Unicode 指南

發🄵 *3.6.15*

Guido van Rossum
and the Python development team

9 月 06, 2021

Python Software Foundation
Email: docs@python.org

Contents

发布版本 1.12

This HOWTO discusses Python support for Unicode, and explains various problems that people commonly encounter when trying to work with Unicode.

1 Unicode 概述

1.1 History of Character Codes

In 1968, the American Standard Code for Information Interchange, better known by its acronym ASCII, was standardized. ASCII defined numeric codes for various characters, with the numeric values running from 0 to 127. For example, the lowercase letter 『a』 is assigned 97 as its code value.

ASCII was an American-developed standard, so it only defined unaccented characters. There was an 『e』, but no 『é』 or 『Í』. This meant that languages which required accented characters couldn't be faithfully represented in ASCII. (Actually the missing accents matter for English, too, which contains words such as 『naïve』 and 『café』, and some publications have house styles which require spellings such as 『coöperate』.)

For a while people just wrote programs that didn't display accents. In the mid-1980s an Apple II BASIC program written by a French speaker might have lines like these:

```
PRINT "MISE A JOUR TERMINEE"
PRINT "PARAMETRES ENREGISTRES"
```

Those messages should contain accents (terminée, paramètre, enregistrés) and they just look wrong to someone who can read French.

In the 1980s, almost all personal computers were 8-bit, meaning that bytes could hold values ranging from 0 to 255. ASCII codes only went up to 127, so some machines assigned values between 128 and 255 to accented characters. Different machines had different codes, however, which led to problems exchanging files. Eventually various commonly used sets of values for the 128–255 range emerged. Some were true standards, defined by the International Organization for Standardization, and some were *de facto* conventions that were invented by one company or another and managed to catch on.

255 characters aren't very many. For example, you can't fit both the accented characters used in Western Europe and the Cyrillic alphabet used for Russian into the 128–255 range because there are more than 128 such characters.

You could write files using different codes (all your Russian files in a coding system called KOI8, all your French files in a different coding system called Latin1), but what if you wanted to write a French document that quotes some Russian text? In the 1980s people began to want to solve this problem, and the Unicode standardization effort began.

Unicode started out using 16-bit characters instead of 8-bit characters. 16 bits means you have 2^16 = 65,536 distinct values available, making it possible to represent many different characters from many different alphabets; an initial goal was to have Unicode contain the alphabets for every single human language. It turns out that even 16 bits isn't enough to meet that goal, and the modern Unicode specification uses a wider range of codes, 0 through 1,114,111 (`0x10FFFF` in base 16).

There's a related ISO standard, ISO 10646. Unicode and ISO 10646 were originally separate efforts, but the specifications were merged with the 1.1 revision of Unicode.

(This discussion of Unicode's history is highly simplified. The precise historical details aren't necessary for understanding how to use Unicode effectively, but if you're curious, consult the Unicode consortium site listed in the References or the Wikipedia entry for Unicode for more information.)

1.2 定义

A **character** is the smallest possible component of a text. 『A』, 『B』, 『C』, etc., are all different characters. So are 『È』and 『Í』. Characters are abstractions, and vary depending on the language or context you're talking about. For example, the symbol for ohms (Ω) is usually drawn much like the capital letter omega (Ω) in the Greek alphabet (they may even be the same in some fonts), but these are two different characters that have different meanings.

The Unicode standard describes how characters are represented by **code points**. A code point is an integer value, usually denoted in base 16. In the standard, a code point is written using the notation `U+12CA` to mean the character with value `0x12ca` (4,810 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
```

Strictly, these definitions imply that it's meaningless to say 『this is character `U+12CA`』. `U+12CA` is a code point, which represents some particular character; in this case, it represents the character 『ETHIOPIC SYLLABLE WI』. In informal contexts, this distinction between code points and characters will sometimes be forgotten.

一个字符在屏幕上或在纸上被表示为一组图形元素，被称为 ** 字形 **。比如，大写字母 A 的字形，是斜向的两笔和水平的一笔，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，找到应被显示的正确字形一般来说是用户图形界面工具箱或者终端的字体渲染器的工作。

1.3 编码

To summarize the previous section: a Unicode string is a sequence of code points, which are numbers from 0 through `0x10FFFF` (1,114,111 decimal). This sequence needs to be represented as a set of bytes (meaning, values from 0 through 255) in memory. The rules for translating a Unicode string into a sequence of bytes are called an **encoding**.

The first encoding you might think of is an array of 32-bit integers. In this representation, the string 「Python」would look like this:

```
   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
```

这个表达方式非常直接，但同时也存在一些问题。

1. 不够方便；不同的处理器对字节的排序不同。
2. 非常浪费空间。多数编码都小于 127，或者 255，所以很多空间都是 `0x00`。上面的字符串 takes 24 bytes compared to the 6 bytes needed for an ASCII representation. Increased RAM usage doesn't matter too much (desktop computers have gigabytes of RAM, and strings aren't usually that large), but expanding our usage of disk and network bandwidth by a factor of 4 is intolerable.
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。
4. Many Internet standards are defined in terms of textual data, and can't handle content with embedded zero bytes.

Generally people don't use this encoding, instead choosing other encodings that are more efficient and convenient. UTF-8 is probably the most commonly supported encoding; it will be discussed below.

Encodings don't have to handle every possible Unicode character, and most encodings don't. The rules for converting a Unicode string into the ASCII encoding, for example, are simple; for each code point:

1. If the code point is < 128, each byte is the same as the value of the code point.

2. If the code point is 128 or greater, the Unicode string can't be represented in this encoding. (Python raises a `UnicodeEncodeError` exception in this case.)

Latin-1, also known as ISO-8859-1, is a similar encoding. Unicode code points 0–255 are identical to the Latin-1 values, so converting to this encoding simply requires converting code points to byte values; if a code point larger than 255 is encountered, the string can't be encoded into Latin-1.

Encodings don't have to be simple one-to-one mappings like Latin-1. Consider IBM's EBCDIC, which was used on IBM mainframes. Letter values weren't in one block: 『a』through 『i』had values from 129 to 137, but 『j』through 『r』were 145 through 153. If you wanted to use EBCDIC as an encoding, you'd probably use some sort of lookup table to perform the conversion, but this is largely an internal detail.

UTF-8 is one of the most commonly used encodings. UTF stands for「Unicode Transformation Format」, and the 『8』means that 8-bit numbers are used in the encoding. (There are also a UTF-16 and UTF-32 encodings, but they are less frequently used than UTF-8.) UTF-8 uses the following rules:

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. A Unicode string is turned into a sequence of bytes containing no embedded zero bytes. This avoids byte-ordering issues, and means UTF-8 strings can be processed by C functions such as `strcpy()` and sent through protocols that can't handle zero bytes.
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。

1.4 参考文献

Unicode Consortium 站点 有 Unicode 规范的字符图表、词汇表和 PDF 版本。为一些困难的阅读做好准备。Unicode 起源和发展的年表 也可在该站点上找到。

To help understand the standard, Jukka Korpela has written an introductory guide to reading the Unicode character tables.

Another good introductory article was written by Joel Spolsky. If this introduction didn't make things clear to you, you should try reading this alternate article before continuing.

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

2 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。

2.1 字符串类型

Since Python 3.0, the language features a `str` type that contain Unicode characters, meaning any string created using `"unicode rocks!"`, `'unicode rocks!'`, or the triple-quoted string syntax is stored as 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' error message.
    print("Fichier non trouvé")
```

You can use a different encoding from UTF-8 by putting a specially-formatted comment as the first or second line of the source code:

```
# -*- coding: <encoding name> -*-
```

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}"  # Using the character name
'\u0394'
>>> "\u0394"                          # Using a 16-bit hex value
'\u0394'
>>> "\U00000394"                      # Using a 32-bit hex value
'\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'
```

Encodings are specified as strings containing the encoding’ s name. Python 3.2 comes with roughly 100 different encodings; see the Python Library Reference at standard-encodings for a list. Some encodings have multiple names; for example, `'latin-1'`, `'iso_8859_1'` and `'8859』` are all synonyms for the same encoding.

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
```

2.2 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。

2.3 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"
... #     ^^^^ two-digit hex escape
... #         ^^^^^^ four-digit Unicode escape
... #                ^^^^^^^^^^ eight-digit Unicode escape
>>> [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 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** 。

2.4 Unicode 属性

The Unicode specification includes a database of information about code points. For each defined code point, the information includes the character’s name, its category, the numeric value if applicable (Unicode has characters representing the Roman numerals and fractions such as one-third and four-fifths). There are also properties related to the code point’s use in bidirectional text and other display-related properties.

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))

# Get numeric value of second character
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缩写。这些被分组为“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。从上面的输出中获取代码，`'Ll'` 表示“字母，小写”，`'No'` 表示“数字，其他”，`'Mn'` 是“标记，非间距”，`'So'` 是“符号，其他”。有关类别代码列表，请参阅 Unicode 字符数据库文档 <http://www.unicode.org/reports/tr44/#General_Category_Values>‘_ 的“通用类别值”部分。

2.5 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]。

2.6 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。
- Pragmatic Unicode, a PyCon 2012 presentation by Ned Batchelder.

str 类型在 Python 库参考文档 textseq 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）。

3 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。

In some areas, it is also convention to use a 「BOM」at the start of UTF-8 encoded files; the name is misleading since UTF-8 is not byte-order dependent. The mark simply announces that the file is encoded in UTF-8. Use the 『utf-8-sig』codec to automatically skip the mark if present for reading such files.

3.1 Unicode 文件名

Most of the operating systems in common use today support filenames that contain arbitrary Unicode characters. Usually this is implemented by converting the Unicode string into some encoding that varies depending on the system. For example, Mac OS X uses UTF-8 while Windows uses a configurable encoding; on Windows, Python uses the name 「mbcs」to refer to whatever the currently configured encoding is. On Unix systems, there will only be a filesystem encoding if you’ ve set the `LANG` or `LC_CTYPE` environment variables; if you haven’ t, the default encoding is 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()` 。

The `os.listdir()` function returns filenames and raises an issue: should it return the Unicode version of filenames, or should it return bytes containing the encoded versions? `os.listdir()` will do both, depending on whether you provided the directory path as bytes or a Unicode string. If you pass a Unicode string as the path, filenames will be decoded using the filesystem’ s encoding and a list of Unicode strings will be returned, while passing a byte path will return the filenames as bytes. For example, assuming the default filesystem encoding is UTF-8, running the following program:

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

(下页继续)

(繼續上一頁)

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

将产生以下输出：

```
amk:~$ python t.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。

Note that on most occasions, the Unicode APIs should be used. The bytes APIs should only be used on systems where undecodable file names can be present, i.e. Unix systems.

3.2 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: 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') )
```

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` 错误处理程序打开文件：

```
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)
```

The `surrogateescape` error handler will decode any non-ASCII bytes as code points in the Unicode Private Use Area ranging from U+DC80 to U+DCFF. These private code points will then be turned back into the same bytes when the `surrogateescape` error handler is used when encoding the data and writing it back out.

3.3 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。

4 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。

Thanks to the following people who have noted errors or offered suggestions on this article: Éric Araujo, Nicholas Bastin, Nick Coghlan, Marius Gedminas, Kent Johnson, Ken Krugler, Marc-André Lemburg, Martin von Löwis, Terry J. Reedy, Chad Whitacre.

索引

P