CÓMO (HOWTO) Unicode

Versión 3.8.18

Guido van Rossum
and the Python development team

febrero 08, 2024

Python Software Foundation
Email: docs@python.org

Índice general

Lanzamiento 1.12

Este CÓMO (*HOWTO*) debate el soporte de Python para la especificación Unicode para representar datos textuales, y explica varios problemas que comúnmente encuentra la gente cuando tratan de trabajar con Unicode.

1 Introducción a Unicode

1.1 Definiciones

Los programas de hoy necesitan poder manejar una amplia variedad de caracteres. Las aplicaciones son a menudo internacionalizadas para mostrar mensajes y resultados en una variedad de idiomas seleccionables por el usuario; Es posible que el mismo programa necesite generar un mensaje de error en inglés, francés, japonés, hebreo o ruso. El contenido web se puede escribir en cualquiera de estos idiomas y también puede incluir una variedad de símbolos *emoji*. El tipo cadena de Python utiliza el estándar Unicode para representar caracteres, lo que permite a los programas de Python trabajar con todos estos caracteres posibles diferentes.

Unicode (https://www.unicode.org/) es una especificación que apunta a listar cada carácter usado por lenguajes humanos y darle a cada carácter su propio código único. La especificación Unicode es continuamente revisada y actualizada para añadir nuevos lenguajes y símbolos.

Un **carácter** es el componente mas pequeño posible de un texto. "A", "B", "C", etc., son todos diferentes caracteres. También lo son *"È"* e *"Í"*. Los caracteres varían dependiendo del lenguaje o del contexto en el que estás hablando. Por ejemplo, Existe un carácter para el «Número Uno Romano», "I", que es distinto de la letra "I" mayúscula. Estos usualmente lucen igual, pero son dos caracteres diferentes que tienen distintos significados.

The Unicode standard describes how characters are represented by **code points**. A code point value is an integer in the range 0 to 0x10FFFF (about 1.1 million values, with some 110 thousand assigned so far). In the standard and in this document, a code point is written using the notation `U+265E` to mean the character with value `0x265e` (9,822 in decimal).

The Unicode standard contains a lot of tables listing characters and their corresponding code points:

```
0061    'a'; LATIN SMALL LETTER A
0062    'b'; LATIN SMALL LETTER B
0063    'c'; LATIN SMALL LETTER C
...
007B    '{'; LEFT CURLY BRACKET
...
2167    'Ⅷ'; ROMAN NUMERAL EIGHT
2168    'Ⅸ'; ROMAN NUMERAL NINE
...
265E    '♞'; BLACK CHESS KNIGHT
265F    '♟'; BLACK CHESS PAWN
...
1F600   '😀'; GRINNING FACE
1F609   '😉'; WINKING FACE
...
```

Strictly, these definitions imply that it's meaningless to say "this is character `U+265E`". `U+265E` is a code point, which represents some particular character; in this case, it represents the character "BLACK CHESS KNIGHT", "♞". In informal contexts, this distinction between code points and characters will sometimes be forgotten.

Un carácter es representado en una pantalla o en papel por un conjunto de elementos gráficos llamado **glifo**. El glifo para una A mayúscula, por ejemplo, es dos trazos diagonales y uno horizontal, aunque los detalles exactos van a depender de la fuente utilizada. La mayoría del código de Python no necesita preocuparse por los glifos*; averiguar el glifo correcto para mostrar es generalmente el trabajo de un kit de herramientas GUI o el renderizador de fuentes de una terminal.

1.2 Encodings

Para resumir la sección anterior: Una cadena Unicode es una secuencia de código de posiciones que son números desde 0 hasta `0x10FFFF` (1114111 decimal). Esta secuencia de código de posiciones necesita ser representada en memoria como un conjunto de **unidades de código**, y las **unidades de código** son mapeadas a bytes de 8 bits. Las reglas para traducir una cadena Unicode a una secuencia de bytes son llamadas **Codificación de carácter**, o sólo una **codificación**.

La primera codificación en que podrías pensar es usar enteros de 32 bits como unidad de código, y luego usar la representación de la CPU de enteros de 32 bits. En esta representación, la cadena «Python» podría verse así:

```
   P           y           t           h           o           n
0x50 00 00 00 79 00 00 00 74 00 00 00 68 00 00 00 6f 00 00 00 6e 00 00 00
   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
```

Esta representación es sencilla pero utilizarla presenta una serie de problemas.

1. No es portable; diferentes procesadores ordenan los bytes de manera diferente.
2. Es un desperdicio de espacio. En la mayoría de los textos, la mayoría de los códigos de posición son menos de 127, o menos de 255, por lo que una gran cantidad de espacio está ocupado por bytes `0x00`. La cadena anterior toma 24 bytes en comparación con los 6 bytes necesarios para una representación ASCII. El aumento en el uso de RAM no importa demasiado (las computadoras de escritorio tienen *gigabytes* de RAM, y las cadenas no suelen ser tan grandes), pero expandir nuestro uso del disco y el ancho de banda de la red en un factor de 4 es intolerable.
3. No es compatible con funciones existentes en C como `strlen()`, para eso se necesitaría una nueva familia de funciones de cadenas.

Por lo tanto esta codificación no es muy utilizada, y la gente prefiere elegir codificaciones que son mas eficientes y convenientes, como UTF-8.

UTF-8 es una de las codificaciones mas utilizadas, y Python generalmente la usa de forma predeterminada. UTF significa *«Unicode Transformation Format»*, y el «8» significa que se utilizan valores de 8 bits en la codificación. (También hay codificaciones UTF-16 y UTF-32, pero son menos frecuentes que UTF-8.) UTF-8 usa las siguientes reglas:

1. Si el código de posición is < 128, es representado por el valor de byte correspondiente.
2. Si el código de posición es >= 128, se transforma en una secuencia de dos, tres, o cuatro bytes, donde cada byte de la secuencia está entre 128 y 255.

UTF-8 tiene varias propiedades convenientes:

1. It can handle any Unicode code point.
2. A Unicode string is turned into a sequence of bytes that contains embedded zero bytes only where they represent the null character (U+0000). This means that UTF-8 strings can be processed by C functions such as `strcpy()` and sent through protocols that can't handle zero bytes for anything other than end-of-string markers.
3. Una cadena de texto ASCII es también texto UTF-8.
4. UTF-8 es bastante compacto; La mayoría de los caracteres comúnmente usados pueden ser representados con uno o dos bytes.
5. Si los bytes están corruptos o perdidos, es posible determinar el comienzo del próximo código de posición y resincronizar. También es poco probable que datos aleatorios de 8 bit se vean como UTF-8 válido.
6. UTF-8 es una codificación orientada a bytes. La codificación especifica que cada carácter está representado por una secuencia específica de uno o más bytes. Esto evita los problemas de ordenamiento de bytes que pueden ocurrir con codificaciones orientadas a números enteros y palabras, como UTF-16 y UTF-32, donde la secuencia de bytes varía según el hardware en el que se codificó la cadena.

1.3 Referencias

El Unicode Consortium site tiene mapas de caracteres, un glosario, y versiones PDF de la especificación Unicode. Esté preparado para alguna dificultad en la lectura. Una cronología del origen y desarrollo de Unicode se encuentra disponible en el sitio.

En el canal de *Youtube Computerphile*, *Tom Scott discute brevemente la historia de Unicode y UTF-8 <https://www.youtube.com/watch?v=MijmeoH9LT4>* (9 minutos 36 segundos).

Para ayudar a entender el estándar, *Jukka Korpela* escribió una guía introductoria para leer tablas de caracteres Unicode.

Otro buen articulo introductorio fue escrito por *Joel Spolsky*. Si esta introducción no aclara las cosas para usted, debería tratar leyendo este articulo alternativo antes de continuar.

Artículos de *Wikipedia* son a menudo útiles. Mire los artículos para «codificación de caracteres» y UTF-8, por ejemplo.

2 Soporte Unicode de Python

Ahora que ya ha aprendido los rudimentos de Unicode, podemos mirar las características de Unicode de Python.

2.1 El tipo cadena

Since Python 3.0, the language's `str` type contains Unicode characters, meaning any string created using `"unicode rocks!"`, `'unicode rocks!'`, or the triple-quoted string syntax is stored as Unicode.

The default encoding for Python source code is UTF-8, so you can simply include a Unicode character in a string literal:

```
try:
    with open('/tmp/input.txt', 'r') as f:
        ...
except OSError:
    # 'File not found' error message.
    print("Fichier non trouvé")
```

Nota al margen: Python 3 también soporta el uso de caracteres Unicode en identificadores:

```
répertoire = "/tmp/records.log"
with open(répertoire, "w") as f:
    f.write("test\n")
```

If you can't enter a particular character in your editor or want to keep the source code ASCII-only for some reason, you can also use escape sequences in string literals. (Depending on your system, you may see the actual capital-delta glyph instead of a u escape.)

```
>>> "\N{GREEK CAPITAL LETTER DELTA}"  # Using the character name
'\u0394'
>>> "\u0394"                          # Using a 16-bit hex value
'\u0394'
>>> "\U00000394"                      # Using a 32-bit hex value
'\u0394'
```

Además, uno puede crear una cadena usando el método `decode()` de la clase `bytes`. Este método recibe una *codificación* como argumento, como `UTF-8`, y opcionalmente un argumento *errores*.

El argumento *errores* especifica la respuesta cuando la cadena ingresada no puede ser convertida de acuerdo a las reglas de codificación. Los posibles valores para este argumento son `'strict'` (levanta una excepción `UnicodeDecodeError`), `'replace'` (use `U+FFFD'', ``CARACTER DE REEMPLAZO} ``), ``'ignore'` (solo deje el carácter fuera del resultado Unicode), o `'backslahsreplace'` (inserta una secuencia de escape `\xNN`). Los siguientes ejemplos muestran las diferencias

```
>>> b'\x80abc'.decode("utf-8", "strict")
Traceback (most recent call last):
    ...
UnicodeDecodeError: 'utf-8' codec can't decode byte 0x80 in position 0:
  invalid start byte
>>> b'\x80abc'.decode("utf-8", "replace")
'\ufffdabc'
>>> b'\x80abc'.decode("utf-8", "backslashreplace")
'\\x80abc'
>>> b'\x80abc'.decode("utf-8", "ignore")
'abc'
```

Las codificaciones son especificadas como cadenas que contienen el nombre de la codificación. Python viene con cerca de 100 codificaciones diferentes; consulta la referencia de la biblioteca de Python en standard-encodings para una lista. Algunas codificaciones tienen múltiples nombres; por ejemplo, `'latin-1'`, `'iso_8859_1'` y `'8859`" son sinónimos para la misma codificación.

Las cadenas de un solo carácter pueden ser creadas también con la función incorporada `chr()`, que toma un entero y retorna una cadena Unicode de longitud 1 que contiene el correspondiente código de posición. La operación inversa es la función incorporada `ord()` que toma una cadena Unicode de un carácter y retorna el código de posición:

```
>>> chr(57344)
'\ue000'
>>> ord('\ue000')
57344
```

2.2 Convirtiendo a Bytes

El método opuesto a `bytes.decode()` es `str.encode()`, que retorna una representación de `bytes` de la cadena Unicode, codificada en la codificación solicitada.

El parámetro *errores* es el mismo que el parámetro del método `decode()` pero soporta algunos manejadores mas.

El siguiente ejemplo muestra los diferentes resultados:

```
>>> u = chr(40960) + 'abcd' + chr(1972)
>>> u.encode('utf-8')
b'\xea\x80\x80abcd\xde\xb4'
>>> u.encode('ascii')
Traceback (most recent call last):
    ...
UnicodeEncodeError: 'ascii' codec can't encode character '\ua000' in
  position 0: ordinal not in range(128)
>>> u.encode('ascii', 'ignore')
b'abcd'
>>> u.encode('ascii', 'replace')
b'?abcd?'
>>> u.encode('ascii', 'xmlcharrefreplace')
b'&#40960;abcd&#1972;'
>>> u.encode('ascii', 'backslashreplace')
```

(continué en la próxima página)

(proviene de la página anterior)

```
b'\\ua000abcd\\u07b4'
>>> u.encode('ascii', 'namereplace')
b'\\N{YI SYLLABLE IT}abcd\\u07b4'
```

Las rutinas de bajo nivel para registrar y acceder a las codificaciones disponibles se encuentran en el módulo `codecs`. La implementación de nuevas codificaciones también requiere comprender el módulo `codecs`. Sin embargo, las funciones de codificación y decodificación retornadas por este módulo generalmente son de nivel más bajo de lo que es cómodo, y escribir nuevas codificaciones es una tarea especializada, por lo que el módulo no se cubrirá en este CÓMO.

2.3 Literales Unicode en código fuente Python

En el código fuente de Python, se pueden escribir puntos de código Unicode específicos utilizando la secuencia de escape `\u`, que es seguida por cuatro dígitos hexadecimales que dan el punto de código. La secuencia de escape `\U` es similar, pero espera ocho dígitos hexadecimales, no cuatro:

```
>>> s = "a\xac\u1234\u20ac\U00008000"
... #     ^^^^ two-digit hex escape
... #         ^^^^^^ four-digit Unicode escape
... #                ^^^^^^^^^^ eight-digit Unicode escape
>>> [ord(c) for c in s]
[97, 172, 4660, 8364, 32768]
```

El uso de secuencias de escape para puntos de código superiores a 127 está bien en pequeñas dosis, pero se convierte en una molestia si está utilizando muchos caracteres acentuados, como lo haría en un programa con mensajes en francés o algún otro lenguaje que utilice acento. También puede ensamblar cadenas usando la función incorporada `chr()`, pero esto es aún más tedioso.

Idealmente, desearía poder escribir literales en la codificación natural de su idioma. Luego, puede editar el código fuente de Python con su editor favorito, que mostrará los caracteres acentuados de forma natural y tendrá los caracteres correctos utilizados en tiempo de ejecución.

Python soporta la escritura de código fuente en UTF-8 de forma predeterminada, pero puede usar casi cualquier codificación si declara la codificación que está utilizando. Esto se hace mediante la inclusión de un comentario especial en la primera o segunda línea del archivo fuente:

```
#!/usr/bin/env python
# -*- coding: latin-1 -*-

u = 'abcdé'
print(ord(u[-1]))
```

La sintaxis está inspirada en la notación de Emacs para especificar variables locales a un archivo. Emacs admite muchas variables diferentes, pero Python solo admite *"coding"*. Los símbolos `-*-` indican a Emacs que el comentario es especial; no tienen importancia para Python pero son una convención. Python busca `coding: name` o `coding=name` en el comentario.

Si no incluye dicho comentario, la codificación predeterminada utilizada será UTF-8 como ya se mencionó. Ver también **PEP 263** para más información.

2.4 Propiedades Unicode

La especificación Unicode incluye una base de datos de información sobre puntos de código. Para cada punto de código definido, la información incluye el nombre del carácter, su categoría, el valor numérico si corresponde (para caracteres que representan conceptos numéricos como los números romanos, fracciones como un tercio y cuatro quintos, etc.). También hay propiedades relacionadas con la visualización, como cómo usar el punto de código en texto bidireccional.

El siguiente programa muestra información sobre varios caracteres e imprime el valor numérico de un carácter en particular:

```
import unicodedata

u = chr(233) + chr(0x0bf2) + chr(3972) + chr(6000) + chr(13231)

for i, c in enumerate(u):
    print(i, '%04x' % ord(c), unicodedata.category(c), end=" ")
    print(unicodedata.name(c))

# Get numeric value of second character
print(unicodedata.numeric(u[1]))
```

Cuando se ejecuta, este imprime:

```
0 00e9 Ll LATIN SMALL LETTER E WITH ACUTE
1 0bf2 No TAMIL NUMBER ONE THOUSAND
2 0f84 Mn TIBETAN MARK HALANTA
3 1770 Lo TAGBANWA LETTER SA
4 33af So SQUARE RAD OVER S SQUARED
1000.0
```

Los códigos de categoría son abreviaturas que describen la naturaleza del carácter. Estos se agrupan en categorías como «Letra», «Número», «Puntuación» o «Símbolo», que a su vez se dividen en subcategorías. Para tomar los códigos de la salida anterior, `'Ll'` significa “Letra, minúscula”, `'No'` significa «Número, otro», `'Mn'` es «Marca, sin espaciado» , y `'So'` es «Símbolo, otro». Consulte la sección the General Category Values section of the Unicode Character Database documentation para obtener una lista de códigos de categoría.

2.5 Comparando cadenas

Unicode agrega algunas complicaciones a la comparación de cadenas, porque el mismo conjunto de caracteres puede representarse mediante diferentes secuencias de puntos de código. Por ejemplo, una letra como “ê” puede representarse como un único punto de código U+00EA, o como U+0065 U+0302, que es el punto de código para “e” seguido de un punto de código para “COMBINING CIRCUMFLEX ACCENT” . Estos producirán la misma salida cuando se impriman, pero uno es una cadena de longitud 1 y el otro es de longitud 2.

Una herramienta para una comparación que no distingue entre mayúsculas y minúsculas es el método `casefold()` que convierte una cadena en una forma que no distingue entre mayúsculas y minúsculas siguiendo un algoritmo descrito por el estándar Unicode. Este algoritmo tiene un manejo especial para caracteres como la letra Alemana “ß” (punto de código U+00DF), que se convierte en el par de letras minúsculas “ss”.

```
>>> street = 'Gürzenichstraße'
>>> street.casefold()
'gürzenichstrasse'
```

Una segunda herramienta es la función `normalize()` del módulo `unicodedata` que convierte las cadenas en una de varias formas normales, donde las letras seguidas de un carácter de combinación se reemplazan con caracteres individuales.

`normalize()` puede usarse para realizar comparaciones de cadenas que no informarán falsamente la desigualdad si dos cadenas usan caracteres combinados de manera diferente:

```
import unicodedata

def compare_strs(s1, s2):
    def NFD(s):
        return unicodedata.normalize('NFD', s)

    return NFD(s1) == NFD(s2)

single_char = 'ê'
multiple_chars = '\N{LATIN SMALL LETTER E}\N{COMBINING CIRCUMFLEX ACCENT}'
print('length of first string=', len(single_char))
print('length of second string=', len(multiple_chars))
print(compare_strs(single_char, multiple_chars))
```

When run, this outputs:

```
$ python3 compare-strs.py
length of first string= 1
length of second string= 2
True
```

El primer argumento para la función `normalize()` es una cadena que proporciona la forma de normalización deseada, que puede ser una de "NFC", "NFKC", "NFD" y "NFKD".

El estándar Unicode también especifica cómo hacer comparaciones sin mayúsculas y minúsculas:

```
import unicodedata

def compare_caseless(s1, s2):
    def NFD(s):
        return unicodedata.normalize('NFD', s)

    return NFD(NFD(s1).casefold()) == NFD(NFD(s2).casefold())

# Example usage
single_char = 'ê'
multiple_chars = '\N{LATIN CAPITAL LETTER E}\N{COMBINING CIRCUMFLEX ACCENT}'

print(compare_caseless(single_char, multiple_chars))
```

Esto imprimirá `Verdadero`. (¿Por qué se invoca dos veces `NFD()`? Debido a que hay algunos caracteres que hacen que `casefold()` devuelva una cadena no normalizada, por lo que el resultado debe normalizarse nuevamente. Consulte la sección 3.13 del Estándar Unicode para una discusión y un ejemplo.)

2.6 Expresiones Regulares Unicode

Las expresiones regulares soportadas por el módulo `re` se pueden proporcionar como bytes o cadenas. Algunas de las secuencias de caracteres especiales como `\d` y `\w` tienen diferentes significados dependiendo de si el patrón se suministra como bytes o una cadena. Por ejemplo, `\d` coincidirá con los caracteres `[0-9]` en bytes, pero en las cadenas coincidirá con cualquier carácter que esté en la categoría `'Nd'`.

La cadena en este ejemplo tiene el número 57 escrito en números tailandeses y árabes:

```
import re
p = re.compile(r'\d+')

s = "Over \u0e55\u0e57 57 flavours"
m = p.search(s)
print(repr(m.group()))
```

Cuando se ejecuta, `\d+` coincidirá con los números tailandeses y los imprimirá. Si proporciona el indicador `re.ASCII` a `compile()`, `\d+` coincidirá con la subcadena «57» en su lugar.

Del mismo modo, `\w` coincide con una amplia variedad de caracteres Unicode pero solo `[a-zA-Z0-9_]` en bytes o si `re.ASCII` se suministra, y `\s` coincidirá con los caracteres de espacio en blanco Unicode o `[ \t\n\r\f\v]`.

2.7 Referencias

Algunas buenas discusiones alternativas sobre el soporte Unicode de Python son:

- Processing Text Files in Python 3, por *Nick Coghlan*.
- Pragmatic Unicode, una presentación de *Ned Batchelder* en PyCon 2012.

El tipo `str` se describe en la referencia de la biblioteca de Python en textseq.

La documentación para el módulo `unicodedata`.

La documentación para el módulo `codecs`.

Marc-André Lemburg hizo una presentación titulada «Python and Unicode» (diapositivas en PDF) <https://downloads.egenix.com/python/Unicode-EPC2002-Talk.pdf>‘_ en EuroPython 2002. Las diapositivas son una excelente descripción general del diseño de las características Unicode de Python 2 (donde el tipo de cadena Unicode se llama `unicode` y los literales comienzan con `u`).

3 Leyendo y escribiendo datos Unicode

Una vez que haya escrito un código que funcione con datos Unicode, el siguiente problema es la entrada/salida. ¿Cómo obtiene cadenas Unicode en su programa y cómo convierte Unicode en una forma adecuada para almacenamiento o transmisión?

Es posible que no necesite hacer nada dependiendo de sus fuentes de entrada y destinos de salida; debe verificar si las bibliotecas utilizadas en su aplicación son compatibles con Unicode de forma nativa. Los analizadores XML a menudo retornan datos Unicode, por ejemplo. Muchas bases de datos relacionales también admiten columnas con valores Unicode y pueden retornar valores Unicode de una consulta SQL.

Los datos Unicode generalmente se convierten a una codificación particular antes de escribirse en el disco o enviarse a través de un socket. Es posible hacer todo el trabajo usted mismo: abra un archivo, lea un objeto de bytes de 8 bits y convierta los bytes con `bytes.decode(codificación)`. Sin embargo, no se recomienda el enfoque manual.

Un problema es la naturaleza de múltiples bytes de las codificaciones; Un carácter Unicode puede ser representado por varios bytes. Si desea leer el archivo en fragmentos de tamaño arbitrario (por ejemplo, 1024 o 4096 bytes), debe escribir un código de manejo de errores para detectar el caso en el que solo una parte de los bytes que codifican un solo carácter Unicode se leen al final de Un trozo. Una solución sería leer todo el archivo en la memoria y luego realizar la decodificación, pero eso le impide trabajar con archivos que son extremadamente grandes; si necesita leer un archivo de 2 GB, necesita 2 GB de RAM. (Más, realmente, ya que por al menos un momento necesitarías tener tanto la cadena codificada como su versión Unicode en la memoria).

La solución sería utilizar la interfaz de decodificación de bajo nivel para detectar el caso de secuencias de codificación parcial. El trabajo de implementar esto ya se ha realizado para usted: la función incorporada `open()` puede retornar

un objeto similar a un archivo que asume que el contenido del archivo está en una codificación especificada y acepta parámetros Unicode para métodos como `read()` y `write()`. Esto funciona a través de los parámetros *enconding* y *errors* de `open()` que se interpretan como los de `str.encode()` y `bytes.decode()`.

Por lo tanto, leer Unicode de un archivo es simple:

```
with open('unicode.txt', encoding='utf-8') as f:
    for line in f:
        print(repr(line))
```

También es posible abrir archivos en modo de actualización, lo que permite leer y escribir:

```
with open('test', encoding='utf-8', mode='w+') as f:
    f.write('\u4500 blah blah blah\n')
    f.seek(0)
    print(repr(f.readline()[:1]))
```

El carácter Unicode `U+FEFF` se usa como marca de orden de bytes (BOM), y a menudo se escribe como el primer carácter de un archivo para ayudar a la auto detección del orden de bytes del archivo. Algunas codificaciones, como UTF-16, esperan que haya una BOM al comienzo de un archivo; cuando se utiliza dicha codificación, la BOM se escribirá automáticamente como el primer carácter y se descartará en silencio cuando se lea el archivo. Existen variantes de estas codificaciones, como "utf-16-le" y "utf-16-be" para codificaciones "little-endian" y "big-endian", que especifican un orden de bytes particular y no omiten la BOM.

En algunas áreas, también es convencional usar una «BOM» al comienzo de los archivos codificados UTF-8; el nombre es engañoso ya que UTF-8 no depende del orden de bytes. La marca simplemente anuncia que el archivo está codificado en UTF-8. Para leer dichos archivos, use el códec "utf-8-sig" para omitir automáticamente la marca si está presente.

3.1 Nombres de archivos Unicode

La mayoría de los sistemas operativos de uso común hoy en día admiten nombres de archivo que contienen caracteres Unicode arbitrarios. Por lo general, esto se implementa convirtiendo la cadena Unicode en alguna codificación que varía según el sistema. Hoy Python está convergiendo sobre el uso de UTF-8: Python en MacOS ha usado UTF-8 para varias versiones, y Python 3.6 también cambió a usar UTF-8 en Windows. En los sistemas Unix, solo habrá una codificación del sistema de archivos si ha configurado las variables de entorno `LANG` o `LC_CTYPE`; si no lo ha hecho, la codificación predeterminada es nuevamente UTF-8.

La función `sys.getfilesystemencoding()` retorna la codificación para usar en su sistema actual, en caso de que desee realizar la codificación manualmente, pero no hay muchas razones para molestarse. Al abrir un archivo para leer o escribir, generalmente puede proporcionar la cadena Unicode como nombre de archivo, y se convertirá automáticamente a la codificación correcta para usted:

```
filename = 'filename\u4500abc'
with open(filename, 'w') as f:
    f.write('blah\n')
```

Las funciones en el módulo `os` como `os.stat()` también aceptarán nombres de archivo Unicode.

La función `os.listdir()` retorna nombres de archivo, lo que plantea un problema: ¿debería retornar la versión Unicode de los nombres de archivo, o debería retornar bytes que contienen las versiones codificadas? `os.listdir()` puede hacer ambas cosas, dependiendo de si proporcionó la ruta del directorio como bytes o una cadena Unicode. Si pasa una cadena Unicode como ruta, los nombres de archivo se decodificarán utilizando la codificación del sistema de archivos y se retornará una lista de cadenas Unicode, mientras que al pasar una ruta de bytes se retornarán los nombres de archivo como bytes. Por ejemplo, suponiendo que la codificación predeterminada del sistema de archivos es UTF-8, ejecutando el siguiente programa:

```
fn = 'filename\u4500abc'
f = open(fn, 'w')
f.close()

import os
print(os.listdir(b'.'))
print(os.listdir('.'))
```

producirá la siguiente salida:

```
$ python listdir-test.py
[b'filename\xe4\x94\x80abc', ...]
['filename\u4500abc', ...]
```

La primera lista contiene nombres de archivos codificados con UTF-8, y la segunda lista contiene las versiones Unicode.

Tenga en cuenta que en la mayoría de las ocasiones, debe seguir usando Unicode con estas API. Las API de bytes solo deben usarse en sistemas donde pueden estar presentes nombres de archivo no codificables; eso es prácticamente solo sistemas Unix ahora.

3.2 Consejos para escribir programas compatibles con Unicode

Esta sección proporciona algunas sugerencias sobre cómo escribir software que maneje Unicode.

El consejo más importante es:

> El software solo debería funcionar con cadenas Unicode internamente, decodificando los datos de entrada lo antes posible y codificando la salida solo al final.

Si intenta escribir funciones de procesamiento que acepten cadenas Unicode y de bytes, encontrará que su programa es vulnerable a errores dondequiera que combine los dos tipos diferentes de cadenas. No hay codificación o decodificación automática: si hace, por ejemplo: `str+bytes`, un `TypeError` se generará.

Cuando se usan datos que provienen de un navegador web u otra fuente no confiable, una técnica común es verificar si hay caracteres ilegales en una cadena antes de usar la cadena en una línea de comando generada o almacenarla en una base de datos. Si está haciendo esto, tenga cuidado de verificar la cadena decodificada, no los datos de bytes codificados; Algunas codificaciones pueden tener propiedades interesantes, como no ser biyectivo o no ser totalmente compatible con ASCII. Esto es especialmente cierto si los datos de entrada también especifican la codificación, ya que el atacante puede elegir una forma inteligente de ocultar el texto malicioso en el flujo de bytes codificado.

Conversión entre codificaciones de archivo

La clase `StreamRecoder` puede convertir de forma transparente entre codificaciones, tomar una secuencia que retorna datos en la codificación 1 y comportarse como una secuencia que retorna datos en la codificación 2.

Por ejemplo, si tiene un archivo de entrada *f* que está en Latin-1, puede envolverlo con `StreamRecoder` para retornar bytes codificados en UTF-8:

```
new_f = codecs.StreamRecoder(f,
    # en/decoder: used by read() to encode its results and
    # by write() to decode its input.
    codecs.getencoder('utf-8'), codecs.getdecoder('utf-8'),

    # reader/writer: used to read and write to the stream.
    codecs.getreader('latin-1'), codecs.getwriter('latin-1') )
```

Archivos en una codificación desconocida

¿Qué puede hacer si necesita hacer un cambio en un archivo, pero no conoce la codificación del archivo? Si sabe que la codificación es compatible con ASCII y solo desea examinar o modificar las partes ASCII, puede abrir el archivo con el manejador de errores `surrogateescape`:

```
with open(fname, 'r', encoding="ascii", errors="surrogateescape") as f:
    data = f.read()

# make changes to the string 'data'

with open(fname + '.new', 'w',
          encoding="ascii", errors="surrogateescape") as f:
    f.write(data)
```

El manejador de errores `surrogateescape` decodificará los bytes que no sean ASCII como puntos de código en un rango especial que va desde U+DC80 a U+DCFF. Estos puntos de código volverán a convertirse en los mismos bytes cuando se use el controlador de error `subrogateescape` para codificar los datos y volver a escribirlos.

3.3 Referencias

Una sección de Mastering Python 3 Input/Output, una charla de *David Beazley* en la PyCon 2010, analiza el procesamiento de texto y el manejo de datos binarios.

El PDF slides for Marc-André Lemburg's presentation «Writing Unicode-aware Applications in Python» discute cuestiones de codificaciones de caracteres, así como también cómo internacionalizar y localizar una aplicación. Estas diapositivas cubren solo Python 2.x.

The Guts of Unicode in Python es una charla de *Benjamin Peterson* en PyCon 2013 que analiza la representación interna de Unicode en Python 3.3.

4 Agradecimientos

El borrador inicial de este documento fue escrito por *Andrew Kuchling*. Desde entonces ha sido revisado por *Alexander Belopolsky, Georg Brandl, Andrew Kuchling y Ezio Melotti*.

293/5000 Gracias a las siguientes personas que notaron errores u ofrecieron sugerencias sobre este artículo: *Éric Araujo, Nicholas Bastin, Nick Coghlan, Marius Gedminas, Kent Johnson, Ken Krugler, Marc-André Lemburg, Martin von Löwis, Terry J. Reedy, Serhiy Storchaka , Eryk Sun, Chad Whitacre, Graham Wideman*.

Índice

P