Chapter 1

Appendix: Text Processing with Unicode

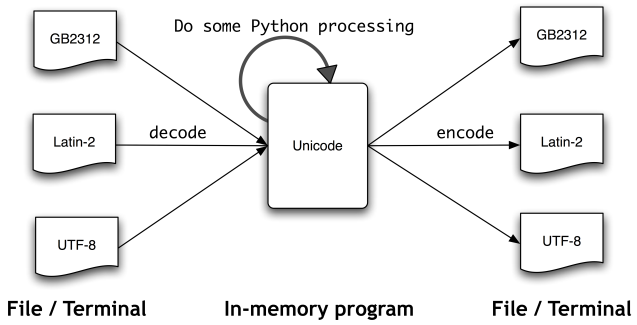
Our programs will often need to deal with different languages, and different character sets. The concept of "plain text" is a fiction. If you live in the English-speaking world you probably use ASCII, possibly without realizing it. If you live in Europe you might use one of the extended Latin character sets, containing such characters as "ø" for Danish and Norwegian, "ő" for Hungarian, "ñ" for Spanish and Breton, and "ň" for Czech and Slovak. In this section, we will give an overview of how to use Unicode for processing texts that use non-ASCII character sets.

1.1 What is Unicode?

Unicode supports over a million characters. Each of these characters is assigned a number, called a **code point**. In Python, code points are written in the form `\u`*XXXX*, where *XXXX* is the number in 4-digit hexadecimal form.

Within a program, Unicode code points can be manipulated directly, but when Unicode characters are stored in files or displayed on a terminal they must be encoded as one or more bytes. Some encodings (such as ASCII and Latin-2) use a single byte, so they can only support a small subset of Unicode, suited to a single language. Other encodings (such as UTF-8) use multiple bytes and can represent the full range of Unicode.

Text in files will be in a particular encoding, so we need some mechanism for translating it into Unicode — translation into Unicode is called **decoding**. Conversely, to write out Unicode to a file or a terminal, we first need to translate it into a suitable encoding — this translation out of Unicode is called **encoding**. The following diagram illustrates.



From a Unicode perspective, characters are abstract entities which can be realized as one or more **glyphs**. Only glyphs can appear on a screen or be printed on paper. A font is a mapping from characters to glyphs.

1.2 Extracting encoded text from files

Let's assume that we have a small text file, and that we know how it is encoded. For example, `polish-lat2.txt`, as the name suggests, is a snippet of Polish text (from the Polish Wikipedia; see http://pl.wikipedia.org/wiki/Biblioteka_Pruska), encoded as Latin-2, also known as ISO-8859-2. The function `nltk.data.find()` locates the file for us.

```
>>> import nltk.data
>>> path = nltk.data.find('samples/polish-lat2.txt')
```

The Python `codecs` module provides functions to read encoded data into Unicode strings, and to write out Unicode strings in encoded form. The `codecs.open()` function takes an encoding parameter to specify the encoding of the file being read or written. So let's import the `codecs` module, and call it with the encoding `'latin2'` to open our Polish file as Unicode.

```
>>> import codecs
>>> f = codecs.open(path, encoding='latin2')
```

For a list of encoding parameters allowed by `codecs`, see http://docs.python.org/lib/standard-encodings.html.

Text read from the file object `f` will be returned in Unicode. As we pointed out earlier, in order to view this text on a terminal, we need to encode it, using a suitable encoding. The Python-specific encoding `unicode_escape` is a dummy encoding that converts all non-ASCII characters into their `\u`*XXXX* representations. Code points above the ASCII 0-127 range but below 256 are represented in the two-digit form `\x`*XX*.

```
>>> lines = f.readlines()
>>> for l in lines:
...     l = l[:-1]
...     uni = l.encode('unicode_escape')
...     print uni
Pruska Biblioteka Pa\u0144stwowa. Jej dawne zbiory znane pod nazw\u0105
"Berlinka" to skarb kultury i sztuki niemieckiej. Przewiezione przez
Niemc\xf3w pod koniec II wojny \u015bwiatowej na Dolny \u015al\u0105sk, zosta\u0142
odnalezione po 1945 r. na terytorium Polski. Trafi\u0142y do Biblioteki
Jagiello\u0144skiej w Krakowie, obejmuj\u0105 ponad 500 tys. zabytkowych
archiwali\xf3w, m.in. manuskrypty Goethego, Mozarta, Beethovena, Bacha.
```

The first line above illustrates a Unicode escape string, namely preceded by the `\u` escape string, namely `\u0144` . The relevant Unicode character will be dislayed on the screen as the glyph ń. In the third line of the preceding example, we see `\xf3`, which corresponds to the glyph ó, and is within the 128-255 range.

In Python, a Unicode string literal can be specified by preceding an ordinary string literal with a `u`, as in `u'hello'`. Arbitrary Unicode characters are defined using the `\u`*XXXX* escape sequence inside a Unicode string literal. We find the integer ordinal of a character using `ord()`. For example:

```
>>> ord('a')
97
```

The hexadecimal 4 digit notation for 97 is 0061, so we can define a Unicode string literal with the appropriate escape sequence:

```
>>> a = u'\u0061'
>>> a
u'a'
>>> print a
a
```

Notice that the Python print statement is assuming a default encoding of the Unicode character, namely ASCII. However, ń is outside the ASCII range, so cannot be printed unless we specify an encoding. In the following example, we have specified that print should use the repr() of the string, which outputs the UTF-8 escape sequences (of the form \x*XX*) rather than trying to render the glyphs.

```
>>> nacute = u'\u0144'
>>> nacute
u'\u0144'
>>> nacute_utf = nacute.encode('utf8')
>>> print repr(nacute_utf)
'\xc5\x84'
```

If your operating system and locale are set up to render UTF-8 encoded characters, you ought to be able to give the Python command

```
print nacute_utf
```

and see ń on your screen.

Note

There are many factors determining what glyphs are rendered on your screen. If you are sure that you have the correct encoding, but your Python code is still failing to produce the glyphs you expected, you should also check that you have the necessary fonts installed on your system.

The module unicodedata lets us inspect the properties of Unicode characters. In the following example, we select all characters in the third line of our Polish text outside the ASCII range and print their UTF-8 escaped value, followed by their code point integer using the standard Unicode convention (i.e., prefixing the hex digits with U+), followed by their Unicode name.

```
>>> import unicodedata
>>> line = lines[2]
>>> print line.encode('unicode_escape')
Niemc\xf3w pod koniec II wojny \u015bwiatowej na Dolny \u015al\u0105sk, zosta\u0142y\n
>>> for c in line:
...     if ord(c) > 127:
...         print '%r U+%04x %s' % (c.encode('utf8'), ord(c), unicodedata.name(c))
'\xc3\xb3' U+00f3 LATIN SMALL LETTER O WITH ACUTE
'\xc5\x9b' U+015b LATIN SMALL LETTER S WITH ACUTE
'\xc5\x9a' U+015a LATIN CAPITAL LETTER S WITH ACUTE
'\xc4\x85' U+0105 LATIN SMALL LETTER A WITH OGONEK
'\xc5\x82' U+0142 LATIN SMALL LETTER L WITH STROKE
```

If you replace the %r (which yields the repr() value) by %s in the format string of the code sample above, and if your system supports UTF-8, you should see an output like the following:

ó U+00f3 LATIN SMALL LETTER O WITH ACUTE
ś U+015b LATIN SMALL LETTER S WITH ACUTE
Ś U+015a LATIN CAPITAL LETTER S WITH ACUTE
ą U+0105 LATIN SMALL LETTER A WITH OGONEK
ł U+0142 LATIN SMALL LETTER L WITH STROKE

Alternatively, you may need to replace the encoding `'utf8'` in the example by `'latin2'`, again depending on the details of your system.

The next examples illustrate how Python string methods and the `re` module accept Unicode strings.

```
>>> line.find(u'zosta\u0142y')
54
>>> line = line.lower()
>>> print line.encode('unicode_escape')
niemc\xf3w pod koniec ii wojny \u015bwiatowej na dolny \u015bl\u0105sk, zosta\u0142
>>> import re
>>> m = re.search(u'\u015b\w*', line)
>>> m.group()
u'\u015bwiatowej'
```

The NLTK `tokenizer` module allows Unicode strings as input, and correspondingly yields Unicode strings as output.

```
>>> from nltk.tokenize import WordTokenizer
>>> tokenizer = WordTokenizer()
>>> tokenizer.tokenize(line)
[u'niemc\xf3w', u'pod', u'koniec', u'ii', u'wojny', u'\u015bwiatowej',
u'na', u'dolny', u'\u015bl\u0105sk', u'zosta\u0142y']
```

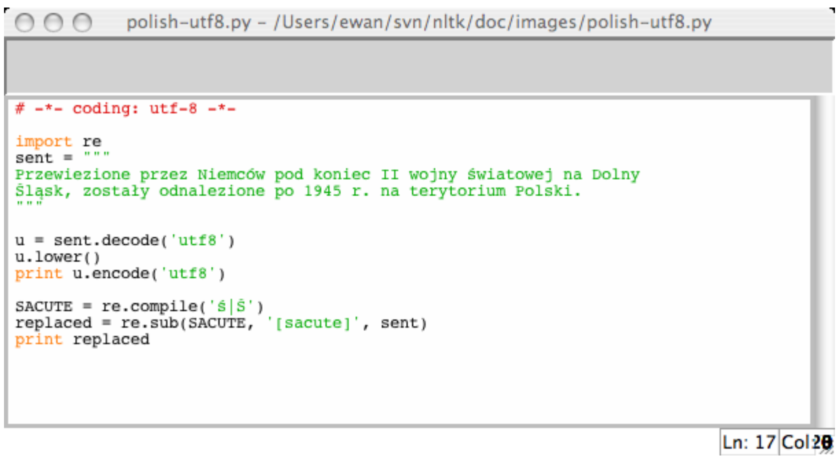
1.3 Using your local encoding in Python

If you are used to working with characters in a particular local encoding, you probably want to be able to use your standard methods for inputting and editing strings in a Python file. In order to do this, you need to include the string `'# -*- coding: <coding> -*-'` as the first or second line of your file. Note that <*coding*> has to be a string like `'latin-1'`, `'big5'` or `'utf-8'`.

Note

If you are using Emacs as your editor, the coding specification will also be interpreted as a specification of the editor's coding for the file. Not all of the valid Python names for codings are accepted by Emacs.

The following screenshot illustrates the use of UTF-8 encoded string literals within the IDLE editor:


polish-utf8.py - /Users/ewan/svn/nltk/doc/images/polish-utf8.py
Ln: 17 Col: 20

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

import re
sent = """
Przewiezione przez Niemców pod koniec II wojny światowej na Dolny
Śląsk, zostały odnalezione po 1945 r. na terytorium Polski.
"""

u = sent.decode('utf8')
u.lower()
print u.encode('utf8')

SACUTE = re.compile('ś|Ś')
replaced = re.sub(SACUTE, '[sacute]', sent)
print replaced
```

Note

The above example requires that an appropriate font is set in IDLE's preferences. In this case, we chose Courier CE.

The above example also illustrates how regular expressions can use encoded strings.

1.3.1 Further Reading

There are a number of online discussions of Unicode in general, and of Python facilities for handling Unicode. The following are worth consulting:

- Jason Orendorff, *Unicode for Programmers*, http://www.jorendorff.com/articles/unicode/.
- A. M. Kuchling, *Unicode HOWTO*, http://www.amk.ca/python/howto/unicode
- Frederik Lundh, *Python Unicode Objects*, http://effbot.org/zone/unicode-objects.htm
- Joel Spolsky, *The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)*, http://www.joelonsoftware.com/articles/Unicode.html