ISSN 2161-3974

JeSLIB 2021; 10(3): e1211
https://doi.org/10.7191/jeslib.2021.1211

Journal of eScience Librarianship
putting the pieces together: theory and practice

Full-Length Paper

Plain Text & Character Encoding: A Primer for Data Curators

Seth Erickson

The Pennsylvania State University, University Park, PA, USA

Abstract

Plain text data consists of a sequence of encoded characters or "code points" from a given standard such as the Unicode Standard. Some of the most common file formats for digital data used in eScience (CSV, XML, and JSON, for example) are built atop plain text standards. Plain text representations of digital data are often preferred because plain text formats are relatively stable, and they facilitate reuse and interoperability. Despite its ubiquity, plain text is not as plain as it may seem. The set of standards used in modern text encoding (principally, the Unicode Character Set and the related encoding format, UTF-8) have complex architectures when compared to historical standards like ASCII. Further, while the Unicode standard has gained in prominence, text encoding problems are not uncommon in research data curation. This primer provides conceptual foundations for modern text encoding and guidance for common curation and preservation actions related to textual data.

Correspondence: Seth Erickson: sre53@psu.edu

Received: April 8, 2021 **Accepted**: June 4, 2021 **Published**: August 11, 2021

Disclosures: The author reports no conflict of interest.

https://doi.org/10.7191/jeslib.2021.1211

Introduction

Character encoding is an often-unconsidered aspect of day-to-day computing. The particularities of encoding, decoding, and displaying textual data can be taken for granted much of the time. However, when things go wrong—when the characters on the screen are very clearly the *wrong* characters—few technical problems provoke the same degree of frustration and anxiety.

Text encoding errors are unsettling in part because plain text is meant to be stable and ubiquitous—it's *just text*, without any formatting or fancy embedded media. What could be simpler? Digital preservationists and data curators hold plain text formats in high regard because they require less specialized software than binary formats. Sophisticated tools for displaying, editing, searching, filtering, and generally "wrangling" plain text are commonplace on modern computers. Similarly, programmers and system designers rely on plain text as a "universal interface" that facilitates interoperability between systems (Raymond 2003). It's not an exaggeration to say that plain text is the glue that binds our digital ecology. Some of the most common file formats for digital data used in eScience (CSV, XML, and JSON, for example) are built atop plain text standards, meaning they are defined as streams of human-readable textual elements.

In addition to providing some guidance for curating plain text and working with character encoding standards, this primer presents a basic argument: plain text isn't so *plain*. Modern text encoding standards are complex when compared to historical standards. Further, assumptions rooted in historical standards can interfere with the effective treatment of textual data. For example, plain text is often conflated with ASCII text (in fact, ASCII is just one plain text standard). This assumption persists, in part because Unicode (the primary modern text standard) was purposefully designed *not* to interfere with it: UTF-8 is backward compatible with ASCII. An unfortunate consequence of this design choice is that an outdated understanding of plain text, one rooted in the legacy of ASCII, has persisted longer than it ought to have. Working with textual data, today, requires an understanding of character encoding that goes beyond the "plainness" of ASCII.

This primer addresses the challenges of curating and preserving plain text in two parts. The first two sections provide conceptual foundations for plain text and introduce key terms from the Unicode Standard. The Unicode Character Set (ISO/IEC 10646) and related formats (UTF-8 and UTF-16) are the focus here because they are central to modern text encoding. Subsequent sections provide guidance for common curation and preservation actions related to textual data: (1) identifying the encoding standard for source data of various types and (2) transforming data to format that supports the Unicode character set.

What is Plain Text?

Plain text is the rudimentary representation of text by a computer. It is rudimentary in the sense that text is modeled as a linear sequence of symbols.

You won't find hierarchical elements like pages and chapters defined by plain text standards. At a low level, the things that computers process, store, and transmit, are composed of bundles of binary information, or bytes. Plain text formats like UTF-8, ASCII, and ISO-8859-1 each define a way to interpret a sequence of bytes as a sequence of written elements. What these written elements consist of—be they characters, accent marks, punctuations, ligatures, or emojis—varies from standard to standard. The Unicode Standard names the constituent elements of plain text *code points*. Plain text is defined more formally as "[c]omputer-encoded text that consists only of a sequence of code points from a given standard, with no other formatting or structural information" (The Unicode Consortium, n.d.).

From ASCII to Unicode

One of the most historically significant text standards, the American Standard Code for Information Interchange (ASCII), defines 128 codes points corresponding to the letters, numerals, punctuation symbols, and device control characters commonly used by US computer manufactures and telecommunications companies in the early 1960s. Because it is based on the English alphabet, ASCII does not support accented characters or characters from non-Latin writing systems. To support writing systems excluded by ASCII, additional standards were needed. This led to an explosion in the number of encoding formats technology companies needed to support if they hoped to reach broad, international markets. In the 1980s, tech companies (initially Xerox and Apple) coordinated their efforts and started work on a single "universal character set" that they hoped would support all the world's writing systems. This effort culminated with the publication of the first version of the Unicode Standard in 1991.

The Unicode Standard: a very brief introduction

The Unicode Standard defines a "universal character set"—a repertoire of letters, symbols, ideograms, and other types of characters that can be used in plain text data (The Unicode Consortium 2020). Unlike ASCII, the Unicode character set can be expanded. The current version (13.0.0, as of this writing) consists of 143,859 coded characters covering 154 modern and historical scripts. Since the publication of the first version in 1991, Unicode's adoption has steadily grown. Today, the vast majority of web content is encoded with UTF-8, one of several *transformation formats* through which the standard is implemented (W3Techs 2017). Unicode's universality—the fact that it aims to support all existing and historical writing systems—comes with added complexity, particularly when compared to historical standards like ASCII. The following sections introduce some of the fundamental aspects of the standard's design.

Code Points and Abstract Characters

A Unicode code point is a numerical value between 0 and 1,114,112 (in decimal), or 0x0 and 0x10FFFF (in hexadecimal). A common convention, used here, is to present code point values in hexadecimal notation, preceded by 'U+'. In the

https://doi.org/10.7191/jeslib.2021.1211

standardization process, values in this range are assigned to *abstract characters*, "unit[s] of information used for the organization, control, or representation of textual data" (The Unicode Consortium 2020). The assignment of a code point to an abstract character results in a *coded character*. Generally speaking, successive versions of the Unicode standard introduce new coded characters that can be used to compose plain text data.

Table 1: Four coded characters (code point/abstract character pairs) from the Unicode character set

Glyph	Code Point	Abstract Character Name
A	U+0041	LATIN CAPITAL LETTER A
Ä	U+00C4	LATIN CAPITAL LETTER A WITH DIAERESIS
◌̈	U+0308	COMBINING DIAERESIS
fi	U+FB01	LATIN SMALL LIGATURE FI

Abstract characters do not necessarily correspond to units of text as one might expect. Table 1 illustrates abstract characters that might otherwise be regarded as either *smaller* than a character (the diaeresis mark, ◌̈) or 'larger' (the ligature, fi). Further, abstract characters can be combined to produce, for example, accented characters: the capital letter 'A' with diaeresis (Ä) can be encoded in Unicode as *either* a single code point (U+00C4) or as the composite of two code points (U+0041 followed by U+0308). Much more can be said about code points, abstract characters, and composite characters. For more on this, see (The Unicode Consortium 2019, Ch. 2).

Unicode's Encoding Formats: UTF-8 and UTF-16

The Unicode Character Set is one of two standards involved in the encoding and decoding of textual data; it defines the collection of abstract characters that may be used in plain text data. However, the actual representation of code point values in data is handled by a separate standard: a character encoding format. Several encoding formats implement the Unicode Character Set but the most well-known are UTF-8 and UTF-16. These differ in the way they represent a sequence of code points as a byte sequence that can be stored as a file or transferred over a network.

UTF-8

UTF-8 is a variable width encoding format, meaning the number of bytes used to represent a code point varies based on the code point's numerical value. UTF-8 uses one byte to represent each of the first 128 code points (U+0000 - U+007F),

the range corresponding to the US-ASCII character set. Code points in the range U+0080 - U+07FF are encoded with two bytes; U+0800 through U+FFFF require three bytes; and four bytes are required for code points between U+10000 and U+10FFFF. UTF-8's chief advantage is that it is backward compatible with US-ASCII.

UTF-16

UTF-16 is a variable width encoding format (like UTF-8) that uses two bytes to represent code points between U+0000 and U+FFFF (however U+D800 - U-DBFF are excluded). Four bytes are required for code points in the range U+010000 through U+10FFFF.UTF-16 is less efficient than UTF-8 for encoding written English, but more efficient for writing systems using coded characters in the U+0800 to U+FFFF range. Many of the most widely used writing systems, including Chinese, Japanese, Korean, and Devanagari, are more efficiently encoded with UTF-16 than UTF-8.

Curation and Preservation Concerns

Preferred Formats

UTF-8 and UTF-16 are the US Library of Congress's recommended formats for textual data (Library of Congress, n.d.). UTF-8 and UTF-16 are based on the Unicode Character Set, so they can be used to encode the same character information. UTF-8 is currently the dominant text encoding format on the web, and newer software applications often use it as the default format for plain text data (W3Techs 2017). There are, however, good reasons to prefer UTF-16 over UTF-8 in some circumstances. As described above, UTF-16 is more efficient than UTF-8 for the purposes of encoding characters used in the some of the world's most widely adopted writing systems (those of South and East Asia, particularly). Data in which these characters are heavily used may require less storage space when encoded with UTF-16 than it would with UTF-8.

Format Identification

Identifying the format that plain text data was encoded with is necessary for using the data effectively. Today, it is often safe to assume that plain text is encoded using UTF-8, due to its increased adoption as a default text format in operating systems and software applications. Nevertheless, curators and preservationists—particularly those working with historical data or data produced with older software—should be prepared to confirm the format for textual data.

The text format may be included in the metadata or the format can be inferred from other features of the data. High-level formats and protocols based on plain text often require the use of a particular encoding format or they include mechanisms to declare the format within the data itself. Some examples are listed below.

JeSLIB 2021; 10(3): e1211
https://doi.org/10.7191/jeslib.2021.1211

- *XML*: The prologue section of an XML document should include the character encoding used in the remainder of the document. For example: <?xml version="1.0" encoding="utf-8"?>
- *HTML*: HTML documents often include a meta tag defining the content's character encoding, for example: <meta http-equiv="content-type" content="text/html; charset=UTF-8" />
- *Excel spreadsheet*: Since 2010, Excel spreadsheets (.xslx files) are XML files, encoded with UTF-8.
- *JSON*: The JSON (Javascript Object Notation) standard states that valid JSON objects are encoded with UTF-8.

Note, however, that the declared format, or the format inferred by the file type, may not correspond to the format used to encode the data. Mismatch between the declared format and the actual format is a source of text encoding problems, described below.

Identification of the encoding format based on the content of the data itself is often necessary. Many common file formats do not provide a mechanism to declare the text encoding format, nor do they require the use of a particular encoding format. One of the most widely used data standards, .CSV (comma separated values), does not include a way of specifying how the text is encoded. Further, even in cases where the encoding format is declared (for example, in an XML file's prologue), the file's content may have been encoded with a different format than the one specified.

Many software tools, like web browsers, word processors, and text editors include features to guess the format of a given piece of text data. Plain text editors oriented toward software development often include this feature. The file command, which can be used from the terminal on Mac OS, Linux, and other Unix-based operating systems, tries to guess a text file's format based on analysis of the first portion of the file. Finally, programming libraries, like chardet for Python, can be used to build scripts and software tools that perform auto-detection.

Text Encoding Problems

Invalid Text Data

When plain text data is decoded using a different standard than the one used to encode it, the text may appear garbled or corrupt. This problem is sometimes referred to as "Mojibake." One way Mojibake manifests is with the appearance of the replacement character (�) in unexpected parts of the text. The replacement character is a special character used by decoding software to indicate invalid input data. For example, if the text "Andrés" is *encoded* with Windows-1252 (a format used in older versions of Microsoft Windows) and then *decoded* with UTF-8, the text may appear as 'Andr�s' because the byte used to represent "é" in

https://doi.org/10.7191/jeslib.2021.1211

Windows-1252 is not valid UTF-8.

```
# Command Line Example (using Mac OS Terminal)
# Terminal decodes as UTF-8, replacement character appears as '?'
# The file data.txt contains an invalid codepoint ("?")
$ cat data.txt
> Andr?s, 1920, ...

# We can test whether the file is valid UTF-8 (it isn't)
$ iconv -f UTF-8 data.txt
> iconv: data.csv:1:4: incomplete character or shift sequence

# Guess the format with the file command
$ file data.txt
> data.txt: ISO-8859 text, with no line terminators

# Use iconv to re-encode as UTF-8 and display data correctly.
$ iconv -f ISO-8859-1 -t UTF-8 data.txt
> Andrés, 1920, ...
```

Misinterpreted Text Data

Plain text data can be technically valid in two different encoding formats but yield different characters depending on which format is used to decode it. This can cause decoding errors that are more difficult to spot than those caused by invalid input data because they do not result in tell-tale replacement characters. Quick format validation checks of the text won't help either. For example, when the left double quotation mark (“) is encoded with UTF-8 and decoded with Windows-1252, the characters (â€œ) will appear instead of the quotation mark. This is because the sequence of bytes used by UTF-8 to represent (“) is the same as the sequence of bytes representing (â€œ) in Windows-1252. The data is *technically* valid in both formats, however its interpretation as UTF-8 is perhaps more linguistically meaningful.

Invalid text data and misinterpreted text data are symptoms of the same underlying issue: the data was decoded with a different format than the one used to encode it; both problems can be addressed by identifying the correct format for decoding the data (see the previous section) and, if necessary, re-encoding the text to a preferred format. (Some techniques for re-encoding are discussed in the next section).

The python library ftfy (Speer 2019) can be used to identify and correct mojibake of this variety.

JeSLIB 2021; 10(3): e1211
https://doi.org/10.7191/jeslib.2021.1211

```
>>> print(fix_text('This text should be in â€œquotesâ€\x9d.'))
This text should be in "quotes".
```

Mixed Format Data

When text data encoded with different formats are combined in the same file or data stream there is no "correct" format that applies to the data as a whole. As a result, decoding the data with one format might work for one part of the text but break another. This problem often plagues websites where user-generated content using different standards is combined on the same page. The solution—a tedious one—is to identify distinct segments of the data that share a common format and re-encode each segment until the entire byte stream shares a single format.

Filenames

While the adoption of the Unicode standard has greatly facilitated the range of possible characters that can be represented in textual data, filenames introduce additional considerations worth mentioning. The file system is the part of the operating system that determines how files are named and identified. Different file systems have different rules concerning about how files are named and the characters that can be included. In addition, correcting text encoding errors in filenames can be challenging because the same tools used to re-encode file *contents* don't help with file *names* (Blewer 2019).

The convmv command can be used on Unix-based operating systems (Linux and MacOS) to re-encode filenames:

```
$ convmv -f OLD_ENCODING -t UTF-8 old-file.txt -o new-file.txt
```

Digital preservation and curation workflows often recommend removing non-ASCII characters from filenames. However, as Arroyo-Ramírez (2016) notes, the practice of removing "illegal" (i.e., accented) characters from filenames can negatively impact the context and meaning of the files.

Tools for Working with Plain Text Data

`file` is a command line program available on Mac OS, Linux, and other Unix-based operating systems that characterizes files. It can be used to guess the format for a text file.

```
# identify the format of the data.txt file
$ file data.txt
> data.txt: ISO-8859 text, with no line terminators
```

`iconv` is a command line program (for Mac OS, Linux, and other Unix-based operating systems) that is used to convert text from one encoding to another. In the example here, it is used to re-encode a file using Window-1225 as UTF-8. (You

https://doi.org/10.7191/jeslib.2021.1211

can get a list of supported formats with the `iconv -l` command).

```
# Convert data.csv from Windows-1252 to UTF-8, save as data.utf8.csv
$ iconv -f WINDOWS-1252 -t UTF-8 data.csv > data.utf8.csv
```

`iconv` can also be used to test whether a file's contents are valid with respect to a particular format. Note that validating a file with respect to a given format does not guarantee that the file was originally encoded with that format; it only means the file could be decoded without error.

```
# Test whether data.csv is valid UTF-8 (file contains invalid UTF-8)
$ iconv -f UTF-8 data.csv
> iconv: data.csv:17:16: cannot convert
```

`convmv` is a command line program available on Mac OS, Linux, and other Unix-based operating systems that can be used to rename and re-encode filenames:

```
$ convmv -f OLD_ENCODING -t UTF-8 old-file.txt -o new-file.txt
```

Conclusion & Recommendations

This primer has introduced modern text encoding with Unicode to help curators resolve potential problems in research data sets. Working with research data often means working with plain text–common data formats, like XML, JSON, and CSV, are based on plain text standards. While the adoption of Unicode has helped alleviate the challenge of conflicting text standards, text encoding errors are not uncommon in data curation work. Further, as this primer has shown, the Unicode standard is significantly more complex than historical standards like ASCII. Unicode is an expandable, "universal" character set with multiple encoding formats, including UTF-8 and UTF-16.

The following recommendations provide guidance for curators working with text-based datasets.

- Unless circumstances demand otherwise, plain text data should be encoded using a Unicode format (e.g., UTF-8). UTF-8 is generally recommended over UTF-16.
- When working with text-based formats that include an encoding declaration (such as XML's encoding attribute), confirm that the data was encoded using the format in the declaration.
- When working with text-based formats that require a particular encoding format (such as JSON, which requires UTF-8), confirm that the data was encoded using the appropriate format.
- When working with text-based formats that do not declare or imply the

underlying text format (like CSV), curators should expect the data to be encoded with a preferred encoding format (UTF-8 or UTF-16) and they should confirm that this is the case.

- When possible, curators should work with researchers to resolve text encoding problems in the researcher's data sets. If curators must transform or convert previously deposited data to resolve text encoding problems, original versions should always be kept.

References

Arroyo-Ramírez, Elvia. 2016. "Invisible Defaults and Perceived Limitations: Processing the Juan Gelman Files." *Medium*. https://medium.com/on-archivy/invisible-defaults-and-perceived-limitations-processing-the-juan-gelman-files-4187fdd36759

Blewer, Ashley. 2019. "Artist_Exhibition-Copy (FINAL)(2).Mov: Preserving Diacritics in Filenames as Significant Properties in Media Conservation." *Ashley Blewer Blog*. https://bits.ashleyblewer.com/blog/2019/06/17/artist-exibition-copy-final-2-preserving-diacritics-in-filenames-as-significant-properties-in-media-conservation

Library of Congress. n.d. "Recommended Formats Statement." https://www.loc.gov/preservation/resources/rfs/index.html

Raymond, Eric S. 2008. *The art of UNIX programming*. Addison-Wesley.

Speer, Robyn. 2019. "Ftfy (Version 5.5.1)." Zenodo. https://doi.org/10.5281/zenodo.2591652

The Unicode Consortium. 2019. The Unicode Standard Version 12.1.0. Mountain View, CA: Unicode Consortium.

———. 2020. The Unicode Standard Version 13.0.0. Mountain View, CA: Unicode Consortium

———. n.d. "Glossary of Unicode Terms." https://www.unicode.org/glossary

W3Techs. 2017. "Historical Trends in the Usage of Character Encodings, September 2017." https://w3techs.com/technologies/history_overview/character_encoding