Unicode HOWTO

リリース *3.6.10*

Guido van Rossum
and the Python development team

12 月 24, 2019

Python Software Foundation
Email: docs@python.org

目 次

第 1 章 Unicode 入門 **3**
1.1 文字コードの歴史 . . . 3
1.2 定義 . . . 4
1.3 エンコーディング . . . 4
1.4 参考資料 . . . 6

第 2 章 Python の Unicode サポート **7**
2.1 文字列型 . . . 7
2.2 バイト列への変換 . . . 8
2.3 Python ソースコード内の Unicode リテラル . . . 9
2.4 Unicode プロパティ . . . 10
2.5 Unicode 正規表現 . . . 10
2.6 参考資料 . . . 11

第 3 章 Unicode データを読み書きする **12**
3.1 Unicode ファイル名 . . . 13
3.2 Unicode 対応のプログラムを書くための Tips . . . 14
3.2.1 ファイルエンコーディングの変換 . . . 14
3.2.2 不明なエンコーディングのファイル . . . 14
3.3 参考資料 . . . 15

第 4 章 謝辞 **16**
索引 . . . 17

リリース 1.12

この HOWTO 文書は Python の Unicode 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でくわす多くの問題について説明します。

第1章　Unicode 入門

1.1 文字コードの歴史

1968 年に American Standard Code for Information Interchange が標準化されました。これは頭文字の ASCII でよく知られています。ASCII は 0 から 127 までの、異なる文字の数値コードを定義していて、例えば、小文字の 'a' にはコード値 97 が割り当てられています。

ASCII はアメリカの開発標準だったのでアクセント無しの文字のみを定義していて、 'e' はありましたが、'' や '' はありませんでした。つまり、アクセント付きの文字を必要とする言語は ASCII できちんと表現するとができません。 (実際には英語でもアクセントが無いために起きる問題がありました、 'nave' や 'caf' のようなアクセントを含む単語や、いくつかの出版社は 'coperate' のような独自のスタイルのつづりを必要とするなど)

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"
```

これらのメッセージはアクセントを含んでいるはず (termine, paramtre, enregistrs) で、フランス語が読める人にとっては単なる間違いのように見られてしまいます。

1980 年代には、ほぼ全てのパーソナルコンピューターは 8-bit で、これは 0 から 255 までの範囲の値を保持できることを意味しました。ASCII コードは最大で 127 までだったので、あるマシンでは 128 から 255 までの値にアクセント付きの文字を割り当てていました。しかし、異なるマシンは異なる文字コードを持っていたため、ファイル交換で問題が起きるようになってきました。結局、128 から 255 まで範囲の値のセットで、よく使われるものが色々と現れました。そのうちいくつかは国際標準化機構 (International Organization for Standardization) によって定義された本物の標準になり、またいくつかはあちこちの会社で開発され、なんとか広まったものが **事実上の** 慣習となっていきました。

255 文字というのは十分多い数ではありません。例えば、西ヨーロッパで使われるアクセント付き文字とロシアで使われるキリルアルファベットの両方は 128 文字以上あるので、128–255 の間におさめることはできません。

異なる文字コードを使ってファイルを作成することは可能です (持っているロシア語のファイル全てを KOI8 と呼ばれるコーディングシステムで、持っているフランス語のファイル全てを別の Latin1 と呼ばれるコーディングシステムにすることで)、しかし、ロシア語の文章を引用するフランス語の文章を書きたい場合にはどうでしょう? 1989 年代にこの問題を解決したいという要望が上って、Unicode 標準化の努力が始まりました。

Unicord は 8-bit 文字の代わりに 16-bit 文字を使うことに取り掛かりました。16 bit ということは 2^16 = 65,536 の異なった値が使え、多くの様々なアルファベットの様々な文字を表現できるということを意味します; 最初の目標は、Unicode に人間の 1 つ 1 つの言語のアルファベットを含めることでした。しかし 16 bit で

さえ、その目標を達成するためには不十分であることが判明し、最新の Unicode 規格では 0 から 1,114,111 (16 進表記で 0x10FFFF) までのより広い範囲の文字コードを使っています。

関連する ISO 標準も ISO 10646 があります。Unicode と ISO 10646 は元々独立した成果でしたが、 Unicode の 1.1 リビジョンで仕様を併合しました。

(この Unicode の歴史についての解説は非常に単純化してあります。Unicode の上手い使い方を理解するのに歴史的な詳細を精密に知る必要はありませんが、もし興味があれば、より詳しい情報は参考文献に載せた Unicode コンソーシアムのサイトや Wikipedia の Unicode の記事 を調べてください。)

1.2 定義

文字 は文章の構成要素の中の最小のものです。'A', 'B', 'C' などは全て異なる文字です。 '' や '' も同様に異なる文字です。文字は抽象的な概念で、言語や文脈に依存してさまざまに変化します。例えば、オーム () はふつう大文字ギリシャ文字のオメガ (Ω) で書かれますが (これらはいくつかのフォントで全く同じ書体かもしれません) しかし、これらは異なる意味を持つ異なる文字とみなされます。

Unicode 標準は文字をどのように **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。コードポイントは通常 16 進数で表記される整数値です。Unicode 標準では、コードポイントは U+12CA という表記を使って書かれ、これは 0x12ca (10 進表記で 4,810) という値を持つ文字ということを意味します。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
```

厳密に言うと、この定義から「これは文字 U+12CA です」と言うのは意味の無いことだと分かります。U+12CA 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、'ETHIOPIC SYLLABLE WI' 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。

1.3 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シーケンスであり、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。このシーケンスはメモリ上ではバイト (つまり 0 から 255 までの値) の集まりとして表される必要があります。Unicode 文字列をバイトのシーケンスとして翻訳する規則を **エンコーディング (encoding)** と呼びます。

まず考え付くエンコーディングは 32-bit 整数の配列でしょう。この表現では、文字列 "Python" は次のようになるでしょう:

```
   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大域が 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。
4. 多くのインターネット標準がテキストデータとして定義されていて、それらはゼロバイトの埋め込まれた内容を扱うことができません。

一般的にこのエンコーディングは使わず、変わりにより効率的で便利な他のエンコーディングが選ばれています。 UTF-8 はたぶん最も一般的にサポートされているエンコーディングです。このエンコーディングについては後で説明します。

エンコーディングは全ての Unicode 文字を扱う必要はありませんし、多くのエンコーディングはそれをしません。Unicode 文字列を ASCII エンコーディングに変換する規則は単純です; それぞれのコードポイントに対して:

1. コードポイントは 128 より小さい場合、コードポイントと同じ値です。
2. コードポイントが 128 以上の場合、Unicode 文字列はエンコーディングで表示することができません。 (この場合 Python は `UnicodeEncodeError` 例外を送出します。)

Latin-1, ISO-8859-1 として知られるエンコーディングも同様のエンコーディングです。Unicode コードポイントの 0–255 は Latin-1 の値と等価なので、このエンコーディングの変換するには、単純にコードポイントをバイト値に変換する必要があります; もしコードポイントが 255 より大きい場合に遭遇した場合、文字列は Latin-1 にエンコードできません。

エンコーディングは Latin-1 のように単純な一対一対応を持っていません。 IBM メインフレームで使われていた IBM の EBCDIC で考えてみます。文字は一つのブロックに収められていませんでした: 'a' から 'i' は 129 から 137 までででしたが、 'j' から 'r' までは 145 から 153 までした。 EBICIC を使いたいと思ったら、おそらく変換を実行するルックアップテーブルの類を使う必要があるでしょう、これは内部の詳細のことになりますが。

UTF-8 は最もよく使われているエンコーディングの 1 つです。UTF は "Unicode Transformation Format" を表していて、'8' は 8-bit 整数がエンコーディングで使われていることを意味しています。(UTF-16 や UTF-32 というエンコーディングもありますが、それらは UTF-8 ほどには頻繁に使われません。) UTF-8 は次のようなルールを使っています:

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 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 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。

1.4 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。

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

また別の 良い入門ガイド を Joel Spolsky が書いています。この HOWTO の入門を読んでも理解が明確にならなかった場合は、続きを読む前にこの記事を読んでみるとよいです。

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

第2章　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

Python 3.0 から、言語は `str` 型を Unicode 文字から成るものとして特徴付けています、つまり `"unicode rocks!"` や `'unicode rocks!'` や三重クオート文字列の構文を使って作られた文字列は Unicode として格納されます。

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 trouv^^c3^^a9")
```

特殊な形式のコメントをソースコードの 1 行目もしくは 2 行目に配置することで、UTF-8 ではないエンコーディングを使うことができます:

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

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

```
r^^c3^^a9pertoire = "/tmp/records.log"
with open(r^^c3^^a9pertoire, "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もできます。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 3.2 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):
    ...
UnicodeEncodeError: 'ascii' codec can't encode character '\ua000' in
```

(次のページに続く)

(前のページからの続き)

```
  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 = 'abcd^^c3^^a9'
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

Unicode 仕様はコードポイントについての情報のデータベースも含んでいます。それぞれの定義されたコードポイントに対して、その情報は文字の名前、カテゴリ、適用可能であれば数値 (Unicode にはローマ数字や 1/3 や 4/5 のような分数を表す文字があります) を含んでいます。また、右から読むテキストと左から読むテキストが混在しているテキストでのそのコードポイントの読む方向や、他の表示に関連した特質もあります。

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 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'` カテゴリー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。

2.6 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-Andr 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しい資料です。

第3章　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?

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の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。

いくつかの領域では、UTF-8 でエンコードされたファイルの先頭に "BOM" を利用する習慣があります; この名前はよく誤解を招きますが、UTF-8 はバイトオーダーに依存しません。"BOM" の印は単にファイルが UTF-8 でエンコーディングされていることを知らせるものです。もし、そのようなファイルを読む場合には、この印を自動的にスキップするために 'utf-8-sig' コーデックを利用してください。

3.1 Unicode ファイル名

多くの OS では現在任意の Unicode 文字を含むファイル名をサポートしています。通常 Unicode 文字列をシステム依存のエンコーディングに変換することによって実装されています。例えば、Mac OS X は UTF-8 を利用し、Windows ではエンコーディングが設定で変更することが可能です; Windows では Python は "mbcs" という名前に現在設定されているエンコーディングを問い合わせて利用します。Unix システムでは LANG や LC_CTYPE 環境変数を設定していれば、それだけがファイルシステムのエンコーディングとなります; もしエンコーディングを設定しなければ、デフォルトエンコーディングは UTF-8 になります。

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。

os.listdir() 関数はファイル名を返しますが、ここで問題が起きます: この関数はファイル名を Unicode で返すべきでしょうか? それともエンコードされたバイト列で返すべきでしょうか? os.listdir() は、ディレクトリのパスをバイト列として渡したか、 Unicode 文字列として渡したかによって、どちらの形式でも返せます。パスを Unicode 文字列として渡した場合、ファイル名はファイルシステムエンコーディングを使ってデコードされ、 Unicode 文字列のリストが返されます。その一方、バイト列のパスを渡すとファイル名をバイト列として返します。例えば、デフォルトのファイルシステムエンコーディングが UTF-8 だと仮定して、以下のプログラムを実行すると:

```
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:

```
amk:~$ python t.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。

ほぼ全ての状況で Unicode API を利用すべきです。bytes API はシステムのデコードされていないファイル名が存在する場合、例えば Unix システム、でのみ利用すべきです。

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。

3.2.1 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') )
```

3.2.2 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 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-Andr 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.

第4章　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: ric Araujo, Nicholas Bastin, Nick Coghlan, Marius Gedminas, Kent Johnson, Ken Krugler, Marc-Andr Lemburg, Martin von Lwis, Terry J. Reedy, Chad Whitacre.

索引

Python Enhancement Proposals
    PEP 263, 10