Programming Unicode

SIMSON L. GARFINKEL

Simson L. Garfinkel is an Associate Professor at the Naval Postgraduate School in Monterey, California. His research interests include computer forensics, the emerging field of usability and security, personal information management, privacy, information policy, and terrorism. He holds six US patents for his computer-related research and has published dozens of journal and conference papers in security and computer forensics.
simsong@acm.org

Many people I work with understand neither Unicode's complexities nor how they must adapt their programming style to take Unicode into account. They were taught to program without any attention to internationalization or localization issues. To them, the world of ASCII is enough—after all, most programming courses stress algorithms and data structures, not how to read Arabic from a file and display it on a screen. But Unicode matters—without it, there is no way to properly display on a computer the majority of names and addresses on the planet. My goal with this article, then, is to briefly introduce the vocabulary and notation of Unicode, to explain some of the issues that arise when encountering Unicode data, and to provide basic information on how to write software that is more or less Unicode-aware. Naturally, this article can't be comprehensive. It should, however, give the Unicode-naïve a starting point. At the end I have a list of references that will provide the Unicode-hungry with additional food for thought.

Unicode is the international standard used by all modern computer systems to define a mapping between information stored inside a computer and the letters, digits, and symbols that are displayed on screens or printed on paper. It's Unicode that says that a decimal 65 stored in a computer's memory should be displayed as a capital letter "A"; that a decimal 97 is a lowercase "a"; and that the "£" is hex 00A3. Every Unicode character has a unique name in English that's written with uppercase letters—for example, POUND SIGN. Every Unicode character also has a unique numeric code, which Unicode calls a "code point," that's written with a U+ followed by a four or five character hex code. One of the most common Unicode characters that will not fit in 8-bits is the EURO SIGN (€, or U+20AC); my favorite character is SNOWMAN (☃, or U+2603). Code points over 65,536 are particularly troublesome. Fortunately, they are mostly used for unusual Chinese characters and letters in ancient scripts, such as AEGEAN NUMBER SIX THOUSAND (𐄇, or U+10127) [1].

As the preceding paragraph demonstrates, Unicode is dramatically more complex than the 7-bit American Standard Code for Information Interchange (ASCII) and so-called "Latin1" (actually ISO 8859-1) systems that it replaces. But Unicode brings more complexity than an expanded character set. For example, Unicode accented characters can be represented with a single code point (for example, LATIN SMALL LETTER E WITH ACUTE—é, or U+00E9), or with a character followed by a so-called combining character (for example, Unicode character LATIN SMALL LETTER E (U+0065) followed by Unicode character COMBINING ACUTE ACCENT (U+0301).

The Unicode standard also has rules for displaying and correctly processing writing systems that go right-to-left such as Arabic, Hebrew, and Dhivehi. Before Unicode, it was a complex task to build computer systems that could properly display text that way, let alone intermix right-to-left with left-to-right. Unicode's support for bidirectional text [9] largely eliminates these problems, making it possible to freely use Arabic words like سلام (Salam) in English text. What's particularly clever is that the characters are stored inside the computer in their logical order, not in the order that they are displayed.

Because you can paste Unicode into Web browsers and email messages, Unicode makes it relatively easy to embellish your writing with a wide variety of interesting symbology, including boxed check marks (☑), circled numbers (both ① and ❶), and even chess pieces (♛). Some programmers discover that it's significantly easier to paste a well-chosen Unicode character on a button than to hire a graphic artist to make an icon. And these icons look really great when magnified or printed, because they are typically encoded in the font as vector graphics, not bitmaps.

So why do so many US programmers and technologists have a poor understanding of Unicode? I suspect it is our cultural heritage at work: by design, most written communications within the United States can be encoded in ASCII. Ever since the move to word processing in the 1980s, it's been exceedingly hard for Americans to type characters that are not part of ASCII. As a result, many of these characters, such as the CENT SIGN ¢ (U+00A2), are now rarely used in the US—even though they were widely available on typewriters in the 1970s.

Indeed, for nearly 30 years it has been possible to have a very lucrative career in the US using only ASCII's 96 printable graphemes, never once having to venture into a world where characters are 8, 16, or 32 bits wide. Alas, the ASCII-ensconced lives of most US programmers is increasingly an anachronism, thanks to the integration of the world's economies, the increasingly international aspects of the computing profession, and even simple demographic trends—more people in the US have accented characters in their names and want those names properly spelled.

Code Points and Characters

Like ASCII and ISO8859-1, at its most fundamental level the Unicode standard defines a mapping between code points and print characters. The big difference is quantity: Unicode's most recent version, Unicode 6.1, defines more than 109,000 printable objects.

Most of the code points map to characters, which the standard sometimes calls "graphemes." A grapheme is a unit of written language such as the letter "a" (U+0061), the number "1" (U+0031), or the Kanji for man "男" (U+7537). Most graphemes are displayed as a single glyph, which is the smallest printable unit of a written language.

Unicode terminology is precise and frequently misused: the confusion is frequently reflected by errors in Unicode implementations. For example, a grapheme (such as the lowercase "a") can be displayed by two distinct glyphs (in the case of an "a", one of the glyphs looks like a circle with an attached vertical line on the right, while the other looks like a slightly smaller circle with a hook on the top and a tail in the lower-right quadrant). Both glyphs are represented by the same code point. But some glyphs can represent two characters—for example, some typesetting programs will typeset the letter "f" (U+0066) followed by the letter "i" (U+0069) as

an “fi” ligature (U+FB01), which obviously has a distinct code point. Typesetting programs do not expect to see a U+FB01 in their input file, for the simple reason that people don’t type ligatures. It’s the typesetting program that replaces the U+0066 followed by the U+0069 with a U+FB01, adjusting spacing as appropriate for a specific font.

Arabic is even more complex. Arabic graphemes are written with different glyphs depending on whether the grapheme appears at the beginning, at the end, or in the middle of a word. In the case of Arabic, most of the letters actually have four code points assigned: one that represents the abstract character and three “presentation” code points that represent each printable form. The abstract character might be used inside a word processor document, while the presentation forms are used in files that are rendered for display—for example, in a PDF file. Microsoft’s implementation of Arabic is different for various versions of Word, Excel, and PowerPoint on the Mac and PC. As a result, a file containing Arabic that looks beautiful on one platform can look horrible on another. Google Docs has similar problems when it is used to edit Arabic documents.

Every modern programming language supports Unicode, but frequently with types and classes that are different from those taught in school. For example, C and C++ programmers should use the wchar_t type to hold a Unicode character (or any other printable character, for that matter), defined in the <wchar.h> header. Unicode strings are best represented in C++ with the STL std::wstring class defined in the <string> header. In Python3 the “string” type can hold 0, 1, or multiple Unicode characters; there is no separate type for an individual character. If you are still using Python2 (and you should stop), the “string” class holds ASCII and you need to specify “unicode” (or u"") to create a Unicode character or string. In Java the “char” primitive type and the “Character” class hold Unicode code points, but only those characters with values less than U+FFFF—what’s called Unicode’s Basic Multilingual Plane. Code points that require more than 16 bits are represented by a pair of code points, as we will see below.

Expanding the Basic Multilingual Plane

Before Unicode, many manufacturers developed systems for representing Chinese or Japanese using 16-bit characters, sometimes called “wide” characters. One of the original goals of Unicode was “Han Unification”—that is, using the same code points to denote ideographs that were the same in the two languages. The designers were able to get nearly all of the characters they needed to fit into the 65,536 code points allowed by a 16-bit word. This 2-byte encoding was called UCS-2, for Universal Character Set, 2-byte encoding.

Alas, 65,536 characters were not sufficient to represent all of the Chinese logograms, let alone characters for all of the world’s ancient languages. Today, Unicode supports a total of 17 “code planes,” each with 65,536 characters. Code Plane 0 is the Basic Multilingual Plane and covers Unicode characters U+0000 through U+FFFF (although U+FFFF is explicitly not a valid Unicode character). Code Plane 1 is mostly used for additional symbols, Plane 2 for additional ideographs, Plane 14 for special purpose, and Planes 15 and 16 for private use. Planes 3–13 are currently unassigned. With so much room to grow, it is highly unlikely that Unicode will ever need to be expanded beyond the 17 code planes—even if we encounter an alien race that has 100,000 or more characters in its written language (see Table 1).

Plane 0:	0000–FFFF	Basic Multilingual Plane (BMP)
Plane 1:	10000–1FFFF	Supplementary Multilingual Plane (SMP)
Plane 2:	20000–2FFFF	Supplementary Ideographic Plane (SIP)
Planes 3–13:	30000–DFFFF	Unassigned
Plane 14:	E0000–EFFFF	Supplementary Special Purpose Plane (SSP)
Planes 15–16:	F0000–10FFFF	Supplementary Private Use Area (S PUA A/B)

Table 1: Unicode code planes

The assignment of code planes is arbitrary, of course, but there is some sensibility in the allocation. Plane 0 was first. The extension mechanism that the standards body adopted allows an additional 4 bits to be optionally specified; Plane 1 has all of those additional bits set to 0, Plane 16 has them all set to 1, and Plane 0 is marked by the absence of those optional bits. Planes 1 and 2 are really the only additional planes that were mapped out by the Committee. Because Planes 3–13 are unassigned, they can be trapped as errors. Frankly, I doubt that characters within Planes 3 through 13 will ever be assigned, although these might conceivably be used for some other purpose at some point in our lifetimes.

Many systems (e.g., Java) were designed to be Unicode-aware back when Unicode only had 16-bit characters. Unicode 2.0's creators thought that they would not be able to get programmers to change their systems to use 32-bit characters—especially when most of the bits would always be 0. Instead, a coding scheme was developed that allowed pairs of 16-bit code points in the BMP to represent characters with code points greater than 65,535.

Consider the SQUARED FREE (🈓, or U+1F193). This code point can be represented with 4 bytes using a sequence of four hex characters, 00 01 F1 93. But the code point can also be represented with a pair of Unicode characters called "surrogates." The 18 bits of code points outside the BMP can be divided into two halves, with 9 bits encoded by a first surrogate in the range D800–DBFF and 9 bits encoded using a second surrogate in the range DC00–DFFF. In the case of U+1F193, the two surrogates are U+D83C and U+DD93. Java always uses surrogates to represent characters outside the BMP.

Python can be compiled to use either 16-bit or 32-bit Unicode code points for its internal representation. If sys.maxunicode is 65535, then your Python interpreter is using surrogates internally to represent characters outside the BMP; if sys.maxunicode is 1114111, the Python interpreter can represent all Unicode characters without surrogates internally [4].

No matter how it is compiled, Python allows code points outside the BMP to be specified with a pair of 16-bit surrogates or as a single 32-bit value. Here we ask Python to print a character string from the two surrogates using the Python "\u" escape, which lets us specify any Unicode character in the BMP with its 4-character hexadecimal code:

```
>>> print("\uD83C\uDD93")
```

And here we use the Unicode "\U" escape and enter an 8-character hexadecimal code to print the same character without the use of surrogates:

```
>>> print("\U0001F193")
```

```
>>>
```

The way C and C++ handle characters outside the BMP depends on the platform. Under GCC/G++ 4.2 on Mac and Linux systems, wchar_t is a 32-bit value, allowing Unicode characters outside the BMP to be stored directly. But compile a program with Microsoft's Visual Studio or the mingw cross-compiler, and wchar_t is a 16-bit quantity. Frankly, it's hard to notice the difference, provided you always allocate memory in sizeof(wchar_t) chunks and never depend on the size of a Unicode string being related in any way to the number of characters that it contains.

All of this is less complicated in practice if you use a high-quality Unicode implementation that hides these details. Indeed, I wrote and deployed several Unicode-aware C++ programs before I realized that sizeof(wchar_t) was 4 on my primary development system but 2 when cross-compiling for Windows.

Normalization and Collation

Since Unicode allows the same string to be represented with many different but logically equivalent sequences of code points, the standard provides a way of normalizing any Unicode sequence of code points so that different strings can be compared for equivalency.

Actually, Unicode has two kinds of equivalencies between characters: "canonical equivalence" and "compatibility equivalence." Canonical equivalence resolves the ambiguity introduced by combining characters. LATIN SMALL LETTER E WITH ACUTE (U+00E9) and LATIN SMALL LETTER E (U+0065) followed by a COMBINING ACUTE ACCENT (U+0301) are considered to be equivalent. Compatibility equivalence is used to denote sequences that have the same semantic meaning but may appear visually distinct—for example, SUPERSCRIPT ONE ("¹", or U+00B9) and DIGIT ONE ("1", or U+0031) have compatibility equivalence, as do the characters LATIN CAPITAL LETTER I ("I" U+0049) and ROMAN NUMERAL ONE ("Ⅰ", U+2160). One of the main uses of compatibility equivalence is to improve the recall of string search, but it can also be used to address some of the many security issues caused by having forms that are visually identical but have different encodings [5a]. These two equivalence algorithms mean that determining whether or not two strings are equal is a multi-step process. First you must decide what kind of equivalence you want. Then both strings must be normalized. Finally, they can be compared [5b].

Yet another issue is the sort order of Unicode characters, something known as "collation." The complexity here is that different languages (and sometimes different usages within the same language) require different sorting of the same characters. A common example is that Swedish sorts "z" (U+007A) before "ö" (U+00F6), but German sorts "ö" before "z". Unicode's combining characters add to the complexity. The Unicode Collation Algorithm provides a unified, locale-aware approach to sorting. Although it's described in Unicode Technical Standard #10 [6], most

programmers will be better off using a collation implementation that's widely used and well-debugged rather than implementing their own. For Python users, the function locale.strcoll performs a basic implementation of the ISO 14651 collation algorithm but not the full Unicode algorithm. For a more complete implementation, use IBM's International Components for Unicode library, which has bindings for C, C++, Java, and Python [3].

Encoding and Decoding

So far, this article has been discussing Unicode in the abstract and has avoided the messy issue of reading and writing Unicode data. The issue is messy because modern computer systems read and write data in 8-bit bytes, but Unicode needs a minimum of 16 bits to represent characters in the BMP and 21 bits [10] to represent all possible code points (or two 16-bit pairs, if surrogates are used).

Early Unicode implementations, such as the one in Microsoft Windows, took the rather straightforward approach of storing everything as 16-bit UCS-2 characters. When Microsoft needed to store Unicode on disk—for example, in a file name—it simply wrote the bytes in the same order that they were stored in memory. This process of transforming abstract code points to a specific set of 8-bit codes stored in a file or sent down a wire is called "encoding."

Clearly, there are two ways for a 16-bit code point such as U+0061 to be encoded: as a 61 followed by a 00 (called "little endian," because the little end comes first), or as a 00 followed by a 61 ("big endian"). Rather than mandating that Unicode be encoded one way or the other, Unicode supports both byte orders. UTF-16LE (UCS Translation Format—16-bit Little Endian) is what Windows uses.

For example, the FAT32 file system stores both legacy ISO8859-1 8.3 filenames and UTF-16LE filenames that can be up to 255 characters long. NTFS file systems store only UTF-16LE filenames. This means that the filename README.TXT is stored as the UTF-16LE sequence 52 00 45 00 41 00 44 00 4d 00 45 00 2e 00 54 00 58 00 54 00. Most of the Windows API functions that operate on files have two versions—one that takes ISO8859-1 names terminated with a 00, and one that takes UTF-16LE "wide" names terminated with a 00 00. For creating files, these are the CreateFile() and CreateFileW() functions.

Similar to the 2-byte encodings, Unicode also supports 4-byte encodings UTF-32LE and UTF-32BE. With the UTF-32LE encoding the string "READ" would encode as 52 00 00 00 45 00 00 00 41 00 00 00 44 00 00 00. Such encodings are rarely used outside of a computer's memory, because of the storage cost.

Unicode provides a special code called the Byte Order Mark (BOM, U+FEFF) that can be stored inside a file and used to unambiguously indicate whether the file is encoded as UTF-16LE, UTF-16BE, UTF-32LE, UTF-32BE, or using the variable-length UTF-8 code (see below). The BOM approach works because the byte-swapped character U+FFFE is defined to be an invalid code point. Thus, by looking at the first 2 or 4 bytes of a file, it is possible to determine the encoding (see Table 2).

Initial Bytes	Encoding
00 00 FE FF	UTF-32, big-endian
FF FE 00 00	UTF-32, little-endian
FE FF	UTF-16, big-endian
FF FE	UTF-16, little-endian
EF BB BF	UTF-8 (see below)

Table 2: Unicode Byte Order Mark signatures, from http://unicode.org/faq/utf_bom.html

C programmers have the biggest problem with UTF-16 and UTF-32 encodings, for the simple reason that the C standard library uses the NULL character (00) to denote the end of a string. The obvious way to avoid this is by using the wide-character version of the string library—for example, use wcslen() instead of strlen(). Personally, I recommend abandoning C-style strings and using the C++ STL std::wstring type instead.

From the preceding examples it may seem that a program can trivially convert between Unicode and ASCII by simply removing or inserting alternating NULL characters. You should never do this!!! This simplistic approach will fail if the Unicode string contains anything other than code points in the range U+0001 through U+007F. Instead, programs should explicitly encode Unicode strings from an abstract internal representation when strings are transformed for operating system APIs, sent over a network connection, or persisted into a file. Likewise, when data is read from an external source it should be decoded from the wire format into the program's internal representation.

UTF-8

UTF-8 is a system for encoding Unicode code points using a variable-length sequence of 8-bit characters. UTF-8 has the property that 7-bit ASCII characters are directly coded as a single UTF-8 byte, making UTF-8 upwards compatible from ASCII. Characters in the range U+0080 through U+07FF are coded as 2 bytes; the remaining characters in the BMP are coded as three bytes; characters outside the BMP are coded as 4 (see Table 3).

The UTF-8 scheme makes it possible to identify the start of a UTF-8 character from within a randomly chosen block of UTF-8 encoded bytes. If the most significant bit is a 0, then the character is a 7-bit UTF-8 character. If the high bits are "10", then it is a continuation character: move forward or backwards until a byte is found that begins "0" or that begins "11". The number of leading "1"s in the first byte of a UTF-8 encoding indicates the number of bytes in the sequence.

As the table makes clear, UTF-8 is great for Americans, since documents coded in UTF-8 are the same size as documents coded in ASCII. For Europeans, the advantage of UTF-8 is that all of their accented characters can be displayed, with only the non-ASCII characters taking up 2 bytes. For the Chinese, UTF-8 is not so good, as for most text it results in a 50% increase in required storage space compared to UTF-16.

Bits	Code	Point	Range	Byte 1	Byte 2	Byte 3	Byte 4
7	U+0000	-	U+007F	0xxxxxxx			
11	U+0080	-	U+07FF	110xxxxx	10xxxxxx		
16	U+0800	-	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
21	U+10000	-	U+1FFFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

Table 3: UTF-8

One of the primary advantages of UTF-8 is that the NULL character is never used. This means that the POSIX APIs can be used more or less transparently with UTF-8 encoded Unicode. The confusion here, however, is that UTF-8 is not Unicode—it is a Unicode encoding. Keeping your program's internal data encoded in UTF-8 is fine as long as the strings are viewed atomically and no string operations such as comparison, search, case change, or splitting ever need to be performed by the program. All of these operations require decoding the UTF-8 sequences into Unicode characters and then re-encoding the Unicode characters back to UTF-8.

Source Code

Encoding and decoding comes up in two primary places when coding: the encoding and decoding of files that are written and read by your program, and the program's source code itself. To see how this works in practice, consider this simple Java program with some embedded Unicode characters:

```
class test {
       public static void main(String[] args) {
            String LETTER_A = "A";
          System.out.println("size("+LETTER_A   +")="+LETTER_A.length());

            String IDEOGRAPH = "男";
          System.out.println("size("+IDEOGRAPH   +")="+IDEOGRAPH.length());

          String SQUARED_FREE = "🆓";
          System.out.println("size("+SQUARED_FREE+")="+SQUARED_FREE.length());
       }
}
```

Java programs are files, of course, so they need to be encoded with a particular Unicode encoding when they are stored in the file system. This file was written with Apple's TextEdit application and saved in UTF-8. In UTF-8 the LETTER_A above takes a single byte, the IDEOGRAPH takes 3 bytes and the SQUARED _FREE takes 4 bytes. So when we run this program we get this result:

```
size(A)=1
size(男)=3
size(🆓)=4
```

Whoops! The values in the Java String are not Unicode code points! Instead they are literally the encoded bytes. This can be confusing (it's confusing to me, at least).

Instead of saving the file in UTF-8, the source file can be saved in UTF-16. I've done that, renaming the class from "test" to "test16." Now when the program is compiled, the Java compiler needs to be told the encoding used by the file:

```
$ javac -encoding utf-16 test16.java
$
```

When I run the program I get these confusing results:

```
$ java test16
size(A)=1
size(?)=1
size(?)=2
$
```

The question marks display result from the fact that the program was run inside a UTF-8 terminal. The A takes a single UCS-2 character, as does the 男. Java's runtime seems willing to convert the A to UTF-8, but not the 男. The SQUARED _FREE is represented in the Java source file with two surrogate pairs (I saved it as UTF-16, remember?), so the length of the string is 2, not 1.

I also tried saving my test program in UTF-32LE and compiling it with the Java compiler, but I got this error:

```
$ javac -encoding utf-32 test32.java
test32.java:1: warning: unmappable character for encoding utf-32
????????????????????????????????????????????????????????????????????????...
...
```

Clearly, my Java compiler does not support UTF-32 for input source encoding. For more information about how Java does this, see the Java documentation for the Character and String classes and the Java tutorial "Working with Unicode" [1a].

Python source code encodings are defined with a "magic comment" [1b] like this:

```
#!/usr/local/bin/python
# coding: utf-8
import os, sys
...
```

Reading and Writing Files

Similar problems occur when attempting to process files. Asked to open a file and infer its coding, some programs will attempt to guess whether the file's contents are in ASCII, UTF-8, or one of the UTF-16 dialects. Unfortunately, it is not always possible to guess correctly, for the simple reason that there are many hex sequences that can be decoded using multiple coding variants. Consider the sequence of hex bytes 41 42 43 44. This could be the UTF-8 sequence "ABCD" (U+0041 U+0042 U+0043 U+0044), but it could also be the UTF-16LE sequence 䅡䅢 (U+4241 U+4443) or the UTF-16BE sequence 格鈴 (U+4142 U+4344).

Python and Java address the encoding issue by allowing the programmer to specify an encoding when a file is opened. For example:

```
f = open("file.txt",mode="r",encoding="utf-8")
```

(In Python2.7, you can get the same functionality with the codecs.open function.)

In Java, one would use:

```
FileInputStream fis = new FileInputStream("file.txt");
InputStreamReader isr = new InputStreamReader(fis,"UTF8");
BufferedReader in = new BufferedReader(isr);
```

Some file formats allow you to specify the encoding inside the file itself, which is something of a trick, because you need to know the encoding in order to decode the file.

For example, if you edit files with EMACS, you can put a local variables line at the top of your file to tell EMACS which coding to use:

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

In XML, encodings are specified on the first line of the file:

```
<?xml version="1.0" encoding="UTF-8"?>
```

Python's XML parsers expect to read this line to determine the coding of the file. As a result, to read an XML file with Python3 and process it with expat, it's necessary to open the file as a binary stream:

```
f = open("file.xml",mode="rb")
p = xml.parser.expat.ParserCreate()
...
p.ParseFile(f)
```

Encoding a Python Unicode string to a particular representation is quite simple. Here's how s, a Python3 Unicode string with my favorite Snowman character, looks in UTF-8, UTF-16LE, and UTF-16BE:

```
>>> s = "This is a Snowman: ☃"
>>> s.encode('utf-8')
b'This is a Snowman: \xe2\x98\x83'
>>> s.encode('utf-16le')
b'T\x00h\x00i\x00s\x00 \x00i\x00s\x00 \x00a\x00 \x00S\x00n\x00o\x00w\
x00m\x00a\x00n\x00:\x00 \x00\x03&'
>>> s.encode('utf-16be')
b'\x00T\x00h\x00i\x00s\x00 \x00i\x00s\x00 \x00a\x00 \x00S\x00n\x00o\
x00w\x00m\x00a\x00n\x00:\x00 &\x03'
>>>
```

For encoding and decoding in C++, I recommend using the open source UTF8-CPP package [2]. The package contains C++ classes and iterators for interconverting between UTF-8, UTF-16, and UTF-32. If you want a more complete (and significantly larger) implementation, IBM's ICU is a better choice.

Encoding Errors

Many programmers get their first unpleasant taste of Unicode when they attempt to read a file and instead of getting data, they get an exception. For example, consider a file "file.txt" that contains the hex characters FF FE FE FF. Try to read this file with Python, and Python will throw an exception:

```
>>> f=open("file.txt")
>>> f.read()
Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
      File "/opt/local/Library/Frameworks/Python.framework/Versions/3.2/
lib/python3.2/codecs.py", line 300, in decode
        (result, consumed) = self._buffer_decode(data, self.errors, final)
UnicodeDecodeError: 'utf8' codec can't decode byte 0xff in position 0: invalid
start byte
>>>
```

Python assumes the file is in UTF-8, and FF is an invalid UTF-8 character. Of course, Python couldn't read this file if opened in text mode with any encoding.

If you are unsure if a file contains valid Unicode encodings, one approach is to open it in binary mode and convert it a line at a time. This approach works best with files encoded as UTF-8, since it's relatively easy to spot the line breaks, but it can be applied to UTF-16 and UTF-32 encoded files as well. If a line contains invalid Unicode, you can try converting it character by character. There are a number of packages that will do this, such as BeautifulSoup's UnicodeDammit class.

For longtime UNIX programmers, this need to open files in "text" or "binary" mode might seem like a step backwards—or into the wacky world of Windows programming. Sadly, the multiplicity of encodings leaves us no choice.

Problems resulting from improper encoding and decoding pervade modern computing systems. If you have ever opened a Web page and seen it filled with white question marks in black diamonds, your browser was showing the Unicode REPLACEMENT CHARACTER (, U+FFFD), because the bytes on the Web page couldn't be decoded using the encoding the Web page specified. This frequently happens on Web pages that specify no encoding at all but contain smart quotes; the default HTTP encoding is ISO 8859-1, and 8859-1 doesn't have any smart quotes in it.

I encountered a more egregious case of bad encoding after a recent trip to Japan. All of the little paper receipts had nicely printed katakana, hiragana, and kanji characters. But the online statement for my American Express credit card for a $39.30 transaction at the Tokyo Muji store looked like this:

MUJIRUSHI RYOHIN *	S L GARFINKEL	39.30
TOKYOTO TOSHIMAKU		
TOKYOTO TOSHIMAKU		
Foreign Spend Amount:	3,064 JAPANESE YEN	
Doing Business As:	MUJIRUSHI RYOHIN	
Merchant Address:	ncgÔcn nÇuh	
	ÏfæÇbiÐæhã 4-27-10	
	Æ]ÊcÄæjæb1Ïæ× 3F	
	JAPAN	

From the selection of characters, it seems that the merchant's bank transmitted the transaction information to American Express as an encoded UTF-8 string, but the data was then decoded by a computer in the US that assumed ISO 8859-1.

You don't have to travel to Japan to see these kinds of problems. A growing number of GNU tools now output UTF-8 error messages. Consider this buggy function:

```
int fail() {
        return "A";
}
```

Here is the error message that g++ generates when it is compiled:

```
$ g++ -Wall fail.cpp
fail.cpp: In function ‘size_t fail()’:
fail.cpp:2: error: invalid conversion from ‘const char*’ to ‘int’
$
```

The compiler outputs the error messages with smart quotes because the environment variables LANG and LC_ALL are set to 'en_US.UTF-8'. The compiler notices this encoding and sends the hex sequence E2 80 98 for the open quote and E2 80 99 for the close quote. If I unset both environment variables, the smart quotes go away:

```
$ unset LC_ALL; unset LANG; g++ -Wall fail.cpp
fail.cpp: In function 'size_t fail()':
fail.cpp:2: error: invalid conversion from 'const char*' to 'int'
$
```

One day I was running my compiles inside EMACS on a remote system and started seeing error messages like this:

```
fail.cpp:3: error: invalid conversion from âconst char*â to âcharâ
```

EMACS was not expecting the compiler to be sending UTF-8, so it assumed the default of ISO 8859-1. It received the E2 and displayed the â character (U+00E2); the 80, 98, and 99 have no mapping in ISO 8859-1, so they were not displayed. (A more clever EMACS implementation might notice that the sequence E2 80 98 is invalid ISO 8859-1 but valid UTF-8 and change the encoding mode for the buffer, but that might cause other problems.)

Conclusion

The information here should help many C, C++, Java, and Python programmers in developing Unicode-aware programs. There's certainly a lot more to learn, though. For those programmers with the time and the interest to delve deeply, here are some suggestions for further reading. Even if you aren't a Python programmer, it's quite instructive to read the Python "Unicode HOWTO." While there's lots to criticize about the Python Unicode implementation, Python3 gets a lot of things right. Python also includes a built-in Unicode database in a module named, aptly enough, unicodedata (http://docs.python.org/library/unicodedata.html). With the database you can trivially convert from code points to Unicode names and access other aspects of the standard.

Resources

Apple's Character Viewer, built into Mac OS X, is a great way to find Unicode characters. You can access it from the Apple menu bar if you enable the appropriate option in the System Preferences.

Wikipedia's articles on Unicode are really excellent.

The Unicode standard is available online. Start with the FAQs [7] and the Technical Reports [5, 6, 8].

Stackoverflow.com is filled with good information about Unicode.

The Web site FileFormat Info contains detailed information on each Unicode character, including test pages to see if your browser will display it. Find the snowman at http://www.fileformat.info/info/unicode/char/2603/.

References

[1] It's quite possible that this character won't show up properly on your system, since many programs still do not properly display Unicode code points over U+FFFF. You can see what AEGEAN NUMBER SIX THOUSAND is supposed to look like and test your browser's functionality at http://www.fileformat.info/info/unicode/char/10127/index.htm.

[1a] The Java Unicode tutorial can be found at http://docs.oracle.com/javase/tutorial/i18n/text/unicode.html. Documentation for the String and Character classes is at http://docs.oracle.com/javase/7/docs/api/java/lang/Character.html and http://docs.oracle.com/javase/7/docs/api/java/lang/String.html.

[1b] For more information on Python's magic comments to specify file encoding, see http://www.python.org/dev/peps/pep-0263/.

[2] See UTF-8 with C++ in a Portable Way: http://utfcpp.sourceforge.net/.

[3] IBM's International Components for Unicode: http://site.icu-project.org/.

[4] Python 3.3 will be able to represent 1, 2, or 4-byte code points internally using the most efficient representation. For more information, see PEP393 Flexible String Representation: http://www.python.org/dev/peps/pep-0393/.

[5a] For a complete discussion of Unicode security issues, please see Unicode Technical Report #36, Unicode Security Considerations: http://unicode.org/reports/tr36/. There is also a very short but somewhat informative Security Issues FAQ at http://unicode.org/faq/security.html.

[5b] See Unicode Technical Annex #15: Unicode Normalization, for an in-depth discussion of the different forms of equivalence and the Unicode Normalization Algorithm: http://unicode.org/reports/tr15/.

[6] TR10—Unicode Collation Algorithm, Unicode Technical Standard #10: http://unicode.org/reports/tr10/.

[7] FAQ: UTF-8, UTF-16, UTF-32, and BOM: http://unicode.org/faq/utf_bom.html.

Python Unicode How To—http://docs.python.org/release/3.1.3/howto/unicode.html.

[8] TR17—Unicode Character Encoding Model, Unicode Technical Report #17: http://unicode.org/reports/tr17/.

[9] See "Unicode Bidirectional Algorithm" (Unicode Standard Annex #9), which formally defines the correct way for conforming Unicode implementations to display bidirectional text: http://unicode.org/reports/tr9.

[10] Twenty-one bits are required to represent an arbitrary code point: 16 bits for the character within the Plane; 1 bit indicating whether the code point is within the BMP or uses one of the higher planes; and 4 bits to represent the higher plane that is in use.