Unicode HOWTO

Release 3.6.15

Guido van Rossum
and the Python development team

setembro 06, 2021

Python Software Foundation
Email: docs@python.org

Sumário

Release 1.12

Este documento fala sobre o suporte do Python para Unicode e explica diversos problemas que as pessoas normalmente encontram quando tentam trabalhar com Unicode.

1 Introdução ao Unicode

1.1 História dos Códigos de Caracteres

Em 1968 a American Standard Code for Information Interchange, conhecida principalmente pelo seu acrônimo ASCII, definiu uma padronização. ASCII definiu códigos numéricos para diversos caracteres, com os valores numércos indo de 0 a 127. Por exemplo, foi definido que o código de valor 97 representaria a letra minúscula 'a'.

O padrão ASCII foi feito para a língua americana, então ele definiu apenas caracteres sem acento. Ele continha 'e' mas não 'é' ou 'í'. Isso significa que as línguas que utilizam caracteres acentuados não poderiam ser representadas de maneira fiel no código ASCII. (Atualmente os acentos importam também para o Inglês, que agora contém palavras como 'naïve' e 'café', e algumas publicações tem estilos que requerem palavras como '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"
```

Essas mensagens deveriam conter acentos (terminée, paramètre, enregistrés) e elas simplesmente estão erradas para qualquer um que saiba ler Francês.

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.

Mas 255 caracteres também não é muita coisa. Por exemplo, você não consegue inserir os caracteres acentuados usados na Europa Oriental e o alfaveto cirílico usado pelos Russos entre os valores de 128-255 porque existem mais que 128 desses caracteres.

Você poderia escrever todos os seus arquivos usando códigos diferentes (todos os seus arquivos Russos em um sistema de códigos chamado KOI8 e todos os seus arquvos em Francês em um sistema de códigos chamado Latin1), mas e se você quisesse escrever um documento Francês que continha citações de um texto Russo? Em 1980 as pessoas queriam resolver esse problema e o eforço de padronização Unicode começou.

O Unicode começou utilizando caracteres de 16-bit ao invés de 8-bit. 16 bits implica 2^16 = 65,536 valores distintos disponíveis, possibilitanto representar diversos caracteres distintos de muitos alfabetos distintos; o objetivo inicial era que o Unicode fosse composto de todos os alfabetos de todas as linguagens humanas. Entretanto, mesmo 16 bits não é suficiente para alcançar esse objetivo, e o padrão Unicode moderno usa uma quantidade maiores de códigos, de 0 a (`0x10FFFF` em base 16).

Existe uma norma ISO semelhante, a ISO 10646. Originalmente, o Unicode e a ISO 10646 eram esforços separados, mas os padrões foram unificados na revisão 1.1 do Unicode.

(Essa discussão da história do Unicode está altamente simplificada. Os detalhes de precisão histórica não são necessários para entender como utilizar o Unicode de forma efetiva, mas se você tiver curiosidade, consulte o site do Unicode consortium listado nas referências ou o artigo da Wikipedia sobre Unicode <https://en.wikipedia.org/wiki/Unicode#History>'_ para mais informações)

1.2 Definições

Um **caracter** é o menor componente possível de um texto. 'A', 'B', 'C', etc., são todos caracteres diferentes. O mesmo vale para 'È' and 'Í'. Caracteres são abstrações e variam de acordo com a língua ou o contexto que você está inserido. Por exemplo, o sómbolo para ohms (Ω) é normalmente desenhado de forma semelhante à letra maiúscula omega (Ω) do alfabeto grego (eles podem ser até iguais em algumas fontes), mas eles são dois caracteres diferentes que contém significados diferentes.

O padrão Unicode descreve como caracteres podem ser representado por **pontos de código**. Ponto de código é um valor inteiro, normalmente denotado na base 16. Por padrão, um ponto de código é escrito usando a notação `U+12CA``para representar caracteres com valor ``0x12ca` (41810 decimais). O padrão Unicode contém uma série de tabelas listando caracteres e seus pontos de códigos correspondentes:

```
0061    'a'; LATIN SMALL LETTER A
0062    'b'; LATIN SMALL LETTER B
0063    'c'; LATIN SMALL LETTER C
...
007B    '{'; LEFT CURLY BRACKET
```

Estritamente, essas definições fazem sem sentido a frase 'esse caracter é `U+12CA`'. `U+12CA` é um ponto de código que representa algum caracter particular; neste caso, ele representa o caracter 'SÍLABA ETÍPICO WI'. Em contextos informais, a distinção entre os pontos de código e caracteres é normalmente esquecida.

Um caracter é representado na tela ou no papel como um conjunto de elementos gráficos que é chamado de **glifo**. O glifo para o A maiúsculo, por exemplo, são dois traços diagonais e um traço horizontal, embora os detalhes exatos dependem da fonte utilizada. Na maior parte do código Python não é preciso se preocupar com glifos; descobrir qual o glifo correto a ser mostrado é normalmente parte do trabalho da ferramenta GUI ou do responsável pela renderização de fontes no terminal.

1.3 Encodings

Resumindo a sessão anterior: uma string Unicode é uma sequência de pontos de código, que são números de 0 a `0x10FFFF` (1,114,111 decimal). Essa sequência precisa ser representada como um conjunto de bytes (ou seja, valores de 0 a 255) em memória. As regras para traduzir uma string Unicode em uma sequência de bytes são chamadas de **enconding**s.

O primeiro enconding que podemos pensar é um vetor de inteiros de 32-bits. Nessa representação, a string "Python" seria descrita da seguinte forma:

```
   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 representação é direta, mas usá-la gera uma série de problemas.

1. Ela não é portátil; diferentes processadores ordenam os bytes de forma diferente.
2. Ela gera desperdício de espaço. Na maior parte dos textos, a maioria dos pontos de código são menores que 127 ou menores que 255, então muito do espaço é ocupado pot bytes `0x00` . A string acima necessita 24 bytes comparado com os 6 bytes necessários em uma representação ASCII. O aumento de uso da memória RAM normalmente não importa tanto (computadores desktop possuem gigabytes de RAM e strings normalmente não são tão grandes), mas expandir o uso de disco ou de banda por um fator de 4 é inaceitável.
3. Ela não é compatível com as funções de C existentes, como `strlen()`, então uma série de novas funções de string serão necessárias.
4. Muitos padrões de internet são definidos por termos de dados textuais e não podem lidar com conteúdo com muitos zero bytes embutidos.

Geralmente não se usa esse tipo de encoding e escolhe-se outros tipos de encoding que são mais eficientes e convenientes. UTF-8 é provavelmente o encoding mais normalmente usado; isso será discutido abaixo.

Encodings não precisam lidar com todos os tipos possíveis de caracter Unicode, e a maioria de fato não suportam todos. As regras para converter uma string Unicode para o encoding ASCII, por exemplo, são simples; para cada code point temos:

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. If the code point is < 128, it's represented by the corresponding byte value.
2. If the code point is >= 128, it's turned into a sequence of two, three, or four bytes, where each byte of the sequence is between 128 and 255.

UTF-8 tem muitas propriedades convenientes:

1. Ela pode lidar com qualquer ponto de código Unicode.
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. Uma string de texto ASCII é também um texto UTF-8 válido.
4. UTF-8 is fairly compact; the majority of commonly used characters can be represented with one or two bytes.
5. If bytes are corrupted or lost, it's possible to determine the start of the next UTF-8-encoded code point and resynchronize. It's also unlikely that random 8-bit data will look like valid UTF-8.

1.4 Referências

The Unicode Consortium site has character charts, a glossary, and PDF versions of the Unicode specification. Be prepared for some difficult reading. A chronology of the origin and development of Unicode is also available on the site.

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 entries are often helpful; see the entries for "character encoding" and UTF-8, for example.

2 Suporte a Unicode no Python

Now that you've learned the rudiments of Unicode, we can look at Python's Unicode features.

2.1 O Tipo String

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.

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")
```

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> -*-
```

Side note: Python 3 also supports using Unicode characters in identifiers:

```
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'
```

In addition, one can create a string using the `decode()` method of `bytes`. This method takes an *encoding* argument, such as `UTF-8`, and optionally an *errors* argument.

The *errors* argument specifies the response when the input string can't be converted according to the encoding's rules. Legal values for this argument are `'strict'` (raise a `UnicodeDecodeError` exception), `'replace'` (use `U+FFFD`, `REPLACEMENT CHARACTER`), `'ignore'` (just leave the character out of the Unicode result), or `'backslashreplace'` (inserts a `\xNN` escape sequence). The following examples show the differences:

```
>>> 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'
```

(continua na próxima página)

(continuação da página anterior)

```
>>> 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.

One-character Unicode strings can also be created with the `chr()` built-in function, which takes integers and returns a Unicode string of length 1 that contains the corresponding code point. The reverse operation is the built-in `ord()` function that takes a one-character Unicode string and returns the code point value:

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

2.2 Convertendo para Bytes

The opposite method of `bytes.decode()` is `str.encode()`, which returns a `bytes` representation of the Unicode string, encoded in the requested *encoding*.

The *errors* parameter is the same as the parameter of the `decode()` method but supports a few more possible handlers. As well as `'strict'`, `'ignore'`, and `'replace'` (which in this case inserts a question mark instead of the unencodable character), there is also `'xmlcharrefreplace'` (inserts an XML character reference), `backslashreplace` (inserts a `\uNNNN` escape sequence) and `namereplace` (inserts a `\N{...}` escape sequence).

The following example shows the different results:

```
>>> 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'
```

The low-level routines for registering and accessing the available encodings are found in the `codecs` module. Implementing new encodings also requires understanding the `codecs` module. However, the encoding and decoding functions returned by this module are usually more low-level than is comfortable, and writing new encodings is a specialized task, so the module won't be covered in this HOWTO.

2.3 Unicode Literals in Python Source Code

In Python source code, specific Unicode code points can be written using the `\u` escape sequence, which is followed by four hex digits giving the code point. The `\U` escape sequence is similar, but expects eight hex digits, not four:

```
>>> 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]
```

Using escape sequences for code points greater than 127 is fine in small doses, but becomes an annoyance if you're using many accented characters, as you would in a program with messages in French or some other accent-using language. You can also assemble strings using the `chr()` built-in function, but this is even more tedious.

Ideally, you'd want to be able to write literals in your language's natural encoding. You could then edit Python source code with your favorite editor which would display the accented characters naturally, and have the right characters used at runtime.

Python supports writing source code in UTF-8 by default, but you can use almost any encoding if you declare the encoding being used. This is done by including a special comment as either the first or second line of the source file:

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

u = 'abcdé'
print(ord(u[-1]))
```

The syntax is inspired by Emacs's notation for specifying variables local to a file. Emacs supports many different variables, but Python only supports 'coding'. The `-*-` symbols indicate to Emacs that the comment is special; they have no significance to Python but are a convention. Python looks for `coding: name` or `coding=name` in the comment.

If you don't include such a comment, the default encoding used will be UTF-8 as already mentioned. See also **PEP 263** for more information.

2.4 Propriedades Unicode

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.

O programa a seguir exibe alguma informação sobre diversos caracteres e imprime o valor numérico de um caractere em 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]))
```

Quando executado, isso 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
```

The category codes are abbreviations describing the nature of the character. These are grouped into categories such as "Letter", "Number", "Punctuation", or "Symbol", which in turn are broken up into subcategories. To take the codes from the above output, `'Ll'` means 'Letter, lowercase', `'No'` means "Number, other", `'Mn'` is "Mark, nonspacing", and `'So'` is "Symbol, other". See the General Category Values section of the Unicode Character Database documentation for a list of category codes.

2.5 Expressões Regulares Unicode

The regular expressions supported by the `re` module can be provided either as bytes or strings. Some of the special character sequences such as `\d` and `\w` have different meanings depending on whether the pattern is supplied as bytes or a string. For example, `\d` will match the characters `[0-9]` in bytes but in strings will match any character that's in the `'Nd'` category.

The string in this example has the number 57 written in both Thai and Arabic numerals:

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

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

When executed, `\d+` will match the Thai numerals and print them out. If you supply the `re.ASCII` flag to `compile()`, `\d+` will match the substring "57" instead.

Similarly, `\w` matches a wide variety of Unicode characters but only `[a-zA-Z0-9_]` in bytes or if `re.ASCII` is supplied, and `\s` will match either Unicode whitespace characters or `[ \t\n\r\f\v]`.

2.6 Referências

Some good alternative discussions of Python's Unicode support are:

- Processing Text Files in Python 3, by Nick Coghlan.
- Pragmatic Unicode, a PyCon 2012 presentation by Ned Batchelder.

The `str` type is described in the Python library reference at textseq.

The documentation for the `unicodedata` module.

The documentation for the `codecs` module.

Marc-André Lemburg gave a presentation titled "Python and Unicode" (PDF slides) at EuroPython 2002. The slides are an excellent overview of the design of Python 2's Unicode features (where the Unicode string type is called `unicode` and literals start with `u`).

3 Reading and Writing Unicode Data

Once you've written some code that works with Unicode data, the next problem is input/output. How do you get Unicode strings into your program, and how do you convert Unicode into a form suitable for storage or transmission?

It's possible that you may not need to do anything depending on your input sources and output destinations; you should check whether the libraries used in your application support Unicode natively. XML parsers often return Unicode data, for example. Many relational databases also support Unicode-valued columns and can return Unicode values from an SQL query.

Unicode data is usually converted to a particular encoding before it gets written to disk or sent over a socket. It's possible to do all the work yourself: open a file, read an 8-bit bytes object from it, and convert the bytes with `bytes.decode(encoding)`. However, the manual approach is not recommended.

One problem is the multi-byte nature of encodings; one Unicode character can be represented by several bytes. If you want to read the file in arbitrary-sized chunks (say, 1024 or 4096 bytes), you need to write error-handling code to catch the case where only part of the bytes encoding a single Unicode character are read at the end of a chunk. One solution would be to read the entire file into memory and then perform the decoding, but that prevents you from working with files that are extremely large; if you need to read a 2 GiB file, you need 2 GiB of RAM. (More, really, since for at least a moment you'd need to have both the encoded string and its Unicode version in memory.)

The solution would be to use the low-level decoding interface to catch the case of partial coding sequences. The work of implementing this has already been done for you: the built-in `open()` function can return a file-like object that assumes the file's contents are in a specified encoding and accepts Unicode parameters for methods such as `read()` and `write()`. This works through `open()`'s *encoding* and *errors* parameters which are interpreted just like those in `str.encode()` and `bytes.decode()`.

Reading Unicode from a file is therefore simple:

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

It's also possible to open files in update mode, allowing both reading and writing:

```
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]))
```

The Unicode character `U+FEFF` is used as a byte-order mark (BOM), and is often written as the first character of a file in order to assist with autodetection of the file's byte ordering. Some encodings, such as UTF-16, expect a BOM to be present at the start of a file; when such an encoding is used, the BOM will be automatically written as the first character and will be silently dropped when the file is read. There are variants of these encodings, such as 'utf-16-le' and 'utf-16-be' for little-endian and big-endian encodings, that specify one particular byte ordering and don't skip the 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 Nomes de arquivos Unicode

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.

The `sys.getfilesystemencoding()` function returns the encoding to use on your current system, in case you want to do the encoding manually, but there's not much reason to bother. When opening a file for reading or writing, you can usually just provide the Unicode string as the filename, and it will be automatically converted to the right encoding for you:

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

Functions in the `os` module such as `os.stat()` will also accept Unicode filenames.

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()

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

will produce the following output:

```
amk:~$ python t.py
[b'filename\xe4\x94\x80abc', ...]
['filename\u4500abc', ...]
```

The first list contains UTF-8-encoded filenames, and the second list contains the Unicode versions.

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 Tips for Writing Unicode-aware Programs

This section provides some suggestions on writing software that deals with Unicode.

A dica mais importante é:

> Software should only work with Unicode strings internally, decoding the input data as soon as possible and encoding the output only at the end.

If you attempt to write processing functions that accept both Unicode and byte strings, you will find your program vulnerable to bugs wherever you combine the two different kinds of strings. There is no automatic encoding or decoding: if you do e.g. `str + bytes`, a `TypeError` will be raised.

When using data coming from a web browser or some other untrusted source, a common technique is to check for illegal characters in a string before using the string in a generated command line or storing it in a database. If you're doing this, be careful to check the decoded string, not the encoded bytes data; some encodings may have interesting properties, such as not being bijective or not being fully ASCII-compatible. This is especially true if the input data also specifies the encoding, since the attacker can then choose a clever way to hide malicious text in the encoded bytestream.

Converting Between File Encodings

The `StreamRecoder` class can transparently convert between encodings, taking a stream that returns data in encoding #1 and behaving like a stream returning data in encoding #2.

For example, if you have an input file *f* that's in Latin-1, you can wrap it with a `StreamRecoder` to return bytes encoded in 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') )
```

Files in an Unknown Encoding

What can you do if you need to make a change to a file, but don't know the file's encoding? If you know the encoding is ASCII-compatible and only want to examine or modify the ASCII parts, you can open the file with the `surrogateescape` error handler:

```
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 Referências

One section of Mastering Python 3 Input/Output, a PyCon 2010 talk by David Beazley, discusses text processing and binary data handling.

The PDF slides for Marc-André Lemburg's presentation "Writing Unicode-aware Applications in Python" discuss questions of character encodings as well as how to internationalize and localize an application. These slides cover Python 2.x only.

The Guts of Unicode in Python is a PyCon 2013 talk by Benjamin Peterson that discusses the internal Unicode representation in Python 3.3.

4 Reconhecimentos

The initial draft of this document was written by Andrew Kuchling. It has since been revised further by Alexander Belopolsky, Georg Brandl, Andrew Kuchling, and Ezio Melotti.

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.

Índice

P