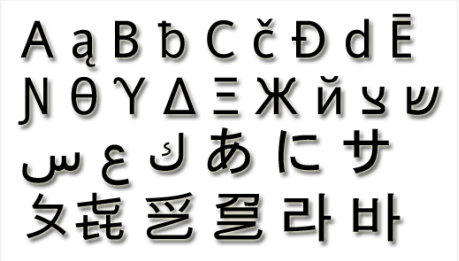
Python 3 para impacientes

"Simple es mejor que complejo" (Tim Peters)

sábado, 12 de julio de 2014

Tipos de cadenas: Unicode, Byte y Bytearray



Una diferencia importante entre Python 2.x y Python 3.x está en la codificación que se utiliza, por defecto, cuando se declaran cadenas de texto. En el primer caso se usa la codificación **ASCII** y en el segundo **Unicode**.

Un poco de historia

La codificación ASCII nos acompaña desde los años 1960 y se desarrolló para tecnologías de 8 bits. ASCII estándar, concretamente, utiliza los 7 primeros bits para codificar la información y el número 8 es el *bit de paridad* usado para controlar errores.

Con los 7 bits se diseñó un juego de 128 caracteres (en binario 2 elevado a 7), partiendo del carácter identificado con el número 0 y terminando en el número 127. Entre ellos, los llamados caracteres de control -no imprimibles- ocupan las primeras 32 posiciones, del 0 al 31; y a partir de la 32 y hasta la 127 se encuentran los usados para representar los caracteres numéricos, alfabéticos para minúsculas, alfabéticos para mayúsculas, signos de puntuación, matemáticos y otros que, en conjunto, son los llamados caracteres imprimibles

(espacio) ! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
@ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [\ ] ^ _
` a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~

Está codificación estaba pensada para el alfabeto inglés; con el español teníamos problemas si pretendíamos, por ejemplo, representar caracteres propios donde aparece la tilde, la diéresis, nuestra particular “ñ” y otros.

A partir de ASCII estándar surgieron distintas tablas que utilizaban ya los 8 bits en lugar de 7 para codificar la información, pudiéndose representar con un bit más (2 elevado a 8) hasta 256 caracteres. Estas tablas de codificación, para mantener la compatibilidad compartían con ASCII los primeros 128 caracteres (del 0 al 127) y el resto (del 128 al 255) se emplearon para codificar todos los caracteres particulares de nuestra lengua (y de otras). Un ejemplo bastante común es la codificación CP-1252 (o Windows-1252):

Buscar

Buscar

Python para impacientes

Python
IPython
EasyGUI
Tkinter
JupyterLab
Numpy

Anexos

Guía urgente de MySQL
Guía rápida de SQLite3

Entradas + populares

Dar color a las salidas en la consola
En Python para dar color a las salidas en la consola (o en la terminal de texto) existen varias posibilidades. Hay un método basado ...

Instalación de Python, paso a paso
Instalación de Python 3.6 A finales de 2016 se produjo el lanzamiento de Python 3.6 . El propósito de esta entrada es mostrar, pas...

Añadir, consultar, modificar y suprimir elementos en Numpy
Acceder a los elementos de un array. [], [,], ... Acceder a un elemento de un array. Para acceder a un elemento se utiliz...

Variables de control en Tkinter
Variables de control Las variables de control son objetos especiales que se asocian a los widgets para almacenar sus valore...

Cálculo con arrays Numpy
Numpy ofrece todo lo necesario para obtener un buen rendimiento cuando se trata de hacer cálculos con arrays. Por como está concebid...

Tkinter: interfaces gráficas en Python
Introducción Con Python hay muchas posibilidades para programar una interfaz gráfica de usuario (GUI) pero Tkinter es fácil d...

Operaciones con fechas y horas. Calendarios
Los módulos datetime y calendar amplían las posibilidades del módulo time que provee funciones para manipular expresiones de ti...

Convertir, copiar, ordenar, unir y dividir arrays Numpy
Esta entrada trata sobre algunos métodos que se utilizan en Numpy para convertir listas en arrays y viceversa; para copiar arrays d...

Tkinter: Tipos de ventanas
Ventanas de aplicación y de diálogos En la entrada anterior tratamos los distintos gestores de geometría que se utilizan para di...

El módulo random
El módulo random de la librería estándar de Python incluye un conjunto de funciones

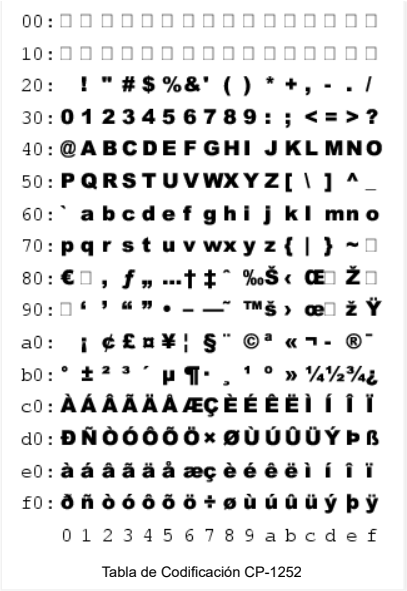


Tabla de Codificación CP-1252

La “Ñ” -mayúscula- es el carácter que ocupa la posición 209 de la tabla y la “ñ” -minúscula-, aparece más adelante, en la número 241.

Fue un gran avance pero, irremediablemente, hizo falta profundizar más para resolver codificaciones como la del árabe o, más complejas, como la china, coreana o japonesa basadas en ideogramas en las que se necesitan representar miles de caracteres, que es mucho más de lo que permiten los 8 bits.

A lo largo del tiempo surgieron otras codificaciones hasta que la globalización y la necesidad de intercambio de información en distintos sistemas y en diferentes idiomas hizo concentrar esfuerzos en el desarrollo de un proyecto universal de codificación llamado **Unicode**.

"**Unicode** es un estándar de codificación de caracteres diseñado para facilitar el tratamiento de textos de múltiples lenguajes, incluido los basados en ideogramas o aquellos usados en textos de lenguas muertas. El término Unicode proviene de los objetivos perseguidos durante el desarrollo del proyecto: universalidad, uniformidad y unicidad".

En **Unicode** los caracteres alfabéticos, los ideogramas y los símbolos se tratan de forma equivalente y se pueden mezclar entre sí en un mismo texto, es decir, es posible representar en un mismo párrafo caracteres del alfabeto árabe, cirílico, latino, ideogramas japoneses y símbolos musicales.

Para hacernos una idea del volumen de caracteres que es capaz de representar Unicode, señalar que su versión 5.1 contiene más de 100.000.

Como comentábamos al principio, en Python 3.x todas las cadenas de texto cuando se declaran son secuencias de caracteres **Unicode**; no existen cadenas codificadas en CP-1252 o en UTF-8 y, por tanto, no sería correcto hablar de codificaciones específicas si no es para decir que es posible convertir una cadena de caracteres en una secuencia de bytes (o viceversa) con una codificación determinada (como UTF-8, por poner un ejemplo).

Algo que debe entenderse (e insiste Mark Pilgrim en su libro *Dive into Python*) es que “los bytes no son caracteres, los bytes son bytes; un carácter es en realidad una abstracción; y una cadena de caracteres es una sucesión de abstracciones”.

Cadenas Unicode, Byte y Bytearray

En Python 3.x las cadenas de caracteres pueden ser de tres tipos: **Unicode**, **Byte** y **Bytearray**.

El tipo **Unicode** permite caracteres de múltiples lenguajes y cada carácter en una cadena tendrá un valor *inmutable*. El tipo **Byte** sólo permitirá caracteres ASCII y los caracteres son también *inmutables*. Y, finalmente, el tipo **Bytearray** es como el tipo Byte pero, en este caso, los caracteres de una cadena si son *mutables*.

Más adelante, con un caso práctico, estudiaremos las características de mutabilidad e inmutabilidad. A continuación, iniciaremos una sesión interactiva en Python 3.x para ver algunos ejemplos en los que se declaran los distintos tipos de cadenas y se estudia cómo hacer conversiones. La función **type**() la utilizaremos con frecuencia para conocer en cada momento el tipo de datos que se obtiene, según las siguientes salidas:

<class 'str'> → cadena Unicode
<class 'bytes'> → cadena Byte
<class 'bytearray'> → cadena Bytearray

Declarar y convertir cadenas

Para **declarar** una cadena de texto **Unicode** es necesario utilizar las comillas (simples o dobles)

que permiten obtener de distintos modos números a...

Archivo

julio 2014 (1)

python.org

pypi.org

Sitios

- ActivePython
- Anaconda
- Bpython
- Django
- Flask
- Ipython
- IronPython
- Matplotlib
- MicroPython
- Numpy
- Pandas
- Pillow
- PortablePython
- PyBrain
- PyCharm
- PyDev
- PyGame
- Pypi
- PyPy
- Pyramid
- Python.org
- PyTorch
- SciPy.org
- Spyder
- Tensorflow
- TurboGears

para delimitarla.

```
>>> lenguaje = "Python"
>>> type(lenguaje)
<class 'str'>
```

Para **declarar** una cadena de texto **Byte** es necesario emplear las comillas (simples o dobles) para delimitarla y anteponer el carácter “b”:

```
>>> lenguaje = b"Python"
>>> type(lenguaje)
<class 'bytes'>
```

Para **declarar** una segunda cadena **Unicode**, en la que se incluye la “**ñ**”:

```
>>> pais = "España"
>>> type(pais)
<class 'str'>
```

Pero **declarar** una segunda cadena **Byte** en la que se incluya la “**ñ**” no es posible, en principio, porque este carácter no se encuentra en la tabla de ASCII estándar. Se producirá un error de sintaxis.

```
>>> pais = b"España"
  File "<stdin>", line 1
SyntaxError: bytes can only contain ASCII literal characters.
>>> type(pais)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pais' is not defined
```

¿Qué podemos hacer? ¿Hay alguna forma de representar la “ñ” como una cadena tipo Byte? Hay una posibilidad que consiste en **convertir** una **cadena Unicode en Byte**, utilizando la función ***bytes***() en la que indicaremos alguna codificación determinada.

Con la codificación “utf-8” sería:

```
>>> pais = bytes("España", "utf-8")
>>> print(pais)
b'Espa\xc3\xb1a'
>>> type(pais)
<class 'bytes'>
```

Con la codificación “latin1” sería:

```
>>> pais = bytes("España", "latin1")
>>> print(pais)
b'Espa\xf1a'
>>> type(pais)
<class 'bytes'>
```

Con la codificación “cp-1252” sería:

```
>>> pais = bytes("España", "cp1252")
>>> print(pais)
b'Espa\xf1a'
>>> type(pais)
<class 'bytes'>
```

Con la codificación “cp-1252” y con la “Ñ” en mayúsculas sería:

```
>>> pais = bytes("EspaÑa", "cp1252")
>>> print(pais)
b'Espa\xd1a'
>>> type(pais)
<class 'bytes'>
```

Observa los dos ejemplos anteriores. En ellos, la “ñ” en minúsculas al imprimirse es representada en hexadecimal con el número “f1” (\xf1”) que en decimal se corresponde con el número 241. En cambio, la “Ñ” en mayúsculas es representada en hexadecimal con el número “d1” (“\xd1”) que en decimal es el número 209.

Otra forma de **convertir** una **cadena Unicode a Byte**, mediante la función **encode**()

```
>>> pais = "España"
>>> pais.encode("cp1252")
b'Espa\xf1a'
```

Para hacer lo inverso, es decir, **convertir** una **cadena Byte a Unicode** podemos utilizar la función **decode**(). Es este caso, para decodificar es imprescindible especificar la codificación adecuada con la que se obtendrá el resultado correcto.

```
>>> pais = "España"
>>> pais2 = pais.encode("cp1252")
>>> pais2.decode("cp1252")
'España'
```

Si intentamos decodificar con una codificación inadecuada podemos llevarnos una sorpresa, en forma de error,

```
>>> pais = "España"
>>> pais2 = pais.encode("cp1252")
>>> pais2.decode("utf-8")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xf1 in position 4: invalid continuation byte
```

o bien, obtener una conversión no deseada:

```
>>> pais = "España"
>>> pais2 = pais.encode("cp1252")
>>> pais2.decode("utf-16")
'獩摩慤'
```

También, en caso de que se produzca un error de conversión podemos hacer que éste sea ignorado:

```
>>> pais = "España"
>>> pais2 = pais.encode("cp1252")
>>> pais2.decode("utf-8", “ignore”)
'Espaa'
```

o bien, forzar un cambio de caracteres aunque no sea coherente:

```
>>> pais = "España"
>>> pais2 = pais.encode("cp1252")
>>> pais2.decode("utf-8", “replace”)
'Espa�a'
```

que podría haberse expresado de otra forma diferente:

```
>>> pais2.decode(errors="replace")
'Espa�a'
```

Para **declarar** una cadena **ByteArray** utilizaremos la función **bytearray**():

```
>>> vehiculo = bytearray("camion", "cp1252")
>>> type(vehiculo)
<class 'bytearray'>
```

Una cadena **Bytearray** es como **Byte** pero mutable, es decir, podemos acceder a una posición de la secuencia y cambiar el carácter.

En el siguiente ejemplo el carácter número 243 que se corresponde con “ó” lo asignamos a la quinta posición de la secuencia (índice número 4). Después, la decodificaremos utilizando la misma codificación con la que se creó la cadena bytearray y mostraremos el resultado obtenido con codificación Unicode.

```
>>> vehiculo[4] = 243
>>> print(vehiculo)
bytearray(b'cami\xf3n')
>>> vehiculo = vehiculo.decode("cp1252")
>>> print(vehiculo)
camión
>>> type(vehiculo)
<class 'str'>
```

Si intentamos lo mismo con una cadena Unicode o Byte obtendríamos un error porque estos tipos son inmutables:

```
>>> vehiculo = "camion"
>>> vehiculo[4] = 243
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment

>>> vehiculo = bytes("camion", "cp1252")
>>> vehiculo[4] = 243
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'bytes' object does not support item assignment
```

Codificación de archivos de texto

En Python 3.x y Python 2.x cuando se desea escribir texto en un archivo utilizando el modificador "**w**", no es posible escribir cadenas Unicode que contengan caracteres no ASCII sin ser codificados previamente (aunque se perderían en la conversión los caracteres no ASCII):

```
>>> pais = bytes(pais, "utf-8")
>>> pais = pais.decode("ascii", "ignore")
```

Para escribir cadenas Unicode (sin perder los caracteres no ASCII) utilizar en la función **open**() el modificador "**wb**". Para leer, "**rb**":

```
>>> pais = "España"
>>> with open('paises.txt', 'wb') as f:
...     f.write(pais.encode("cp1252"))
...
```

Referencias:

Inmersión en Python 3. Capítulo 4. Cadenas de Texto.
Traducción al español José Miguel González Aguilera
del libro de Mark Pilgrim sobre Python 3, *Dive Into Python 3*

Encoding and Decoding Strings (in Python 3.x)
Les De Shay – Python Central (http://www.pythoncentral.io)

Tipos de cadena en Python 3: Unicode, byte y bytearray.
Carlos Santana Roldán (http://www.codejobs.biz)

"Unicode", "ASCII" y "CP-1252" (https://es.wikipedia.org)

Relacionado:
Evaluar, ejecutar y compilar cadenas

Ir al índice del Tutorial

Publicado por Pherkad en 8:27

Etiquetas: Python3

Entrada más reciente Inicio Entrada antigua