

---

# Unicode HOWTO

リリース 3.8.20

Guido van Rossum  
and the Python development team

12 月 09, 2024

## 目次

1	Unicode 入門	2
1.1	定義	2
1.2	エンコーディング	3
1.3	参考資料	4
2	Python の Unicode サポート	5
2.1	文字列型	5
2.2	バイト列への変換	6
2.3	Python ソースコード内の Unicode リテラル	7
2.4	Unicode プロパティ	8
2.5	Comparing Strings	9
2.6	Unicode 正規表現	10
2.7	参考資料	11
3	Unicode データを読み書きする	11
3.1	Unicode ファイル名	12
3.2	Unicode 対応のプログラムを書くための Tips	13
3.3	参考資料	15
4	謝辞	15
	索引	16

---

リリース 1.12

この HOWTO 文書は、文字データの表現のための Unicode 仕様の Python におけるサポートについて論じ、さらに Unicode を使おうというときによく出喰わす多くの問題について説明します。

## 1 Unicode 入門

### 1.1 定義

今日のプログラムは広範囲の文字を扱える必要があります。アプリケーションは国際化され、ユーザーが選べる様々な言語でメッセージや出力を表示します; 同じプログラムが、英語、フランス語、日本語、ヘブライ語、ロシア語でエラーメッセージを出力する必要があるかもしれません。Web コンテンツはどんな言語でも書かれる可能性がありますし、様々な絵文字が含まれることもあります。Python の文字列型は文字表現のための Unicode 標準を使っていて、Python プログラムは有り得る様々な文字を全て扱えます。

Unicode (<https://www.unicode.org/>) は、人類の言語で使われる全ての文字を列挙し、それぞれの文字自身の一意な符号を与えるのを目的とした仕様です。Unicode 仕様は継続的に改訂され、新しい言語や記号を追加する更新がなされています。

**文字** は文章の最小の構成要素です。'A', 'B', 'C' などは全て異なる文字です。'È' と 'Í' も同様に異なる文字です。文字は、話している言語や文脈によって変わってきます。例えば、「ローマ数字の 1」という文字 'I' は大文字の 'I' とは別の文字です。両者は通常は同じに見えますが、異なる意味を持つ別々の 2 つの文字です。

Unicode 標準は文字がどのように **コードポイント (code points)** で表現されるのかを記述しています。コードポイントの値は 0 から 0x10FFFF までの範囲の整数です (約 110 万個の値であり、今のところ 11 万個くらいが割り当てられています)。Unicode 標準やこのドキュメントでは、コードポイントを U+265E という記法を使って書き、これはその文字が値 0x265e (10 進数では 9,822) を持つことを意味します。

Unicode 標準は、文字とそれに対応するコードポイントを列挙した多くの表を含んでいます:

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
...	

厳密には、この定義から「これは文字 U+265E です」と言うのは意味の無いことだと分かります。U+265E はコードポイントであり、それはある特定の文字を表しているのです; この場合では、'BLACK CHESS KNIGHT', '♘' という文字を表しています。形式ばらない文脈では、このコードポイントと文字の区別は忘れ去られることもあります。

文字は画面や紙面上では **グリフ (glyph)** と呼ばれるグラフィック要素の組で表示されます。大文字の A のグリフは例えば、厳密な形は使っているフォントによって異なりますが、斜めの線と水平の線です。たいていの Python コードではグリフの心配をする必要はありません; 一般的には表示する正しいグリフを見付けることは GUI toolkit や端末のフォントレンダラーの仕事です。

## 1.2 エンコーディング

前の節をまとめると: Unicode 文字列はコードポイントの列であり、コードポイントとは 0 から 0x10FFFF (10 進表記で 1,114,111) までの数値です。このコードポイント列はメモリ上では **コードユニット** 列として表され、その **コードユニット** 列は 8-bit のバイト列にマップされます。Unicode 文字列をバイト列として翻訳する規則を **文字エンコーディング** または単に **エンコーディング** と呼びます。

The first encoding you might think of is using 32-bit integers as the code unit, and then using the CPU's representation of 32-bit integers. In this representation, the string "Python" might 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

この表現は直接的でわかりやすい方法ですが、この表現を使うにはいくつかの問題があります。

1. 可搬性がない; プロセッサが異なるとバイトの順序づけも変わってしまいます。
2. 無駄な領域が多いです。多くの文書では、コードポイントは 127 未満もしくは 255 未満が多数派を占め、そのため多くの領域が 0x00 というバイトで埋め尽くされます。上の文字列は、ASCII 表現では 6 バイトなのに対し、24 バイトのサイズになっています。RAM の使用量が増加するのはそれほど問題にはなりません (デスクトップコンピュータはギガバイト単位の RAM を持っており、通常、文字列はそんな大きさにはなりません) が、ディスクとネットワーク帯域が 4 倍多く使われてしまうのは我慢できるものではありません。
3. `strlen()` のような現存する C 関数と互換性がありません、そのためワイド文字列関数一式が新たに必要となります。

Therefore this encoding isn't used very much, and people instead choose other encodings that are more efficient and convenient, such as UTF-8.

UTF-8 is one of the most commonly used encodings, and Python often defaults to using it. UTF stands for "Unicode Transformation Format", and the '8' means that 8-bit values are used in the encoding. (There are also UTF-16 and UTF-32 encodings, but they are less frequently used than UTF-8.) UTF-8 uses the

following rules:

1. コードポイントが 128 未満だった場合、対応するバイト値で表現します。
2. コードポイントが 128 以上の場合、128 から 255 までのバイトからなる、2、3 または 4 バイトのシーケンスに変換します。

UTF-8 はいくつかの便利な性質を持っています:

1. 任意の Unicode コードポイントを扱うことができる。
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. ASCII テキストの文字列は UTF-8 テキストとしても有効です。
4. UTF-8 はかなりコンパクトです; よく使われている文字の大多数は 1 バイトか 2 バイトで表現できます。
5. バイトが欠落したり、失われた場合、次の UTF-8 でエンコードされたコードポイントの開始を決定し、再同期することができる可能性があります。同様の理由でランダムな 8-bit データは正当な UTF-8 とみなされにくくなっています。
6. UTF-8 is a byte oriented encoding. The encoding specifies that each character is represented by a specific sequence of one or more bytes. This avoids the byte-ordering issues that can occur with integer and word oriented encodings, like UTF-16 and UTF-32, where the sequence of bytes varies depending on the hardware on which the string was encoded.

## 1.3 参考資料

[Unicode コンソーシアムのサイト](#) には文字の図表、用語辞典、PDF 版の Unicode 仕様があります。これ読むのはそれなりに難しいので覚悟してください。Unicode の起源と発展の [年表](#) もサイトにあります。

On the Computerphile Youtube channel, Tom Scott briefly [discusses the history of Unicode and UTF-8](#) (9 minutes 36 seconds).

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 の記事はしばしば役に立ちます; 例えば、"[character encoding](#)" や [UTF-8](#) の記事を読んでみてください。

## 2 Python の Unicode サポート

ここまでで Unicode の基礎を学びました、ここから Python の Unicode 機能に触れます。

### 2.1 文字列型

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.

Python ソースコードのデフォルトエンコーディングは UTF-8 なので、文字列リテラルの中に Unicode 文字をそのまま含めることができます:

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

追記: Python3 は Unicode 文字を使った識別子もサポートしています:

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

エディタである特定の文字が入力できなかったり、とある理由でソースコードを ASCII のみに保ちたい場合は、文字列リテラルでエスケープシーケンスが使えます。(使ってるシステムによっては、`u` でエスケープされた文字列ではなく、実物の大文字のラムダのグリフが見えるかもしれません。):

```
>>> "\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'
```

加えて、`bytes` クラスの `decode()` メソッドを使って文字列を作ることができます。このメソッドは UTF-8 のような値を `encoding` 引数に取り、オプションで `errors` 引数を取ります。

`errors` 引数は、入力文字列に対しエンコーディングルールに従った変換ができなかったときの対応方法を指定します。この引数に使える値は `'strict'` (`UnicodeDecodeError` を送出する)、`'replace'` (REPLACEMENT CHARACTER である `U+FFFD` を使う)、`'ignore'` (結果となる Unicode から単に文字を除く)、`'backslashreplace'` (エスケープシーケンス `\xNN` を挿入する) です。次の例はこれらの違いを示しています:

```
>>> 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'
```

エンコーディングはエンコーディングの名前を含んだ文字列で指定されます。Python はおよそ 100 の異なるエンコーディングに対応しています; 一覧は Python ライブラリリファレンスの `standard-encodings` を参照してください。いくつかのエンコーディングは複数の名前を持っています; 例えば、`'latin-1'` と `'iso_8859_1'` と `'8859'` は全て同じエンコーディングの別名です。

Unicode 文字列の一つの文字は `chr()` 組み込み関数で作成することができます、この関数は整数を引数にとり、対応するコードポイントを含む長さ 1 の Unicode 文字列を返します。逆の操作は `ord()` 組み込み関数です、この関数は一文字の Unicode 文字列を引数にとり、コードポイント値を返します:

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

## 2.2 バイト列への変換

`bytes.decode()` とは処理が逆向きのメソッドが `str.encode()` です。このメソッドは、Unicode 文字列を指定された *encoding* でエンコードして、`bytes` による表現で返します。

`errors` 引数は `decode()` メソッドのパラメータと同じものですが、サポートされているハンドラの数がもう少し多いです。 `'strict'`、`'ignore'`、`'replace'` (このメソッドでは、エンコードできなかった文字の代わりに疑問符を挿入する) の他に、`'xmlcharrefreplace'` (XML 文字参照を挿入する) と `backslashreplace` (エスケープシーケンス `\nNNNN` を挿入する)、`namereplace` (エスケープシーケンス `\N{...}` を挿入する) があります。

次の例では、それぞれの異なる処理結果が示されています:

```
>>> u = chr(40960) + 'abcd' + chr(1972)
>>> u.encode('utf-8')
b'\xea\x80\x80abcd\xde\xb4'
>>> u.encode('ascii')
Traceback (most recent call last):
...
```

(次のページに続く)

```
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'
```

利用可能なエンコーディングを登録したり、アクセスしたりする低レベルのルーチンは `codecs` モジュールにあります。新しいエンコーディングを実装するには、`codecs` モジュールを理解していることも必要になります。しかし、このモジュールのエンコードやデコードの関数は、使い勝手が良いというより低レベルな関数で、新しいエンコーディングを書くのは特殊な作業なので、この HOWTO では扱わないことにします。

## 2.3 Python ソースコード内の Unicode リテラル

Python のソースコード内では、特定のコードポイントはエスケープシーケンス `\u` を使い、続けてコードポイントを 4 桁の 16 進数を書きます。エスケープシーケンス `\U` も同様です、ただし 4 桁ではなく 8 桁の 16 進数を使います:

```
>>> 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]
```

127 より大きいコードポイントに対してエスケープシーケンスを使うのは、エスケープシーケンスがあまり多くないうちは有効ですが、フランス語等のアクセントを使う言語でメッセージのような多くのアクセント文字を使う場合には邪魔になります。文字を `chr()` 組み込み関数を使って組み上げることもできますが、それはさらに長くなってしまいうでしょう。

理想的にはあなたの言語の自然なエンコーディングでリテラルを書くことでしょう。そうなれば、Python のソースコードをアクセント付きの文字を自然に表示するお気に入りのエディタで編集し、実行時に正しい文字が得られます。

Python はデフォルトでは UTF-8 ソースコードを書くことができます、ただしどのエンコーディングを使うかを宣言すればほとんどのエンコーディングを使えます。それはソースファイルの一行目や二行目に特別なコメントを含めることによってできます:

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

u = 'abcdé'
print(ord(u[-1]))
```

この構文は Emacs のファイル固有の変数を指定する表記から影響を受けています。Emacs は様々な変数をサポートしていますが、Python がサポートしているのは 'coding' のみです。-\*- の記法は Emacs に対してコメントが特別であることを示します。これは Python にとって意味はありませんが慣習で使われています。Python はコメント中に coding: name または coding=name を探します。

このようなコメントを含んでいない場合、すでに述べた通り、使われるデフォルトエンコーディングは UTF-8 になります。より詳しい情報は [PEP 263](#) を参照してください。

## 2.4 Unicode プロパティ

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 (for characters representing numeric concepts such as the Roman numerals, fractions such as one-third and four-fifths, etc.). There are also display-related properties, such as how to use the code point in bidirectional text.

以下のプログラムはいくつかの文字に対する情報を表示し、特定の文字の数値を印字します:

```
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]))
```

実行すると、このように出力されます:

```
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
```

カテゴリーコードは文字の性質を略記で表したものです。カテゴリーコードは "Letter"、"Number"、"Punctuation"、"Symbol" などのカテゴリーに分類され、さらにサブカテゴリーに細分化されます。上記の出力からコー



ドを拾うと、'Ll' は 'Letter, lowercase'、'No' は "Number, other"、'Mn' は "Mark, nonspacing"、'So' は "Symbol, other" を意味しています。カテゴリーコードの一覧は [Unicode Character Database 文書の General Category Values 節](#) を参照してください。

## 2.5 Comparing Strings

Unicode adds some complication to comparing strings, because the same set of characters can be represented by different sequences of code points. For example, a letter like 'ê' can be represented as a single code point U+00EA, or as U+0065 U+0302, which is the code point for 'e' followed by a code point for 'COMBINING CIRCUMFLEX ACCENT'. These will produce the same output when printed, but one is a string of length 1 and the other is of length 2.

One tool for a case-insensitive comparison is the `casefold()` string method that converts a string to a case-insensitive form following an algorithm described by the Unicode Standard. This algorithm has special handling for characters such as the German letter 'ß' (code point U+00DF), which becomes the pair of lowercase letters 'ss'.

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

A second tool is the `unicodedata` module's `normalize()` function that converts strings to one of several normal forms, where letters followed by a combining character are replaced with single characters. `normalize()` can be used to perform string comparisons that won't falsely report inequality if two strings use combining characters differently:

```
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))
```

実行すると、このように出力されます:

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

(次のページに続く)

```
length of second string= 2
True
```

The first argument to the `normalize()` function is a string giving the desired normalization form, which can be one of 'NFC', 'NFKC', 'NFD', and 'NFKD'.

The Unicode Standard also specifies how to do caseless comparisons:

```
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))
```

This will print `True`. (Why is `NFD()` invoked twice? Because there are a few characters that make `casefold()` return a non-normalized string, so the result needs to be normalized again. See section 3.13 of the Unicode Standard for a discussion and an example.)

## 2.6 Unicode 正規表現

`re` モジュールがサポートしている正規表現はバイト列や文字列として与えられます。`\d` や `\w` などのいくつかの特殊な文字シーケンスは、そのパターンがバイト列として与えられたのか文字列として与えられたのかによって、異なる意味を持ちます。例えば、`\d` はバイト列では `[0-9]` の範囲の文字と一致しますが、文字列では `'Nd'` カテゴリにある任意の文字と一致します。

この例にある文字列には、タイ語の数字とアラビア数字の両方で数字の 57 が書いてあります。

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

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

実行すると、`\d+` はタイ語の数字と一致し、それを出力します。フラグ `re.ASCII` を `compile()` に渡した場合、`\d+` は先程とは違って部分文字列 `"57"` に一致します。

同様に、`\w` は非常に多くの Unicode 文字に一致しますが、バイト列の場合もしくは `re.ASCII` が渡された場合は `[a-zA-Z0-9_]` にしか一致しません。`\s` は文字列では Unicode 空白文字に、バイト列では `[\t\n\r\f\v]` に一致します。

## 2.7 参考資料

Python の Unicode サポートについての参考になる議論は以下の 2 つです:

- Nick Coghlan による [Processing Text Files in Python 3](#)
- Ned Batchelder による PyCon 2012 での発表 [Pragmatic Unicode](#)

`str` 型については Python ライブラリリファレンスの `textseq` で解説されています。

`unicodedata` モジュールについてのドキュメント。

`codecs` モジュールについてのドキュメント。

Marc-André Lemburg は EuroPython 2002 で ["Python and Unicode"](#) というタイトルのプレゼンテーション (PDF スライド) を行いました。このスライドは Python 2 の Unicode 機能 (Unicode 文字列型が `unicode` と呼ばれ、リテラルは `u` で始まります) の設計について概観する素晴らしい資料です。

## 3 Unicode データを読み書きする

一旦 Unicode データに対してコードが動作するように書き終えたら、次の問題は入出力です。プログラムは Unicode 文字列をどう受けとり、どう Unicode を外部記憶装置や送受信装置に適した形式に変換するのでしょうか?

入力ソースと出力先に依存しないような方法は可能です; アプリケーションに利用されているライブラリが Unicode をそのままサポートしているかを調べなければいけません。例えば XML パーサーは大抵 Unicode データを返します。多くのリレーショナルデータベースも Unicode 値の入ったコラムをサポートしていますし、SQL の問い合わせで Unicode 値を返すことができます。

Unicode のデータはディスクに書き込まれるにあたって通常、特定のエンコーディングに変換されます。推奨はされませんが、これを手動で行うことも可能です。ファイルを開き、8 バイトオブジェクトを読み込み、バイト列を `bytes.decode(encoding)` で変換することにより実現できます。

1 つの問題はエンコーディングのマルチバイトという性質です; 1 つの Unicode 文字はいくつかのバイトで表現され得ます。任意のサイズのチャンク (例えば、1024 もしくは 4096 バイト) にファイルの内容を読み込みたい場合、ある 1 つの Unicode 文字をエンコードしたバイト列が、チャンクの末尾での一部分のみ読み込まれる場合のエラー処理のためのコードを書く必要があります。1 つの解決策はファイル全体をメモリに読み込み、デコード処理を実行することですが、こうしてしまうと非常に大きなファイルを処理するときの妨げになります; 2 GiB のファイルを読み込む必要がある場合、2 GiB の RAM が必要になります。(実際には、少なくともある瞬間では、エンコードされた文字列と Unicode 文字列の両方をメモリに保持する必要があるため、より多くのメモリが必要です。)

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()`.

そのためファイルから Unicode を読むのは単純です:

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

読み書きの両方ができる update モードでファイルを開くことも可能です:

```
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 文字 U+FEFF は byte-order mark (BOM) として使われ、ファイルのバイト順の自動判定を支援するために、ファイルの最初の文字として書かれます。UTF-16 のようないくつかのエンコーディングは、ファイルの先頭に BOM があることを要求します; そのようなエンコーディングが使われるとき、自動的に BOM が最初の文字として書かれ、ファイルを読むときに暗黙の内に取り除かれます。これらのエンコーディングには、リトルエンディアン (little-endian) 用の 'utf-16-le' やビッグエンディアン (big-endian) 用の 'utf-16-be' というような変種があり、これらは特定の 1 つのバイト順を指定していて BOM をスキップしません。

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. For reading such files, use the 'utf-8-sig' codec to automatically skip the mark if present.

### 3.1 Unicode ファイル名

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. Today Python is converging on using UTF-8: Python on MacOS has used UTF-8 for several versions, and Python 3.6 switched to using UTF-8 on Windows as well. 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 again UTF-8.

`sys.getfilesystemencoding()` 関数は現在のシステムで利用するエンコーディングを返し、エンコーディングを手動で設定したい場合利用します、ただしわざわざそうする積極的な理由はありません。読み書きのためにファイルを開く時には、ファイル名を Unicode 文字列として渡すだけで正しいエンコーディングに自動的に変更されます:

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

`os.stat()` のような `os` モジュールの関数も Unicode のファイル名を受け付けます。

The `os.listdir()` function returns filenames, which raises an issue: should it return the Unicode version of filenames, or should it return bytes containing the encoded versions? `os.listdir()` can 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('.'))
```

以下の出力結果が生成されます:

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

最初のリストは UTF-8 でエンコーディングされたファイル名を含み、第二のリストは Unicode 版を含んでいます。

Note that on most occasions, you should can just stick with using Unicode with these APIs. The bytes APIs should only be used on systems where undecodable file names can be present; that's pretty much only Unix systems now.

## 3.2 Unicode 対応のプログラムを書くための Tips

この章では Unicode を扱うプログラムを書くためのいくつかの提案を紹介します。

最も重要な助言としては:

ソフトウェアは内部では Unicode 文字列のみを利用し、入力データはできるだけ早期にデコードし、出力の直前でエンコードすべきです。

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.

web ブラウザから来るデータやその他の信頼できないところからのデータを利用する場合、それらの文字列から生成したコマンド行の実行や、それらの文字列をデータベースに蓄える前に文字列の中に不正な文字が含まれていないか確認するのが一般的です。もしそういう状況になった場合には、エンコードされたバイトデータではなく、デコードされた文字列のチェックを入念に行なって下さい; いくつかのエンコーディングは問題となる性質を持っています、例えば全単射でなかったり、完全に ASCII 互換でないなど。入力データがエンコーディングを指定している場合でもそうして下さい、なぜなら攻撃者は巧みに悪意あるテキストをエンコードした文字列の中に隠すことができるからです。

## ファイルエンコーディングの変換

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') )
```

## 不明なエンコーディングのファイル

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 a special range running from U+DC80 to U+DCFF. These code points will then turn back into the same bytes when the `surrogateescape` error handler is used to encode the data and write it back out.

### 3.3 参考資料

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

Marc-André Lemburg のプレゼンテーション “Writing Unicode-aware Applications in Python” の PDF スライドが <https://downloads.egenix.com/python/LSM2005-Developing-Unicode-aware-applications-in-Python.pdf> から入手可能です、そして文字エンコーディングの問題と同様にアプリケーションの国際化やローカライズについても議論されています。このスライドは Python 2.x のみをカバーしています。

[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 謝辞

このドキュメントの最初の草稿は Andrew Kuchling によって書かれました。それからさらに Alexander Belopolsky, Georg Brandl, Andrew Kuchling, Ezio Melotti らで改訂が重ねられています。

この記事中の誤りの指摘や提案を申し出てくれた以下の人々に感謝します: 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.

索引

P

Python Enhancement Proposals  
PEP 263, 8

