CHAPTER 4
Bad Data Lurking in Plain Text

Josh Levy, PhD

> *This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.*
>
> —Doug McIlroy

Bad data is often delivered with a warning or an apology such as, "This dump is a real mess, but maybe you'll find something there." Some bad data comes with a more vacuous label: "This is plain text, tab-delimited. It won't give you any trouble."

In this article, I'll present data problems I've encountered while performing seemingly simple analysis of data stored in plain text files and the strategies I've used to get past the problems and back to work. The problems I'll discuss are:

1. Unknown character encoding
2. Misrepresented character encoding
3. Application-specific characters leaking into plain text

I'll use snippets of Python code to illustrate these problems and their solutions. My demo programs will run against a stock install of Python 2.7.2 without any additional requirements. There are, however, many excellent Open Source libraries for text processing in Python. Towards the end of the article, I'll survey a few of my favorites. I'll conclude with a set of exercises that the reader can perform on publicly available data.

Which Plain Text Encoding?

McIlroy's advice above is incredibly powerful, but it must be taken with a word of caution: *some text streams are more universal than others*. A text encoding is the mapping between the characters that can occur in a plain text file and the numbers computers use to represent them. A program that joins data from multiple sources may misbehave if its inputs were written using different text encodings.

This is a problem I encountered while matching names listed in plain text files. My client had several lists of names that I received in plain text files. Some lists contained names of people with whom the client conducted business; others contained the names of known bad actors with whom businesses are forbidden from transacting. The lists were provided as-is, with little or no accompanying documentation. The project was part of an audit to determine which, if any, of the client's partners were on the bad actors lists. The matching software that I wrote was only a part of the solution. The suspected matches it identified were then sent to a team of human reviewers for further investigation.

In this setting, there were asymmetric costs for errors in the matching software. The cost of a false positive—a case where the investigative team rejects a suspected match—is the cost of the human effort to refute the match. The cost of a false negative is the risk to which the client is exposed when the investigative team is not alerted to a match. The client wanted the ability to tune the matching software's error profile, that is, to be able to choose between being exposed to fewer false positives at the expense of potentially more false negatives, or vice-versa. Had I not recognized and worked around the text encoding issues described below, the client would have underestimated its risk of false negative errors.

Let's discuss the basics of text encoding to understand why multiple text encodings are in use today. Then let's discuss how we can detect and normalize the encoding used on any given plain text file. I became computer literate in a time and place when plain text meant ASCII. ASCII is 7-bit encoding with codes for 95 printable characters shown in Table 4-1. ASCII encodes the letters of the English alphabet in upper- and lowercase, the Arabic numerals, and several punctuation and mathematical symbols.

When I write a number without a prefix, such as *65*, it can be assumed to be in decimal notation. I will use the prefix *0x* to indicate numbers in hexadecimal format, for example *0x41*.

Table 4-1. Printable ASCII characters

(32)	*!* (33)	*"* (34)	*#* (35)	*$* (36)	*%* (37)	*&* (38)	*'* (39)
((40)	*)* (41)	* (42)	*+* (43)	*,* (44)	*-* (45)	*.* (46)	*/* (47)
0 (48)	*1* (49)	*2* (50)	*3* (51)	*4* (52)	*5* (53)	*6* (54)	*7* (55)
8 (56)	*9* (57)	*:* (58)	*;* (59)	*<* (60)	*=* (61)	*>* (62)	*?* (63)
@ (64)	*A* (65)	*B* (66)	*C* (67)	*D* (68)	*E* (69)	*F* (70)	*G* (71)
H (72)	*I* (73)	*J* (74)	*K* (75)	*L* (76)	*M* (77)	*N* (78)	*O* (79)
P (80)	*Q* (81)	*R* (82)	*S* (83)	*T* (84)	*U* (85)	*V* (86)	*W* (87)
X (88)	*Y* (89)	*Z* (90)	*[* (91)	\ (92)	*]* (93)	^ (94)	_ (95)
` (96)	*a* (97)	*b* (98)	*c* (99)	*d* (100)	*e* (101)	*f* (102)	*g* (103)
h (104)	*i* (105)	*j* (106)	*k* (107)	*l* (108)	*m* (109)	*n* (110)	*o* (111)
p (112)	*q* (113)	*r* (114)	*s* (115)	*t* (116)	*u* (117)	*v* (118)	*w* (119)
x (120)	*y* (121)	*z* (122)	*{* (123)	| (124)	*}* (125)	~ (126)	(127)

ASCII omits letters and symbols such as *ñ*, *ß*, *£*, and *¡* that are used in Western European, or Western Latin, languages. Several 8-bit encoding schemes with support for Western European character sets were developed in the 1980s. These schemes were incompatible with each other. The incompatibilities between these competing standards were confounded by the historical coincidence that the Euro symbol € was invented after the 8-bit encodings had gained traction. Microsoft was able to inject € into a blank space in encoding known as Code Page 1252; IBM modified its encoding known as Code Page 850, creating Code Page 858 to add €; and the ISO-8859-15 standard was created from ISO-8859-1 to support €. Table 4-2 lists six of the most common 8-bit encodings of Western Latin character sets. Table 4-3 shows examples of the incompatibilities between these encodings, by showing how they represent some uppercase, non-English letters. For example, the character *Ÿ* is missing from IBM Code Page 858 and ISO-8859-1 and is represented by *159* in Windows Code Page 1252, by *190* in ISO-8859-15, and by *217* in MacRoman.

Table 4-2. 8-Bit Western Latin encoding schemes

Encoding	Operating System	Region	Has €
Code Page 437	DOS	USA	N
Code Page 850	DOS	Europe	N
Code Page 858	DOS	Europe	Y
Code Page 1252	Windows		Y
ISO-8859-1	ISO Standard		N
ISO-8859-15	ISO Standard		Y
MacRoman	Macintosh		Y

Table 4-3. Encodings of some non-ASCII Western Latin letters

	Code Page			ISO-8859-x		Mac Roman	Unicode
	437	858	1252	-1	-15		
À		183	192	192	192	203	192
Æ	146	146	198	198	198	174	198
Ç	128	128	199	199	199	130	199
È		212	200	200	200	233	200
Ì		222	204	204	204	237	204
Ð		209	208	208	208		208
Ñ	165	165	209	209	209	132	209
Ö	153	153	214	214	214	133	214
Ù		235	217	217	217	244	217
Þ		232	222	222	222		222
ß	225	225	223	223	223	167	223
Œ			140		188	206	338
Š			138		166		352
Ÿ			159		190	217	376

In the late 1980s, as the problems caused by these incompatible encodings were recognized, work began on Unicode, a universal character encoding. Unicode is now well supported on all major computing platforms in use today. The original version of Unicode used a 16-bit encoding for each character. That was initially believed to provide enough code points (numeric encodings of symbols) to represent all characters in all languages that had been printed in the previous year. Unicode has since been expanded to support more than one million code points, more than one hundred thousand of which have been assigned. There are Unicode code pages (groups of code points) that represent both living and historic languages from around the globe. I can keep a list of my favorite drinks: *קוקה-קולה*, *weißbier*, *清酒*, *piña colada*, and so on in a single Unicode plain text file.

Strictly speaking, a Unicode string is a sequence of code points. Unicode code points are often written with the prefix *U+* in front of a hexadecimal number. For example, *U+41* specifies code point 0x41 and corresponds to the letter *A*. A serialization scheme is needed to map between sequences of code points and sequences of bytes. The two most popular Unicode serialization schemes are UTF-16 and UTF-8. Unicode code points U+D800 - U+DFFF have been permanently reserved for the UTF-16 encoding scheme. In UTF-16, code points U+0000 - U+D7FF and U+E000 - U+FFFF are written as they are (i.e., by the same 16-bit value). The remaining assigned code points fall in the range U+010000 - U+10FFFF, and are serialized to a pair of 16-bit values: the first from the range 0xD800 - 0xDBFF and the second from the range 0xDC00 - 0xDCFF. In UTF-8,

code points are serialized to between one and four 8-bit values. The number of bytes needed increases with the code point and was designed so that the ASCII characters can be written in a single byte (the same value as in ASCII), the Western European characters can be written in two or fewer bytes, and the most commonly used characters can be written in three or fewer bytes.

The widespread adoption of Unicode, UTF-8, and UTF-16 has greatly simplified text processing. Unfortunately, some legacy programs still generate output in other formats, and that can be a source of bad data. One of the trickiest cases has to do with confusion between Windows Code Page 1252 and ISO-8859-1. As seen in Table 4-3, many characters have the same representation in Code Page 1252, ISO-8859-1, and ISO-8859-15. The differences we've seen so far are in the letters added to ISO-8859-15 that were not present in ISO-8859-1. Punctuation is a different story. Code Page 1252 specifies 18 non-alphanumeric symbols that are not found in ISO-8859-1. These symbols are listed in Table 4-4. € is found in ISO-8859-15, but at a different code point (164). The other 17 are not found in ISO-8859-15 either. Some of the symbols that are representable by Code Page 1252 but not by ISO-8859-1 are the Smart Quotes—quote marks that slant towards the quoted text. Some Windows applications replace straight quotes with Smart Quotes as the user types. Certain versions of those Windows applications had export to XML or HTML functionality that incorrectly reported the ISO-8859-1 encoding used, when Code Page 1252 was the actual encoding used.

Table 4-4. Code Page 1252 symbols (conflicts with ISO-8859-1 & ISO-8859-15)

€ (128)	‚ (130)	„ (132)	… (133)	† (134)	‡ (135)
‰ (137)	‹ (139)	‘ (145)	’ (146)	“ (147)	” (148)
• (149)	– (150)	— (151)	˜ (152)	™ (153)	› (155)

Consider the example in Example 4-1. In this Python code, *s* is a byte string that contains the Code Page 1252 encodings of the smart quotes (code points 147 and 148) and the Euro symbol (code point 128). When we erroneously treat *s* as if it had been encoded with ISO-8859-1, something unexpected happens. Code points (128, 147, and 148) are control characters (not printable characters) in ISO-8859-1. Python prints them invisibly. The printed string appears to have only 11 characters, but the Python *len* function returns 14. Code Page 1252 encoding masquerading as ISO-8859-1 is bad data.

Example 4-1. Smart quotes and ISO-8859-1

```
>>> bytes = [45,147, 128, 53, 44, 32, 112, 108, 101,
... 97, 115, 101, 148,45]
>>> s = ''.join(map(chr, bytes))
>>> print s
-??5, please?-
>>> print(s.decode('cp1252'))
-“€5, please”-
>>> print(s.decode('iso-8859-1'))
```

```
-5, please-
>>> print(len(s.decode('cp1252')))
14
>>> print(len(s.decode('iso-8859-1')))
14
```

We've now seen that text with an unknown encoding can be bad data. What's more, text with a misrepresented encoding, as can happen with the Code Page 1252 / ISO-8859-1 mix-up, can also be bad data.

Guessing Text Encoding

The Unix *file* tool determines what type of data is in a file. It understands a wide variety of file types, including some plain text character encodings. The Python script in Example 4-2 generates some text data in different encodings. The function *make_alnum_sample* iterates through the first *n* Unicode code points looking for alpha-numeric characters. The parameter *codec* specifies an encoding scheme that is used to write out the alpha-numeric characters.

Example 4-2. Generating test data

```
>>> def make_alnum_sample(out, codec, n):
    """
    Look at the first n unicode code points
    if that unicode character is alphanumeric
    and can be encoded by codec write the encoded
    character to out
    """
    for x in xrange(n):
        try:
            u = unichr(x)
            if u.isalnum():
                bytes = u.encode(codec)
                out.write(bytes)
        except:
            # skip u if codec cannot represent it
            pass
    out.write('\n')

>>> codecs = ['ascii', 'cp437', 'cp858', 'cp1252',
... 'iso-8859-1', 'macroman', 'utf-8', 'utf-16']
>>> for codec in codecs:
   with open('../%s_alnum.txt' % codec, 'w') as out:
      make_alnum_sample(out, codec, 512)
```

The results of running the files generated in Example 4-2 through *file* are shown in Example 4-3. On my system, *file* correctly identified the ASCII, ISO-8859, UTF-8, and

UTF-16 files. *file* was not able to infer the type of the files containing bytes that map to alphanumeric characters in Code Page 1252, Code Page 437, Code Page 858, or MacRoman. From the output of *file*, we know that those files contain some 8-bit text encoding, but we do not yet know which one.

Example 4-3. Output from file command

```
$ file *alnum.txt
ascii_alnum.txt:      ASCII text
cp1252_alnum.txt:     Non-ISO extended-ASCII text
cp437_alnum.txt:      Non-ISO extended-ASCII text,
↪ with LF, NEL line terminators
cp858_alnum.txt:      Non-ISO extended-ASCII text,
↪ with LF, NEL line terminators
iso-8859-1_alnum.txt: ISO-8859 text
macroman_alnum.txt:   Non-ISO extended-ASCII text,
↪ with LF, NEL line terminators
utf-16_alnum.txt:     Little-endian UTF-16 Unicode
↪ text, with very long lines, with no line
↪ terminators
utf-8_alnum.txt:      UTF-8 Unicode text,
↪ with very long lines
```

One way to resolve this problem is to inspect snippets of text containing non-ASCII characters, manually evaluating how they would be generated by different encoding schemes. Python code that does this is shown in Example 4-4. Let's revisit the byte string from Example 4-1. Pretend for a moment that we don't know the contents of that string. Running it through *test_codecs* and *stream_non_ascii_snippets*, we see that sequence of bytes is valid in Code Page 858, Code Page 1252, and MacRoman. A human looking at the results of *test_codecs* can make a judgment as to which encoding makes sense. In this case, smart quotes and € contextually make more sense than the other options, and we can infer that the text is encoded by Code Page 1252.

Example 4-4. Snippets of non-ASCII text

```
>>> def stream_non_ascii_snippets(s, n_before=15,
... n_after=15):
    """
    s is a byte string possibly containing non-ascii
    characters
    n_before and n_after specify a window size

    this function is a generator for snippets
    containing the n_before bytes before a non-ascii
    character, the non-ascii byte itself, and the
    n_after bytes that follow it.
    """
    for idx, c in enumerate(s):
        if ord(c) > 127:
            start = max(idx - n_before, 0)
```

```
            end = idx + n_after + 1
            yield(s[start:end])
>>> CODECS = ['cp858', 'cp1252', 'macroman']
>>> def test_codecs(s, codecs=CODECS):
    """
    prints the codecs that can decode s to a Unicode
    string and those unicode strings
    """
    max_len = max(map(len, codecs))
    for codec in codecs:
        try:
            u = s.decode(codec)
            print(codec.rjust(max_len) + ': ' + u)
        except:
            pass
>>> bytes = [45,147, 128, 53, 44, 32, 112, 108, 101,
... 97, 115, 101, 148,45]
>>> s = ''.join(map(chr, bytes))
>>> test_codecs(next(stream_non_ascii_snippets(s)))
   cp858: -ôÇ5, pleaseö-
  cp1252: -“€5, please”-
macroman: -ìÄ5, pleaseî-
```

The *stream_non_ascii_snippets* function in Example 4-4 lets the user explore the non-ASCII bytes sequentially, in the order in which they occur in the byte string. An alternative, presented in Example 4-5, is to consider the frequency of occurrence of non-ASCII bytes in the string. The set of unique non-ASCII bytes might be enough to eliminate some encodings from consideration, and the user may benefit from inspecting snippets containing specific characters. The test string with which we've been working isn't the most interesting, because it is short and no non-ASCII character repeats. However, Example 4-5 shows how these ideas could be implemented.

Example 4-5. Frequency-count snippets of non-ASCII text

```
>>> from collections import defaultdict
>>> from operator import itemgetter
>>> def get_non_ascii_byte_counts(s):
    """
    returns {code point: count}
    for non-ASCII code points
    """
    counts = defaultdict(int)
    for c in s:
        if ord(c) > 127:
            counts[ord(c)] += 1
    return counts
>>> def stream_targeted_non_ascii_snippets(s,
... target_byte, n_before=15, n_after=15):
    """
    s is a byte string possibly containing non-ascii
    characters
```

```
    target_byte is code point
    n_before and n_after specify a window size

    this function is a generator for snippets
    containing the n_before bytes before
    target_byte, target_byte itself, and the n_after
    bytes that follow it.
    """
    for idx, c in enumerate(s):
        if ord(c) == target_byte:
            start = max(idx - n_before, 0)
            end = idx + n_after + 1
            yield(s[start:end])
>>> sorted(get_non_ascii_byte_counts(s).items(),
... key=itemgetter(1,0), reverse=True)
[(148, 1), (147, 1), (128, 1)]
>>> it = stream_targeted_non_ascii_snippets(s, 148,
... n_before=6)
>>> test_codecs(next(it))
   cp858: pleaseö-
  cp1252: please”-
macroman: pleaseî-
```

Normalizing Text

When mashing up data from multiple sources, it is useful to normalize them to either UTF-8 or UTF-16 depending on which is better supported by the tools you use. I typically normalize to UTF-8.

Let us revisit the file *macroman_alnum.txt* that was generated in Example 4-2. We know from its construction that it contains 8-bit MacRoman encodings of various alphanumeric characters. Example 4-6 shows standard Unix tools operating on this file. The first example, using *cat*, shows that the non-ASCII characters do not render correctly on my system. *iconv* is a Unix tool that converts between character sets. It converts *from* the encoding specified with the `-f` parameter *to* the encoding specified with `-t`. Output from *iconv* gets written to *STDOUT*. In Example 4-6, we allow it to print, and we see that the non-ASCII characters display correctly. In practice, we could redirect the output from *iconv* to generate a new file with the desired encoding.

Example 4-6. Normalizing text with Unix tools

```
$ cat macroman_alnum.txt
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnop
↪ qrstuvwxyz????????????????????????????????????????
↪ ????????????????????
$ iconv -f macroman -t utf-8 macroman_alnum.txt
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnop
↪ qrstuvwxyzªµºÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÑÒÓÔÕÖØÙÚÛÜßàáâãäåæçè
↪ éêëìíîïñòóôõöøùúûüÿıŒœŸƒ
```

Using *iconv* to re-encode text can be consistent with McIlroy's Unix philosophy. One might write a tool to perform some specific text processing operation to transform one UTF-8 document into another. We can design this tool to "do one thing and do it well." Its code can be simpler than a program that has to detect and normalize its input because it can always assume it is working with UTF-8. If it needs to process an non-UTF-8 dataset, that input can be piped through *iconv*. Similarly, its output needs to be consumed by a process with different encoding assumptions; it too can be piped through *iconv*.

Python has excellent support for decoding byte strings to Unicode. I often find it useful to write functions that "do one thing and do it well" operating on Unicode input, and then to wrap them with a bit of boilerplate to decode text from other formats. I tend to use this technique when I'm developing systems that allow their user to reference input that they do not control (for example, by passing in a URL). Two functions that decode byte strings to Unicode are *str.decode* and *codecs.open*. Both are illustrated in Example 4-7.

Example 4-7. Normalizing text from Python

```
>>> with open('macroman_alnum.txt') as f:
    print(f.readline())
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnop
↪ qrstuvwxyz?????? ????????????????□?????????????
↪ ??????????????????
>>> with open('macroman_alnum.txt') as f:
    print(f.readline().decode('macroman'))
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnop
↪ qrstuvwxyzªµºÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÑÒÓÔÕÖØÙÚÛÜßàáâãäåæçè
↪ éêëìíîïñòóôõöøùúûüÿıŒœŸƒ
>>> import codecs
>>> with codecs.open('macroman_alnum.txt',
... encoding='macroman') as f:
    print(f.readline())
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnop
↪ qrstuvwxyzªµºÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÑÒÓÔÕÖØÙÚÛÜßàáâãäåæçè
↪ éêëìíîïñòóôõöøùúûüÿıŒœŸƒ
```

The first example of Example 4-7 simply shows that the byte string for what we know to be MacRoman-encoded text does not render correctly within Python. In the second example, we use *str.decode* to produce the correct Unicode string, which in turn prints correctly. The last example of Example 4-7 uses *codecs.open* when reading the file. *codecs.open* is called in the same manner as the ordinary *open* function, and returns an object that behaves like an ordinary *file* except that it automatically decodes byte strings to Unicode when reading, and it automatically encodes Unicode to the specified encoding when writing. In the last example of Example 4-7, the call to *f.readline* returns a properly decoded Unicode string. The call to *print* handles the Unicode string correctly.

Thus far, we have discussed how text with an unknown or misrepresented character encoding can be bad data for a text processing application. A join operation using text

keys may not behave as expected if different encodings are used on its inputs. We have seen strategies for detecting character encoding using the Unix `file` command or custom Python code. We have also seen how text with a known representation can be normalized to a standard encoding such as UTF-8 or UTF-16 from Python code or by using *iconv* on the command line. Next, we will discuss another bad data problem: application-specific characters leaking into plain text.

Problem: Application-Specific Characters Leaking into Plain Text

Some applications have characters or sequences of characters with application-specific meanings. One source of bad text data I have encountered is when these sequences leak into places where they don't belong. This can arise anytime the data flows through a tool with a restricted vocabulary.

One project where I had to clean up this type of bad data involved text mining on web content. The users of one system would submit content through a web form to a server where it was stored in a database and in a text index before being embedded in other HTML files for display to additional users. The analysis I performed looked at dumps from various tools that sat on top of the database and/or the final HTML files. That analysis would have been corrupted had I not detected and normalized these application-specific encodings:

- URL encoding
- HTML encoding
- Database escaping

In most cases, the user's browser will URL encode the content before submitting it. In general, any byte can be URL encoded by % followed by its hex code. For example, *ȳ* is Unicode code point U+233. Its UTF-8 representation is two bytes: 0xC3A9, and the URL encoding of its UTF-8 representation is *%C3%A9*. URL encoding typically gets applied to non-ASCII characters if present, as well as to the following ASCII symbols, each of which has a special meaning when it appears in a URL: *;/?:@&=+$,*. Because *%* is used in the encoded string, raw *%* characters in the input need to be encoded. The ASCII code for *%* is *0x25*, so the encoding *%25* is used. The blank space is ASCII code 0x20, but it is typically encoded as + rather than *%20*. The server application must decode the URL encoding to recover exactly what the user has entered.

In general, URL encoding artifacts leaking into plain text is not a serious problem. URL encoding and decoding of form submissions happens behind the scenes in most web application frameworks. Even if URL encoded text did leak into other parts of an application, it would be easily detectable by the lack of spaces and abundance of + and

%XX codes. One notable exception is when analyzing a list of URLs or URL fragments. In that case, it may be worthwhile to ensure that all of the URLs have been decoded consistently. Example 4-8 uses Python's *urllib.urlencode* function to URL encode text, and then *urllib.unquote* functions to decode the URL encoded text.

Example 4-8. Decoding URL encoded text

```
>>> import urllib
# urlencode generates a query string from a dict
>>> urllib.urlencode({'eqn': '1+2==3'})
'eqn=1%2B2%3D%3D3'
# unquote decodes a URL encoded string
>>> s = 'www.example.com/test?eqn=1%2B2%3D%3D3'
>>> urllib.unquote(s)
'www.example.com/test?eqn=1+2==3'
```

Let's return to our application collecting user-generated content for inclusion on other web pages. Our users enter their content into a form, their browser URL encodes it at submission time, our application decodes it, and we have their content ready to display on one of our pages. If we write the submitted text as-is on our web pages, evil-doers will submit content containing their own HTML markup for things like link spamming, Rickrolling[1], or executing arbitrary JavaScript. Best practice for this type of application is to HTML encode user submissions so they get rendered as they were typed. HTML encoding a string replaces some characters with an entity reference: a sequence beginning with *&* followed by the name or code point of a character, followed by *;*. Example 4-9 gives an example of HTML encoding text.

Example 4-9. Decoding HTML encoded text

```
>>> import cgi
>>> import HTMLParser
>>> s = u'<script>//Do Some Évîl</script>'
>>> encoded = cgi.escape(s).encode('ascii',
... 'xmlcharrefreplace')
>>> print(encoded)
&lt;script&gt;//Do Some &#201;v&#238;l&lt;/script
↪ &gt;
>>> print(HTMLParser.HTMLParser().unescape(encoded))
<script>//Do Some Évîl</script>
```

The call to `cgi.escape` in Example 4-9 replaces the angle brackets < and > with the named entities `<` and `>` respectively. `unicode.encode(…, xmlcharrefre`

1. To "Rickroll" someone is to trick them into clicking a link that plays the music video for Rick Astley's song "Never Gonna Give You Up"

`place)` replaces the non-ASCII characters `É (U+C9)` and `î (U+EE)` with their numeric entities: `&201;` and `&238;` (0xC9 = 201, 0xEE=238). When a browser encounters the encoded string `<script>//Do Some &#201;v&#238;l</script>` it will display *<script>//Do Some Évîl</script>*, but it will not actually execute the evil script.

It is a reasonable decision to have a web application store HTML encoded strings in its database. That decision ensures that raw text submitted by the users won't appear in our other pages, and it may speed up the server-side rendering time for those pages. However, if we decide to text mine the user-submitted content, we'll need to understand how the content is formatted in database dumps, and we'll want to decode the HTML entity references before processing it.

I've actually seen redundantly HTML-encoded strings such as `&amp;lt;` in what was supposed to be a plain text dump. That data presumably passed through multiple web applications and databases before I got my hands on it. Example 4-10 expands on code from Example 4-9 to decode repeatedly HTML-encoded strings inside a `while` loop.

Example 4-10. Decoding redundantly HTML encoded text

```
>>> # add a few more layers of encoding
>>> ss = cgi.escape(encoded).encode('ascii',
... 'xmlcharrefreplace')
>>> ss = cgi.escape(ss).encode('ascii',
... 'xmlcharrefreplace')
>>> print(ss)
&amp;amp;lt;script&amp;amp;gt;//Do Some &amp;amp;
↪ #201;v&amp;amp;#238;l&amp;amp;lt;/script
↪ &amp;amp;gt;
>>> # now decode until length becomes constant
>>> while len(ss) != len(parser.unescape(ss)):
    ss = parser.unescape(ss)
    print(ss)
&amp;lt;script&amp;gt;//Do Some &amp;#201;v&amp;
↪ #238;l&amp;lt;/script&amp;gt;
&lt;script&gt;//Do Some &#201;v&#238;l&lt;/script
↪ &gt;
<script>//Do Some Évîl</script>
```

HTML encoding all user-submitted text is a step towards preventing malicious users from launching attacks when our pages are rendered. A well-engineered web application will also take precautions to protect itself from attacks that exploit its form submission handlers. A common example of such an attack is the SQL injection attack, where the attacker tries to trick a form handler into running user-supplied SQL statements. There is a brilliant example of a SQL injection attack in the famous XKCD comic about "Little Bobby Tables" (*http://xkcd.com/327/*).

The characters ', ;, --, and /* are often exploited in SQL injection attacks. They are used to terminate strings (') and statements (;), and to begin comments that span single (--)

or multiple lines (/*). There are two main strategies for defending against SQL injection attacks. The first uses a database feature called "prepared statements" that separates a SQL statement from its parameters, eliminating the possibility that a maliciously crafted parameter could terminate the original statement and launch an attack. When prepared statements are used, the special characters listed above can occur as-is in the database and dump files exported from the database. The second strategy is to detect and escape those special strings. When this technique is used, a text processing application operating on a dump from the database will need to decode the escaped strings back to their normal forms.

As we've seen, URL encoding, HTML encoding, and SQL escaping can all leak into text generated by a web application. Another case where encoding/decoding rules need to be implemented in text processing applications comes up when data is exported from a spreadsheet or database to a flat file such as CSV. Many tools will let the user specify which characters are used for field delimiters and which are used for quoting. Quoting is necessary if the export field delimiter happens to appear in the original data. In Example 4-11, we see a simple example simulating a dump of Name and Job Description from a database into a .CSV file

Example 4-11. Quoted CSV

```
>>> import StringIO
>>> import csv
>>> # s behaves like a file opened for reading
>>>  s = StringIO.StringIO('''Name,Job Description
"Bolton, Michael ""Mike""","Programmer"
Bolton,Michael "Mike",Programmer''')
>>> # When we count the fields per line,
>>> # str.split is confused by Name
>>> map(len, [line.split(',') for line in s])
[2, 3, 3]
>>> # csv.reader understands quoted name
>>> s.seek(0)
>>> map(len, csv.reader(s))
[2, 2, 3
>>> s.seek(0)
>>> data = [row for row in csv.reader(s)]
>>> # with quotes the comma in the name
>>> # is not a delimiter
>>> data[1][0]
'Bolton, Michael "Mike"'
>>> # without quotes all commas are delimiters
>>> data[2][0]
'Bolton'
```

The difference between the data rows of *s* in Example 4-11 is that values are quoted in the first, similar to what MySQL's `mysqldump --fields-enclosed-by="` would produce. Values in the second data row are not quoted. The Python functions `str.split` and

`unicode.split` are simple ways to extract fields from a line of comma-delimited text. They treat all commas as delimiters, a behavior that is incorrect for this data, where the name field contains a non-delimiting comma. Python's `csv.reader` allows the non-delimiting commas to occur within quoted strings, so it correctly parses the data line where the values are quoted. Mismatched columns when parsing delimited text is a bad data problem. I recommend quoting when exporting text from a database or spreadsheet, and using `csv.reader` rather than `str.split` as a parser.

If we don't understand the processes that create plain text files, application-specific characters may inadvertently leak in and affect text analysis. By understanding these processes and normalizing text before working with it, we can avoid this type of bad data.

Text Processing with Python

We've discussed bad data problems caused by unknown or misrepresented text encodings. We've also discussed bad data problems caused by application-specific encodings or escape characters leaking into plain text dumps. We've used small examples written in Python to expose these problems and their solutions. Table 4-5 summarizes the Python functions we've used in these demonstrations.

Table 4-5. Python reference

Function	Notes	Listings
str.decode	Converts byte string to Unicode string	Example 4-1
		Example 4-4
		Example 4-7
unicode.encode	Converts Unicode string to byte string	Example 4-2
unichr	Maps number (Unicode code point) to character	Example 4-2
ord	Gets number (code point) from byte string or Unicode strings	Example 4-5
codecs.open	Reads and decodes Unicode strings from a file of byte strings	Example 4-7
urllib.urlencode	URL encodes a dictionary	Example 4-8
urllib.unquote	Decodes URL-encoded string	Example 4-8
cgi.escape	HTML encodes characters. Used with unicode.encode('ascii', 'xmlcharrefreplace')	Example 4-9
		Example 4-10
HTMLParser.unescape	Decodes HTML-encoded string	Example 4-9
		Example 4-10
csv.reader	Parses delimited text	Example 4-11

The functions listed in Table 4-5 are good low-level building blocks for creating text processing and text mining applications. There are a lot of excellent Open Source Python libraries for higher level text analysis. A few of my favorites are listed in Table 4-6.

Table 4-6. Third-party Python reference

Library	Notes
NLTK	Parsers, tokenizers, stemmers, classifiers
BeautifulSoup	HTML & XML parsers, tolerant of bad inputs
gensim	Topic modeling
jellyfish	Approximate and phoenetic string matching

These tools provide a great starting point for many text processing, text mining, and text analysis applications.

Exercises

1. The results shown in Example 4-3 were generated when the *n* parameter to *make_alnum_sample* was set to *512*. Do the results change for other values of *n*? Try with *n=128*, *n=256*, and *n=1024*.
2. Example 4-4 shows possible interpretations of the string *s* for three character encodings: Code Page 858, Code Page 1252, and MacRoman. Is every byte string valid for all three of those encodings? What happens if you run *test_codecs(''.join(map(chr, range(256))))*.
3. Example 4-6 shows *iconv* converting text from MacRoman to UTF-8. What happens if you try to convert the same text from MacRoman to ASCII? What if you try to convert the same text from ISO-8859-1 to UTF-8?

The Office of the Superintendent of Financial Institutions, Canada publishes a list of individuals connected to terrorism financing. As of July 9, 2012, the list can be downloaded from *http://bit.ly/S1l9WK*. The following exercises refer to that list as "the OSFI list."

4. How is the OSFI list encoded?
5. What are the most popular non-ASCII symbols (not necessarily letters) in the OSFI list?
6. What are the most popular non-ASCII letters in the OSFI list?
7. Is there any evidence of URL encoding in the OSFI list?
8. Is there any evidence of HTML encoding in the OSFI list?