ftfy Documentation

Release 4.0

Rob Speer

February 25, 2016

`ftfy` fixes Unicode that's broken in various ways. It works in Python 2.7, Python 3.2, or later.

The goal of ftfy is to **take in bad Unicode and output good Unicode**, for use in your Unicode-aware code. This is different from taking in non-Unicode and outputting Unicode, which is not a goal of ftfy. It also isn't designed to protect you from having to write Unicode-aware code. ftfy helps those who help themselves.

Of course you're better off if your input is decoded properly and has no glitches. But you often don't have any control over your input; it's someone else's mistake, but it's your problem now.

`ftfy` will do everything it can to fix the problem.

Note: Time is marching on. ftfy 4.x supports Python 2.7 and 3.x, but when ftfy 5.0 is released, it will probably only support Python 3.

If you're running on Python 2, ftfy 4.x will keep working for you. You don't have to upgrade to 5.0. You can save yourself a headache by adding `ftfy < 5` to your requirements, making sure you stay on version 4.

See *Future versions of ftfy* for why this needs to happen.

Mojibake

The most interesting kind of brokenness that ftfy will fix is when someone has encoded Unicode with one standard and decoded it with a different one. This often shows up as characters that turn into nonsense sequences (called "mojibake"):

- The word `schön` might appear as `schÃ¶n`.
- An em dash (`--`) might appear as `â€”`.
- Text that was meant to be enclosed in quotation marks might end up instead enclosed in `â€œ` and `â€<9d>`, where `<9d>` represents an unprintable character.

This causes your Unicode-aware code to end up with garbage text because someone else (or maybe "someone else") made a mistake.

This happens very often to real text. It's often the fault of software that makes it difficult to use UTF-8 correctly, such as Microsoft Office and some programming languages. The *`ftfy.fix_encoding()`* function will look for evidence of mojibake and, when possible, it will undo the process that produced it to get back the text that was supposed to be there.

Does this sound impossible? It's really not. UTF-8 is a well-designed encoding that makes it obvious when it's being misused, and a string of mojibake usually contains all the information we need to recover the original string.

When ftfy is tested on multilingual data from Twitter, it has a false positive rate of less than 1 per million tweets.

CHAPTER 2

Other fixes

Any given text string might have other irritating properties, possibly even interacting with the erroneous decoding. The main function of ftfy, `ftfy.fix_text()`, will fix other problems along the way, such as:

- The text could contain HTML entities such as `&amp;` in place of certain characters, when you would rather see what the characters actually are.
- For that matter, it could contain instructions for a text terminal to do something like change colors, but you are not sending the text to a terminal, so those instructions are just going to look like `^[[30m;` or something in the middle of the text.
- The text could write words in non-standard ways for display purposes, such as using the three characters ` o p` for the word "flop". This can happen when you copy text out of a PDF, for example.
- It might not be in *NFC normalized* form. You generally want your text to be NFC-normalized, to avoid situations where unequal sequences of codepoints can represent exactly the same text. You can also opt for ftfy to use the more aggressive *NFKC normalization*.

Note: Before version 4.0, ftfy used NFKC normalization by default. This covered a lot of helpful fixes at once, such as expanding ligatures and replacing "fullwidth" characters with their standard form. However, it also performed transformations that lose information, such as converting ™ to `TM` and `H2O` to `H2O`.

The default, starting in ftfy 4.0, is to use NFC normalization unless told to use NFKC normalization (or no normalization at all). The more helpful parts of NFKC are implemented as separate, limited fixes.

There are other interesting things that `ftfy` can do that aren't part of the `ftfy.fix_text()` pipeline, such as:

- `ftfy.explain_unicode()`: show you what's going on in a string, for debugging purposes
- `ftfy.fixes.decode_escapes()`: do what everyone thinks the built-in `unicode_escape` codec does, but this one doesn't *cause* mojibake

CHAPTER 3

Encodings ftfy can handle

`ftfy` can't fix all possible mix-ups. Its goal is to cover the most common encoding mix-ups while keeping false positives to a very low rate.

`ftfy` can understand text that was decoded as any of these single-byte encodings:

- Latin-1 (ISO-8859-1)
- Windows-1252 (cp1252 – used in Microsoft products)
- Windows-1251 (cp1251 – the Russian version of cp1252)
- MacRoman (used on Mac OS 9 and earlier)
- cp437 (used in MS-DOS and some versions of the Windows command prompt)

when it was actually intended to be decoded as one of these variable-length encodings:

- UTF-8
- CESU-8 (what some people think is UTF-8)

It can also understand text that was intended as Windows-1252 but decoded as Latin-1. That's the very common case where things like smart-quotes and bullets turn into single weird control characters.

However, ftfy cannot understand other mixups between single-byte encodings, because it is extremely difficult to detect which mixup in particular is the one that happened.

We also can't handle the non-UTF encodings used for Chinese, Japanese, and Korean, such as `shift-jis` and `gb18030`. See issue #34 for why this is so hard.

But remember that the input to `ftfy` is Unicode, so it handles actual CJK *text* just fine. It just can't discover that a CJK *encoding* introduced mojibake into the text.

CHAPTER 4

Using ftfy

The main function, *ftfy.fix_text()*, will run text through a sequence of fixes. If the text changed, it will run them through again, so that you can be sure the output ends up in a standard form that will be unchanged by *ftfy.fix_text()*.

All the fixes are on by default, but you can pass options to turn them off. Check that the default fixes are appropriate for your use case. For example:

- You should set `fix_entities` to False if the output is meant to be interpreted as HTML.
- You should set `fix_character_width` to False if you want to preserve the spacing of CJK text.
- You should set `uncurl_quotes` to False if you want to preserve quotation marks with nice typography. You could even consider doing quite the opposite of `uncurl_quotes`, running smartypants on the result to make all the punctuation nice.

If the only fix you need is to detect and repair decoding errors (mojibake), then you should use *ftfy.fix_encoding()* directly.

Changed in version 4.0: The default normalization was changed from `'NFKC'` to `'NFC'`. The new options *fix_latin_ligatures* and *fix_character_width* were added to implement some of the less lossy parts of NFKC normalization on top of NFC.

`ftfy.`**`fix_text`**`(`*text, fix_entities='auto', remove_terminal_escapes=True, fix_encoding=True, fix_latin_ligatures=True, fix_character_width=True, uncurl_quotes=True, fix_line_breaks=True, fix_surrogates=True, remove_control_chars=True, remove_bom=True, normalization='NFC', max_decode_length=1000000*`)`

Given Unicode text as input, fix inconsistencies and glitches in it, such as mojibake.

Let's start with some examples:

```
>>> print(fix_text('uÌˆnicode'))
ünicode
```

```
>>> print(fix_text('Broken text&hellip; it&#x2019;s ﬂubberiﬁc!',
...                normalization='NFKC'))
Broken text... it's flubberific!
```

```
>>> print(fix_text('HTML entities &lt;3'))
HTML entities <3
```

```
>>> print(fix_text('<em>HTML entities &lt;3</em>'))
<em>HTML entities &lt;3</em>
```

```
>>> print(fix_text('\001\033[36;44mI&#x92;m blue, da ba dee da ba '
...                'doo&#133;\033[0m', normalization='NFKC'))
I'm blue, da ba dee da ba doo...
```

```
>>> # This example string starts with a byte-order mark, even if
>>> # you can't see it on the Web.
>>> print(fix_text('\ufeffParty like\nit&rsquo;s 1999!'))
Party like
it's 1999!
```

```
>>> print(fix_text(''))
LOUD NOISES
```

```
>>> len(fix_text('' * 100000))
200000
```

```
>>> len(fix_text(''))
0
```

Based on the options you provide, ftfy applies these steps in order:

- If `fix_entities` is True, replace HTML entities with their equivalent characters. If it's "auto" (the default), then consider replacing HTML entities, but don't do so in text where you have seen a pair of actual angle brackets (that's probably actually HTML and you shouldn't mess with the entities).
- If `remove_terminal_escapes` is True, remove sequences of bytes that are instructions for Unix terminals, such as the codes that make text appear in different colors.
- If `fix_encoding` is True, look for common mistakes that come from encoding or decoding Unicode text incorrectly, and fix them if they are reasonably fixable. See `fixes.fix_encoding` for details.
- If `uncurl_quotes` is True, replace various curly quotation marks with plain-ASCII straight quotes.
- If `fix_latin_ligatures` is True, then ligatures made of Latin letters, such as , will be separated into individual letters. These ligatures are usually not meaningful outside of font rendering, and often represent copy-and-paste errors.
- If `fix_character_width` is True, half-width and full-width characters will be replaced by their standard-width form.
- If `fix_line_breaks` is true, convert all line breaks to Unix style (CRLF and CR line breaks become LF line breaks).
- If `fix_surrogates` is true, ensure that there are no UTF-16 surrogates in the resulting string, by converting them to the correct characters when they're appropriately paired, or replacing them with ufffd otherwise.
- If `fix_control_characters` is true, remove all C0 control characters except the common useful ones: TAB, CR, LF, and FF. (CR characters may have already been removed by the `fix_line_breaks` step.)
- If `remove_bom` is True, remove the Byte-Order Mark if it exists. (This is a decoded Unicode string. It doesn't have a "byte order".)
- If `normalization` is not None, apply the specified form of Unicode normalization, which can be one of 'NFC', 'NFKC', 'NFD', and 'NFKD'.
  - The default normalization, NFC, combines characters and diacritics that are written using separate code points, such as converting "e" plus an acute accent modifier into "é", or converting "ka" () plus a dakuten into the single character "ga" (). Unicode can be converted to NFC form without any change in its meaning.

– If you ask for NFKC normalization, it will apply additional normalizations that can change the meanings of characters. For example, ellipsis characters will be replaced with three periods, all ligatures will be replaced with the individual characters that make them up, and characters that differ in font style will be converted to the same character.

- If anything was changed, repeat all the steps, so that the function is idempotent. "&amp;amp;" will become "&", for example, not "&amp;".

`fix_text` will work one line at a time, with the possibility that some lines are in different encodings, allowing it to fix text that has been concatenated together from different sources.

When it encounters lines longer than `max_decode_length` (1 million codepoints by default), it will not run the `fix_encoding` step, to avoid unbounded slowdowns.

If you're certain that any decoding errors in the text would have affected the entire text in the same way, and you don't mind operations that scale with the length of the text, you can use `fix_text_segment` directly to fix the whole string in one batch.

`ftfy.`**`fix_text_segment`**(*text*, *fix_entities='auto'*, *remove_terminal_escapes=True*, *fix_encoding=True*, *fix_latin_ligatures=True*, *fix_character_width=True*, *uncurl_quotes=True*, *fix_line_breaks=True*, *fix_surrogates=True*, *remove_control_chars=True*, *remove_bom=True*, *normalization='NFC'*)

Apply fixes to text in a single chunk. This could be a line of text within a larger run of `fix_text`, or it could be a larger amount of text that you are certain is in a consistent encoding.

See `fix_text` for a description of the parameters.

`ftfy.`**`fix_encoding`**(*text*)

Fix text with incorrectly-decoded garbage ("mojibake") whenever possible.

This function looks for the evidence of mojibake, formulates a plan to fix it, and applies the plan. It determines whether it should replace nonsense sequences of single-byte characters that were really meant to be UTF-8 characters, and if so, turns them into the correctly-encoded Unicode character that they were meant to represent.

The input to the function must be Unicode. If you don't have Unicode text, you're not using the right tool to solve your problem.

`fix_encoding` decodes text that looks like it was decoded incorrectly. It leaves alone text that doesn't.

```
>>> print(fix_encoding('Ãºnico'))
único
```

```
>>> print(fix_encoding('This text is fine already :þ'))
This text is fine already :þ
```

Because these characters often come from Microsoft products, we allow for the possibility that we get not just Unicode characters 128-255, but also Windows's conflicting idea of what characters 128-160 are.

```
>>> print(fix_encoding('This â€” should be an em dash'))
This — should be an em dash
```

We might have to deal with both Windows characters and raw control characters at the same time, especially when dealing with characters like 0x81 that have no mapping in Windows. This is a string that Python's standard `.encode` and `.decode` methods cannot correct.

```
>>> print(fix_encoding('This text is sad .â\x81”.'))
This text is sad .⁔.
```

However, it has safeguards against fixing sequences of letters and punctuation that can occur in valid text. In the following example, the last three characters are not replaced with a Korean character, even though they could be.

```
>>> print(fix_encoding('not such a fan of Charlotte Brontë...”'))
not such a fan of Charlotte Brontë...”
```

This function can now recover some complex manglings of text, such as when UTF-8 mojibake has been normalized in a way that replaces U+A0 with a space:

```
>>> print(fix_encoding('The more you know ðŸŒ '))
The more you know
```

Cases of genuine ambiguity can sometimes be addressed by finding other characters that are not double-encoded, and expecting the encoding to be consistent:

```
>>> print(fix_encoding('AHÅ™, the new sofa from IKEA®'))
AHÅ™, the new sofa from IKEA®
```

Finally, we handle the case where the text is in a single-byte encoding that was intended as Windows-1252 all along but read as Latin-1:

```
>>> print(fix_encoding('This text was never UTF-8 at all\x85'))
This text was never UTF-8 at all...
```

The best version of the text is found using *`ftfy.badness.text_cost()`*.

`ftfy.`**`fix_file`**(*input_file*, *encoding=None*, *fix_entities='auto'*, *remove_terminal_escapes=True*, *fix_encoding=True*, *fix_latin_ligatures=True*, *fix_character_width=True*, *uncurl_quotes=True*, *fix_line_breaks=True*, *fix_surrogates=True*, *remove_control_chars=True*, *remove_bom=True*, *normalization='NFC'*)

Fix text that is found in a file.

If the file is being read as Unicode text, use that. If it's being read as bytes, then we hope an encoding was supplied. If not, unfortunately, we have to guess what encoding it is. We'll try a few common encodings, but we make no promises. See the `guess_bytes` function for how this is done.

The output is a stream of fixed lines of text.

`ftfy.`**`explain_unicode`**(*text*)

A utility method that's useful for debugging mysterious Unicode.

It breaks down a string, showing you for each codepoint its number in hexadecimal, its glyph, its category in the Unicode standard, and its name in the Unicode standard.

```
>>> explain_unicode('(°°) ')
U+0028  (       [Ps] LEFT PARENTHESIS
U+256F         [So] BOX DRAWINGS LIGHT ARC UP AND LEFT
U+00B0  °       [So] DEGREE SIGN
U+25A1         [So] WHITE SQUARE
U+00B0  °       [So] DEGREE SIGN
U+0029  )       [Pe] RIGHT PARENTHESIS
U+256F         [So] BOX DRAWINGS LIGHT ARC UP AND LEFT
U+FE35         [Ps] PRESENTATION FORM FOR VERTICAL LEFT PARENTHESIS
U+0020          [Zs] SPACE
U+253B         [So] BOX DRAWINGS HEAVY UP AND HORIZONTAL
U+2501         [So] BOX DRAWINGS HEAVY HORIZONTAL
U+253B         [So] BOX DRAWINGS HEAVY UP AND HORIZONTAL
```

CHAPTER 5

Non-Unicode strings

When first using ftfy, you might be confused to find that you can't give it a bytestring (the type of object called `str` in Python 2).

ftfy fixes text. Treating bytestrings as text is exactly the kind of thing that causes the Unicode problems that ftfy has to fix. So if you don't give it a Unicode string, ftfy will point you to the Python Unicode HOWTO.

Reasonable ways that you might exchange data, such as JSON or XML, already have perfectly good ways of expressing Unicode strings. Given a Unicode string, ftfy can apply fixes that are very likely to work without false positives.

CHAPTER 6

A note on encoding detection

If your input is a mess of unmarked bytes, you might want a tool that can just statistically analyze those bytes and predict what encoding they're in.

`ftfy` is not that tool. The `ftfy.guess_bytes()` function it contains will do this in very limited cases, but to support more encodings from around the world, something more is needed.

You may have heard of `chardet`. Chardet is admirable, but it doesn't completely do the job either. Its heuristics are designed for multi-byte encodings, such as UTF-8 and the language-specific encodings used in East Asian languages. It works badly on single-byte encodings, to the point where it will output wrong answers with high confidence.

`ftfy.guess_bytes()` doesn't even try the East Asian encodings, so the ideal thing would combine the simple heuristic of `ftfy.guess_bytes()` with the multibyte character set detection of `chardet`. This ideal thing doesn't exist yet.

Accuracy

ftfy uses Twitter's streaming API as an endless source of realistic sample data. Twitter is massively multilingual, and despite that it's supposed to be uniformly UTF-8, in practice, any encoding mistake that someone can make will be made by someone's Twitter client.

We check what ftfy's `fix_encoding()` heuristic does to this data, and we aim to have the rate of false positives be indistinguishable from zero.

A pre-release version of ftfy was evaluated on 30,880,000 tweets received from Twitter's streaming API in April 2015. There was 1 false positive, and it was due to a bug that has now been fixed.

When looking at the changes ftfy makes, we found:

- `ftfy.fix_text()`, with all default options, will change about 1 in 18 tweets.
- With stylistic changes (`fix_character_width` and `uncurl_quotes`) turned off, `ftfy.fix_text()` will change about 1 in every 300 tweets.
- `ftfy.fix_encoding()` alone will change about 1 in every 8500 tweets.

We sampled 1000 of these `ftfy.fix_encoding()` changes for further evaluation, and found:

- 980 of them correctly restored the text.
- 12 of them incompletely or incorrectly restored the text, when a sufficiently advanced heuristic might have been able to fully recover the text.
- 8 of them represented text that had lost too much information to be fixed.
- None of those 1000 changed correct text to incorrect text (these would be false positives).

In all the data we've sampled, including from previous versions of ftfy, there are only three false positives that remain that we know of:

```
fix_encoding('aaaaa') == 'ôaaaaa'
fix_encoding('ESSE CARA AI QUEM É¿') == 'ESSE CARA AI QUEM '
fix_encoding('``hogwarts nao existe, voce nao vai pegar o trem pra lá´´')
  == '``hogwarts nao existe, voce nao vai pegar o trem pra l'
```

CHAPTER 8

Command-line tool

ftfy can be used from the command line. By default, it takes UTF-8 input and writes it to UTF-8 output, fixing problems in its Unicode as it goes.

Here's the usage documentation for the `ftfy` command:

```
usage: ftfy [-h] [-o OUTPUT] [-g] [-e ENCODING] [-n NORMALIZATION]
            [--preserve-entities]
            [filename]

ftfy (fixes text for you), version 4.0.0

positional arguments:
  filename              The file whose Unicode is to be fixed. Defaults to -,
                        meaning standard input.

optional arguments:
  -h, --help            show this help message and exit
  -o OUTPUT, --output OUTPUT
                        The file to output to. Defaults to -, meaning standard
                        output.
  -g, --guess           Ask ftfy to guess the encoding of your input. This is
                        risky. Overrides -e.
  -e ENCODING, --encoding ENCODING
                        The encoding of the input. Defaults to UTF-8.
  -n NORMALIZATION, --normalization NORMALIZATION
                        The normalization of Unicode to apply. Defaults to
                        NFC. Can be "none".
  --preserve-entities   Leave HTML entities as they are. The default is to
                        decode them, as long as no HTML tags have appeared in
                        the file.
```

CHAPTER 9

Module documentation

9.1 *ftfy.fixes*: how individual fixes are implemented

This module contains the individual fixes that the main fix_text function can perform.

`ftfy.fixes.`**`unescape_html`**`(`*text*`)`

Decode all three types of HTML entities/character references.

Code by Fredrik Lundh of effbot.org. Rob Speer made a slight change to it for efficiency: it won't match entities longer than 8 characters, because there are no valid entities like that.

```
>>> print(unescape_html('&lt;tag&gt;'))
<tag>
```

`ftfy.fixes.`**`remove_terminal_escapes`**`(`*text*`)`

Strip out "ANSI" terminal escape sequences, such as those that produce colored text on Unix.

```
>>> print(remove_terminal_escapes(
...     "\033[36;44mI'm blue, da ba dee da ba doo...\033[0m"
... ))
I'm blue, da ba dee da ba doo...
```

`ftfy.fixes.`**`uncurl_quotes`**`(`*text*`)`

Replace curly quotation marks with straight equivalents.

```
>>> print(uncurl_quotes('\u201chere\u2019s a test\u201d'))
"here's a test"
```

`ftfy.fixes.`**`fix_latin_ligatures`**`(`*text*`)`

Replace single-character ligatures of Latin letters, such as '', with the characters that they contain, as in 'fi'. Latin ligatures are usually not intended in text strings (though they're lovely in *rendered* text). If you have such a ligature in your string, it is probably a result of a copy-and-paste glitch.

We leave ligatures in other scripts alone to be safe. They may be intended, and removing them may lose information. If you want to take apart nearly all ligatures, use NFKC normalization.

```
>>> print(fix_latin_ligatures("ue"))
fluffiest
```

`ftfy.fixes.`**`fix_character_width`**`(`*text*`)`

The ASCII characters, katakana, and Hangul characters have alternate "halfwidth" or "fullwidth" forms that help text line up in a grid.

If you don't need these width properties, you probably want to replace these characters with their standard form, which is what this function does.

Note that this replaces the ideographic space, U+3000, with the ASCII space, U+20.

```
>>> print(fix_character_width(""))
LOUD NOISES
>>> print(fix_character_width(""))   # this means "U-turn"
U
```

ftfy.fixes.**fix_line_breaks**(*text*)

Convert all line breaks to Unix style.

This will convert the following sequences into the standard \n line break:

- CRLF (\r\n), used on Windows and in some communication protocols
- CR (\r), once used on Mac OS Classic, and now kept alive by misguided software such as Microsoft Office for Mac
- LINE SEPARATOR (\u2028) and PARAGRAPH SEPARATOR (\u2029), defined by Unicode and used to sow confusion and discord
- NEXT LINE (\x85), a C1 control character that is certainly not what you meant

The NEXT LINE character is a bit of an odd case, because it usually won't show up if `fix_encoding` is also being run. \x85 is very common mojibake for \u2026, HORIZONTAL ELLIPSIS.

```
>>> print(fix_line_breaks(
...     "This string is made of two things:\u2029"
...     "1. Unicode\u2028"
...     "2. Spite"
... ))
This string is made of two things:
1. Unicode
2. Spite
```

For further testing and examples, let's define a function to make sure we can see the control characters in their escaped form:

```
>>> def eprint(text):
...     print(text.encode('unicode-escape').decode('ascii'))
```

```
>>> eprint(fix_line_breaks("Content-type: text/plain\r\n\r\nHi."))
Content-type: text/plain\n\nHi.
```

```
>>> eprint(fix_line_breaks("This is how Microsoft \r trolls Mac users"))
This is how Microsoft \n trolls Mac users
```

```
>>> eprint(fix_line_breaks("What is this \x85 I don't even"))
What is this \n I don't even
```

ftfy.fixes.**fix_surrogates**(*text*)

Replace 16-bit surrogate codepoints with the characters they represent (when properly paired), or with otherwise.

```
>>> high_surrogate = unichr(0xd83d)
>>> low_surrogate = unichr(0xdca9)
>>> print(fix_surrogates(high_surrogate + low_surrogate))

>>> print(fix_surrogates(low_surrogate + high_surrogate))

```

The above doctest had to be very carefully written, because even putting the Unicode escapes of the surrogates in the docstring was causing various tools to fail, which I think just goes to show why this fixer is necessary.

`ftfy.fixes.`**`remove_control_chars`**`(`*text*`)`

Remove all ASCII control characters except for the important ones.

This removes characters in these ranges:

- U+0000 to U+0008
- U+000B
- U+000E to U+001F
- U+007F

It leaves alone these characters that are commonly used for formatting:

- TAB (U+0009)
- LF (U+000A)
- FF (U+000C)
- CR (U+000D)

Feel free to object that FF isn't "commonly" used for formatting. I've at least seen it used.

`ftfy.fixes.`**`remove_bom`**`(`*text*`)`

Remove a byte-order mark that was accidentally decoded as if it were part of the text.

```
>>> print(remove_bom("\ufeffWhere do you want to go today?"))
Where do you want to go today?
```

`ftfy.fixes.`**`decode_escapes`**`(`*text*`)`

Decode backslashed escape sequences, including \x, \u, and \U character references, even in the presence of other Unicode.

This is what Python's "string-escape" and "unicode-escape" codecs were meant to do, but in contrast, this actually works. It will decode the string exactly the same way that the Python interpreter decodes its string literals.

```
>>> factoid = '\\u20a1 is the currency symbol for the colón.'
>>> print(factoid[1:])
u20a1 is the currency symbol for the colón.
>>> print(decode_escapes(factoid))
₡ is the currency symbol for the colón.
```

Even though Python itself can read string literals with a combination of escapes and literal Unicode – you're looking at one right now – the "unicode-escape" codec doesn't work on literal Unicode. (See http://stackoverflow.com/a/24519338/773754 for more details.)

Instead, this function searches for just the parts of a string that represent escape sequences, and decodes them, leaving the rest alone. All valid escape sequences are made of ASCII characters, and this allows "unicode-escape" to work correctly.

This fix cannot be automatically applied by the `ftfy.fix_text` function, because escaped text is not necessarily a mistake, and there is no way to distinguish text that's supposed to be escaped from text that isn't.

`ftfy.fixes.`**`fix_one_step_and_explain`**`(`*text*`)`

Performs a single step of re-decoding text that's been decoded incorrectly.

Returns the decoded text, plus a "plan" for how to reproduce what it did.

`ftfy.fixes.`**`apply_plan`**`(`*text, plan*`)`

Apply a plan for fixing the encoding of text.

The plan is a list of tuples of the form (operation, encoding, cost):

- `operation` is 'encode' if it turns a string into bytes, 'decode' if it turns bytes into a string, and 'transcode' if it keeps the type the same.
- `encoding` is the name of the encoding to use, such as 'utf-8' or 'latin-1', or the function name in the case of 'transcode'.
- The `cost` does not affect how the plan itself works. It's used by other users of plans, namely `fix_encoding_and_explain`, which has to decide *which* plan to use.

9.2 *ftfy.badness*: measures the "badness" of text

Heuristics to determine whether re-encoding text is actually making it more reasonable.

`ftfy.badness.`**`sequence_weirdness`**`(`*`text`*`)`

Determine how often a text has unexpected characters or sequences of characters. This metric is used to disambiguate when text should be re-decoded or left as is.

We start by normalizing text in NFC form, so that penalties for diacritical marks don't apply to characters that know what to do with them.

The following things are deemed weird:

- Lowercase letters followed by non-ASCII uppercase letters
- Non-Latin characters next to Latin characters
- Un-combined diacritical marks, unless they're stacking on non-alphabetic characters (in languages that do that kind of thing a lot) or other marks
- C1 control characters
- Adjacent symbols from any different pair of these categories:
  - Modifier marks
  - Letter modifiers
  - Non-digit numbers
  - Symbols (including math and currency)

The return value is the number of instances of weirdness.

`ftfy.badness.`**`text_cost`**`(`*`text`*`)`

An overall cost function for text. Weirder is worse, but all else being equal, shorter strings are better.

The overall cost is measured as the "weirdness" (see `sequence_weirdness()`) plus the length.

9.3 *ftfy.bad_codecs*: support some encodings Python doesn't like

Give Python the ability to decode some common, flawed encodings.

Python does not want you to be sloppy with your text. Its encoders and decoders ("codecs") follow the relevant standards whenever possible, which means that when you get text that *doesn't* follow those standards, you'll probably fail to decode it. Or you might succeed at decoding it for implementation-specific reasons, which is perhaps worse.

There are some encodings out there that Python wishes didn't exist, which are widely used outside of Python:

- "utf-8-variants", a family of not-quite-UTF-8 encodings, including the ever-popular CESU-8 and "Java modified UTF-8".

- "Sloppy" versions of character map encodings, where bytes that don't map to anything will instead map to the Unicode character with the same number.

Simply importing this module, or in fact any part of the `ftfy` package, will make these new "bad codecs" available to Python through the standard Codecs API. You never have to actually call any functions inside `ftfy.bad_codecs`.

However, if you want to call something because your code checker insists on it, you can call `ftfy.bad_codecs.ok()`.

A quick example of decoding text that's encoded in CESU-8:

```
>>> import ftfy.bad_codecs
>>> print(b'\xed\xa0\xbd\xed\xb8\x8d'.decode('utf-8-variants'))
```

9.3.1 "Sloppy" encodings

Decodes single-byte encodings, filling their "holes" in the same messy way that everyone else does.

A single-byte encoding maps each byte to a Unicode character, except that some bytes are left unmapped. In the commonly-used Windows-1252 encoding, for example, bytes 0x81 and 0x8D, among others, have no meaning.

Python, wanting to preserve some sense of decorum, will handle these bytes as errors. But Windows knows that 0x81 and 0x8D are possible bytes and they're different from each other. It just hasn't defined what they are in terms of Unicode.

Software that has to interoperate with Windows-1252 and Unicode – such as all the common Web browsers – will pick some Unicode characters for them to map to, and the characters they pick are the Unicode characters with the same numbers: U+0081 and U+008D. This is the same as what Latin-1 does, and the resulting characters tend to fall into a range of Unicode that's set aside for obselete Latin-1 control characters anyway.

These sloppy codecs let Python do the same thing, thus interoperating with other software that works this way. It defines a sloppy version of many single-byte encodings with holes. (There is no need for a sloppy version of an encoding without holes: for example, there is no such thing as sloppy-iso-8859-2 or sloppy-macroman.)

The following encodings will become defined:

- sloppy-windows-1250 (Central European, sort of based on ISO-8859-2)
- sloppy-windows-1251 (Cyrillic)
- sloppy-windows-1252 (Western European, based on Latin-1)
- sloppy-windows-1253 (Greek, sort of based on ISO-8859-7)
- sloppy-windows-1254 (Turkish, based on ISO-8859-9)
- sloppy-windows-1255 (Hebrew, based on ISO-8859-8)
- sloppy-windows-1256 (Arabic)
- sloppy-windows-1257 (Baltic, based on ISO-8859-13)
- sloppy-windows-1258 (Vietnamese)
- sloppy-cp874 (Thai, based on ISO-8859-11)
- sloppy-iso-8859-3 (Maltese and Esperanto, I guess)
- sloppy-iso-8859-6 (different Arabic)
- sloppy-iso-8859-7 (Greek)
- sloppy-iso-8859-8 (Hebrew)

- sloppy-iso-8859-11 (Thai)

Aliases such as “sloppy-cp1252” for “sloppy-windows-1252” will also be defined.

Only sloppy-windows-1251 and sloppy-windows-1252 are used by the rest of ftfy; the rest are rather uncommon.

Here are some examples, using `ftfy.explain_unicode` to illustrate how sloppy-windows-1252 merges Windows-1252 with Latin-1:

```
>>> from ftfy import explain_unicode
>>> some_bytes = b'\x80\x81\x82'
>>> explain_unicode(some_bytes.decode('latin-1'))
U+0080  \x80    [Cc] <unknown>
U+0081  \x81    [Cc] <unknown>
U+0082  \x82    [Cc] <unknown>
```

```
>>> explain_unicode(some_bytes.decode('windows-1252', 'replace'))
U+20AC  €       [Sc] EURO SIGN
U+FFFD         [So] REPLACEMENT CHARACTER
U+201A  ,       [Ps] SINGLE LOW-9 QUOTATION MARK
```

```
>>> explain_unicode(some_bytes.decode('sloppy-windows-1252'))
U+20AC  €       [Sc] EURO SIGN
U+0081  \x81    [Cc] <unknown>
U+201A  ,       [Ps] SINGLE LOW-9 QUOTATION MARK
```

9.3.2 Variants of UTF-8

This file defines a codec called “utf-8-variants” (or “utf-8-var”), which can decode text that’s been encoded with a popular non-standard version of UTF-8. This includes CESU-8, the accidental encoding made by layering UTF-8 on top of UTF-16, as well as Java’s twist on CESU-8 that contains a two-byte encoding for codepoint 0.

This is particularly relevant in Python 3, which provides no other way of decoding CESU-8 [1].

The easiest way to use the codec is to simply import `ftfy.bad_codecs`:

```
>>> import ftfy.bad_codecs
>>> result = b'here comes a null! \xc0\x80'.decode('utf-8-var')
>>> print(repr(result).lstrip('u'))
'here comes a null! \x00'
```

The codec does not at all enforce “correct” CESU-8. For example, the Unicode Consortium’s not-quite-standard describing CESU-8 requires that there is only one possible encoding of any character, so it does not allow mixing of valid UTF-8 and CESU-8. This codec *does* allow that, just like Python 2’s UTF-8 decoder does.

Characters in the Basic Multilingual Plane still have only one encoding. This codec still enforces the rule, within the BMP, that characters must appear in their shortest form. There is one exception: the sequence of bytes `0xc0 0x80`, instead of just `0x00`, may be used to encode the null character `U+0000`, like in Java.

If you encode with this codec, you get legitimate UTF-8. Decoding with this codec and then re-encoding is not idempotent, although encoding and then decoding is. So this module won’t produce CESU-8 for you. Look for that functionality in the sister module, “Breaks Text For You”, coming approximately never.

[1] In a pinch, you can decode CESU-8 in Python 2 using the UTF-8 codec: first decode the bytes (incorrectly), then encode them, then decode them again, using UTF-8 as the codec every time.

9.4 *ftfy.chardata* and *ftfy.build_data*: trivia about characters

These files load information about the character properties in Unicode 7.0. Yes, even if your version of Python doesn't support Unicode 7.0. This ensures that ftfy's behavior is consistent across versions. This gives other modules access to the gritty details about characters and the encodings that use them.

`ftfy.chardata.`**`chars_to_classes`**`(`*string*`)`

Convert each Unicode character to a letter indicating which of many classes it's in.

See build_data.py for where this data comes from and what it means.

`ftfy.chardata.`**`possible_encoding`**`(`*text*, *encoding*`)`

Given text and a single-byte encoding, check whether that text could have been decoded from that single-byte encoding.

In other words, check whether it can be encoded in that encoding, possibly sloppily.

`ftfy.build_data.`**`make_char_data_file`**`(`*do_it_anyway=False*`)`

Build the compressed data file 'char_classes.dat' and write it to the current directory.

If you run this, run it in Python 3.5 or later, even though that requires an alpha version at the time of writing this code. It will run in earlier versions, but you won't get the Unicode 7 standard, leading to inconsistent behavior.

To protect against this, running this in the wrong version of Python will raise an error unless you pass `do_it_anyway=True`.

CHAPTER 10

Future versions of ftfy

ftfy has full support for Python 2.7, even including a backport of Unicode 7 character classes to Python 2. But given the sweeping changes to Unicode in Python, it's getting inconvenient to add new features to ftfy that work the same on both versions.

ftfy 5.0, when it is released, will probably only support Python 3.

If you want to see examples of why ftfy is particularly difficult to maintain on two versions of Python (which is more like three versions because of Python 2's "wide" and "narrow" builds), take a look at functions such as `ftfy.bad_codecs.utf8_variants.mangle_surrogates()` and `ftfy.compatibility._narrow_unichr_workaround()`.

We're following the lead of venerable projects such as jQuery. jQuery dropped support for IE 6-8 in version 2.0, and in doing so, it became much simpler and faster for people who didn't need to support old versions of IE in their web applications. Meanwhile, jQuery 1.9 remained there for those who needed it.

Similarly, ftfy 5.0 will reduce the size and complexity of the code greatly, but ftfy 4.x will remain there. If you're running on Python 2, please make sure that `ftfy < 5` is in your requirements list, not just `ftfy`.

Index