Unicode 指南

发布版本: 1.12

本文介绍了 Python 对表示文本数据的 Unicode 规范的支持，并对各种 Unicode 常见使用问题做了解释。

Unicode 概述

定义

如今的程序需要能够处理各种各样的字符。应用程序通常做了国际化处理，用户可以选择不同的语言显示信息和输出数据。同一个程序可能需要以英语、法语、日语、希伯来语或俄语输出错误信息。网页内容可能由这些语言书写，并且可能包含不同的表情符号。Python 的字符串类型采用 Unicode 标准来表示字符，使得 Python 程序能够正常处理所有这些不同的字符。

Unicode 规范（https://www.unicode.org/）旨在罗列人类语言所用到的所有字符，并赋予每个字符唯一的编码。该规范一直在进行修订和更新，不断加入新的语种和符号。

一个 **字符** 是文本的最小组件。'A'、'B'、'C' 等都是不同的字符。'È' 和 'Í' 也一样。字符会随着语言或者上下文的变化而变化。比如，'Ⅰ' 是一个表示 "罗马数字 1" 的字符，它与大写字母 'I' 不同。他们往往看起来相同，但这是两个有着不同含义的字符。

Unicode 标准描述了字符是如何用 **码位（code point）** 表示的。码位的取值范围是 0 到 0x10FFFF 的整数（大约 110 万个值，实际分配的数字 没有那么多）。在 Unicode 标准和本文中，码位采用 `U+265E` 的形式，表示值为 `0x265e` 的字符（十进制为 9822）。

Unicode 标准中包含了许多表格，列出了很多字符及其对应的码位。

```
0061    'a'; 拉丁字母 A 小写
0062    'b'; 拉丁字母 B 小写
0063    'c'; 拉丁字母 C 小写
...
007B    '{'; 左花括号
...
2167    'Ⅷ'; 罗马数字八
2168    'Ⅸ'; 罗马数字九
...
265E    '♞'; 国际象棋黑马
265F    '♟'; 国际象棋黑兵
...
1F600   '😀'; 微笑脸
1F609   '😉'; 眨眼脸
...
```

严格地说，上述定义暗示了以下说法是没有意义的："这是字符 `U+265E`"。`U+265E` 只是一个码位，代表某个特定的字符；这里它代表了字符 "国际象棋黑骑士" '♞'。在非正式的上下文中，有时会忽略

码位和字符的区别。

一个字符在屏幕或纸上被表示为一组图形元素，被称为 **字形（glyph）** 。比如，大写字母 A 的字形，是两笔斜线和一笔横线，而具体的细节取决于所使用的字体。大部分 Python 代码不必担心字形，找到正确的显示字形通常是交给 GUI 工具包或终端的字体渲染程序来完成。

编码

上一段可以归结为：一个 Unicode 字符串是一系列码位（从 0 到 `0x10FFFF` 或者说十进制的 1,114,111 的数字）组成的序列。这一序列在内存中需被表示为一组 **码元（code unit）** ， **码元** 会映射成包含八个二进制位的字节。将 Unicode 字符串翻译成字节序列的规则称为 **字符编码** ，或者 **编码** 。

大家首先会想到的编码可能是用 32 位的整数作为代码位，然后采用 CPU 对 32 位整数的表示法。字符串 “Python” 用这种表示法可能会如下所示：

```
   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
```

这种表示法非常直白，但也存在 一些问题。

1. 不具可移植性；不同的处理器的字节序不同。
2. 非常浪费空间。 在大多数文本中，大部分码位都小于 127 或 255，因此字节 `0x00` 占用了大量空间。相较于 ASCII 表示法所需的 6 个字节，以上字符串需要占用 24 个字节。RAM 用量的增加没那么要紧（台式计算机有成 GB 的 RAM，而字符串通常不会有那么大），但要把磁盘和网络带宽的用量增加 4 倍是无法忍受的。
3. 与现有的 C 函数（如 `strlen()` ）不兼容，因此需要采用一套新的宽字符串函数。

因此这种编码用得不多，人们转而选择其他更高效、更方便的编码，比如 UTF-8。

UTF-8 是最常用的编码之一，Python 往往默认会采用它。UTF 代表“Unicode Transformation Format”，'8' 表示编码采用 8 位数。（UTF-16 和 UTF-32 编码也是存在的，但其使用频率不如 UTF-8。）UTF-8 的规则如下：

1. 如果码位 < 128，则直接用对应的字节值表示。
2. 如果码位 >= 128，则转换为 2、3、4 个字节的序列，每个字节值都位于 128 和 255 之间。

UTF-8 有几个很方便的特性：

1. 可以处理任何 Unicode 码位。
2. Unicode 字符串被转换为一个字节序列，仅在表示空（null ）字符（U+0000）时才会包含零值的字节。这意味着 `strcpy()` 之类的C 函数可以处理 UTF-8 字符串，而且用那些不能处理字符串结束符之外的零值字节的协议也能发送。
3. ASCII 字符串也是也是也是合法的 UTF-8 文本。
4. UTF-8 相当紧凑；大多数常用字符均可用一两个字节表示。
5. 如果字节数据被损坏或丢失，则可以找出下一个 UTF-8 码点的开始位置并重新开始同步。随机的 8 位数据也不太可能像是有效的 UTF-8 编码。

6. UTF-8 是一种面向字节的编码。编码规定了每个字符由一个或多个字节的序列表示。这避免了整数和双字节编码（如 UTF-16 和 UTF-32）可能出现的字节顺序问题，那时的字节序列会因执行编码的硬件而异。

参考文献

Unicode Consortium 站点 包含 Unicode 规范的字符图表、词汇表和 PDF 版本。请做好准备，有些内容读起来有点难度。该网站上还提供了 Unicode 起源和发展的 年表 。

在 Computerphile 的 Youtube 频道上，Tom Scott 简要地 讨论了 Unicode 和 UTF-8 （9 分 36 秒）的历史。

为了帮助理解该标准，Jukka Korpela 编写了阅读 Unicode 字符表的 介绍性指南 。

Joel Spolsky 撰写了另一篇不错的介绍性文章 <https://www.joelonsoftware.com/2003/10/08/the-absolute-minimum-every-software-developer-absolutely-positively-must-know-about-unicode-and-character- set-no-excuses/>`_ 。如果本文没让您弄清楚，那应在继续之前先试着读读这篇文章。

Wikipedia 条目通常也有帮助；请参阅“字符编码”和 UTF-8 的条目，例如：

Python对Unicode的支持

现在您已经了解了 Unicode 的基础知识，可以看下 Python 的 Unicode 特性。

字符串类型

从 Python 3.0 开始， `str` 类型包含了 Unicode 字符，这意味着用 `"unicode rocks!"`、`'unicode rocks!'` 或三重引号字符串语法创建的任何字符串都会存储为 Unicode。

Python 源代码的默认编码是 UTF-8，因此可以直接在字符串中包含 Unicode 字符：

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

旁注：Python 3 还支持在标识符中使用 Unicode 字符：

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

如果无法在编辑器中输入某个字符，或出于某种原因想只保留 ASCII 编码的源代码，则还可以在字符串中使用转义序列。（根据系统的不同，可能会看到真的大写 Delta 字体而不是 u 转义符。）：

```
>>> "\N{GREEK CAPITAL LETTER DELTA}"  # 使用字符名称
'\u0394'
>>> "\u0394"                          # 使用 16 比特位十六进制数值
```

```
'\u0394'
>>> "\U00000394"                    # 使用 32 比特位十六进制数值
'\u0394'
```

此外，可以用 bytes 的 decode() 方法创建一个字符串。 该方法可以接受 *encoding* 参数，比如可以为 `UTF-8` ，以及可选的 *errors* 参数。

若无法根据编码规则对输入字符串进行编码，*errors* 参数指定了响应策略。 该参数的合法值可以是 `'strict'` (触发 UnicodeDecodeError 异常)、`'replace'` (用 `U+FFFD`、`REPLACEMENT CHARACTER`)、`'ignore'` (只是将字符从 Unicode 结果中去掉)，或 `'backslashreplace'` (插入一个 `\xNN` 转义序列)。 以下示例演示了这些不同的参数:

```
>>> 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'
```

编码格式以包含编码格式名称的字符串来指明。 Python 有大约 100 种不同的编码格式；清单详见 Python 库参考文档 标准编码。 一些编码格式有多个名称，比如 `'latin-1'`、`'iso_8859_1'` 和 `'8859` 都是指同一种编码。

利用内置函数 chr() 还可以创建单字符的 Unicode 字符串，该函数可接受整数参数，并返回包含对应码位的长度为 1 的 Unicode 字符串。内置函数 ord() 是其逆操作，参数为单个字符的 Unicode 字符串，并返回码位值:

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

转换为字节

bytes.decode() 的逆方法是 str.encode() ，它会返回 Unicode 字符串的 bytes 形式，已按要求的 *encoding* 进行了编码。

参数 *errors* 的意义与 decode() 方法相同，但支持更多可能的handler。除了 `'strict'` 、`'ignore'` 和 `'replace'` （这时会插入问号替换掉无法编码的字符），还有 `'xmlcharrefreplace'` （插入一个 XML 字符引用）、 `backslashreplace` （插入一个 `\uNNNN` 转义序列）和 `namereplace` （插入一个 `\N{...}` 转义序列 ）。

以下例子演示了各种不同的结果:

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

用于注册和访问可用编码格式的底层函数，位于 `codecs` 模块中。 若要实现新的编码格式，则还需要了解 `codecs` 模块。 不过该模块返回的编码和解码函数通常更为底层一些，不大好用，编写新的编码格式是一项专业的任务，因此本文不会涉及该模块。

Python 源代码中的 Unicode 文字

在 Python 源代码中，可以用 `\u` 转义序列书写特定的 Unicode 码位，该序列后跟 4 个代表码位的十六进制数字。`\U` 转义序列用法类似，但要用8 个十六进制数字，而不是 4 个：

```
>>> s = "a\xac\u1234\u20ac\U00008000"
... #     ^^^^ 两位十六进制数转义
... #         ^^^^^^ 四位 Unicode 转义
... #                ^^^^^^^^^^ 八位 Unicode 转义
>>> [ord(c) for c in s]
[97, 172, 4660, 8364, 32768]
```

对大于 127 的码位使用转义序列，数量不多时没什么问题，但如果要用到很多重音字符，这会变得很烦人，类似于程序中的信息是用法语或其他使用重音的语言写的。也可以用内置函数 `chr()` 拼装字符串，但会更加乏味。

理想情况下，都希望能用母语的编码书写文本。还能用喜好的编辑器编辑 Python 源代码，编辑器要能自然地显示重音符，并在运行时使用正确的字符。

默认情况下，Python 支持以 UTF-8 格式编写源代码，但如果声明要用的编码，则几乎可以使用任何编码。只要在源文件的第一行或第二行包含一个特殊注释即可：

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

u = 'abcdé'
print(ord(u[-1]))
```

上述语法的灵感来自于 Emacs 用于指定文件局部变量的符号。Emacs 支持许多不同的变量，但 Python 仅支持“编码”。 `-*-` 符号向 Emacs 标明该注释是特殊的；这对 Python 没有什么意义，只是一种约定。Python 会在注释中查找 `coding: name` 或 `coding=name` 。

如果没有这种注释，则默认编码将会是前面提到的 UTF-8。更多信息请参阅 **PEP 263** 。

Unicode属性

Unicode 规范包含了一个码位信息数据库。对于定义的每一个码位，都包含了字符的名称、类别、数值（对于表示数字概念的字符，如罗马数字、分数如三分之一和五分之四等）。还有有关显示的属性，比如如何在双向文本中使用码位。

以下程序显示了几个字符的信息，并打印一个字符的数值：

```
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))

# 获取第二个字符的数值
print(unicodedata.numeric(u[1]))
```

当运行时，这将打印出：

```
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
```

类别代码是描述字符性质的一个缩写。分为“字母”、“数字”、“标点符号”或“符号”等类别，而这些类别又分为子类别。就以上输出的代码而言，`'Ll'` 表示“字母，小写”，`'No'` 表示“数字，其他”，`'Mn'` 表示“标记，非空白符”，`'So'` 是“符号，其他”。有关类别代码的清单，请参阅 Unicode 字符库文档 的“通用类别值”部分。

字符串比较

Unicode 让字符串的比较变得复杂了一些，因为同一组字符可能由不同的码位序列组成。例如，像“ê”这样的字母可以表示为单码位 U+00EA，或是 U+0065 U+0302，即“e”的码位后跟“COMBINING CIRCUMFLEX ACCENT”的码位。虽然在打印时会产生同样的输出，但一个是长度为 1 的字符串，另一个是长度为 2 的字符串。

一种不区分大小写比较的工具是字符串方法 `casefold()` ，将按照 Unicode 标准描述的算法将字符串转换为不区分大小写的形式。该算法对诸如德语字母“ß”（代码点 U+00DF）之类的字符进行了特殊处理，变为一对小写字母“ss”。

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

第二个工具是 unicodedata 模块的 normalize() 函数，该函数可将字符串转换为几种规范化形式之一，即用单字符替换后面带一个组合字符的多个字母。 normalize() 可被用于执行字符串比较，如果两个字符串使用不同的组合字符，也不会错误地报告两者不相等:

```
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))
```

当运行时，这将输出:

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

normalize() 函数的第一个参数是个字符串，给出所需的规范化形式，可以是“NFC”、“NFKC”、“NFD”和“NFKD”之一。

Unicode 标准还设定了如何进行不区分大小写的比较:

```
import unicodedata

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

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

# 使用示例
single_char = 'ê'
multiple_chars = '\N{LATIN CAPITAL LETTER E}\N{COMBINING CIRCUMFLEX ACCENT}'

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

这将打印 `True`。 （为什么 `NFD()` 会两次被唤起？ 因为有几个字符会使 casefold() 返回非规范化的字符串，所以需要再次对结果进行规范化处理。 有关讨论和示例，请参阅 Unicode 标准第 3.13 节）。

Unicode 正则表达式

re 模块支持的正则表达式可以用字节串或字符串的形式提供。有一些特殊字符序列，比如 `\d` 和 `\w` 具有不同的含义，具体取决于匹配模式是以字节串还是字符串形式提供的。例如，`\d` 将匹配字节串中的字符 `[0-9]`，但对于字符串将会匹配 `'Nd'` 类别中的任何字符。

上述示例中的字符串包含了泰语和阿拉伯数字书写的数字 57：

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

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

执行时，`\d+` 将匹配上泰语数字并打印出来。如果向 `compile()` 提供的是 `re.ASCII` 标志，`\d+` 则会匹配子串 "57"。

类似地，`\w` 将匹配多种 Unicode 字符，但对于字节串则只会匹配 `[a-zA-Z0-9_]`，如果指定 `re.ASCII`，`\s` 将匹配 Unicode 空白符或 `[ \t\n\r\f\v]`。

参考文献

关于 Python 的 Unicode 支持，其他还有一些很好的讨论：

- 用 Python 3 处理文本文件，作者 Nick Coghlan。
- 实用的 Unicode，Ned Batchelder 在 PyCon 2012 上的演示。

`str` 类型在 Python 库参考文档 文本序列类型 --- str 中有介绍。

`unicodedata` 模块的文档

`codecs` 模块的文档

Marc-André Lemburg 在 EuroPython 2002 上做了一个题为“Python 和 Unicode”（PDF 幻灯片）<https://downloads.egenix.com/python/Unicode-EPC2002-Talk.pdf>`_ 的演示文稿。该幻灯片很好地概括了 Python 2 的 Unicode 功能设计（其中 Unicode 字符串类型称为 `unicode`，文字以 `u` 开头）。

Unicode 数据的读写

既然处理 Unicode 数据的代码写好了，下一个问题就是输入/输出了。如何将 Unicode 字符串读入程序，如何将 Unicode 转换为适于存储或传输的形式呢？

根据输入源和输出目标的不同，或许什么都不用干；请检查一下应用程序用到的库是否原生支持 Unicode。例如，XML 解析器往往会返回 Unicode 数据。许多关系数据库的字段也支持 Unicode 值，并且 SQL 查询也能返回 Unicode 值。

在写入磁盘或通过套接字发送之前，Unicode 数据通常要转换为特定的编码。可以自己完成所有工作：打开一个文件，从中读取一个 8 位字节对象，然后用 `bytes.decode(encoding)` 对字节串进行转换。但是，不推荐采用这种全人工的方案。

编码的多字节特性就是一个难题； 一个 Unicode 字符可以用几个字节表示。 如果要以任意大小的块（例如 1024 或 4096 字节）读取文件，那么在块的末尾可能只读到某个 Unicode 字符的部分字节，这就需要编写错误处理代码。 有一种解决方案是将整个文件读入内存，然后进行解码，但这样就没

法处理很大的文件了；若要读取 2 GB 的文件，就需要 2 GB 的 RAM。（其实需要的内存会更多些，因为至少有一段时间需要在内存中同时存放已编码字符串及其 Unicode 版本。）

解决方案是利用底层解码接口去捕获编码序列不完整的情况。这部分代码已经是现成的：内置函数 `open()` 可以返回一个文件类的对象，该对象认为文件的内容采用指定的编码，`read()` 和 `write()` 等方法接受 Unicode 参数。只要用 `open()` 的 *encoding* 和 *errors* 参数即可，参数释义同 `str.encode()` 和 `bytes.decode()` 。

因此从文件读取 Unicode 就比较简单了：

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

也可以在更新模式下打开文件，以便同时读取和写入：

```
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]))
```

Unicode 字符 `U+FEFF` 用作字节顺序标记（BOM），通常作为文件的第一个字符写入，以帮助自动检测文件的字节顺序。某些编码（例如 UTF-16）期望在文件开头出现 BOM；当采用这种编码时，BOM 将自动作为第一个字符写入，并在读取文件时会静默删除。这些编码有多种变体，例如用于 little-endian 和 big-endian 编码的 “utf-16-le” 和 “utf-16-be”，会指定一种特定的字节顺序并且不会忽略 BOM。

在某些地区，习惯在 UTF-8 编码文件的开头用上“BOM”；此名称具有误导性，因为 UTF-8 与字节顺序无关。此标记只是声明该文件以 UTF-8 编码。要读取此类文件，请使用“utf-8-sig”编解码器自动忽略此标记。

Unicode 文件名

当今大多数操作系统都支持包含任意 Unicode 字符的文件名。 通常这是通过将 Unicode 字符串转换为某种根据具体系统而定的编码格式来实现的。 如今的 Python 倾向于使用 UTF-8：MacOS 上的 Python 已经在多个版本中使用了 UTF-8，而 Python 3.6 也已在 Windows 上改用了 UTF-8。 在 Unix 系统中，将只有一个 文件系统编码格式。 如果你已设置了 `LANG` 或 `LC_CTYPE` 环境变量的话；如果未设置，则默认编码格式还是 UTF-8。

`sys.getfilesystemencoding()` 函数将返回要在当前系统采用的编码，若想手动进行编码时即可用到，但无需多虑。在打开文件进行读写时，通常只需提供 Unicode 字符串作为文件名，会自动转换为合适的编码格式：

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

`os` 模块中的函数也能接受 Unicode 文件名，如 `os.stat()` 。

`os.listdir()` 函数返回文件名，这引发了一个问题：它应该返回文件名的 Unicode 版本，还是应该返回包含已编码版本的字节串？ 这两者 `os.listdir()` 都能做到，具体取决于你给出的目录路径是字节串还是 Unicode 字符串形式的。 如果你传入一个 Unicode 字符串作为路径，文件名将使用文件系统的编码格式进行解码并返回一个 Unicode 字符串列表，而传入一个字节串形式的路径则将返回字节串形式的文件名。 例如，假定默认 文件系统编码 为 UTF-8，运行以下程序:

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

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

将产生以下输出：

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

第一个列表包含 UTF-8 编码的文件名，第二个列表则包含 Unicode 版本的。

请注意，大多时候应该坚持用这些 API 处理 Unicode。字节串 API 应该仅用于可能存在不可解码文件名的系统；现在几乎仅剩 Unix 系统了。

识别 Unicode 的编程技巧

本节提供了一些关于编写 Unicode 处理软件的建议。

最重要的技巧如下：

> 程序应只在内部处理 Unicode 字符串，尽快对输入数据进行解码，并只在最后对输出进行编码。

如果尝试编写的处理函数对 Unicode 和字节串形式的字符串都能接受，就会发现组合使用两种不同类型的字符串时，容易产生差错。没办法做到自动编码或解码：如果执行 `str + bytes`，则会触发 `TypeError`。

当要使用的数据来自 Web 浏览器或其他不受信来源时，常用技术是在用该字符串生成命令行之前，或要存入数据库之前，先检查字符串中是否包含非法字符。请仔细检查解码后的字符串，而不是编码格式的字节串数据；有些编码可能具备一些有趣的特性，例如与 ASCII 不是一一对应或不完全兼容。如果输入数据还指定了编码格式，则尤其如此，因为攻击者可以选择一种巧妙的方式将恶意文本隐藏在经过编码的字节流中。

在文件编码格式之间进行转换

`StreamRecoder` 类可以在两种编码之间透明地进行转换，参数为编码格式为 #1 的数据流，表现行为则是编码格式为 #2 的数据流。

假设输入文件 *f* 采用 Latin-1 编码格式，即可用 `StreamRecoder` 包装后返回 UTF-8 编码的字节串：

```
new_f = codecs.StreamRecoder(f,
    # en/decoder: 被 read() 用来编码其结果
    # 并被 write() 用来解码其输入。
    codecs.getencoder('utf-8'), codecs.getdecoder('utf-8'),

    # reader/writer: 用于读取和写入流。
    codecs.getreader('latin-1'), codecs.getwriter('latin-1') )
```

编码格式未知的文件

若需对文件进行修改，但不知道文件的编码，那该怎么办呢？如果已知编码格式与 ASCII 兼容，并且只想查看或修改 ASCII 部分，则可利用 `surrogateescape` 错误处理 handler 打开文件：

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

# 对字符串“data”进行更改

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

`surrogateescape` 错误处理 handler 会把所有非 ASCII 字节解码为 U+DC80 至 U+DCFF 这一特殊范围的码位。当 `surrogateescape` 错误处理 handler用于数据编码并回写时，这些码位将转换回原样。

参考文献

David Beazley 在 PyCon 2010 上的演讲 掌握 Python 3 输入/输出 中，有一节讨论了文本和二进制数据的处理。

Marc-André Lemburg 演示的PDF 幻灯片“在 Python 中编写支持 Unicode 的应用程序”，讨论了字符编码问题以及如何国际化和本地化应用程序。这些幻灯片仅涵盖 Python 2.x。

Python Unicode 实质 是 Benjamin Peterson 在 PyCon 2013 上的演讲，讨论了 Unicode 在 Python 3.3 中的内部表示。

致谢

本文初稿由 Andrew Kuchling 撰写。此后，Alexander Belopolsky、Georg Brandl、Andrew Kuchling 和 Ezio Melotti 作了进一步修订。

感谢以下各位指出本文错误或提出建议：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。