# How to Use UTF-8 with Python



**[ 2005-October-01 20:15 ]**

Tim Bray describes why Unicode and UTF-8 are wonderful much better than I could, so go read that for an overview of what Unicode is, and why all your programs should support it. What I'm going to tell you is how to use Unicode, and specifically UTF-8, with one of the coolest programming languages, Python, but I have also written an introduction to Using Unicode in C/C++. Python has good support for Unicode, but there are a few tricks that you need to be aware of. I spent more than a few hours learning these tricks, and I'm hoping that by reading this you won't have to. This is a very quick and dirty introduction. If you need in depth knowledge, or need to learn about Unicode in Java or Windows, see Unicode for Programmers. [*Updated 2015-09-28*: **Warning**: This should still be accurate for Python 2 (email if not), but if you are using Python 3, this is probably not correct because it handles strings differently.]

## The Basics

There are two types of strings in Python: byte strings and Unicode strings. As you may have guessed, a byte string is a sequence of bytes. When needed, Python uses your computer's default locale to convert the bytes into characters. On Mac OS X, the default locale is actually UTF-8, but everywhere else, the default is probably ASCII. This creates a byte string:

```
byteString = "hello world! (in my default locale)"
```

And this creates a Unicode string:

```
unicodeString = u"hello Unicode world!"
```

Convert a byte string into a Unicode string and back again:

```
s = "hello byte string"
u = unicode( s )
backToBytes = u.encode()
```

The previous code uses your default character set to perform the conversions. However, relying on the locale's character set is a bad idea, since your application is likely to break as soon as someone from Thailand tries to run it on their computer. In most cases it is probably better to explicitly specify the encoding of the string:

```
s = "hello normal string"
u = unicode( s, "utf-8" )
backToBytes = u.encode( "utf-8" )
```

Now, the byte string `s` will be treated as a sequence of UTF-8 bytes to create the Unicode string `u`. The next line stores the UTF-8 representation of `u` in the byte string `backToBytes`.

## Working With Unicode Strings

Thankfully, everything in Python is supposed to treat Unicode strings identically to byte strings. However, you need to be careful in your own code when testing to see if an object is a string. Do *not* do this:

```
if isinstance( s, str ): # BAD: Not true for Unicode strings!
```

Instead, use the generic string base class, `basestring`:

```
if isinstance( s, basestring ): # True for both Unicode and byte strings
```

## Reading UTF-8 Files

You can manually convert strings that you read from files, however there is an easier way:

```
import codecs
fileObj = codecs.open( "someFile", "r", "utf-8" )
u = fileObj.read() # Returns a Unicode string from the UTF-8 bytes in the file
```

The codecs module will take care of all the conversions for you. You can also open a file for writing and it will convert the Unicode strings you pass in to `write` into whatever encoding you have chosen. However, take a look at the note below about the byte-order marker (BOM).

## Working with XML and minidom

I use the minidom module for my XML needs mostly because I am familiar with it. Unfortunately, it only handles byte strings so you need to encode your Unicode strings before passing them to minidom functions. For example:

```
import xml.dom.minidom
xmlData = u"<français>Comment ça va ? Très bien ?</français>"
dom = xml.dom.minidom.parseString( xmlData )
```

The last line raises an exception: `UnicodeEncodeError: 'ascii' codec can't encode character '\ue7' in position 5: ordinal not in range(128)`. To work around this error, encode the Unicode string into the appropriate format before passing it to minidom, like this:

```
import xml.dom.minidom
xmlData = u"<français>Comment ça va ? Très bien ?</français>"
dom = xml.dom.minidom.parseString( xmlData.encode( "utf-8" ) )
```

Minidom can handle any format of byte string, such as Latin-1 or UTF-16. However, it will only work reliably if the XML document has an encoding declaration (eg. `<?xml version="1.0" encoding="Latin-1"?>`). If the encoding declaration is missing, minidom assumes that it is UTF-8. In is a good habit to include an encoding declaration on all your XML documents, in order to guarantee compatability on all systems.

When you get XML out of minidom by calling `dom.toxml()` or `dom.toprettyxml()`, minidom returns a Unicode string. You can also pass in an additional `encoding="utf-8"` parameter to get an encoded byte string, perfect for writing out to a file.

## The Byte-Order Marker (BOM)

UTF-8 files sometimes start with a byte-order marker (BOM) to indicate that they are encoded in UTF-8. This is commonly used on Windows. On Mac OS X, applications (eg. TextEdit) ignore the BOM and remove it if the file is saved again. The W3C HTML Validator warns that older applications may not be able to handle the BOM. Unicode effectively ignores the marker, so it should not matter when reading the file. You may wish to add this to the beginning of your files to determine if they are encoded in ASCII or UTF-8. The `codecs` module provides the constant for you to do this:

```
out = file( "someFile", "w" )
out.write( codecs.BOM_UTF8 )
out.write( unicodeString.encode( "utf-8" ) )
out.close()
```

You need to be careful when using the BOM and UTF-8. Frankly, I think this is a bug in Python, but what do I know. Python will decode the value of the BOM into a Unicode character, instead of ignoring it. For example (tested with Python 2.3):

```
>>> codecs.BOM_UTF16.decode( "utf16" )
u''
>>> codecs.BOM_UTF8.decode( "utf8" )
u'\ufeff'
```

For UTF-16, Python decoded the BOM into an empty string, but for UTF-8, it decoded it into a character. Why is there a difference? I think the UTF-8 decoder should do the same thing as the UTF-16 decoder and strip out the BOM. However, it doesn't, so you will probably need to detect it and remove it yourself, like this:

```
import codecs
if s.beginswith( codecs.BOM_UTF8 ):
        # The byte string s begins with the BOM: Do something.
        # For example, decode the string as UTF-8

if u[0] == unicode( codecs.BOM_UTF8, "utf8" ):
        # The unicode string begins with the BOM: Do something.
        # For example, remove the character.

# Strip the BOM from the beginning of the Unicode string, if it exists
u.lstrip( unicode( codecs.BOM_UTF8, "utf8" ) )
```

## Writing Python Scripts in Unicode

As you may have noticed from the examples on this page, you can actually write Python scripts in UTF-8. Variables must be in ASCII, but you can include Chinese comments, or Korean strings in your source files. In order for this to work correctly, Python needs to know that your script file is not ASCII. You can do this in one of two ways. First, you can place a UTF-8 byte-order marker at the beginning of your file, if your editor supports it. Secondly, you can place the following special comment in the first or second lines of your script:

```
# -*- coding: utf-8 -*-
```

Any ASCII-compatible encoding is permitted. For details, see the Defining Python Source Code Encodings specification.

## Other Resources

- Using Unicode in C/C++ by Evan Jones - A brief tour of how to use Unicode in standard C/C++ applications.
- On the Goodness of Unicode by Tim Bray - An essay about why you should support Unicode.
- The [...] Minimum Every Software Developer [...] Must Know About Unicode [...] by Joel Spolsky - Another essay about why Unicode is good, and an introduction to how it works.
- Characters vs. Bytes by Tim Bray - An introduction to the details of Unicode encoding.
- Unicode in Python by Thijs van der Vossen - Another quick and dirty introduction to Python's Unicode support.
- Python Unicode Objects by Fredrik Lundh - A collection of tips about Python's Unicode support, like using it in regular expressions.
- Unicode for Programmers by Jason Orendorff - A detailed guide to Unicode, geared towards Python, Java, and Windows programmers.
- Unicode Home Page - The official web site for the Unicode specifications.