Introducing The chardet Module

WE'LL COVER THE FOLLOWING ^

- UTF-N With A BOM
- Escaped Encodings
- Multi-Byte Encodings
- Single-Byte Encodings
- windows-1252

Before we set off porting the code, it would help if you understood how the code worked! This is a brief guide to navigating the code itself. The `chardet` library is too large to include inline here, but you can [download it from chardet.feedparser.org](http://chardet.feedparser.org).

> Encoding detection is really language detection in drag.

The main entry point for the detection algorithm is `universaldetector.py`, which has one class, `UniversalDetecto`r. (You might think the main entry point is the detect function in `chardet/__init__.py`, but that's really just a convenience function that creates a UniversalDetector object, calls it, and returns its result.)

There are 5 categories of encodings that `UniversalDetector` handles:

1. UTF-N with a Byte Order Mark (BOM). This includes UTF-8, both Big-Endian and Little-Endian variants of UTF-16, and all 4 byte-order variants of UTF-32.
2. Escaped encodings, which are entirely 7-bit ASCII compatible, where non-ASCII characters start with an escape sequence. Examples: ISO-2022-JP (Japanese) and HZ-GB-2312 (Chinese).

3. Multi-byte encodings, where each character is represented by a variable number of bytes. Examples: BIG5 (Chinese), SHIFT_JIS (Japanese), EUC-KR (Korean), and UTF-8 without a BOM.
4. Single-byte encodings, where each character is represented by one byte. Examples: KOI8-R (Russian), WINDOWS-1255 (Hebrew), and TIS-620 (Thai).
5. WINDOWS-1252, which is used primarily on Microsoft Windows by middle managers who wouldn't know a character encoding from a hole in the ground.

UTF-N With A BOM

If the text starts with a bom, we can reasonably assume that the text is encoded in UTF-8, UTF-16, or UTF-32. (The bom will tell us exactly which one; that's what it's for.) This is handled inline in `UniversalDetector`, which returns the result immediately without any further processing.

Escaped Encodings

If the text contains a recognizable escape sequence that might indicate an escaped encoding, `UniversalDetector` creates an `EscCharSetProber` (defined in `escprober.py`) and feeds it the text.

`EscCharSetProber` creates a series of state machines, based on models of HZ-GB-2312, ISO-2022-CN, ISO-2022-JP, and ISO-2022-KR (defined in `escsm.p`y). `EscCharSetProber` feeds the text to each of these state machines, one byte at a time. If any state machine ends up uniquely identifying the encoding, `EscCharSetProber` immediately returns the positive result to `UniversalDetector`, which returns it to the caller. If any state machine hits an illegal sequence, it is dropped and processing continues with the other state machines.

Multi-Byte Encodings

Assuming no BOM, `UniversalDetector` checks whether the text contains any high-bit characters. If so, it creates a series of "probers" for detecting multi-byte encodings, single-byte encodings, and as a last resort, `windows-1252`.

The multi-byte encoding prober, `MBCSGroupProber` (defined in `mbcsgroupprober.py`), is really just a shell that manages a group of other `mbcsgroupprober.py`), is really just a shell that manages a group of other probers, one for each multi-byte encoding: BIG5, GB2312, EUC-TW, EUC-KR,

EUC-JP, SHIFT_JIS, and UTF-8. `MBCSGroupProber` feeds the text to each of these encoding-specific probers and checks the results. If a prober reports that it has found an illegal byte sequence, it is dropped from further processing (so that, for instance, any subsequent calls to `UniversalDetector.feed()` will skip that prober). If a prober reports that it is reasonably confident that it has detected the encoding, `MBCSGroupProber` reports this positive result to `UniversalDetector`, which reports the result to the caller.

Most of the multi-byte encoding probers are inherited from `MultiByteCharSetProber` (defined in `mbcharsetprober.py`), and simply hook up the appropriate state machine and distribution analyzer and let `MultiByteCharSetProber` do the rest of the work. `MultiByteCharSetProber` runs the text through the encoding-specific state machine, one byte at a time, to look for byte sequences that would indicate a conclusive positive or negative result. At the same time, `MultiByteCharSetProber` feeds the text to an encoding-specific distribution analyzer.

The distribution analyzers (each defined in `chardistribution.py`) use language-specific models of which characters are used most frequently. Once `MultiByteCharSetProber` has fed enough text to the distribution analyzer, it calculates a confidence rating based on the number of frequently-used characters, the total number of characters, and a language-specific distribution ratio. If the confidence is high enough, `MultiByteCharSetProber` returns the result to `MBCSGroupProber`, which returns it to `UniversalDetector`, which returns it to the caller.

The case of Japanese is more difficult. Single-character distribution analysis is not always sufficient to distinguish between `EUC-JP` and `SHIFT_JIS`, so the `SJISProber` (defined in `sjisprober.py`) also uses 2-character distribution `analysis. SJISContextAnalysis` and `EUCJPContextAnalysis` (both defined in `jpcntx.py` and both inheriting from a common `JapaneseContextAnalysis` class) check the frequency of Hiragana syllabary characters within the text. Once enough text has been processed, they return a confidence level to `SJISProber`, which checks both analyzers and returns the higher confidence level to `MBCSGroupProber`.

Single-Byte Encodings

Single-Byte Encodings

Seriously, where's my Unicode pony?

The single-byte encoding prober, `SBCSGroupProber` (defined in `sbcsgroupprober.py`), is also just a shell that manages a group of other probers, one for each combination of single-byte encoding and language: `windows-1251`, `KOI8-R`, `ISO-8859-5`, `MacCyrillic`, `IBM855`, and `IBM866` (Russian); `ISO-8859-7` and `windows-1253` (Greek); `ISO-8859-5` and `windows-1251` (Bulgarian); `ISO-8859-2` and `windows-1250` (Hungarian); `TIS-620` (Thai); `windows-1255` and `ISO-8859-8` (Hebrew).

`SBCSGroupProber` feeds the text to each of these encoding+language-specific probers and checks the results. These probers are all implemented as a single class, `SingleByteCharSetProber` (defined in `sbcharsetprober.py`), which takes a language model as an argument. The language model defines how frequently different 2-character sequences appear in typical text. `SingleByteCharSetProber` processes the text and tallies the most frequently used 2-character sequences. Once enough text has been processed, it calculates a confidence level based on the number of frequently-used sequences, the total number of characters, and a language-specific distribution ratio.

Hebrew is handled as a special case. If the text appears to be Hebrew based on 2-character distribution analysis, `HebrewProber` (defined in `hebrewprober.py`) tries to distinguish between Visual Hebrew (where the source text actually stored "backwards" line-by-line, and then displayed verbatim so it can be read from right to left) and Logical Hebrew (where the source text is stored in reading order and then rendered right-to-left by the client). Because certain characters are encoded differently based on whether they appear in the middle of or at the end of a word, we can make a reasonable guess about direction of the source text, and return the appropriate encoding `(windows-1255` for Logical Hebrew, or `ISO-8859-8` for Visual Hebrew).

windows-1252

If `UniversalDetector` detects a high-bit character in the text, but none of the other multi-byte or single-byte encoding probers return a confident result, it creates a `Latin1Prober` (defined in `latin1prober.py`) to try to detect English text in a `windows-1252` encoding. This detection is inherently unreliable text in a `windows-1252` encoding. This detection is inherently unreliable, because English letters are encoded in the same way in many different encodings. The only way to distinguish `windows-1252` is through commonly used symbols like smart quotes, curly apostrophes, copyright symbols, and the like. `Latin1Prober` automatically reduces its confidence rating to allow more accurate probers to win if at all possible.