite usar la sintaxis del antediluviano, presente en todos los lenguajes, printf(), para controlar exactamente cómo se muestra una cadena de texto.

```
>>> print math.pi
3.14159265359
>>> "%.2f" % math.pi
3.14
>>> "%05.2f" % math.pi
03.14
>>> potencia = math.e ** 100
>>> "%.2f ^ %.2f = %.4g" % (math.e, 100, potencia)
'2.72 ^ 100.00 = 2.688e+43'
```

9. printf en Python

```
>>> "%s por %d es %f" % ("tres", 2, 6.0)
'tres por 2 es 6.000000'
>>> "%0*d" % (5, 3)
'00003'
>>> "%+.*f" % (5, math.pi)
'+3.14159'
```

No obstante, desde Python 2.6 lo recomendable es usar str.format(): más sofisticado, flexible, extensible y que puede trabajar con tuplas y diccionarios de forma natural. ¡El operador de % para el formateo de cadenas se considera obsoleto!

PEP 3101: Advanced String Formatting [2006]

http://www.python.org/dev/peps/pep-3101/

10. str.format()

Tanto el operador % como str.format() permiten acceder a argumentos por su nombre, lo que es especialmente útil si necesitamos usar más de una vez la misma cadena. Pero str.format() puede hacer muchas cosas más, como acceder a atributos de los argumentos, o hacer conversiones a str() y repr().

Acceso a los argumentos por posición

```
>>> '{0}{1}{0}'.format('abra', 'cad')
'abracadabra'
```

Acceso a los atributos de un argumento por posición

```
>>> "Parte real: {0.real}".format(2+3j)
'Parte real: 2.0'
```

10. str.format()

Acceso a argumentos por nombre

```
>>> kwargs = {'nombre' : 'Pedro', 'apellido' : 'Medina'}
>>> "{nombre} {apellido} se llama {nombre}".format(kwargs)
'Pedro Medina se llama Pedro'
```

Acceso a atributos de argumentos por nombre

```
>>> kwargs = {'numero' : 1+5j}
>>> "Parte imaginaria: {numero.imag}".format(**kwargs)
'Parte imaginaria: 5.0'
```

Format Specification Mini-Language

http://docs.python.org/2/library/string.html#formatstrings

11. Cadenas multilínea

Una opción es usar barras invertidas

```
>>> parrafo = "En un lugar de la Mancha, de cuyo nombre " \
...           "no quiero acordarme, no ha mucho tiempo " \
...           "que vivía un hidalgo de los de lanza en " \
...           "astillero, adarga antigua, rocín flaco y " \
...           "galgo corredor."
>>> parrafo[:24]
'En un lugar de la Mancha'
>>> parrafo[-17:]
'y galgo corredor.'
```

Pero, en el caso de trabajar con cadenas muy largas, puede ser molesto tener que terminar cada línea con \, sobre todo porque nos impide reformatear el párrafo con nuestro entorno de desarrollo o editor de texto — ¡Emacs, por supuesto!

11. Cadenas multilínea

Las comillas triples respetan todo lo que hay dentro de ellas, incluyendo saltos de línea. No es necesario escapar nada: todo se incluye directamente en la cadena.

```
>>> parrafo = """En un lugar de la Mancha,
... de cuyo nombre no quiero
... acordarme
...
... """
>>> print parrafo
En un lugar de la Mancha,
de cuyo nombre no quiero
acordarme


>>>
```

11. Cadenas multilínea

Una opción menos conocida es que podemos dividir nuestra cadena en líneas, una por línea, y rodearlas con paréntesis. Python evalúa lo que hay dentro de los mismos y de forma automática las concatena:

```
>>> parrafo =  ("Es, pues, de saber, que este "
...             "sobredicho hidalgo, los ratos "
...             "que estaba ocioso")
>>> len(parrafo.split())
13
```

Un ejemplo más sencillo:

```
>>> "uno" "dos"
'unodos'
```

12. Listas por comprensión vs generadores

Las listas por comprensión nos permiten construir, en un único paso, una lista donde cada elemento es el resultado de aplicar algunas operaciones a todos (o algunos de) los miembros de otra secuencia o iterable.

Por ejemplo, para calcular el cuadrado de los números [0, 9]:

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

Esto es equivalente a:

```
>>> resultado = []
>>> for x in range(10):
...     resultado.append(x ** 2)
```

12. Listas por comprensión vs generadores

El problema, a veces, es que las listas por comprensión construyen... pues eso, una lista, almacenando todos los valores en memoria a la vez.

No intentéis esto en casa

```
>>> [x ** 2 for x in xrange(10000000000000000)]
# Tu ordenador explota
```

12. Listas por comprensión vs generadores

Si sólo necesitamos los valores una vez, y de uno en uno, podemos utilizar una expresión generadora: son idénticos a las listas por comprensión, pero usando paréntesis.

```
>>> cuadrados = (x ** 2 for x in range(1, 11))
>>> cuadrados
<generator object <genexpr> at 0x7f8b82504eb0>
>>> next(cuadrados)
1
>>> next(cuadrados)
4
```

PEP 289: Generator Expressions

http://www.python.org/dev/peps/pep-0289/

12. Listas por comprensión vs generadores

Equivalente a:

```
>>> def cuadrados(seq):
...     for x in seq:
...         yield x ** 2
...
>>> g = cuadrados(range(1, 11))
>>> next(g)
1
>>> next(g)
4
```

Generator Tricks for Systems Programmers (David M. Beazley)

http://www.dabeaz.com/generators/

13. Comprensión de conjuntos y diccionarios

En Python 2.7.x+ y 3.x+ podemos crear no sólo listas, sino también conjuntos y diccionarios por comprensión.

Un conjunto con los múltiplos de siete <= 100:

```
>>> {x for x in range(1, 101) if not x % 7}
set([98, 35, 70, 7, 42, 77, 14, 49, 84, 21, 56, 91, 28, 63])
```

Otra forma de hacerlo:

```
>>> mult7 = set(x for x in range(1, 101) if not x % 7)
>>> len(mult7)
14
>>> 25 in mult7
False
```

13. Comprensión de conjuntos y diccionarios

Un diccionario con el cubo de los números [0, 9]:

```
>>> cubos = {x : x ** 3 for x in range(10)}
>>> cubos[2]
8
>>> cubos[8]
512
```

Otra forma de hacerlo:

```
>>> dict((x, x ** 3) for x in range(5))
{0: 0, 1: 1, 2: 8, 3: 27, 4: 64}
```

PEP 274: Dict Comprehensions [2001]

http://www.python.org/dev/peps/pep-0274/

14. Repeticiones innecesarias en 'if' compuestos

A veces podemos simplificar algunas expresiones lógicas. Por ejemplo, si tenemos una serie de valores y queremos comprobar si nuestra variable es igual a alguno de ellos:

No escribas esto:

```
>> x = 4
>> x == 3 or x == 5 or x == 7:
False
```

Mejor hacerlo así:

```
>> x in (3, 5, 7)
False
```

14. Repeticiones innecesarias en 'if' compuestos

any() devuelve True si al menos un elemento del iterable que recibe evalúa a verdadero. También devuelve False si el iterable está vacío.

```
>>> any([False, True, False])
True
>>> any([False, False])
False
>>> any([])
False
```

14. Repeticiones innecesarias en 'if' compuestos

¿Hay algún número par?

```
>>> numeros = [3, 7, 8, 9]
>>> any([not x % 2 for x in numeros])
True
```

Aún mejor: podemos usar un generador

```
>>> numeros = [3, 7, 9, 11]
>>> any(not x % 2 for x in numeros)
False
```

Built-in Functions: any()

http://docs.python.org/2/library/functions.html#any

14. Repeticiones innecesarias en 'if' compuestos

all() devueve True si todos los elementos del iterable evalúan a verdadero, o si éste está vacío.

```
>>> all([False, True, False])
False
>>> all([True])
True
>>> all([])
True
```

14. Repeticiones innecesarias en 'if' compuestos

¿Son pares todos los números?

```
>>> numeros = [1, 4, 6, 8]
>>> all(not x % 2 for x in numeros)
False
```

14. Repeticiones innecesarias en 'if' compuestos

Infinitamente mejor que la alternativa de, quizás:

```
>>> numeros = [1, 4, 6, 8]
>>> son_pares = True
>>> for x in numeros:
...     if x % 2:
...         son_pares = False
...         break
...
>>> son_pares
False
```

Built-in Functions: all()

http://docs.python.org/2/library/functions.html#all

15. pow(x, y, z)

La función pow() acepta, opcionalmente, un tercer argumento, z. En caso de especificarse, la función devuelve pow(x, y) % z. No es sólo práctico en algunos escenarios, sino que su implementación es más rápida que calcularlo nosotros en dos pasos.

```
>>> pow(2, 3)
8
>>> pow(2, 3, 6)
2
>>> pow(2, 5, 17)
15
```

Built-in Functions: pow()

http://docs.python.org/2/library/functions.html#pow

16. sorted() - parámetro 'key'

Desde Python 2.4, tanto el método list.sort() como la función sorted() aceptan el parámetro key, con el que se especifica la función que se ejecuta para cada elemento y a partir de cuyo resultado se ordena la secuencia.

Ordenar una serie de números por su valor absoluto:

```
>>> nums = [-8, 3, 5, -1, 3]
>>> sorted(nums, key=abs)
[-1, 3, 3, 5, -8]
```

Una lista de cadenas lexicográficamente:

```
>>> gnu = ["GNU", "is", "Not", "Unix"]
>>> sorted(gnu)
['GNU', 'Not', 'Unix', 'is']
```

16. sorted() - parámetro 'key'

Ahora por su longitud:

```
>>> sorted(gnu, key=len)
['is', 'GNU', 'Not', 'Unix']
```

Desde 2.2, las ordenaciones en Python son estables: al tener 'GNU' y 'Not' la misma longitud, se mantiene el orden original. Esto permite ordenar en función de dos o más criterios, en diferentes pasos.

16. sorted() - parámetro 'key'

Podemos ordenar usando nuestras propias funciones

Ordenar por el número de vocales:

```
>>> def nvocales(palabra):
...     contador = 0
...     for letra in palabra:
...         if letra.lower() in "aeiou":
...             contador += 1
...     return contador
...
>>> lame = "LAME Ain't an MP3 Encoder".split()
>>> sorted(lame, key=nvocales)
['MP3', 'an', 'LAME', "Ain't", 'Encoder']
```

16. sorted() - parámetro 'key'

O usando funciones anónimas

Ordenar por la parte compeja:

```
>>> nums = [7+9j, 4-5j, 3+2j]
>>> sorted(nums, key=lambda x: x.imag)
[(4-5j), (3+2j), (7+9j)]
```

Ordenar por el segundo elemento de la tupla:

```
>>> puntos = [(2, 3), (5, 1), (3, 4), (8, 2)]
>>> sorted(puntos, key=lambda x: x[1])
[(5, 1), (8, 2), (2, 3), (3, 4)]
```

Sorting Mini-HOW TO

http://wiki.python.org/moin/HowTo/Sorting/

17. Por qué "python" > [130, 129]

Listas y tuplas se ordenan lexicográficamente, comparando uno a uno los elementos. Para ser iguales, ambas secuencias han de tener el mismo tipo, la misma longitud y sus elementos ser iguales.

```
>>> [1, 2] == [1, 2]
True
>>> [1, 2] == [3, 4]
False
>>> [1, 2] == (1, 2)
False
```

17. Por qué "python" > [130, 129]

En caso de no tener el mismo número de elementos, la comparación de las dos secuencias está determinada por el primer elemento en el que sus valores diferen.

```
>>> (4, 2, 5) > (1, 3, 2)
True
>>> [2, 3, 1] >= [2, 4, 2]
False
```

17. Por qué "python" > [130, 129]

Si el elemento correspondiente, donde los valores difieren, no existe, la secuencia más corta va antes.

```
>>> (2, 3) > (1,2,3)
True
>>> [1, 2] < [1, 2, 3]
True
```

17. Por qué "python" > [130, 129]

Los objetos de tipos diferentes nunca se consideran iguales, y se ordenan de forma consistente pero arbitraria. En CPython, el criterio escogido es el del nombre de sus tipos.

```
>>> "python" > [130, 129]      # "str" > "list"
True
>>> {1 : "uno"} > [1, 2, 3]    # "dict" < "list"
False
>>> (23,) > [34, 1]            # "tuple" > "list"
True
```

Stack Overflow: Maximum of two tuples

http://stackoverflow.com/a/9576006/184363

17. Por qué "python" > [130, 129]

Pero un tipo de dato numérico va siempre antes si lo comparamos con uno no numérico.

```
>>> 23 < "Python"
True
>>> 23 < (3, 1, 4)
True
>>> 23 > [3, 1, 4]
False
>>> 23 < {"pi" : 3.1415}
True
```

Stack Overflow: How does Python compare string and int?

http://stackoverflow.com/a/3270689/184363

17. Por qué "python" > [130, 129]

Esto es un detalle de la implementación de CPython.

Nuestro código nunca debería depender de que valores de diferente tipos se ordenen así. Y, en cualquier caso, ¿qué haces comparando valores de diferente tipo? ¿Por qué querrías hacer eso?

En Py3K el comportamiento ya es el que esperaríamos:

```
>>> 23 < "Python"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: int() < str()
```

18. Aplanando una lista con sum()

sum(iterable[, start])

La función sum() acepta, opcionalmente, un segundo argumento, start, cuyo valor por defecto es cero, al cual se van sumando los elementos de iterable.

```
>>> sum([1, 2, 3])
6
>>> sum([1, 2, 3], 10)
16
>>> sum(range(1, 10))
45
>>> sum([100], -1)
99
```

18. Aplanando una lista con sum()

Podemos usar una lista vacía, [], como start, para aplanar una lista de listas. La función sum(), al 'sumarlas', lo que hará será concatenarlas una detrás de otra.

```
>>> sum([[1, 2], [3], [4, 5], [6, 7, 8]], [])
[1, 2, 3, 4, 5, 6, 7, 8]
>>> sum([[4, 5], [6, 7]], range(4))
[0, 1, 2, 3, 4, 5, 6, 7]
```

18. Aplanando una lista con sum()

Es importante que start sea una lista. De lo contrario, sum() intentará añadir cada una de las sublistas a un entero, cero. Y eso no se puede hacer.

```
>>> sum([[4, 5], [6, 7]])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'list'
```

Stack Overflow - Flattening a list with sum():

http://stackoverflow.com/a/6632253/184363

19. El operador ternario

El operator ternario (o condicional) nos permite escoger entre dos expresiones en función de una tercera, la condición.

En casi todos los lenguajes de programación tiene la misma forma:

```
test ? a : b
```

Por ejemplo, en C:

```
int nsecs = (counter > 5) ? 5 : 0;
```

19. El operador ternario

En Python se conocen como expresiones condicionales, y existen desde la versión 2.5. Su sintaxis es:

```
a if test else b
```

```
>>> x = 2
>>> "uno" if x == 1 else "otra cosa"
'otra cosa'
```

19. El operador ternario

```
>>> hora = 17
>>> msg = ("Es la" if hora == 1 else "Son las")
>>> msg + " {0}h".format(hora)
'Son las 17h'
```

Eleva al cuadrado si x es impar, divide entre dos de lo contrario:

```
>>> x = 13
>>> y = x ** 2 if x % 2 else x / 2
>>> y
169
```

PEP 308: Conditional Expressions [2003]

http://www.python.org/dev/peps/pep-0308/

20. Incremento != 1 en el operador slice

x[start:end:step]

El tercer índice, opcional, del operador slice, especifica el tamaño del paso — cuánto avanzamos cada vez. Por defecto es uno, y por eso:

Los tres primeros elementos

```
>>> x = range(1, 10)
>>> x[:3]
[0, 1, 2]
```

20. Incremento != 1 en el operador slice

Los cuatro últimos

```
>>> x[-4:]
[6, 7, 8, 9]
```

Del segundo al quinto

```
>>> x[1:5]
[1, 2, 3, 4]
```

20. Incremento != 1 en el operador slice

Podemos usar un paso distinto de uno

Elementos en posiciones pares...

```
>>> x[::2]
[1, 3, 5, 7, 9]
```

... e impares

```
>>> x[1::2]
[0, 2, 4, 6, 8]
```

20. Incremento != 1 en el operador slice

Recorrer los primeros nueve elementos, de tres en tres

```
>>> x[:9:3]
[0, 3, 6]
```

Hacia adelante, empezamos a contar en cero, terminamos en n-1

20. Incremento != 1 en el operador slice

Un paso negativo permite recorrer la lista hacia atrás

Invierte la lista

```
>>> x[::-1]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Recorre la lista hacia atrás, de dos en dos

```
>>> x[::-2]
[9, 7, 5, 3, 1]
```

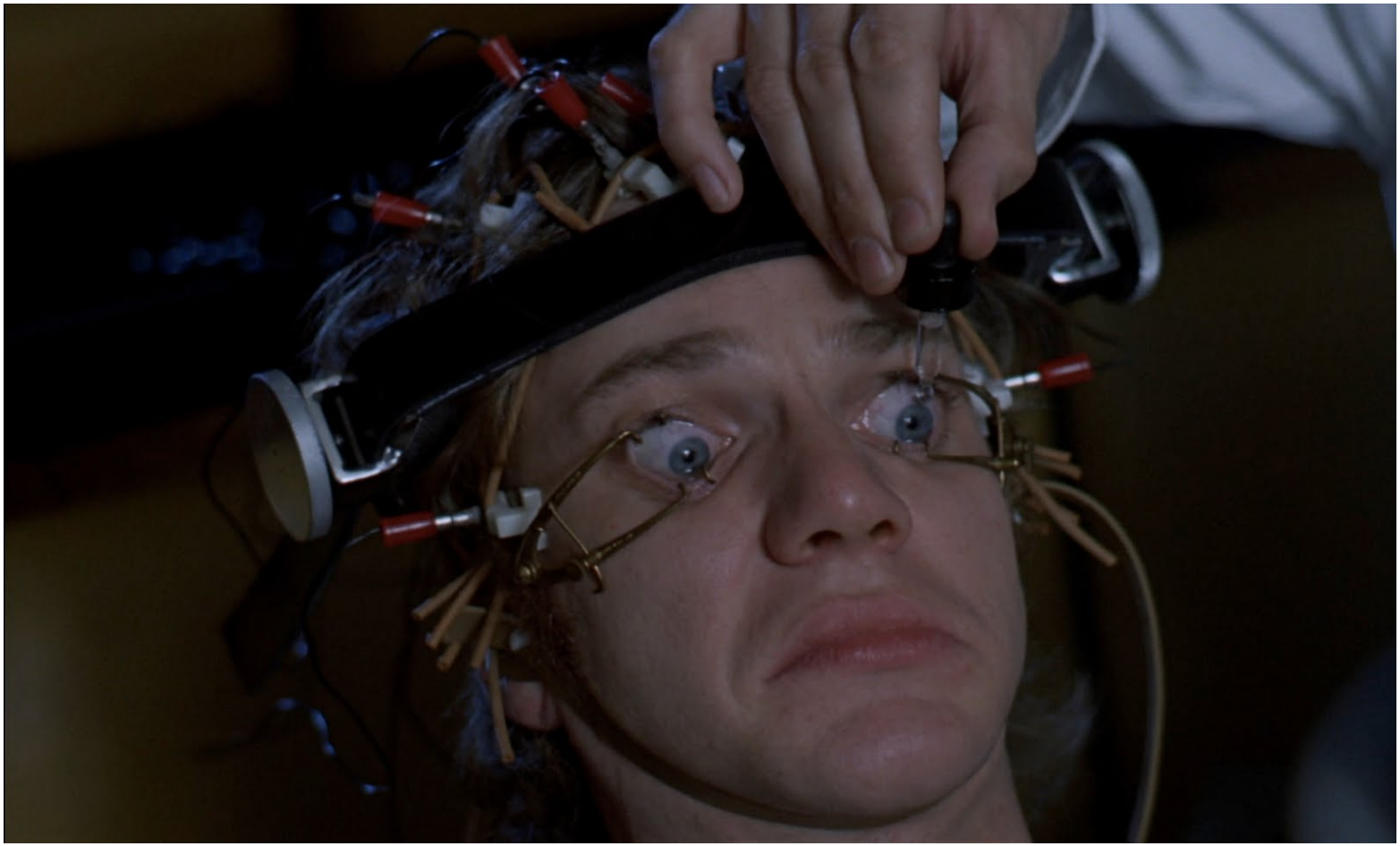
20. Incremento != 1 en el operador slice

Del último hasta el tercero, no inclusive, hacia atrás

```
>>> x[:2:-1]
[9, 8, 7, 6, 5, 4, 3]
```

Hacia atrás, empezamos a contar en -1, terminamos en -len

Respira profundamente



21. La variable _

En el modo interactivo (Python shell), la variable _ siempre contiene el valor del último *statement* ejecutado. Es muy útil si necesitamos reusar el último valor que hemos calculado y olvidamos almacenarlo en una variable:

```
>>> 3 * 4
12
>>> _ + 1
13
>>> _ ** 2
169
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> _[-1]
9
```

21. La variable _

En modo no-interactivo (un script en Python cualquiera), _ es un nombre más de variable. Por convención, suele usarse como nombre de variable para indicar que es desechable (*throwaway variable*), cuando necesitamos una pero no llegamos a usarla.

```
>>> for _ in range(10):
...     print "¡Hola, mundo!"
```

```
>>> import os.path
>>> path = "curriculum.tex"
>>> _, ext = os.path.splitext(path)
>>> ext
'.tex'
```

22. Funciones lambda, y qué opina GvR

Antes hemos usado funciones lambda (o anónimas) para definir sobre la marcha funciones sencillas que sólo necesitábamos una vez. Recordemos:

Ordenar por la parte compeja:

```
>>> nums = [7+9j, 4-5j, 3+2j]
>>> sorted(nums, key=lambda x: x.imag)
[(4-5j), (3+2j), (7+9j)]
```

Las funciones lambda son muy útiles y cómodas en algunas ocasiones. Provienen del mundo de la programación funcional, al igual que las funciones de orden superior map(), filter() y reduce().

22. Funciones lambda, y qué opina GvR

Pero a Guido van Rossum no le gustan.

Nunca le convencieron y, de hecho, estuvieron a punto de desaparecer en Py3K.

El nombre de lambda puede dar lugar a confusión, ya que su semántica no es la misma que en otros lenguajes. La idea de lambda era sólo la de servir de herramienta sintática para definir funciones anónimas. El problema es que nadie ha sido capaz todavía de encontrar una forma mejor forma esto, por lo que lambda sigue entre nosotros.

Origins of Python's "Functional" Features:

`http://python-history.blogspot.com.es/2009/04/origins-of-pythons-functional-features.html`

22. Funciones lambda, y qué opina GvR

Así que lambda sigue existiendo, pero lo Pythónico es usar el módulo operator. Más legible, más claro, más explícito.

Ordenando de nuevo por la parte compeja...

```
>>> import operator
>>> nums = [7+9j, 4-5j, 3+2j]
>>> sorted(nums, key=operator.attrgetter('imag'))
[(4-5j), (3+2j), (7+9j)]
```

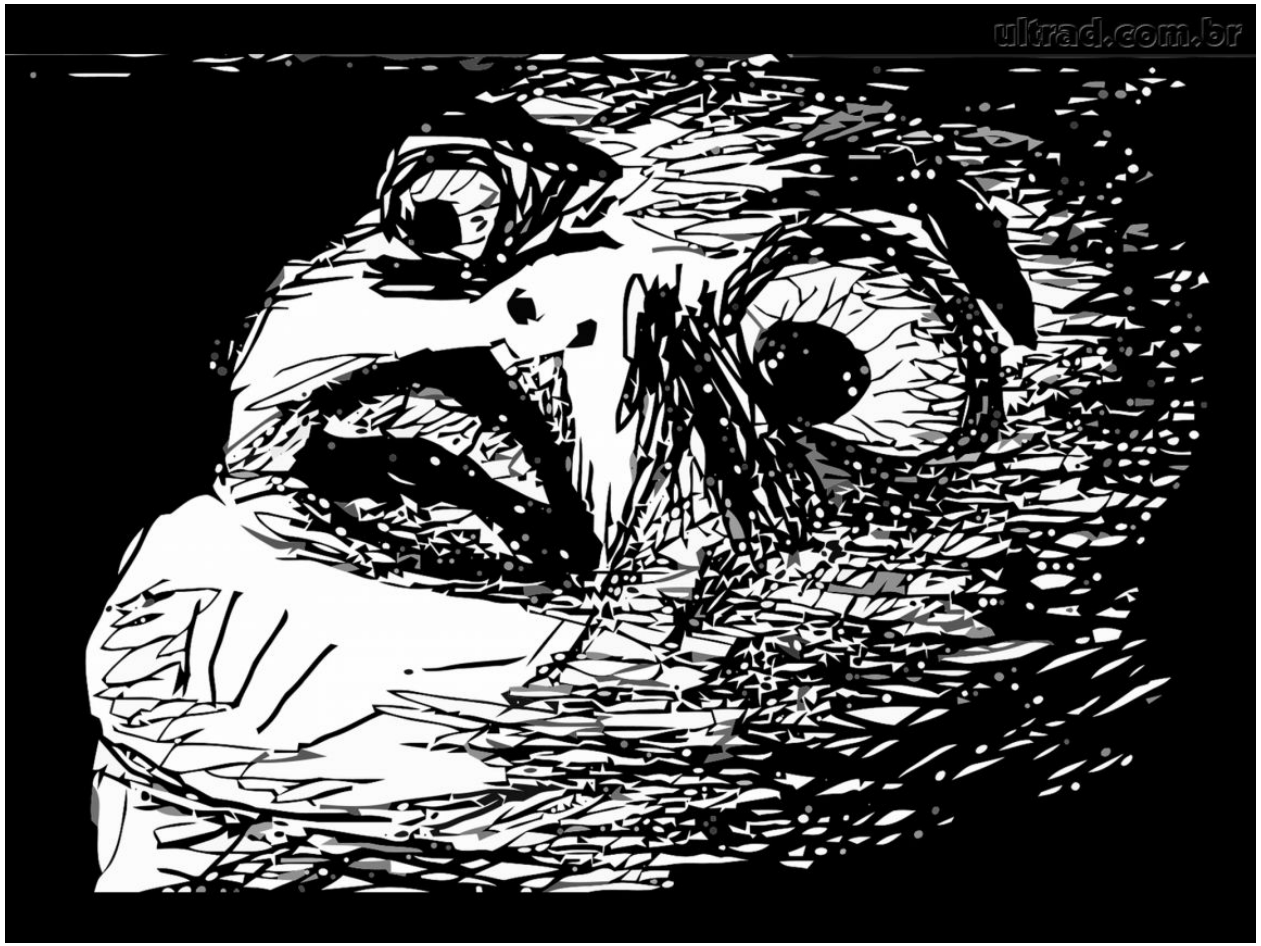
... y por el segundo elemento de la tupla:

```
>>> import operator
>>> puntos = [(2, 3), (5, 1), (3, 4), (8, 2)]
>>> sorted(puntos, key=operator.itemgetter(1))
[(5, 1), (8, 2), (2, 3), (3, 4)]
```

23. Parámetros por defecto mutables

```
>>> def inserta_uno(valores=[]):
...     valores.append(1)
...     return valores
...
>>> inserta_uno(range(3))
[0, 1, 2, 1]
>>> inserta_uno()
[1]
>>> inserta_uno()
[1, 1]
>>> inserta_uno()
[1, 1, 1]
```

23. Parámetros por defecto mutables



23. Parámetros por defecto mutables

El motivo por el que esto ocurre es que las funciones en Python son objetos de primera clase, no sólo un trozo más de código: se evalúan cuando son definidas, incluyendo parámetros por defecto, y por tanto su estado puede cambiar de una vez a otra.

Es decir, que los parámetros por defecto se evalúan sólo una vez, en el momento en el que la función se define.

23. Parámetros por defecto mutables

La solución es usar un parámetro de sustitución, en lugar de modificar directamente el valor por defecto.

```
>>> def inserta_uno(valores=None):
...     if valores is None:
...         valores = []
...     valores.append(1)
...     return valores
...
>>> inserta_uno()
[1]
>>> inserta_uno()
[1]
>>> inserta_uno()
[1]
```

23. Parámetros por defecto mutables

¿Y entonces esto?

```
>>> def suma(numeros, inicio=0):
...     for x in numeros:
...         inicio += x
...     return inicio
...
>>> suma([1, 2])
3
>>> suma([3, 4])
7
>>> suma([1, 2, 3])
6
```

23. Parámetros por defecto mutables

La razón es que, a diferencia de las listas, los enteros en Python son inmutables, por lo que no pueden modificarse de una llamada a otra. Por tanto, inicio siempre es cero cuando llamamos a la función.

Call By Object

`http://effbot.org/zone/call-by-object.htm`

Default Parameter Values in Python

`http://effbot.org/zone/default-values.htm`

24. *args y **kwargs

Nuestras funciones pueden recibir un número variable de argumentos.

```
>>> def imprime_punto2D(x, y):
...     print "({0}, {1})".format(x, y)
...
>>> def imprime_punto3D(x, y, z):
...     print "({0}, {1}, {2})".format(x, y, z)
...
>>> imprime_punto2D(1, 2)
(1, 2)
>>> imprime_punto3D(4, 3, 3)
(4, 3, 3)
```

24. *args y **kwargs

Podríamos generalizarlo a cualquier número de dimensiones:

```
>>> def imprime_punto(coords):
...     print "({0})".format(", ".join(str(x) for x in coords))
...
>>> imprime_punto([1, 2])
(1, 2)
>>> imprime_punto([1, 2, 3, 4])
(1, 2, 3, 4)
```

Pero esto nos obliga a pasar las coordenadas en una secuencia.

24. *args y **kwargs

Añadiendo un asterisco al nombre del parámetro cuando definimos la función podemos hacer que Python los agrupe todos a partir de ese punto en una tupla.

```
>>> def imprime_punto(*coords):
...     print "coordenadas: ", coords
...     print "({0})".format(", ".join(str(x) for x in coords))
...
>>> imprime_punto(2, 4)
coordenadas: (2, 4)
(2, 4)
>>> imprime_punto(1, 3, 2)
coordenadas: (1, 3, 2)
(1, 3, 2)
```

24. *args y **kwargs

Los parámetros listados antes del que tiene el asterisco se comportan normalmente.

```
>>> def imprime(x, y, *resto):
...     print "x:", x
...     print "y:", y
...     print "resto:", resto
...
>>> imprime(1, 2, 3, 7)
x: 1
y: 2
resto: (3, 7)
>>> imprime(5, 3)
x: 5
y: 3
resto: ()
```

24. *args y **kwargs

A la inversa, podemos usar el asterisco al llamar una función para desempaquetar una sequencia y asignar sus elementos, uno a uno, a los diferentes parámetros que la función recibe.

```
>>> def imprime_punto2D(x, y):
...     print "({0}, {1})".format(x, y)
...
coords = (4, 5)
>>> imprime_punto2D(coords)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: imprime_punto2D() takes exactly 2 arguments (1 given)
>>> imprime_punto2D(*coords)
(4, 5)
```

24. *args y **kwargs

Lo mismo puede hacerse, pero usando el doble asterisco, con los argumentos nombrados (*keyword arguments*): Python agrupa todos los que recibe en un diccionario.

```
>>> def imprime_punto(**coordenadas):
...     print "coordenadas: ", coordenadas
...     for clave, valor in coordenadas.items():
...         print "{0} = {1}".format(clave, valor)
...
>>> imprime_punto(x = 4, y = 8)
coordenadas:  {'y': 8, 'x': 4}
y = 8
x = 4
>>> imprime_punto(z = 3, x = 7, y = 2)
coordenadas:  {'y': 2, 'x': 7, 'z': 3}
y = 2
x = 7
z = 3
```

24. *args y **kwargs

Y, también, a la hora de llamar a una función podemos usar ** para desempaquetar un diccionario y pasar cada uno de sus elementos como un argumento nombrado.

```
>>> def imprime_punto2D(x = 0, y = 0):
...     print "x = {0}".format(x)
...     print "y = {0}".format(y)
...
>>> coords = {'x' : 1}
>>> imprime_punto2D(**coords)
x = 1
y = 0
>>> coords = {'x' : 3, 'y' : 7}
>>> imprime_punto2D(**coords)
x = 3
y = 7
```

24. *args y **kwargs

Es habitual usar *args y **kwargs como nombres de variables cuando definimos una función que recibe un número variable de parámetros o parámetros nombrados, respectivamente.

En GiHub hay 710K+ ejemplos en los que se usa esa notación:

`https://github.com/search?l=python&q=*args%2C+**kwargs&ref=searchresults&type=Code`

Magnus Lie Hetland — Beginning Python
Collecting Parameters (capítulo 6, "Abstraction")

`http://hetland.org/writing/beginning-python-2/`

25. for-else

El bucle for admite, opcionalmente, la cláusula else: este bloque se ejecuta sólo si el bucle for ha terminado *limpiamente* — es decir, si no se ha usado break para terminar antes de tiempo.

No usamos break, else se ejecuta

```
>>> for x in range(3):
...     print x
... else:
...     print "¡Hemos terminado!"
...
0
1
2
¡Hemos terminado!
```

25. for-else

Usamos break, else no se ejecuta

```
>>> for x in range(3):
...     print x
...     break
... else:
...     print "¡Hemos terminado otra vez!"
...
0
```

Esta construcción, for-else, nos ahorra la necesidad de usar una variable de bandera para almacenar el estado de, por ejemplo, una búsqueda.

25. for-else

Para lanzar una excepción si un elemento no ha sido encontrado en una secuencia, por ejemplo, podríamos usar una variable que fuera False y que, sólo si en algún momento encontramos el elemento a buscar, pasamos a True. Al terminar el bucle, actuamos en función del valor de esta variable.

```
def comprueba_que_esta(x, secuencia):
    """ Lanza ValueError si x no está en la secuencia """

    encontrado = False
    for elemento in secuencia:
        if elemento == x:
            encontrado = True
            break

    if not encontrado:
        msg = "{0} no está en {1}".format(x, secuencia)
        raise ValueError(msg)
```

25. for-else

La cláusula else nos permite escribirlo así:

```
def comprueba_que_esta(x, secuencia):
    for elemento in secuencia:
        if elemento == x:
            break
    else:
        msg = "{0} no está en {1}".format(x, secuencia)
        raise ValueError(msg)
```

25. for-else

Los bucles while también admiten la cláusula else.

Hay a quien no convence el nombre — y no sin motivo: de primeras, podríamos creer que else sólo se ejecuta si el bucle for no llega a ejecutarse nunca. Las razones detrás de este nombre son históricas, por cómo los compiladores implementaban los bucles while.

La forma de visualizar el else es que casi siempre está emparejado con un if: el que está dentro del bucle for. Lo que estamos diciendo es "*si ninguna de las veces el if dentro del for es cierta, entonces ejecuta el bloque de else*".

Ned Batchelder: For/else

`http://nedbatchelder.com/blog/201110/forelse.html`

26. try-except-else-finally

La construcción try-except-finally admite también la cláusula else: este bloque de código se ejecuta sólo si no se ha lanzado ninguna excepción.

Esto nos permite minimizar el código que escribimos dentro de la cláusula try, reduciendo el riesgo de que except capture una excepción que realmente no queríamos proteger con el try.

26. try-except-else-finally

Descarga presentación, descomprime, actualiza log

```
import os
import urllib
import zipfile

url = "http://github.com/vterron/PyConES-2013/archive/master.zip"
path = "charla-pycon.zip"

try:
    urllib.urlretrieve(url, path)
    myzip = zipfile.ZipFile(path)
    myzip.extractall()  # puede lanzar IOError

    # Tambien puede lanzar IOError
    with open("descargas-log", "at") as fd:
        fd.write("{0}\n".format(path))

except IOError:
    print "He capturado IOError"finally:os.unlink(path)
```

¿Cuál de los dos IOError he capturado? No lo sabemos

26. try-except-else-finally

Por supuesto, podríamos evitar este problema anidando try-elses:

```
try:
    urllib.urlretrieve(url, path)
    myzip = zipfile.ZipFile(path)
    try:
        myzip.extractall()
try:
            with open("descargas-log", "at") as fd:
                fd.write("{0}\n".format(path))
        except IOError:
            print "Error escribiendo fichero"

    except IOError:
        print "Error descomprimiendo fichero ZIP"

finally:
    os.unlink(path)
```

Pero, como dice el Zen de Python, "*Flat is better than nested*"

26. try-except-else-finally

Utilizando la cláusula else basta con:

```
try:
    urllib.urlretrieve(url, path)
    myzip = zipfile.ZipFile(path)
    myzip.extractall()
except IOError:
    print "Error descomprimiendo fichero ZIP"
else:
    with open("descargas-log", "at") as fd:
        fd.write("{0}\n".format(path))
finally:
    os.unlink(path)
```

27. Relanzamiento de excepciones

raise a secas, dentro del bloque de la cláusula except, relanza la excepción con la traza (traceback) original intanta. Esto es muy útil si queremos hacer algo más antes de lanzar la excepción.

```
>>> x = 1
>>> y = 0
>>> try:
...     z = x / y
... except ZeroDivisionError:
...     print "Imposible dividir. Abortando..."
...     raise
...
Imposible dividir. Abortando...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: integer division or modulo by zero
```

27. Relanzamiento de excepciones

Pero, claro, raise relanza la última excepción. En caso de que dentro del except exista la posibilidad de que se lance una segunda excepción, podemos almacenarla para asegurarnos de que no la perdemos.

En Python 3000:

```
import os

try:
    path = "resultado.txt"
    with open(path, "wt") as fd:
        resultado = 1 / 0
        fd.write(str(resultado))

except BaseException as e:

    try:
        os.unlink(path)
    except Exception as sub_e:
        print("Ha ocurrido otro error:")
        print(sub_e)

    raise e
```

27. Relanzamiento de excepciones

Esto produce:

```
>>> python3.1 relanza-excepcion.py
Ha ocurrido otro error:
[Errno 13] Permission denied: '/root/.bashrc'
Traceback (most recent call last):
  File "relanza.py", line 18, in <module>
    raise e
  File "relanza.py", line 8, in <module>
    resultado = 1 / 0
ZeroDivisionError: int division or modulo by zero
```

27. Relanzamiento de excepciones

En Python 2 también podemos usar la notación except Exception, e (sin los paréntesis), pero desde 2.6+ la keyword as permite que nuestro código sea mucho menos ambiguo.

Python try...except comma vs 'as' in except

`http://stackoverflow.com/a/2535770/184363`

PEP 3110: Catching Exceptions in Python 3000:

`http://www.python.org/dev/peps/pep-3110/`

28. __str__() vs __repr__()

El objetivo de __repr__() es ser inequívoco.

Este método, ejecutado vía repr(), devuelve una cadena de texto con la representación única del objeto. Se usa sobre todo para depurar errores, por lo que la idea es que incluya toda la información que necesitamos — por ejemplo, intentando entender qué ha fallado analizando unos logs.

El objetivo de __str__() es ser legible.

La cadena que devuelve str() no tiene otro fin que el de ser fácil de comprender por humanos: cualquier cosa que aumente la legibilidad, como eliminar decimales inútiles o información poco importante, es aceptable.

28. __str__() vs __repr__()

El módulo datetime nos proporciona un buen ejemplo:

```
>>> import datetime
>>> today = datetime.datetime.now()
>>> str(today)
'2013-11-20 13:51:53.006588'
>>> repr(today)
'datetime.datetime(2013, 11, 20, 13, 51, 53, 6588)'
```

28. __str__() vs __repr__()

Idealmente, la cadena devuelta por __repr__() debería ser aquella que, pasada a eval(), devuelve el mismo objeto. Al fin y al cabo, si eval() es capaz de reconstruir el objeto a partir de ella, esto garantiza que contiene toda la infomación necesaria:

```
>>> today = datetime.datetime.now()
>>> repr(today)
'datetime.datetime(2013, 11, 20, 13, 54, 33, 934577)'
>>> today == eval(repr(today))
True
```

28. __str__() vs __repr__()

Por cierto, ¡usar eval(repr(obj)) para serializar objetos no es buena idea! Es poco eficiente y, mucho peor, peligroso. Usa pickle mejor.

Eval really is dangerous:

`http://nedbatchelder.com/blog/201206/eval_really_is_dangerous.html`

28. __str__() vs __repr__()

__repr__() es para desarrolladores, __str__() para usuarios.

En caso de que nuestra clase defina __repr__() pero no __str__(), la llamada a str() también devuelve repr(). Así que el único que de verdad tenemos que implementar es __repr__().

Difference between __str__ and __repr__ in Python:

http://stackoverflow.com/q/1436703/184363

29. Implementando __hash__() en nuestras clases

El método __hash__() devuelve un entero que representa el valor hash del objeto. Es lo que se usa en los diccionarios, por ejemplo, para relacionar cada clave con su valor asociado. El único requisito es que los objetos que se consideren iguales (==) han de devolver también el mismo valor hash.

En caso de que no definamos __hash__(), nuestras clases devuelven id() – un valor único para cada objeto, ya que es su dirección en memoria.

29. Implementando __hash__() en nuestras clases

```
class Punto(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

p1 = Punto(1, 7)
p2 = Punto(2, 3)
p3 = Punto(2, 3)

print "id(p1)  : ", id(p1)
print "hash(p1): ", hash(p1)
print "hash(p2): ", hash(p2)
print "hash(p3): ", hash(p3)
```

Esto muestra por pantalla, por ejemplo:

```
id(p1)  :  140730517360144
hash(p1):  140730517360144
hash(p2):  140730516993872
hash(p3):  140730516993936
```

29. Implementando __hash__() en nuestras clases

Una forma de calcular el hash de nuestra clase, en los casos que no son demasiado complejos, es devolver el hash de una tupla que agrupa los atributos de nuestra clase. Rápido y sencillo:

```
class Punto(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __hash__(self):
        return hash((self.x, self.y))

p1 = Punto(3, 9)
p2 = Punto(3, 9)

print "id(p1)   : ", id(p1)
print "id(p2)   : ", id(p2)
print "hash(p1): ", hash(p1)
print "hash(p2): ", hash(p2)
```

29. Implementando __hash__() en nuestras clases

Ejecutado, esto muestra:

```
id(p1)   :   140050627503952
id(p2)   :   140050627504016
hash(p1):   3713083797000648531
hash(p2):   3713083797000648531
```

Python: What's a correct and good way to implement __hash__()?

http://stackoverflow.com/q/2909106/184363

29. Implementando __hash__() en nuestras clases

En caso de que dos objetos devuelvan el mismo hash, aún podemos usarlos en diccionarios. Esto es así porque los elementos se indexan no sólo por su valor hash, sino que Python también comprueba que no son iguales (==). Es sólo si son iguales en los dos sentidos que los dos objetos no pueden existir en el mismo diccionario.

Difference between __str__ and __repr__ in Python:

`http://stackoverflow.com/q/1436703/184363`

How hash collisions are resolved in Python dictionaries

`http://www.shutupandship.com/2012/02/how-hash-collisions-are-resolved-in.html`

30. {} y dict()

Habitualmente los diccionarios se contruyen usando

```
'uno': 1, 'dos': 2, 'tres': 3
```

También podemos hacerlo, sin embargo, utilizando dict(). Esto tiene la ventaja de que podemos definir los elementos del diccionario como argumentos nombrados — esto nos ahorra tener que poner comillas alrededor de las claves que sean cadenas de texto:

```
dict(uno = 1, dos = 2, tres = 3)
```

30. {} y dict()

Una limitación de dict() es que las claves de nuestro diccionario sólo pueden ser identificadores (nombres de variable) válidos en Python. Por el contrario, {} admite cualquier identificador.

Esto es perfectamente legal

```
>>> d = {1 : "uno", 2 : "dos", 3 : "tres"}
>>> d[1]
'uno'
```

Pero un entero no es un nombre de variable válido:

```
>>> dict(1 = "uno", 2 = "dos", 3 = "tres")
  File "<stdin>", line 1
SyntaxError: keyword can't be an expression
```

30. {} y dict()

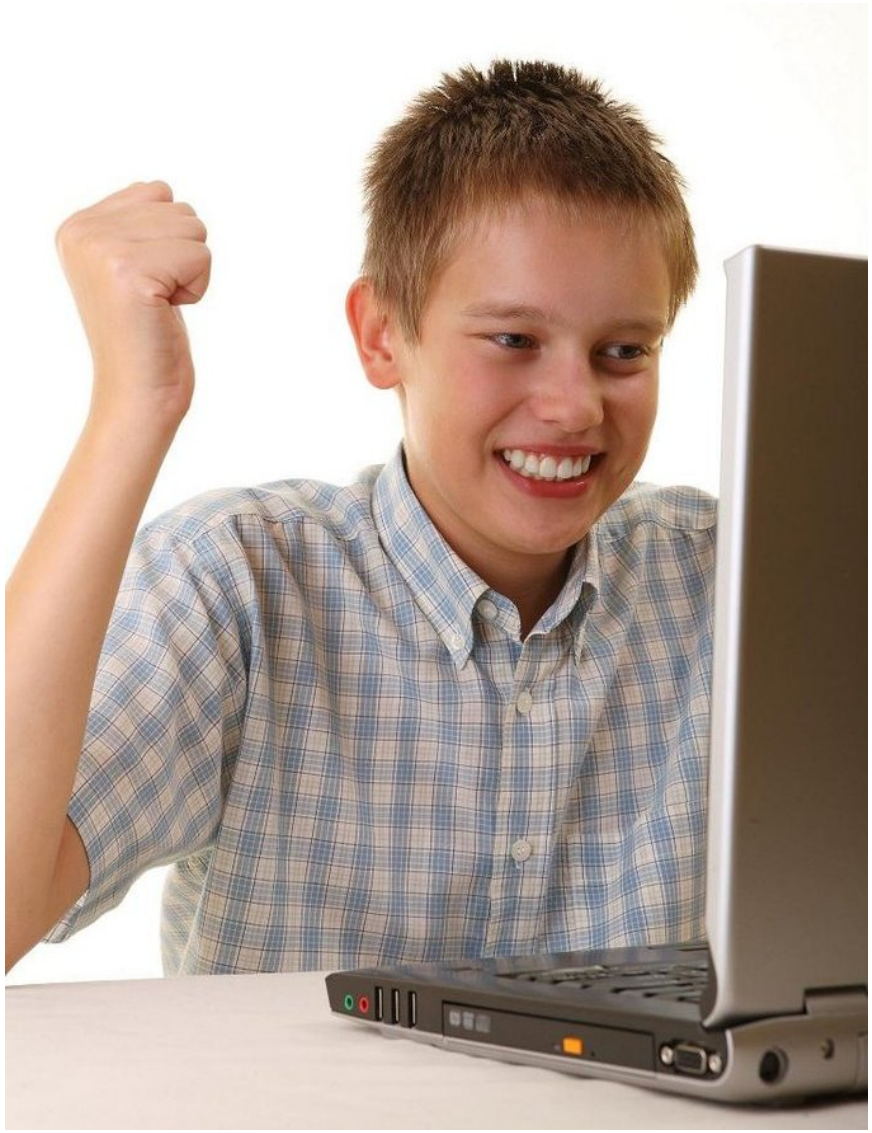
Esto es, en esencia, una cuestión de estilo – hay quien lo considera más cómodo y legible, pero dict() también tiene sus detractores.

Doug Hellmann, por ejemplo, ha cuantificado que es seis veces más lento que {}, y que consume más memoria.

The Performance Impact of Using dict() Instead of {} in CPython 2.7

`http://doughellmann.com/2012/11/the-performance-impact-of-using-dict-instead-of-in-cpython-2-7-2.html`

A veces nos sentimos así



31. dict.__missing__()

En las subclases de dict que definen __missing__(), el acceso a claves que no estén presentes en el diccionario devuelve lo que este método devuelva.

```
class MyDict(dict):
    """ Diccionario que devuelve -1 si la clave no existe """

    def __missing__(self, key):
        print "'{0}' no encontrado".format(key)
        return -1

>>> d = MyDict()
>>> d[1] = 3
>>> d[1]
3
>>> d[5]
'5' no encontrado
-1
```

31. dict.__missing__()

__missing__() no puede ser una variable – ha de ser un método, que recibirá como argumento la clave que no se ha encontrado.

Algo que no va a funcionar:

```
class MyDict(dict):
    """ Diccionario que devuelve -1 si la clave no existe """

    __missing__ = -1

>>> d = MyDict()
>>> d[7]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable
```

32. collections.defaultdict

Normalmente es mucho mejor, no obstante, usar defaultdict: se comporta como un diccionario a todos con los efectos, con la única diferencia de que al crearlo especificamos el valor que tendrán por defecto las claves a las que accedamos que no existan.

Para implementar un contador, en vez de hacer esto...

```
mydict = dict()
for letra in palabra:
    if letra in mydict:
        mydict[letra] += 1
    else:
        mydict[letra] = 1
```

32. collections.defaultdict

...o incluso esto...

```
mydict = dict()
for letra in palabra:
    try:
        mydict[letra] += 1
    except KeyError:
        mydict[letra] = 1
```

32. collections.defaultdict

Mucho mejor así:

```
import collections
mydict = collections.defaultdict(int)
for letra in palabra:
    mydict[letra] += 1
```

¡Pero esto es sólo un ejemplo! En código real nunca implementéis un contador así! Lo tenéis ya hecho por gente mucho más capaz que nosotros, y disponible desde Python 2.7, en este mismo módulo: collections.Counter.

32. collections.defaultdict

Valor por defecto: []

```
collections.defaultdict(list)
```

Valor por defecto: set()

```
collections.defaultdict(set)
```

Valor por defecto: {}

```
d = collections.defaultdict(dict)
d[0][3] = 3.43
d[1][5] = 1.87
```

32. collections.defaultdict

Para usar valores por defecto diferentes de cero o vacíos, tenemos que tener presente que lo que defaultdict.__init__() recibe es una función que acepta cero argumentos, que es la que se ejecuta cada vez que la clave no se encuentra.

Valor por defecto: -7

```
collections.defaultdict(lambda: -7)
```

Valor por defecto: números [0, 7]

```
collections.defaultdict(lambda: range(8))
```

33. collections.namedtuple

Devuelve un tipo de dato, subclase de tuple, que podemos usar para crear tuplas que además nos permiten acceder también por atributos.

Con namedtuple estamos definiendo nuestra propia clase en una línea de código – no deja de estar bien, aunque sean clases sencillas.

La clase Punto(x, y, z)

```
>>> Punto = collections.namedtuple('Punto', ['x', 'y', 'z'])
>>> Punto.__mro__
(<class '__main__.Punto'>, <type 'tuple'>, <type 'object'>
```

33. collections.namedtuple

Y ahora usándola:

```
>>> dest = Punto(1, 4, 3)
>>> dest.x
1
>>> dest.y
4
>>> dest[2]
3
```

Understanding Python's iterator, iterable, and iteration protocols

http://stackoverflow.com/q/9884132/184363

34. itertools.chain()

La función chain(), del módulo itertools, crea un iterador que devuelve, uno a uno, elementos de cada uno de los iterables que recibe, recorriéndolos todos.

```
>>> import itertools
>>> x = [1, 2, 3]
>>> y = [8, 9, 10]
>>> for numero in itertools.chain(x, y):
...     print numero
...
1
2
3
8
9
10
```

34. itertools.chain()

Aplanar una lista de listas, como hicimos antes:

```
>>> it = itertools.chain([1, 2], [3], [4, 5], [6, 7, 8])
>>> it
<itertools.chain object at 0x2536590>
>>> list(it)
[1, 2, 3, 4, 5, 6, 7, 8]
>>> list(it)
[]
```

Utilizando * para desempaquetar las sublistas:

```
>>> sublistas = [[4, 5], [6, 7]]
>>> list(itertools.chain(*sublistas))
[4, 5, 6, 7]
```

34. itertools.chain()

Pero chain() acepta mucho más que listas: le podemos pasar cualquier cosa sobre la que se pueda iterar. Esto incluye generadores, conjuntos, diccionarios y cualquier objeto cuyos elementos podemos recorrer de uno en uno.

Dos conjuntos:

```
>>> s1 = set([1, 2, 3])
>>> s2 = set([6, 5, 4])
>>> list(itertools.chain(s1, s2))
[1, 2, 3, 4, 5, 6]
```

Diccionaro (claves) y conjunto:

```
>>> d1 = {1 : "uno", 2 : "dos"}
>>> s1 = set(["tres", "cuatro"])
>>> list(itertools.chain(d1, d2))
[1, 2, 'cuatro', 'tres']
```

35. functools.partial()

En el módulo functools, la función partial() nos permite *congelar* otra función, fijándole argumentos de forma que podamos llamarla de forma más sencilla y corta.

```
>>> import functools
>>> eleva_dos = functools.partial(pow, 2)
```

Y ahora usándola:

```
>>> eleva_dos(3)
8
>>> eleva_dos(10)
1024
```

35. functools.partial()

Es importante tener presente que los argumentos que nosotros le pasamos a la función *congelada* se añaden después de los que hemos definido en partial(). Es necesario dar un rodeo si queremos crear funciones parciales que añadan los argumentos por el comienzo – por ejemplo, para devolver pow(x, 3).

Python: Why is functools.partial necessary?

`http://stackoverflow.com/a/3252425/184363`

implementing functools.partial that prepends additional arguments

`http://stackoverflow.com/q/11831070/184363`

36. Pasando funciones a iter()

iter(function, sentinel)

El uso más habitual de iter() es pasarle un único argumento — el objeto sobre el que queremos iterar. Pero también podemos pasarle una función, que es ejecutada una y otra vez: en el momento en el que uno de los valores que devuelve sea igual a sentinel, nos detenemos.

36. Pasando funciones a iter()

Lee un fichero hasta la primera línea vacía:

```
with open('currículum.tex') as fd:
    for linea in iter(fd.readline, '\n'):
        print linea
```

36. Pasando funciones a iter()

Genera números aleatorios hasta llegar a cero:

```
import random

def aleatorio():
    return random.randint(-10, 10)

for numero in iter(aleatorio, 0):
    print numero
```

37. Autocompletado en el intérprete

En nuestro fichero .pythonrc, estas líneas bastan para habilitar el autocompletado, que se activa al pulsar tabulador, como estamos acostumbrados a hacer en la línea de comandos.

```
import readline
import rlcompleter
readline.parse_and_bind("tab: complete")
```

¡La variable de entorno PYTHONSTARTUP debe apuntar a este fichero!

38. Historia de comandos

También podemos habilitar la historia de todo lo que hemos escrito en el intérprete de Python, accesibles pulsando las teclas de navegación vertical. Al igual que en una terminal cualquiera, la historia se guarda en un fichero, por lo que podemos reusar los comandos de una sesión a otra.

```
import atexit
import os.path
import readline

history_file = os.path.expanduser('~/.python_history')
if os.path.exists(history_file):
    readline.read_history_file(history_file)
func = readline.write_history_file
atexit.register(func, history_file)
```

39. Límite de memoria

Antes hemos mencionado que crear una lista de tamaño excesivo podría usar toda la memoria del ordenador, a veces con trágicas consecuencias. Usando esto podemos impedir que Python use más del 3/4 de la memoria total de nuestro equipo, abortando la ejecución si sobrepasamos este límite.

```
def get_ram_size():
    with open('/proc/meminfo', 'rt') as fd:
        regexp = "^MemTotal:\s*(\d+) kB$"
        match = re.match(regexp, fd.read(), re.MULTILINE)
        return int(match.group(1)) * 1024 # kB to bytes
MAX_RAM = get_ram_size() * 0.75
resource.setrlimit(resource.RLIMIT_AS,
                   (MAX_RAM, resource.RLIM_INFINITY))
```

39. Límite de memoria

Esto es lo que ocurre entonces si abarcamos demasiado:

```
>>> list(xrange(10e12))
Traceback (most recent call last):
  File ‘‘<stdin>’’, line 1, in <module>
MemoryError
```

Por si alguien quiere reusar mi fichero .pythonrc

http://github.com/vterron/dotfiles

40. a, *b, c = range(5)

Un detalle precioso de Py3K

```
>>> primero, *resto = range(5)
>>> primero
0
>>> resto
[1, 2, 3, 4]
>>> primero, *resto, ultimo = range(10)
>>> primero
0
>>> resto
[1, 2, 3, 4, 5, 6, 7, 8]
>>> ultimo
9
```

Programadores Python Shaolín