Python 2 vs Python 3

La función print

La principal diferencia entre la versión 2 y la versión 3 reside en el uso de la función de print. Mientras que en Python 2 podíamos usarla sin paréntesis, en la versión 3 debemos usarla con paréntesis envolviendo al objeto a mostrar por línea de comandos. La versión 2 no tiene problemas con los paréntesis, pero la versión 3 genera una excepción si no se incluyen.

	Python 2	**Python 3**

Ejemplo	`print 'Python', python_version()`
	`print (‘Hola Mundo’)`
	`print ‘Hola Mundo’`
		`print ('Python', python_version())`
		`print (‘Hola Mundo’)`
		`print ‘Hola Mundo’`

Salida	`Python 2.7.6`
	`Hola Mundo`
	`Hola Mundo`
		`Python 2.7.6`
		`Hola Mundo`
		`File "<ipython-input-3-139a7c5835bd>", line 1`
		`print 'Hola Mundo'`
		`^`
		`SyntaxError: invalid syntax`

División entera

En Python 2.x cuando realizamos la división entre enteros el resultado es otro entero, perdiendo los posibles decimales de la división. En cambio, en Python 3.x, la división entre enteros da como resultado un número real, es decir, con decimales. Luego debemos tener cuidado con este cambio.

	Python 2	**Python 3**
Ejemplo	`print ‘3/2={0}’.format(3/2)` `print ‘3/2={0}’.format(3/2.0)`	`print (‘3/2=’,3/2)` `print (‘3/2=’,3/2.0)`
Salida	`3/2=1` `3/2=1.5`	`3/2=1.5` `3/2=1.5`

Cadenas Unicode

En Python 2.x teníamos dos tipos de cadenas, str() y el tipo Unicode(), pero no teníamos un tipo byte.

En Python 3.x tenemos cadenas Unicode utf-8 y dos tipos de clases byte: byte y bytearray.

	Python 2	**Python 3**

Ejemplo	`print type(Unicode(‘cadena’))`
	`print type(b’cadena’)`
	`‘cadena ’+b’no da error’`
		`print (‘cadena utf-8 \u03BCnico\u0394é ‘)`
		`print (type(b’bytes para los datos’))`

		print(type(bytearray(b'array'))) 'cadena '+b'bytes da error'
Salida	<type 'unicode'> <type 'str'> 'cadena no da error'	cadena utf-8 µnicoΔé! <class 'bytes'> <class 'bytearray'> TypeError Traceback (most recent call last) <ipython-input-13-d3e8942ccf81> in <module>() ----> 1 'cadena '+b'bytes da error' TypeError: Can't convert 'bytes' object to str implicitly

Excepciones

Mientras que en Python 2.x tenemos disponibles las dos versiones que nos permiten lanzar excepciones, en Python 3.x debemos utilizar la versión con paréntesis.

	Python 2	Python 3
Ejemplo	raise IOError, "error en un fichero" raise IOError("error en un fichero"	raise IOError, "error en un fichero" raise IOError("error en un fichero"
Salida	IOError Traceback (most recent call last) <ipython-input-8-25f049caebb0> in <module>() ----> 1 raise IOError, "error en un fichero" IOError: error en un fichero ------------------------------ IOError Traceback (most recent call last) <ipython-input-8-25f049caebb0> in <module>() ----> 1 raise IOError("error en un fichero ") IOError: error en un fichero	File "<ipython-input-10-25f049caebb0>", line 1 raise IOError, "error en un fichero" SyntaxError: invalid syntax ------------------------------ IOError Traceback (most recent call last) <ipython-input-8-25f049caebb0> in <module>() ----> 1 raise IOError("error en un fichero ") IOError: error en un fichero

También tenemos una diferencia entre Python 3 y Python 2 a la hora de gestionar las excepciones. En Python 3 debemos usar la palabra clave 'as'.

	Python 2	Python 3

Ejemplo	`try:`
	`  funcion_exc`
	`except NameError, err:`
	`  print err,"→ mensaje"`
		`try:`
		`  funcion_exc`
		`except NameError as err:`
		`  print (err,"→ mensaje")`

Salida	`name 'funcion_exc' is not defined --> our error message`	`name 'funcion_exc' is not defined --> our error message`

Bucles

En Python 2.x usamos frecuentemente la función xrange para crear una lista que nos permita iterar. Dado que xrange es una función perezosa es mucho más eficiente que la función estándar range(). En Python 3.x desparece esta función que es sustituida por range() directamente.

Otra novedad en Python 3.x es que las variables definidas dentro de los bucles como índices de iteración no modifican a las variables del entorno global como ocurre en Python 2.x.

También hay que tener en cuenta que las listas comprimidas tienen una semántica diferente, están más cerca de los generadores dentro de un constructor list(), ya que la variables locales a la definición no son visibles fuera de la misma.

	Python 2	Python 3

Ejemplo	`i = 1`
	`print 'before: i =', i`
	
	`print 'comprehension: ', [i for i in range(5)]`
	
	`print 'after: i =', i`
		`lista=(letra for letra in 'abcdefg')`
		`next(lista)`
		`lista.next()`

Salida	`'b'`	`'a'`
		`AttributeError`
		`Traceback (most recent call last)`
		
		`<ipython-input-14-125f388bb61b> in <module>()`
		`----> 1 lista.next()`
		
		`AttributeError: 'generator' object has no attribute 'next'`

Iteradores

Uno de los métodos más populares para iterar sobre listas es el método next cuya sintaxis ha cambiado entre las versiones 2.x y 3.x. Mientras que en Python 2.x podemos invocarlo como un método de la lista (lista.next()); en Python 3.x sólo podemos llamarlo sobre la lista directamente (next(lista)).

	Python 2	Python 3

Ejemplo	`lista=(letra for letra in 'abcdefg')`
	`lista.next()`
	`next(lista)`
		`lista=(letra for letra in 'abcdefg')`
		`next(lista)`
		`lista.next()`

Salida	`'b'`	`'a'`
		`AttributeError`
		`Traceback (most recent call last)`
		
		`<ipython-input-14-125f388bb61b> in <module>()`
		`----> 1 lista.next()`
		
		`AttributeError: 'generator' object has no attribute 'next'`

Este cambio significa que algunos métodos nos van a devolver iteradores en lugar de listas para recorrer colecciones, como van a ser los métodos zip(), map(), los métodos keys(), values() e ítem() de los diccionarios entre otros.

Comparar tipos

Otra novedad es que en caso de intentar comparar tipos de datos que no se puede ordenar en el caso de Python 3 se produce la excepción TypeError.

	Python 2	Python 3
Ejemplo	`print "cadena"> [1,2]`	`print("cadena"> [1,2])`

Salida	`False`	`---------------------------------------------------------------------------`
		`TypeError`
		`Traceback (most recent call last)`
		`<ipython-input-16-a9031729f4a0> in <module>()`
		`----> 1 print("cadena"> [1,2])`

Redondeo

Python 3 ha cambiado la forma ahora de redondear decimales cuando resulta en un empate (.5) en los últimos dígitos significativos. Ahora, en Python 3, los decimales se redondean al número par más cercano. Aunque es un inconveniente para la portabilidad de código, supuestamente es una mejor forma de redondear en comparación con el redondeo anterior, ya que evita el sesgo hacia los grandes números

Compatibilidad entre versiones

Python 3.x introduce ciertas palabras clave y características que no son compatibles con Python 2.x. Para garantizar que nuestro código en Python 2.x es compatible con la versión 3.X debemos acordarnos de usar siempre los paréntesis. Además podemos agregar características de Python 3.x mediante el módulo __future__. Por ejemplo, si quisiéramos utilizar la división con decimales bastaría con importar en nuestro módulo la división de la siguiente manera:

```
from __future__ import division
```

Otras características que podemos importar son:

Característica	Disponible desde la version	Obligatoria desde la versión
nested_scopes	2.1.0b1	2.2
generators	2.2.0a1	2.3
division	2.2.0a2	3.0
absolute_import	2.5.0a1	3.0
with_statement	2.5.0a1	2.6
print_function	2.6.0a2	3.0
unicode_literals	2.6.0a2	3.0

Bibliografía utilizada para este resumen

- What's new in Python 3.0: https://docs.python.org/3.0/whatsnew/3.0.html
- The key differences between Python 2.7.x and Python 3.x with examples: http://sebastianraschka.com/Articles/2014_python_2_3_key_diff.html
- Porting Python 2 code to Python 3: https://docs.python.org/3/howto/pyporting.html
- 10 awesome features of Python that you can't use because you refuse to upgrade to Python 3: http://www.asmeurer.com/python3-presentation/slides.html#1