
Unicode HOWTO

リリース 2.7.15

**Guido van Rossum
and the Python development team**

2月16, 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 2.x の Unicode サポート	7
2.1	Unicode 型	7
2.2	Python ソースコード内の Unicode リテラル	9
2.3	Unicode プロパティ	10
2.4	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
3.3	参考資料	15
第 4 章	更新履歴と謝辞	16

リリース 1.03

この HOWTO 文書は Python 2.x 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。この文書の Python 3 版は <<https://docs.python.org/3/howto/unicode.html>> をご覧ください。

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、'à' や 'é' 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ことができません。(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)

しばらくの間は単にアクセントが表示されないプログラムを書きました。1980 年半ばのフランス語で出版された Apple II の BASIC プログラムを見た記憶を辿ると、そこにはこんな行が含まれていました:

```
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むべきで、フランス語を読める人から見ると単に間違いとみなされます。

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。

Unicode は 8-bit の文字の代わりに 16-bit の文字を使うことにとりかかりました。16bit 使うということは $2^{16} = 65,536$ の異なる値が利用可能だということを意味します、これによって多くの異なるアルファベット上の多くの異なる文字を表現することができます; 最初の目標は Unicode が人間が使う個々の言語のア

ルファベットを含むことでした。あとになってこの目標を達成するには 16bit でさえも不十分であることがわかりました、そして最新の Unicode 規格は 0–1,114,111 (16 進表記で 0x10ffff) までのより広い文字コードの幅を使っています。

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います。平均的な Python プログラマは歴史的な詳細を気にする必要は無いと考えています; 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を参考にして下さい。)

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 (Ω) 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が) 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 標準は文字がコードポイント (**code points**) でどう表現するかを記述しています。コードポイントは整数値で、ふつう 16 進表記で書かれます。標準的にはコードポイントは U+12ca のような表記を使って書かれます、U+12ca は 0x12ca (10 進表記で 4810) を意味しています。Unicode 標準は文字とコードポイントを対応させる多くのテーブルを含んでいます:

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

厳密にいうとこれらの定義は「この文字は U+12ca です」ということを意味していません。U+12ca はコードポイントで、特定の文字を示しています; この場合では、’ETHIOPIC SYLLABLE WI’ を示しています。細かく気にしない文脈の中ではコードポイントと文字の区別は忘れられることがよくあります。

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 0 から 0x10ffff までの数値であるコードポイントのシーケンスで、シーケンスはメモリ上でバイト (0 から 255 までの値) の組として表現される必要があります。バイト列を Unicode 文字列に変換する規則を **エンコーディング (encoding)** と呼びます。

最初に考えるであろうエンコーディングは 32-bit 整数の配列でしょう。この表示では、”Python” という文字列はこうみえます:

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. 空間を無駄に使ってしまいます。多くの文書では、コードポイントの多くは 127 か 255 より小さいため多くの空間がゼロバイトに占有されます。上の文字列は ASCII 表示では 6 バイトを必要だったのに対して 24 バイトを必要としています。RAM の使用料の増加はたいした問題ではありませんが (デスクトップコンピュータは RAM をメガバイト単位で持っていますし、文字列はそこまで大きい容量にはなりません)、しかし、ディスクとネットワークの帯域が 4 倍増えることはとても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。例えば Python のデフォルトエンコーディングの 'ascii' エンコーディング。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 まででしたが、'j' から 'r' までは 145 から 153 まででした。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。

UTF-8 は最もよく使われるエンコーディングの一つです。UTF は "Unicode Transformation Format" からとられていて、8 はエンコーディングに 8-bit の数字を使うことを意味しています。(同じく UTF-16 エンコーディングもあります、しかしこちらは UTF-8 ほど頻繁に使われていません。) UTF-8 は以下の規則を利用します:

1. コードポイントが 128 より小さい場合、対応するバイト値で表現。
2. コードポイントは 128 から 0x7fff の間の場合、128 から 255 までの 2 バイト値に変換。

3. 0x7ff より大きいコードポイントは 3 か 4 バイト列に変換し、バイト列のそれぞれのバイトは 128 から 255 の間をとる。

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. Unicode 文字列をゼロバイトで埋めないバイト文字列に変換する。これによってバイト順の問題を解決し、UTF-8 文字列を `strcpy()` のような C 関数で処理することができ、そしてゼロバイトを扱うことができないプロトコル経由で送信することができます。
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; コードポイントの多くは 2 バイトに変換され、値が 128 より小さければ、1 バイトしか占有しません。
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 <<http://www.unicode.org>> には文字の図表や用語辞典、そして Unicode 仕様の PDF があります。読むのは簡単ではないので覚悟して下さい。 <<http://www.unicode.org/history/>> は Unicode の起源と発展の年表です。

標準についての理解を助けるために Jukka Korpela が Unicode の文字表を読むための導入ガイドを書いてあります、 <<https://www.cs.tut.fi/~jkorpela/unicode/guide.html>> から入手可能です。

もう一つのよい入門記事 <<http://www.joelonsoftware.com/articles/Unicode.html>> を Joel Spolsky が書いています。もしこの HOWTO の入門が明解に感じなかった場合には、続きを読む前にこの記事を読んでみるべきです。

Wikipedia の記事はしばしば役に立ちます; 試しに "character encoding" <http://en.wikipedia.org/wiki/Character_encoding> の記事と UTF-8 <<http://en.wikipedia.org/wiki/UTF-8>> の記事を読んでみて下さい。

第2章 Python 2.x の Unicode サポート

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

2.1 Unicode 型

Unicode 文字列は Python の組み込み型の一つ `unicode` 型のインスタンスとして表現されます。`basestring` と呼ばれる抽象クラスから派生しています、`str` 型の親戚でもあります；そのため `isinstance(value, basestring)` で文字列型かどうか調べることができます。Python 内部では Unicode 文字列は 16-bit, 32-bit 整数のどちらかで表現され、どちらが使われるかは Python インタプリタがどうコンパイルされたかに依存します。

`unicode()` コンストラクタは `unicode(string[, encoding, errors])` という用法を持っています。この引数は全て 8-bit 文字列でなければいけません。最初の引数は指定したエンコーディングを使って Unicode に変換されます；`encoding` 引数を渡さない場合、変換には ASCII エンコーディングが使われます、そのため 127 より大きい文字はエラーとして扱われます：

```
>>> unicode('abcdef')
u'abcdef'
>>> s = unicode('abcdef')
>>> type(s)
<type 'unicode'>
>>> unicode('abcdef' + chr(255))
Traceback (most recent call last):
...
UnicodeDecodeError: 'ascii' codec can't decode byte 0xff in position 6:
ordinal not in range(128)
```

`errors` 引数は入力文字列がエンコーディング規則に従って変換できないときの対応を指定します。この引数に有効な値は `'strict'` (`UnicodeDecodeError` を送出する)、`'replace'` (`U+FFFD`, `'REPLACEMENT CHARACTER'` を追加する)、または `'ignore'` (結果の Unicode 文字列から文字を除くだけ) です。以下の例で違いを示します：

```
>>> unicode('\x80abc', errors='strict')
Traceback (most recent call last):
...
UnicodeDecodeError: 'ascii' codec can't decode byte 0x80 in position 0:
ordinal not in range(128)
>>> unicode('\x80abc', errors='replace')
u'\ufffdabc'
>>> unicode('\x80abc', errors='ignore')
u'abc'
```

エンコーディングはエンコーディング名を含む文字列によって指定されます。Python 2.7 ではエンコーディングはおよそ 100 に及びます；一覧は Python ライブラリレファレンスの `standard-encodings` を参照して下さい

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', 'iso_8859_1', そして '8859' これらは全て同じエンコーディングの別称です。

Unicode 文字列の一つの文字は `unichr()` 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:

```
>>> unichr(40960)
u'\ua000'
>>> ord(u'\ua000')
40960
```

`unicode` 型のインスタンスは多くの 8-bit 文字列型と同じ検索や書式指定のためのメソッドを持っています:

```
>>> s = u'Was ever feather so lightly blown to and fro as this multitude?'
>>> s.count('e')
5
>>> s.find('feather')
9
>>> s.find('bird')
-1
>>> s.replace('feather', 'sand')
u'Was ever sand so lightly blown to and fro as this multitude?'
>>> s.upper()
u'WAS EVER FEATHER SO LIGHTLY BLOWN TO AND FRO AS THIS MULTITUDE?'
```

これらのメソッドの引数は Unicode 文字列または 8-bit 文字列が使えることに注意して下さい。8-bit 文字列は操作に使われる前に Unicode に変換されます; Python デフォルトの ASCII エンコーディングが利用されるため、127 より大きい文字列は例外を引き起します:

```
>>> s.find('Was\x9f')
Traceback (most recent call last):
...
UnicodeDecodeError: 'ascii' codec can't decode byte 0x9f in position 3:
ordinal not in range(128)
>>> s.find(u'Was\x9f')
-1
```

文字列操作を行なう多くの Python コードはコードの変更無しに Unicode 文字列を扱うことができるでしょう。(入出力に関しては Unicode のための更新が必要になります; 詳しくは後で述べます。)

別の重要なメソッドは `.encode([encoding], [errors='strict'])` があります、このメソッドは Unicode 文字列を要求したエンコーディングでエンコードされた 8-bit 文字列を返します。 `errors` パラメータは `unicode()` コンストラクタのパラメータと同様ですが、もう一つ可能性が追加されています; 同様のものとして 'strict', 'ignore', そして 'replace' があり、さらに XML 文字参照を使う 'xmlcharrefreplace' を渡すことができます:

```
>>> u = unichr(40960) + u'abcd' + unichr(1972)
>>> u.encode('utf-8')
'\xea\x80\x80abcd\xde\xb4'
>>> u.encode('ascii')
Traceback (most recent call last):
...
UnicodeEncodeError: 'ascii' codec can't encode character u'\ua000' in
position 0: ordinal not in range(128)
```

(次のページに続く)

(前のページからの続き)

```
>>> u.encode('ascii', 'ignore')
'abcd'
>>> u.encode('ascii', 'replace')
'?abcd?'
>>> u.encode('ascii', 'xmlcharrefreplace')
'&#40960;abcd&#1972;'
```

Python の 8-bit 文字列は `.decode([encoding], [errors])` メソッドを持っています、これは与えたエンコーディングを使って文字列を解釈します:

```
>>> u = unichr(40960) + u'abcd' + unichr(1972)      # Assemble a string
>>> utf8_version = u.encode('utf-8')                 # Encode as UTF-8
>>> type(utf8_version), utf8_version
(<type 'str'>, '\xea\x80\x80abcd\xde\xb4')
>>> u2 = utf8_version.decode('utf-8')                 # Decode using UTF-8
>>> u == u2                                           # The two strings match
True
```

`codecs` モジュールに利用可能なエンコーディングを登録したり、アクセスする低レベルルーチンがあります。しかし、このモジュールが返すエンコーディングとデコーディング関数はふつう低レベルすぎて快適とはいえません、そのためここで `codecs` モジュールについて述べないことにします。もし、全く新しいエンコーディングを実装する必要がある場合は、`codecs` モジュールのインターフェースについて学ぶ必要があります、しかし、エンコーディングの実装は特殊な作業なので、ここでは扱いません。このモジュールについて学ぶには Python ドキュメントを参照して下さい。

`codecs` モジュールの中で最も使われるのは `codecs.open()` 関数です、この関数は入出力の節で議題に挙げます。

2.2 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 Unicode リテラルは `'u'` または `'U'` の文字を最初に付けた文字列として書かれます: `u'abcdefghijkl'`。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。

Unicode リテラルは 8-bit 文字列と同じエスケープシーケンスを使うことができます、使えるエスケープシーケンスには `\x` も含みます、ただし `\x` は 2 桁の 16 進数しかとることができないので任意のコードポイントを表現することはできません。8 進エスケープは 8 進数の 777 を示す `U+01ff` まで使うことができます。

```
>>> s = u"a\xac\u1234\u20ac\u00008000"
... #      ^^^^ two-digit hex escape
... #      ^^^^^ four-digit Unicode escape
... #      ^^^^^^^^^ eight-digit Unicode escape
>>> for c in s: print ord(c),
...
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 `unichr()` 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ます。

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 Unicode 文字列を任意のエンコーディングで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しなければいけません。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 = 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す。Python はコメント中に `coding: name` または `coding=name` を探します。

このコメントを含まない場合には、デフォルトエンコーディングとして ASCII が利用されます。Python のバージョンが 2.4 より前の場合には Euro-centric と Latin-1 が文字列リテラルのデフォルトエンコーディングであると仮定されていました; Python 2.4 では 127 より大きい文字でも動作しますが、警告を発することになります。例えば、以下のエンコーディング宣言のないプログラムは:

```
#!/usr/bin/env python
u = u'abcd^c3^a9'
print ord(u[-1])
```

これを Python 2.4 で動作させたときには、以下の警告が出力されます:

```
amk:~$ python2.4 p263.py
sys:1: DeprecationWarning: Non-ASCII character '\xe9'
      in file p263.py on line 2, but no encoding declared;
      see https://www.python.org/peps/pep-0263.html for details
```

Python 2.5 以降ではより厳格になり、文法エラーになります:

```
amk:~$ python2.5 p263.py
File "/tmp/p263.py", line 2
SyntaxError: Non-ASCII character '\xc3' in file /tmp/p263.py
      on line 2, but no encoding declared; see
      https://www.python.org/peps/pep-0263.html for details
```

2.3 Unicode プロパティ

Unicode 仕様はコードポイントについての情報データベースを含んでいます。定義された各コードポイントに対して、情報は文字の名前、カテゴリ、適用可能ならば数値 (Unicode にはローマ数字や $1/3$ や $4/5$ のような分数などの文字があります) を含んでいます。コードポイントを左右どちらから読むのか等表示に関連したプロパティもあります。

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 = unichr(233) + unichr(0x0bf2) + unichr(3972) + unichr(6000) + unichr(13231)

for i, c in enumerate(u):
    print i, '%04x' % ord(c), unicodedata.category(c),
    print unicodedata.name(c)

# Get numeric value of second character
print unicodedata.numeric(u[1])
```

実行時には、このように印字されます:

```
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
```

カテゴリコードは文字の性質を簡単に説明するものです。カテゴリの分類は "Letter", "Number", "Punctuation" または "Symbol" で、さらにサブカテゴリに分かれます。上に出ている出力結果を例にとると 'Ll' は 'Letter, lowercase' を意味していて、'No' は "Number, other" を意味しています、'Mn' は "Mark, nonspacing" で 'So' は "Symbol, other" です。カテゴリコードの一覧は <http://www.unicode.org/reports/tr44/#General_Category_Values> を参照して下さい。

2.4 参考資料

Unicode と 8-bit 文字型については Python ライブラリレファレンスの `typeseq` に記述があります。

`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" という題のプレゼンテーションを行ないました。彼のスライドの PDF バージョンが <<https://downloads.egenix.com/python/Unicode-EPC2002-Talk.pdf>> から入手できます。これは、Python の Unicode 機能のデザインの素晴らしい概観になっています。

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 データは大抵の場合、ディスクに書き込んだりソケットを通して送られる前に特定のエンコーディングに変換されます。それらを自分自身で行なうことは可能です: ファイルを開いて、8-bit 文字列を読み、文字列を `unicode(str, encoding)` で変換します。しかし、この手動での操作は推奨できません。

問題はエンコーディングがマルチバイトであるという性質からきています; 一つの Unicode 文字は数バイトで表現されます。ファイルを任意のサイズ (1K または 4K) を単位 (chunk) として読みたい場合、読み込みの単位 (chunk) の最後にエンコーディングされた一つの Unicode 文字のバイト列の一部のみだった状況に対するエラー処理コードを書く必要がでます。一つの解決策としてはメモリ上にファイル全体を読み込んでから、デコードを実行するという方法があります、しかし巨大なファイルを扱うときに問題が起きます; 2Gb のファイルを読む場合、2Gb の RAM が必要です。(正確にいうとより多くの RAM が必要です、少なくともある時点ではエンコードする文字列と Unicode に変換した文字列の両方がメモリ上に必要とされるために)

解決策は文字コードのシーケンスが途中で切れる問題を捉える低レベルのデコーディングインターフェースを使うことです。このインターフェースの実装は既に行なわれています: `codecs` モジュールは `open()` 関数を含んでいます、この関数はファイルの内容が指定したエンコーディングであると仮定されるファイルオブジェクトを返し、`.read()` and `.write()` のようなメソッドに対して Unicode パラメータを受けつけます。

関数の引数は `open(filename, mode='rb', encoding=None, errors='strict', buffering=1)` です。mode は 'r', 'w', または 'a' が受け付けられ、通常の組み込み関数 `open()` 関数の引数と同様です; ファイルを更新するには '+' を加えます。buffering は標準の関数の引数と同様です。encoding は使うエンコーディングを文字列で与えます; もし None にした場合は 8-bit 文字列を受け付ける通常の Python のファイルオブジェクトが返されます。それ以外の引数の場合には、ラップされたオブジェクトが返され、データは必要に応じて変換されたラッパーオブジェクトから読み書きされます。errors はエンコーディングエラーに対する動作を指定します、これは例の如く 'strict', 'ignore' そして 'replace' のうちのどれかをとりまします。

そのためファイルから Unicode を読むのは単純です:

```
import codecs
f = codecs.open('unicode.rst', encoding='utf-8')
```

(次のページに続く)

(前のページからの続き)

```
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:

```
f = codecs.open('test', encoding='utf-8', mode='w+')
f.write(u'\u4500 blah blah blah\n')
f.seek(0)
print repr(f.readline()[:1])
f.close()
```

Unicode 文字 U+FEFF は byte-order-mark (BOM) として利用されます、そしてファイルのバイト順の自動判定の役立てるためにファイルの最初の文字として書かれます。いくつかのエンコーディング、たとえば UTF-16 では BOM がファイルの最初に存在することになっています; そのようなエンコーディングが利用されるときには BOM は最初の文字として自動的に書き込まれ、ファイルの読み込み時には暗黙の内に除かれます。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) とビッグエンディアン (big-endian) に対して 'utf-16-le' と 'utf-16-be' のようにエンコーディングの変種が存在します、これらは特定のバイト順を示すもので、BOM をスキップしません。

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 ASCII 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 = u'filename\u4500abc'
f = open(filename, 'w')
f.write('blah\n')
f.close()
```

`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。

ファイル名を返す `os.listdir()` は問題を引き起こします: この関数はファイル名を返すべきでしょうか、それともエンコードされた内容の 8-bit 文字列を返すべきでしょうか? `os.listdir()` は与えられたディレクトリへのパスが 8-bit 文字列か Unicode 文字列で与えたかに応じてその両方を返します。パスを Unicode 文字列で与えた場合、ファイル名はファイルシステムのエンコーディングを利用してデコードされ、Unicode 文字列のリストが返されます、8-bit パスを与えるとファイル名は 8-bit 文字列で返されます。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 と仮定される場合、以下のプログラムを実行すると:

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

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

以下の出力結果が生成されます:

```
amk:~$ python t.py
['.svn', 'filename\xe4\x94\x80abc', ...]
[u'.svn', u'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。

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:

ソフトウェア内部の動作には Unicode 文字列のみを利用し、出力時に特定のエンコーディングに変換する。

UTF-8 と 8-bit 文字列の両方を処理する関数を書こうとすると、異なる種類の文字列を結合する際にバグが生じやすいことに気づくでしょう。Python のデフォルトエンコーディングは ASCII なので、ASCII の値 127 より大きい文字が入力データにあった場合、これは ASCII エンコーディングで扱えないために、`UnicodeDecodeError` が発生します。

この問題を見逃がすのは簡単です、ソフトウェアに対してアクセントを含まないデータのみでテストを行えばよいのです; 全てはうまく動作しているように見えます、しかし実際には最初に 127 より大きい文字を試みたユーザにバグが待ち構えていることになります。第二の助言は:

テストデータには 127 より大きい文字を含み、さらに 255 より大きい文字を含むことが望ましい。

Web ブラウザからのデータやその他の信用できないところからのデータを使う場合には、コマンド行の生成やデータベースへの記録の前に不正な文字に対するチェックを行なうことが一般的です。もしコマンド行生成やデータベース記録を行なう場合には、文字列が利用または保存できる形式になっているかを一度は注意深く確かめる必要があります; 文字を偽装するためにエンコーディングを利用することは可能です。このことは入力データのエンコーディングが指定されている場合にも可能です; 多くのエンコーディングはチェック用の文字単独をそのままにしておきますが、Python は 'base64' のような単独の文字を変更するエンコーディングも含んでいます。

例えば、Unicode のファイル名を取るコンテキストマネジメントシステムがあるとし、そして '/' 文字を含むパスを拒否したいとします。するとこのコードのように書くでしょう:

```
def read_file (filename, encoding):
    if '/' in filename:
```

(次のページに続く)

(前のページからの続き)

```
raise ValueError("'" not allowed in filenames")
unicode_name = filename.decode(encoding)
f = open(unicode_name, 'r')
# ... return contents of file ...
```

しかし、攻撃者が 'base64' エンコーディングを指定できる場合、攻撃者はシステムファイルを読むために '/etc/passwd' の文字列を base-64 でエンコードした 'L2V0Yy9wYXNzd2Q=' を渡すことができます。上のコードは文字 '/' をエンコードした形式で探し、デコードした結果が危険な文字となる場合を見逃してしまいます。

3.3 参考資料

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。

第4章 更新履歴と謝辞

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: Nicholas Bastin, Marius Gedminas, Kent Johnson, Ken Krugler, Marc-Andr Lemburg, Martin von Lwis, Chad Whitacre.

Version 1.0: posted August 5 2005.

Version 1.01: posted August 7 2005. Corrects factual and markup errors; adds several links.

Version 1.02: posted August 16 2005. Corrects factual errors.

Version 1.03: posted June 20 2010. Notes that Python 3.x is not covered, and that the HOWTO only covers 2.x.