CHAPTER 10

Unicode

If you've ever come across text in a foreign language that contains lots of question mark characters in unexpected positions or if you've written Python code that causes an exception such as the following one to be raised, then chances are you have run into a problem with character sets, encodings, and Unicode:

```
UnicodeDecodeError: 'ascii' codec can't decode byte 0xff in position 6:➥
ordinal not in range(128)
```

Many developers try to avoid getting involved with Unicode because these error messages seem obscure and difficult to fix, but the good news is that Python has great Unicode support, so with a little effort, you will be able to banish these problems from your applications entirely as well as properly support languages other than English. This chapter and the next will show you how.

Note All the libraries that come with Pylons have Unicode support, so it is always best to use Unicode in a Pylons application. The Python 3.0 language will treat all strings as Unicode by default, so Unicode support will become even more standard across all Python libraries in the future.

A Brief History

As I'm sure you are aware, computers operate on binary numbers that can be thought of as a collection of 1s and 0s. For example, the binary number 1110 represents the decimal number 14. Each 1 or 0 in the binary number is called a *bit*. A binary number made up from seven 1s and 0s is called a *7-bit* number and can represent all the decimal numbers from 0 to 127.

In the early days of computers, people wanted to be able to represent characters as well as binary numbers, and at the time, the most important characters were unaccented English letters, numbers, and punctuation, which could all be represented by a number between 0 and 127. These numbers can therefore all be stored in binary with just seven 1s and 0s (in other words, in 7 bits). The character set defined by these numbers eventually became standardized as ASCII. In the ASCII character set, *P* is represented by 80, and *y* is represented by 121. Python understands ASCII, so you can find out the codes for characters with the built-in `ord()` function like this:

```
>>> print ord("P"), ord("y"), ord("l"), ord("o"), ord("n"), ord("s") ➥
80 121 108 111 110 115
```

You can also find out a character from its ASCII representation with `chr()` like this:

```
>>> ''.join([chr(80), chr(121), chr(108), chr(111), chr(110), chr(115)]) ➥
'Pylons'
```

Computers of the day used 8-bit bytes in their calculations. These can represent the decimal numbers 0 to 255, so people quickly realized that an extra 128 characters were available. Different people assigned these extra numbers to different characters, and before long, these different collections of extra characters were also standardized into sets known as *code pages*. As an example, code page 857 is for Turkish characters, and code page 861 is for Icelandic characters. The code page system was adequate for representing most Western languages as long as you used the correct code page for the language you wanted to represent and as long as you didn't want to work on a document that contained two different languages at once.

It quickly became apparent, though, that code pages would not be suitable for representing every language. Asian languages in particular can contain many more than 256 characters, so a system was needed that represented a much wider set of characters. This is where Unicode came in.

The origins of Unicode date back to 1987 when Joe Becker, Lee Collins, and Mark Davis started investigating the practicalities of creating a universal character set. In the following year, Joe Becker published a draft proposal for an "international/multilingual text character encoding system, tentatively called Unicode." In this document, entitled "Unicode 88," he outlined a model where every script and character in modern usage could be represented in 16 bits. It soon became clear that people would also want to be able to represent scripts and characters that weren't in modern-day use, and over successive releases of the Unicode standard more scripts and characters were added until the most recent version, Unicode 5.1, was released in April 2008 with more than 100,000 characters. As you can imagine, this requires more than the 16 bits of Joe Becker's original draft of Unicode 88.

Introducing Unicode

Unicode is an industry standard allowing computers to consistently represent and manipulate text expressed in most of the world's writing systems. Unlike ASCII, where each character is represented in 7 bits, Unicode characters are represented by something called a *code point*, which is effectively an abstract integer ID for that character. For example, the characters in *Pylons* could be represented by the Unicode code points `U+0050`, `U+0079`, `U+006C`, `U+006F`, `U+006E`, and `U+0073`.

If you've worked with hexadecimal numbers, you might notice that the last two characters of each code point correspond to the decimal representation of the corresponding character in the ASCII character set for the word *Pylons* shown in the previous paragraph. This is because the first 256 Unicode code points were made identical to the numbers representing the characters in the ISO 8859-1 encoding of the Latin alphabet (less formally known as Latin-1). This in turn shares the first 128 characters and their respective codes with the ASCII character set. I'll return to the significance of this backward compatibility in a moment.

Unicode also has the concept of an *encoding*. One way of encoding Unicode code points into binary numbers on a disk would be to store each code point as a 32-bit (4-byte) number (since a 32-bit number is more than capable of storing every possible Unicode code point). This might seem sensible at first, but representing Unicode code points on disk in this way would take up a lot of space, especially if you used only those characters with low code points such as those also represented in ISO 8859-1 and ASCII because each character would be using 4 bytes when it really needed only one.

Another way of storing the values would be to use a variable number of bytes for each character. Those with low code points such as the unaccented English characters could be stored in 1 byte, and those with much higher code points such as Arabic or Chinese characters would use more than 1 byte. This would mean that all the Unicode characters could be represented if necessary, but the most commonly used ones (the unaccented English characters) could be represented in just 1 byte. This is exactly what happens in the UTF-8 encoding, which you will probably have come across.

UTF-8 is one of the most popular encodings for Python programmers, so much so that Python 3.0 will assume that files you open are encoded in UTF-8 unless you say otherwise.

Encoding Unicode characters with a variable number of bytes for each character as UTF-8 has an interesting side effect. It means that UTF-8 encoded Unicode for the characters represented by the ASCII character set has the same binary representation as ASCII itself. This means computers can treat UTF-8 encoded Unicode as ASCII without any errors being raised as long as characters used are in the first 128 Unicode code points. This explains why your application might already be working perfectly well with certain Unicode strings even though you haven't made a special effort to work with any character set except ASCII. This is also why as soon as a character such as `£` or `é` is entered, the application will break because these are not ASCII characters; therefore, treating their UTF-8 encoded versions as ASCII will cause the kind of `UnicodeDecodeError` shown at the start of the chapter.

Luckily, working properly with Unicode is very straightforward, so you shouldn't need to rely on the backward compatibility of the UTF-8 encoding for the ASCII characters.

Before you look at Unicode in Python, I'll recap the important points:

- Unicode can represent pretty much any character in any writing system in widespread use today as well as some historical characters.
- Unicode uses code points to represent characters, and the way these map to bits on disk depends on the encoding.
- The most popular encoding is UTF-8, which has several convenient properties:
  - It can handle any Unicode code point.
  - A string of ASCII text is also valid UTF-8 encoded Unicode.
  - UTF-8 doesn't use much storage space; the majority of code points are turned into 2 bytes, and values less than 128 occupy only 1 byte.

Unicode in Python 2

In Python 2, Unicode strings are expressed as instances of the built-in `unicode` type. Under the hood, Python represents Unicode strings as either 16-bit or 32-bit integers, depending on the way the Python interpreter was compiled. Python 3 will treat all strings as Unicode automatically, but the discussion in this chapter relates only to the Unicode handling of recent Python 2 releases such as Python 2.4, 2.5, and 2.6.

Unicode Literals

In Python source code, Unicode literals are written as strings prefixed with the `u` or `U` character (although you will hardly ever see the uppercase version used).

```
>>> u'abcd'
>>> U'efgh'
```

You can also use `"`, `"""`, or `'''` versions too. For example:

```
>>> u"""This
... is a multiline
... Unicode string"""
```

Individual code points can be written using the escape sequence `\u` followed by four hex digits specifying the code point. You can also use `\U` followed by eight hex digits instead of four. Unicode literals can also use the same escape sequences as 8-bit strings including `\x`, but this takes only two hex digits, so it can't express many of the available code points. You can add characters to Unicode strings using the `unichr()` built-in function, and you can find out what the ordinal is with `ord()`, which you also saw earlier in the chapter when it was used with ASCII characters.

Here is an example demonstrating the different alternatives:

```
>>> s = u"\x66\u0072\u0061\U0000006e" + unichr(231) + u"ais"
>>> for c in s:
...     print ord(c),
...
97 102 114 97 110 231 97 105 115
```

Here `\x66` is a two-digit hex escape, `\u0072` and `\u0061` are four-digit Unicode escapes, and `\u0000006e` is an eight-digit Unicode escape. The example also demonstrates the use of `unichr()`. The word made in this is as follows:

```
>>> print s
français
```

■Note If you are working with Unicode in detail, you might be interested in the `unicodedata` module, which can be used to find out Unicode properties such as a character's name, category, numeric value, and the like.

Handling Errors

Now that you have seen how to write Unicode literals, let's look at how you can create Unicode strings with the `unicode()` constructor. Here is an example:

```
>>> cost = unicode('50.00')
>>> cost
u'50.00'
>>> type(cost)
<type 'unicode'>
```

Let's see what happens if you try to concatenate the `cost` Unicode string with a normal ASCII string:

```
>>> '$' + cost
u'$50.00'
```

Python decodes the string `'$'` from ASCII to Unicode, concatenates the two Unicode strings, and returns the result.

Now let's try to use a `£` sign instead of a `$`. The `£` character is not an ASCII character, so you have to represent it by its ordinal, which is 163. Let's see what happens:

```
>>> chr(163) + u'50.00'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'ascii' codec can't decode byte 0xa3 in ➥
position 0: ordinal not in range(128)
```

In this case, because `£` is not an ASCII character, when Python internally calls `unicode(chr(163))` to try to decode it from ASCII to Unicode, an error occurs.

Python's `unicode()` constructor takes three arguments, including an `errors` argument that determines what should happen in a situation like this:

`string`: This is the Python string to decode to Unicode.

`encoding`: This is an optional encoding to specify how the string is currently encoded. If you don't specify an encoding, ASCII will be used, so characters with code points greater than 127 will be treated as errors.

`errors`: This specifies how to handle any errors. This can be one of the following: the string `"strict"` (the default), which results in a `ValueError` being raised if an invalid character is found; `"ignore"`, which simply results in any errors being silently ignored; or `"replace"`, which causes the official Unicode replacement character, `U+FFFD` (usually displayed �), to be inserted instead.

Let's explore what happens if you perform the conversion explicitly and use the `errors` option:

```
>>> unicode(chr(163), errors='ignore') + u'50.00'
u'50.00'
>>> unicode(chr(163), errors='replace') + u'50.00'
u'\ufffd50.00'
```

As you can see, using `'ignore'` silently ignores the problem, and using `'replace'` results in the Unicode character `U+FFFD` being inserted in place of the pound sign. Neither of these is quite what you want. The solution to the problem lies in understanding the `encoding` option. Let's look at encoding and decoding Unicode data next.

Decoding Unicode

Unicode strings are simply a series of Unicode code points. When you are converting an ASCII or UTF-8 string to Unicode, you are actually *decoding* it; when you are converting from Unicode to UTF-8 or ASCII, you are *encoding* it. This is why the error in the example said that the ASCII codec could not *decode* the byte `0xa3` from ASCII to Unicode. You might be used to thinking of ASCII as the "natural" representation of characters and anything else to be an encoding, but this is not the way you should think with Unicode. You should always think of the Unicode code point as the "natural" representation and anything else as being a particular encoding.

The `0xa3` characters that appeared in the `UnicodeDecodeError` message are hex for 163, which represents the `£` sign. The error occurred because this is outside the ASCII range. However, this character is present in the ISO 8859-1 character set. If you tell Python that the data it is decoding is encoded with the `'iso_8859_1'` character set, you get the result you expected:

```
>>> unicode(chr(163), encoding='iso_8859_1') + cost
u'\xa350.00'
```

Notice that because 163 can be represented in just two hex digits, Python chose to use the `\x` representation rather than its `\u` or `\U` representation of Unicode characters.

Let's print the result:

```
>>> print u'\xa350.00'
£50.00
```

Be aware that not all terminals will be able to display all Unicode characters when printed like this; it will depend on the encoding of the terminal and the fonts available on the system.

Encoding Unicode

Now that you've seen how to decode to Unicode, let's see how to encode it. All Python Unicode objects have an `encode()` method that takes the encoding you want to use as its argument. It is used like this:

```
>>> u'$50.00'.encode('utf-8')
'$50.00'
>>> u'$50.00'.encode('ascii')
'$50.00'
```

As you can see, `u'$50.00'`, when encoded to UTF-8, is the same as the ASCII representation. The same cannot be said for `u'£50.00'` because this isn't an ASCII character, as I've already explained:

```
>>> u'\xa350.00'.encode('utf-8')
'\xc2\xa350.00'
>>> print '\xc2\xa350.00'
£50.00
>>> u'\xa350.00'.encode('ascii')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode character u'\xa3' in ➥
position 0: ordinal not in range(128)
```

Once again, you get the familiar `UnicodeEncodeError`, this time specifying that the encoding failed as you would expect.

■ **Note** Python supports many more character encodings besides the ones mentioned in this chapter; you can find the full list at `http://docs.python.org/lib/standard-encodings.html`.

Python Source Code Encoding

If you are working with non-ASCII characters in your application, you are likely to also want to be able to use them in your Python source code. Although you could manually escape each character you use in a Unicode literal, Python 2.4 and newer let you define the encoding you are using at the top of the source file like this:

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

This special setting tells Python to treat the source code as UTF-8 encoded Unicode. This allows you to use Unicode characters in the source code itself as long as you remember to set your editor to save the file in UTF-8. Windows users who use the SciTE editor can specify the encoding of their file from the menu using the File ➤ Encoding menu option. Vim users can set the encoding with `set encoding=utf8`.

If you use a non-ASCII character, which is still part of the ISO 8859-1 character set, in your source file (such as the £ character) but fail to specify an encoding, versions of Python newer than 2.4 will assume that you are using the ISO 8859-1 character set but will still issue a warning:

```
sys:1: DeprecationWarning: Non-ASCII character '\xe9' in file testas.py on line
2, but no encoding declared; see http://www.python.org/peps/pep-0263.html for de
tails
```

You can correct this by specifying the correct encoding at the top of your source file. If you use other characters but fail to specify an encoding or if you forget to save the file in the encoding you have specified, Python will give an error.

If you look back at Chapter 8, you'll see that the base template you created for the SimpleSite tutorial starts with this line:

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

When Mako creates a Python version of the template in the Mako cache, the ## characters get converted to a single # character. This results in the Python file having the correct encoding definition at the top, which allows you to use Unicode characters within the template source file as long as your editor encodes the file to UTF-8 when you saves it.

Unicode and Files

To write Unicode data to a file, you will need to encode it first. Likewise, when reading encoded Unicode from a file, it will need to be decoded. The easiest way to handle this in Python is to use the `codecs` module. Here is an example of how to read Unicode from a UTF-8 encoded file:

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

Each line will have been automatically decoded to a Unicode string. Here's an example of writing Unicode to a file encoded in ISO 8859-1:

```
f = codecs.open('unicode.txt', encoding='latin-1', mode='w')
f.write(u"\x66\u0072\u0061\U0000006e" + unichr(231) + u"ais")
f.close()
```

I've used `latin-1` here to demonstrate that Python will accept a number of different descriptions if an encoding has multiple names in common use. Reading/writing files in different encodings is almost as easy as normal Python file operations.

It is also possible to use Unicode strings as file names if the underlying filesystem supports Unicode file names. For example:

```
filename = u"\x66\u0072\u0061\U0000006e" + unichr(231) + u"ais"
f = open(filename, 'w')
f.write('Bonjour!\n')
f.close()
```

Other functions such as `os.listdir()` will return Unicode if you pass them a Unicode argument and will try to return strings if you pass an ordinary 8-bit string. Add the previous code to a file called `test.py`, and then add the following afterward:

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

If you ran `python test.py`, you would see the following output:

```
['Fran\xcc\xa7ais', 'test.py']
[u'Fran\u0327ais', u'test.py']
```

As you can see from the second line, when `os.listdir()` is given a Unicode argument, it returns Unicode strings.

Unicode Considerations in Pylons Programming

There are three main rules when dealing with using Unicode in a Pylons application:

> *The main rule is that your application should use Unicode for all strings internally, decoding any input to Unicode as soon as it enters the application and encoding the Unicode to UTF-8 or another encoding only on output.* If you perform all the decoding right at the edge of your application, as soon as it is passed any encoded Unicode data, then it will be obvious where any problems are caused. If you fail to do this and some of the data your application receives is badly encoded, it is possible your application will crash in an obscure place or, worse, that the badly encoded data poses a security risk.
>
> *The second rule is to always test your application with characters greater than 127 wherever possible.* If you fail to do this, you might think your application is working fine, but as soon as your users do put in non-ASCII characters, you will have problems. Using Arabic is always a good test, and `http://www.google.ae` is a good source of sample text.
>
> *The third rule is to always do any checking of a string for illegal characters once it's in the form that will be used or stored; otherwise, the illegal characters might be disguised.*

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

```
# DO NOT DO THIS
def read_file(filename):
    if '/' in filename:
        raise ValueError("'/' not allowed in filenames")
    unicode_name = filename.decode('base64')
    f = open(unicode_name, 'r')
    # ... return contents of file ...
```

This is incorrect because the check was performed before the actual data to be used was decoded. An attacker could have passed the data `L2V0Yy9wYXNzd2Q=`, which is the base-64 encoded form of the string `'/etc/passwd'`. The previous code would have resulted in this file being opened and returned to the browser, which wasn't what you expected. Instead, decode the data first and then perform the check. Although this is obvious advice when using the base-64 encoding where the encoded version looks very different from the original, it is less obvious when using UTF-8 where you could easily forget you are not using a decoded string.

Those are the three basic rules, so now I will cover some of the places you might want to perform Unicode decoding in a Pylons application.

Request Parameters

Pylons automatically decodes incoming form parameters into Unicode objects so that when you access `request.POST`, `request.GET`, or `request.params` in your application, the values are already Unicode strings. Only parameter values (not their associated names) are decoded to Unicode by default. Since parameter names commonly map directly to Python variable names (which are restricted to the ASCII character set), it's usually preferable to handle them as strings.

You can change the encoding used to decode the request information by setting the `request.charset` attribute.

Templating

Pylons uses Mako as its default templating language. Mako handles all content as Unicode internally. It only deals in raw strings upon the final rendering of the template just before it returns a value from the `render()` function. The encoding of the rendered string can be configured; Pylons sets the default value to UTF-8. To change this value, edit your project's `config/environment.py` file, and update the `output_encoding` argument to `TemplateLookup`:

```
# Create the Mako TemplateLookup, with the default auto-escaping
config['pylons.app_globals'].mako_lookup = TemplateLookup(
    directories=paths['templates'],
    module_directory=os.path.join(app_conf['cache_dir'], 'templates'),
    input_encoding='utf-8', output_encoding='utf-8',
    imports=['from webhelpers.html import escape'],
    default_filters=['escape'])
```

The `input_encoding` argument specifies the encoding that Mako expects the templates to have if they don't have an explicit `## -*- coding: utf-8 -*-` comment at the top of the file. You can find more information about Unicode in Mako at `http://www.makotemplates.org/docs/unicode.html`.

Output Encoding

Web pages should always be generated with a specific encoding, most likely UTF-8. At the very least, that means you should specify the following in the `<head>` section of your HTML:

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
```

You can specify the character set for the HTTP response from within a controller action using the Pylons `response` global:

```
response.charset = 'utf8'
```

This will automatically add the character set to the end of the `Content-type` header, and most browsers will trust this over the value in the `<meta>` tag. When you return a Unicode string from the controller action, it will be encoded using the character specified by `response.charset`, but if you return a non-Unicode string, it will be passed directly to the browser without being encoded again because it is assumed you have already encoded it.

The web browser will usually submit form data back to the server using the same character set as that used in the page containing the form. You should therefore try to make sure you are using the same character set in `request.charset` and `response.charset`. The defaults of UTF-8 are a good choice, though.

Databases

Another place where you will have to think about encoding and decoding is the database. You should encode to whichever encoding the database expects immediately before executing a query and decode to Unicode immediately after receiving results from the database.

SQLAlchemy has a `Unicode` column type that you can use to store Unicode characters. If you use this column type, SQLAlchemy will be responsible for handling the encoding and decoding for you so that you don't need to worry about it yourself. If you look back at the model for the SimpleSite application, you'll see that you have already been using `Unicode` columns. This is good practice because you never know when a user of your application might place a non-ASCII character in a form field, and it is best to be able to handle that situation.

A Complete Request Cycle

Now that you've seen the various places your Pylons applications might have to deal with Unicode, I'll take you through an example request cycle and explain exactly what happens in terms of encoding and decoding Unicode.

Start the SimpleSite application, and edit a page by visiting `http://localhost:5000/page/view/1`. Try copying and pasting some Chinese or Arabic text into the content field. (Just search Google for the words *Arabic* or *Chinese characters*, and some of the results are bound to contain suitable sample text.) When you save the page, the text will be sent to your application as UTF-8 (since this is the encoding of the page). Pylons will then receive the request, and the form fields will be decoded to Unicode in the Pylons `request` global. The code within the `view()` action then retrieves the value of the content field from `request.params` where it has already been decoded to a Unicode string. It then sets the `.content` attribute of the `page` object using the Unicode value. When the session is committed, the page object is automatically flushed. SQLAlchemy takes over and performs the necessary encoding before sending the content to the underlying database engine.

When the saved page is redisplayed, SQLAlchemy will issue a query to fetch the content and decode the results it fetches to Unicode. The page's content data is passed as a Unicode string to a Mako template where it is rendered. The `render()` function will obtain the result from Mako and return the entire template as a UTF-8 encoded string, which is then returned from the `view()` action. Pylons assembles the response using the UTF-8 encoded response from the action and any settings in the `response` global. Because `response.charset` is set to `'utf-8'`, Pylons adds the following header to the response:

```
Content-Type: text/html; charset=utf-8
```

Pylons then returns the response to the browser. The browser knows to expect UTF-8 because of the `charset=utf=8` part of the previous header and decodes the content that follows to its Unicode representation so that it can correctly display the text.

Summary

You should now understand the history of Unicode, how to use it in Python, and where to apply Unicode encoding and decoding in a Pylons application. You should also be able to use Unicode in your web app; remember that the main rule is to use UTF-8 to talk to the world, performing the necessary encoding and decoding at the very edge of your application (or letting Pylons do it for you!).

Now that you know how to handle multiple different characters and scripts, it is time to turn your attention to how to write a Pylons application that is designed to be able to be used by people from different countries at the same time, customizing the language used on each request for each user. You'll learn this in the next chapter.