LING115 Lecture Note
Session: Unicode

1. Character encoding[1]

A text is a string of symbols or characters. The letters, digits, and control characters that we type and see on the screen are all characters by themselves. The set of characters that we can type and visualize is called a *character set*.

The computers recognize and process each character as a string of binary digits – 0s and 1s. Each character we type is mapped to a number, or more precisely a binary digit string representing that number. The computer programs that we use map that number to a proper visual representation for us to see. For example, the lower-case `a` is normally mapped to 97 or a binary digit string representing 97 (e.g. 01100001), so that the computer can understand it. The number 97 in turn is mapped to `a` by computer programs like Firefox so that we can understand it. This mapping between characters and numbers is called a *character encoding*. The number that a character is mapped to is called a *code point*.

If you think about it, the mapping is completely arbitrary. After all, why should `a` be mapped to 97? Why not 162? It is really a matter of what people agree on. People don't always agree so there are different character encodings that people use. This can be a problem. You may not be able to properly read a text file that I have created unless we both use the same character encoding.

Initially, people used the ASCII code: a character encoding that involves a set of 128 characters consisting of printing characters which label typical English QWERTY keyboards and control characters: see http://www.robelle.com/library/smugbook/ascii.txt . Most computer programs consider this to be the default character encoding so we don't always have to worry about character encoding.

Obviously, the ASCII code did not have all the necessary characters for some writing systems. So people extended the ASCII code and came up with various other character encodings. Many of those encodings are tailored for specific writing systems: a character encoding for the Korean writing system, a character for the Japanese writing system, etc. One notable exception is the Unicode system, which aims to cover the characters of every writing system used on the planet.

2. Unicode

Not surprisingly, the Unicode system covers a very large set of characters: the latest version covers more than 100,000 characters and has the potential to cover over one million characters. While the Unicode system has a standard character set, there are different Unicode character

---

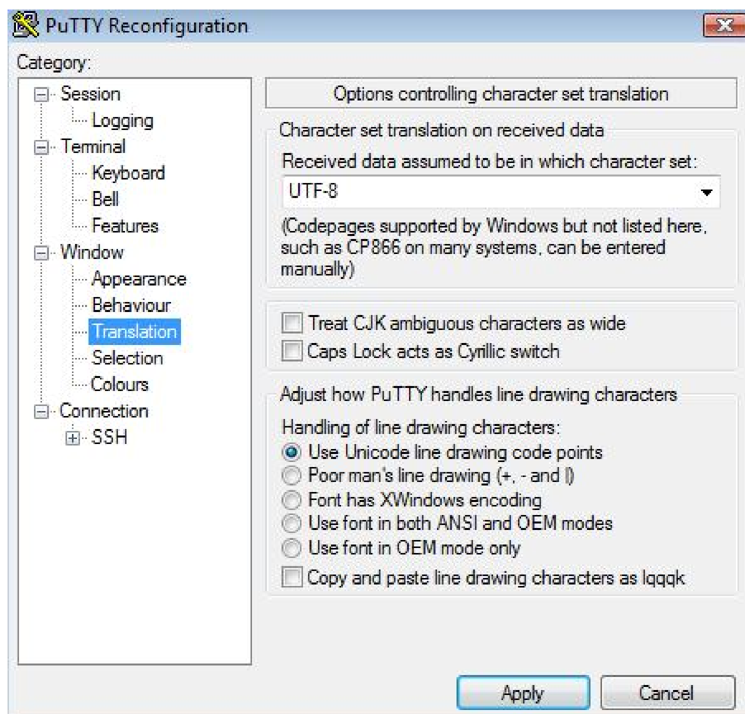
[1] Joel Spolsky has a very good section on character encodings at http://www.joelonsoftware.com/articles/Unicode.html . I suggest you read it. The take-home message is that you need to know what the character encoding of a text file is before you try to make any sense of it.

encodings: UTF-8, UTF-16, etc. So again, the same character may be represented by different code-points in UTF-8 than in UTF-16, for example. In this class, we use UTF-8 (8-bit Unicode Transformation Format). It is the most widely used among Unicode character encodings. But more importantly, it is backward-compatible with ASCII: we can still read text encoded in ASCII even if we choose UTF-8 as our character encoding since both ASCII and UTF-8 map each ASCII character to the same number (code point).

The code points in UTF-8 are specified as U+ followed by a hexadecimal number. Hexadecimal is a system where a number is specified using 16 symbols: 0-9 and A-F representing 10-15. So a hexadecimal symbol that appears just before another hexadecimal symbol is 16 times larger, just as a decimal symbol (0-9) that appears before another decimal symbol is 10 times larger. Anyway, a UTF-8 code point would look like U+004A or U+0061. U+004A represents the 74th symbol in the character set (16*4 + 1*10 = 74), which is `J`. U+0061 represents the 97th symbol in the character set (16*6 + 1*1 = 97), which is `a`. You can look up a UTF-8 character table for the code points: see http://www.utf8-chartable.de/ , for example.

3. UTF-8 in PuTTY

Your computer program must know what the character encoding of a text file is in order to interpret the text properly. Some programs come with the ability to guess or detect the character encoding of their input files. Unfortunately, PuTTY doesn't come with this ability and we must configure its settings to interpret the text in UTF-8. To do this, right click on the title bar, click Choose settings..., and reconfigure Window > Translation as follows.



4. Inserting UTF-8 characters in Vim

We might occasionally want to insert UTF-8 characters in the Vim text-editor program. This will often be the case if we are writing a program that processes a corpus of a language other than English. Here is how you can insert UTF-8 characters in Vim:

(1) In menu mode, enter `:set encoding=utf-8`
(2) Enter insert mode.
(3) Hit `Ctrl+v`.
(4) Enter code point: first hit `u` and then type the hexadecimal number. For example, hit `u` and then `00E1` to insert á.

5. UTF-8 in Python

As I mentioned in the previous section, you will often work with text files in UTF-8 character encoding if you are working with multilingual corpora. Below is a little bit of how to use UTF-8 with Python.

5.1. Byte-string vs. Unicode-string

Python supports two kinds of strings: byte strings and Unicode strings. The string objects we have been using so far were byte strings: we specify a string in single/double quotes and Python interprets it as a sequence of bytes[2]. To specify a Unicode string, we add `u` before the string in quotes as in the following example:

```
unicode_string = u'San Jos\u00E9'
```

Note that `\u00E9 (=U+00E9)` is the UTF-8 code point for é. The `\u` at the beginning is means `U+`.

To process a Unicode-string in a meaningful way, Python must first convert it to a byte-string. This is done by using the `encode` method specific to a Unicode-string. For example, the following tells Python that `unicode_string` is encoded in UTF-8 and orders Python to convert it to a byte-string.

```
byte_string = unicode_string.encode('utf-8')
```

If we want to convert a byte-string into a Unicode-string we use the `decode` method as in the following example:

```
unicode_string_2 = byte_string.decode('utf-8')
```

[2] A byte is a string of eight bits (zeros or ones). A single character is represented by a single byte when we use the ASCII code. So when Python interprets a string object as a byte string, it is assuming that we are using the ASCII code.

5.2. UTF-8 literals in Python

By default, Python assumes that strings are encoded in ASCII. So if we want to use a character encoding other than ASCII, we have to declare it before we begin our code. This is done by adding a special comment either as the first or the second line of code:

```
# -*- coding: <encoding> -*-
```

For example, if we were to use UTF-8, we would declare

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

The above comment would be in the second line if the first line of the code were reserved for invoking the Python interpreter: `#!/usr/bin/python`.

Once you declare the character encoding, you can include non-ASCII characters in your code as long as they are covered by the character encoding system you declared.