Unicode HOWTO

Release 2.7.11

Guido van Rossum
and the Python development team

December 05, 2015

Python Software Foundation
Email: docs@python.org

Contents

1 Introduction to Unicode **1**
1.1 History of Character Codes . . . 1
1.2 Definitions . . . 2
1.3 Encodings . . . 3
1.4 References . . . 4

2 Python 2.x's Unicode Support **4**
2.1 The Unicode Type . . . 4
2.2 Unicode Literals in Python Source Code . . . 6
2.3 Unicode Properties . . . 7
2.4 References . . . 8

3 Reading and Writing Unicode Data **8**
3.1 Unicode filenames . . . 9
3.2 Tips for Writing Unicode-aware Programs . . . 9
3.3 References . . . 10

4 Revision History and Acknowledgements **10**

Release 1.03

This HOWTO discusses Python 2.x's support for Unicode, and explains various problems that people commonly encounter when trying to work with Unicode. For the Python 3 version, see <https://docs.python.org/3/howto/unicode.html>.

1 Introduction to Unicode

1.1 History of Character Codes

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

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

For a while people just wrote programs that didn’t display accents. I remember looking at Apple][BASIC programs, published in French-language publications in the mid-1980s, that had lines like these:

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

Those messages should contain accents, and they just look wrong to someone who can read French.

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

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

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

Unicode started out using 16-bit characters instead of 8-bit characters. 16 bits means you have 2^16 = 65,536 distinct values available, making it possible to represent many different characters from many different alphabets; an initial goal was to have Unicode contain the alphabets for every single human language. It turns out that even 16 bits isn’t enough to meet that goal, and the modern Unicode specification uses a wider range of codes, 0-1,114,111 (0x10ffff in base-16).

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

(This discussion of Unicode’s history is highly simplified. I don’t think the average Python programmer needs to worry about the historical details; consult the Unicode consortium site listed in the References for more information.)

1.2 Definitions

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

The Unicode standard describes how characters are represented by **code points**. A code point is an integer value, usually denoted in base 16. In the standard, a code point is written using the notation U+12ca to mean the character with value 0x12ca (4810 decimal). The Unicode standard contains a lot of tables listing characters and their corresponding code points:

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

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

A character is represented on a screen or on paper by a set of graphical elements that's called a **glyph**. The glyph for an uppercase A, for example, is two diagonal strokes and a horizontal stroke, though the exact details will depend on the font being used. Most Python code doesn't need to worry about glyphs; figuring out the correct glyph to display is generally the job of a GUI toolkit or a terminal's font renderer.

1.3 Encodings

To summarize the previous section: a Unicode string is a sequence of code points, which are numbers from 0 to 0x10ffff. This sequence needs to be represented as a set of bytes (meaning, values from 0-255) in memory. The rules for translating a Unicode string into a sequence of bytes are called an **encoding**.

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

```
   P           y           t           h           o           n
0x50 00 00 00 79 00 00 00 74 00 00 00 68 00 00 00 6f 00 00 00 6e 00 00 00
   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
```

This representation is straightforward but using it presents a number of problems.

1. It's not portable; different processors order the bytes differently.
2. It's very wasteful of space. In most texts, the majority of the code points are less than 127, or less than 255, so a lot of space is occupied by zero bytes. The above string takes 24 bytes compared to the 6 bytes needed for an ASCII representation. Increased RAM usage doesn't matter too much (desktop computers have megabytes of RAM, and strings aren't usually that large), but expanding our usage of disk and network bandwidth by a factor of 4 is intolerable.
3. It's not compatible with existing C functions such as `strlen()`, so a new family of wide string functions would need to be used.
4. Many Internet standards are defined in terms of textual data, and can't handle content with embedded zero bytes.

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

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

1. If the code point is < 128, each byte is the same as the value of the code point.
2. If the code point is 128 or greater, the Unicode string can't be represented in this encoding. (Python raises a `UnicodeEncodeError` exception in this case.)

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

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

UTF-8 is one of the most commonly used encodings. UTF stands for "Unicode Transformation Format", and the '8' means that 8-bit numbers are used in the encoding. (There's also a UTF-16 encoding, but it's 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 between 128 and 0x7ff, it's turned into two byte values between 128 and 255.
3. Code points >0x7ff are turned into three- or four-byte sequences, where each byte of the sequence is between 128 and 255.

UTF-8 has several convenient properties:

1. It can handle any Unicode code point.
2. A Unicode string is turned into a string 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. A string of ASCII text is also valid UTF-8 text.
4. UTF-8 is fairly compact; the majority of code points are turned into two bytes, and values less than 128 occupy only a single byte.
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 References

The Unicode Consortium site at <http://www.unicode.org> has character charts, a glossary, and PDF versions of the Unicode specification. Be prepared for some difficult reading. <http://www.unicode.org/history/> is a chronology of the origin and development of Unicode.

To help understand the standard, Jukka Korpela has written an introductory guide to reading the Unicode character tables, available at <http://www.cs.tut.fi/~jkorpela/unicode/guide.html>.

Another good introductory article was written by Joel Spolsky <http://www.joelonsoftware.com/articles/Unicode.html>. 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" <http://en.wikipedia.org/wiki/Character_encoding> and UTF-8 <http://en.wikipedia.org/wiki/UTF-8>, for example.

2 Python 2.x's Unicode Support

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

2.1 The Unicode Type

Unicode strings are expressed as instances of the `unicode` type, one of Python's repertoire of built-in types. It derives from an abstract type called `basestring`, which is also an ancestor of the `str` type; you can therefore check if a value is a string type with `isinstance(value, basestring)`. Under the hood, Python represents Unicode strings as either 16- or 32-bit integers, depending on how the Python interpreter was compiled.

The `unicode()` constructor has the signature `unicode(string[, encoding, errors])`. All of its arguments should be 8-bit strings. The first argument is converted to Unicode using the specified encoding; if you leave off the `encoding` argument, the ASCII encoding is used for the conversion, so characters greater than 127 will be treated as errors:

```
>>> 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)
```

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' (add U+FFFD, 'REPLACEMENT CHARACTER'), or 'ignore' (just leave the character out of the Unicode result). The following examples show the differences:

```
>>> 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'
```

Encodings are specified as strings containing the encoding's name. Python 2.7 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 `unichr()` 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:

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

Instances of the `unicode` type have many of the same methods as the 8-bit string type for operations such as searching and formatting:

```
>>> 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?'
```

Note that the arguments to these methods can be Unicode strings or 8-bit strings. 8-bit strings will be converted to Unicode before carrying out the operation; Python's default ASCII encoding will be used, so characters greater than 127 will cause an exception:

```
>>> 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
```

Much Python code that operates on strings will therefore work with Unicode strings without requiring any changes to the code. (Input and output code needs more updating for Unicode; more on this later.)

Another important method is `.encode([encoding], [errors='strict'])`, which returns an 8-bit string version of the Unicode string, encoded in the requested encoding. The `errors` parameter is the same as the parameter of the `unicode()` constructor, with one additional possibility; as well as 'strict', 'ignore', and 'replace', you can also pass 'xmlcharrefreplace' which uses XML's character references. The following example shows the different results:

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

Python's 8-bit strings have a `.decode([encoding], [errors])` method that interprets the string using the given encoding:

```
>>> 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
```

The low-level routines for registering and accessing the available encodings are found in the `codecs` module. However, the encoding and decoding functions returned by this module are usually more low-level than is comfortable, so I'm not going to describe the `codecs` module here. If you need to implement a completely new encoding, you'll need to learn about the `codecs` module interfaces, but implementing encodings is a specialized task that also won't be covered here. Consult the Python documentation to learn more about this module.

The most commonly used part of the `codecs` module is the `codecs.open()` function which will be discussed in the section on input and output.

2.2 Unicode Literals in Python Source Code

In Python source code, Unicode literals are written as strings prefixed with the 'u' or 'U' character: `u'abcdefghijk'`. Specific 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 8 hex digits, not 4.

Unicode literals can also use the same escape sequences as 8-bit strings, including `\x`, but `\x` only takes two hex digits so it can't express an arbitrary code point. Octal escapes can go up to U+01ff, which is octal 777.

```
>>> 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
```

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 `unichr()` 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 Unicode literals in any encoding, but you have to 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 = 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 ASCII. Versions of Python before 2.4 were Euro-centric and assumed Latin-1 as a default encoding for string literals; in Python 2.4, characters greater than 127 still work but result in a warning. For example, the following program has no encoding declaration:

```
#!/usr/bin/env python
u = u'abcdé'
print ord(u[-1])
```

When you run it with Python 2.4, it will output the following warning:

```
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 and higher are stricter and will produce a syntax error:

```
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 Properties

The Unicode specification includes a database of information about code points. For each code point that's defined, 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.

The following program displays some information about several characters, and prints the numeric value of one particular character:

```
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])
```

When run, this prints:

```
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 <http://www.unicode.org/reports/tr44/#General_Category_Values> for a list of category codes.

2.4 References

The Unicode and 8-bit string types are described in the Python library reference at typesseq.

The documentation for the `unicodedata` module.

The documentation for the `codecs` module.

Marc-André Lemburg gave a presentation at EuroPython 2002 titled "Python and Unicode". A PDF version of his slides is available at <https://downloads.egenix.com/python/Unicode-EPC2002-Talk.pdf>, and is an excellent overview of the design of Python's Unicode features.

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 string from it, and convert the string with `unicode(str, 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, 1K or 4K), 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 2Gb file, you need 2Gb 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 `codecs` module includes a version of the `open()` function that returns 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()`.

The function's parameters are `open(filename, mode='rb', encoding=None, errors='strict', buffering=1)`. `mode` can be `'r'`, `'w'`, or `'a'`, just like the corresponding parameter to the regular built-in `open()` function; add a `'+'` to update the file. `buffering` is similarly parallel to the standard function's parameter. `encoding` is a string giving the encoding to use; if it's left as `None`, a regular Python file object that accepts 8-bit strings is returned. Otherwise, a wrapper object is returned, and data written to or read from the wrapper object will be converted as needed. `errors` specifies the action for encoding errors and can be one of the usual values of 'strict', 'ignore', and 'replace'.

Reading Unicode from a file is therefore simple:

```
import codecs
f = codecs.open('unicode.rst', encoding='utf-8')
for line in f:
    print repr(line)
```

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

```
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 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.

3.1 Unicode filenames

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 ASCII.

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

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

`os.listdir()`, which returns filenames, raises an issue: should it return the Unicode version of filenames, or should it return 8-bit strings containing the encoded versions? `os.listdir()` will do both, depending on whether you provided the directory path as an 8-bit string 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 an 8-bit path will return the 8-bit versions of the filenames. For example, assuming the default filesystem encoding is UTF-8, running the following program:

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

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

will produce the following output:

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

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

3.2 Tips for Writing Unicode-aware Programs

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

The most important tip is:

> Software should only work with Unicode strings internally, converting to a particular encoding on output.

If you attempt to write processing functions that accept both Unicode and 8-bit strings, you will find your program vulnerable to bugs wherever you combine the two different kinds of strings. Python's default encoding is ASCII, so whenever a character with an ASCII value > 127 is in the input data, you'll get a `UnicodeDecodeError` because that character can't be handled by the ASCII encoding.

It's easy to miss such problems if you only test your software with data that doesn't contain any accents; everything will seem to work, but there's actually a bug in your program waiting for the first user who attempts to use characters > 127. A second tip, therefore, is:

> Include characters > 127 and, even better, characters > 255 in your test data.

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 string once it's in the form that will be used or stored; it's possible for encodings to be used to disguise characters. This is especially true if the input data also specifies the encoding; many encodings leave the commonly checked-for characters alone, but Python includes some encodings such as `'base64'` that modify every single character.

For example, let's say you have a content management system that takes a Unicode filename, and you want to disallow paths with a '/' character. You might write this code:

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

However, if an attacker could specify the `'base64'` encoding, they could pass `'L2V0Yy9wYXNzd2Q='`, which is the base-64 encoded form of the string `'/etc/passwd'`, to read a system file. The above code looks for `'/'` characters in the encoded form and misses the dangerous character in the resulting decoded form.

3.3 References

The PDF slides for Marc-André Lemburg's presentation "Writing Unicode-aware Applications in Python" are available at <https://downloads.egenix.com/python/LSM2005-Developing-Unicode-aware-applications-in-Python.pdf> and discuss questions of character encodings as well as how to internationalize and localize an application.

4 Revision History and Acknowledgements

Thanks to the following people who have noted errors or offered suggestions on this article: Nicholas Bastin, Marius Gedminas, Kent Johnson, Ken Krugler, Marc-André Lemburg, Martin von Löwis, 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.