chardet Documentation

Release 5.2.0

Mark Pilgrim, Dan Blanchard, Ian Cordasco

Apr 05, 2024

CONTENTS

Character encoding auto-detection in Python. As smart as your browser. Open source.

CHAPTER

ONE

DOCUMENTATION

1.1 Frequently asked questions

1.1.1 What is character encoding?

When you think of "text", you probably think of "characters and symbols I see on my computer screen". But computers don't deal in characters and symbols; they deal in bits and bytes. Every piece of text you've ever seen on a computer screen is actually stored in a particular *character encoding*. There are many different character encodings, some optimized for particular languages like Russian or Chinese or English, and others that can be used for multiple languages. Very roughly speaking, the character encoding provides a mapping between the stuff you see on your screen and the stuff your computer actually stores in memory and on disk.

In reality, it's more complicated than that. Many characters are common to multiple encodings, but each encoding may use a different sequence of bytes to actually store those characters in memory or on disk. So you can think of the character encoding as a kind of decryption key for the text. Whenever someone gives you a sequence of bytes and claims it's "text", you need to know what character encoding they used so you can decode the bytes into characters and display them (or process them, or whatever).

1.1.2 What is character encoding auto-detection?

It means taking a sequence of bytes in an unknown character encoding, and attempting to determine the encoding so you can read the text. It's like cracking a code when you don't have the decryption key.

1.1.3 Isn't that impossible?

In general, yes. However, some encodings are optimized for specific languages, and languages are not random. Some character sequences pop up all the time, while other sequences make no sense. A person fluent in English who opens a newspaper and finds "txzqJv 2!dasd0a QqdKjvz" will instantly recognize that that isn't English (even though it is composed entirely of English letters). By studying lots of "typical" text, a computer algorithm can simulate this kind of fluency and make an educated guess about a text's language.

In other words, encoding detection is really language detection, combined with knowledge of which languages tend to use which character encodings.

1.1.4 Who wrote this detection algorithm?

This library is a port of the auto-detection code in Mozilla. I have attempted to maintain as much of the original structure as possible (mostly for selfish reasons, to make it easier to maintain the port as the original code evolves). I have also retained the original authors' comments, which are quite extensive and informative.

You may also be interested in the research paper which led to the Mozilla implementation, A composite approach to language/encoding detection.

1.1.5 Yippie! Screw the standards, I'll just auto-detect everything!

Don't do that. Virtually every format and protocol contains a method for specifying character encoding.

- HTTP can define a `charset` parameter in the `Content-type` header.
- HTML documents can define a `<meta http-equiv="content-type">` element in the `<head>` of a web page.
- XML documents can define an `encoding` attribute in the XML prolog.

If text comes with explicit character encoding information, you should use it. If the text has no explicit information, but the relevant standard defines a default encoding, you should use that. (This is harder than it sounds, because standards can overlap. If you fetch an XML document over HTTP, you need to support both standards *and* figure out which one wins if they give you conflicting information.)

Despite the complexity, it's worthwhile to follow standards and respect explicit character encoding information. It will almost certainly be faster and more accurate than trying to auto-detect the encoding. It will also make the world a better place, since your program will interoperate with other programs that follow the same standards.

1.1.6 Why bother with auto-detection if it's slow, inaccurate, and non-standard?

Sometimes you receive text with verifiably inaccurate encoding information. Or text without any encoding information, and the specified default encoding doesn't work. There are also some poorly designed standards that have no way to specify encoding at all.

If following the relevant standards gets you nowhere, *and* you decide that processing the text is more important than maintaining interoperability, then you can try to auto-detect the character encoding as a last resort. An example is my Universal Feed Parser, which calls this auto-detection library only after exhausting all other options.

1.2 Supported encodings

Universal Encoding Detector currently supports over two dozen character encodings.

- `Big5`, `GB2312`/`GB18030`, `EUC-TW`, `HZ-GB-2312`, and `ISO-2022-CN` (Traditional and Simplified Chinese)
- `EUC-JP`, `SHIFT_JIS`, and `ISO-2022-JP` (Japanese)
- `EUC-KR` and `ISO-2022-KR` (Korean)
- `KOI8-R`, `MacCyrillic`, `IBM855`, `IBM866`, `ISO-8859-5`, and `windows-1251` (Russian)
- `ISO-8859-2` and `windows-1250` (Hungarian)
- `ISO-8859-5` and `windows-1251` (Bulgarian)
- `ISO-8859-1` and `windows-1252` (Western European languages)
- `ISO-8859-7` and `windows-1253` (Greek)

- ISO-8859-8 and windows-1255 (Visual and Logical Hebrew)
- TIS-620 (Thai)
- UTF-32 BE, LE, 3412-ordered, or 2143-ordered (with a BOM)
- UTF-16 BE or LE (with a BOM)
- UTF-8 (with or without a BOM)
- ASCII

Warning: Due to inherent similarities between certain encodings, some encodings may be detected incorrectly. In my tests, the most problematic case was Hungarian text encoded as ISO-8859-2 or windows-1250 (encoded as one but reported as the other). Also, Greek text encoded as ISO-8859-7 was often mis-reported as ISO-8859-2. Your mileage may vary.

1.3 Usage

1.3.1 Basic usage

The easiest way to use the Universal Encoding Detector library is with the detect function.

1.3.2 Example: Using the detect function

The detect function takes one argument, a non-Unicode string. It returns a dictionary containing the auto-detected character encoding and a confidence level from 0 to 1.

```
>>> import urllib.request
>>> rawdata = urllib.request.urlopen('http://yahoo.co.jp/').read()
>>> import chardet
>>> chardet.detect(rawdata)
{'encoding': 'EUC-JP', 'confidence': 0.99}
```

1.3.3 Advanced usage

If you're dealing with a large amount of text, you can call the Universal Encoding Detector library incrementally, and it will stop as soon as it is confident enough to report its results.

Create a UniversalDetector object, then call its feed method repeatedly with each block of text. If the detector reaches a minimum threshold of confidence, it will set detector.done to True.

Once you've exhausted the source text, call detector.close(), which will do some final calculations in case the detector didn't hit its minimum confidence threshold earlier. Then detector.result will be a dictionary containing the auto-detected character encoding and confidence level (the same as the chardet.detect function returns).

1.3.4 Example: Detecting encoding incrementally

```
import urllib.request
from chardet.universaldetector import UniversalDetector

usock = urllib.request.urlopen('http://yahoo.co.jp/')
detector = UniversalDetector()
for line in usock.readlines():
    detector.feed(line)
    if detector.done: break
detector.close()
usock.close()
print(detector.result)
```

```
{'encoding': 'EUC-JP', 'confidence': 0.99}
```

If you want to detect the encoding of multiple texts (such as separate files), you can re-use a single `UniversalDetector` object. Just call `detector.reset()` at the start of each file, call `detector.feed` as many times as you like, and then call `detector.close()` and check the `detector.result` dictionary for the file's results.

1.3.5 Example: Detecting encodings of multiple files

```
import glob
from chardet.universaldetector import UniversalDetector

detector = UniversalDetector()
for filename in glob.glob('*.xml'):
    print(filename.ljust(60), end='')
    detector.reset()
    for line in open(filename, 'rb'):
        detector.feed(line)
        if detector.done: break
    detector.close()
    print(detector.result)
```

1.4 How it works

This is a brief guide to navigating the code itself.

First, you should read A composite approach to language/encoding detection, which explains the detection algorithm and how it was derived. This will help you later when you stumble across the huge character frequency distribution tables like `big5freq.py` and language models like `langcyrillicmodel.py`.

The main entry point for the detection algorithm is `universaldetector.py`, which has one class, `UniversalDetector`. (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 BOM. This includes `UTF-8`, both BE and LE 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; its subset, `ISO-8859-1` is widely used for legacy 8-bit-encoded text. chardet, like many encoding detectors, defaults to guessing this encoding when no other can be reliably established.

1.4.1 `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.

1.4.2 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.py`). `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.

1.4.3 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 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`.

1.4.4 Single-byte encodings

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).

1.4.5 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, 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.

1.5 chardet

1.5.1 chardet package

Submodules

chardet.big5freq module

chardet.big5prober module

class chardet.big5prober.**Big5Prober**

Bases: *MultiByteCharSetProber*

property charset_name: str

property language: str

chardet.chardetect module

chardet.chardistribution module

class chardet.chardistribution.**Big5DistributionAnalysis**

Bases: *CharDistributionAnalysis*

get_order(*byte_str: bytes | bytearray*) → int

class chardet.chardistribution.**CharDistributionAnalysis**

Bases: object

ENOUGH_DATA_THRESHOLD = 1024

MINIMUM_DATA_THRESHOLD = 3

SURE_NO = 0.01

SURE_YES = 0.99

feed(*char: bytes | bytearray*, *char_len: int*) → None

feed a character with known length

get_confidence() → float

return confidence based on existing data

get_order(*_: bytes | bytearray*) → int

got_enough_data() → bool

reset() → None

reset analyser, clear any state

class chardet.chardistribution.**EUCJPDistributionAnalysis**

Bases: *CharDistributionAnalysis*

get_order(*byte_str: bytes | bytearray*) → int

class chardet.chardistribution.**EUCKRDistributionAnalysis**

Bases: *CharDistributionAnalysis*

get_order(*byte_str: bytes | bytearray*) → int

class chardet.chardistribution.**EUCTWDistributionAnalysis**

Bases: *CharDistributionAnalysis*

get_order(*byte_str: bytes | bytearray*) → int

class chardet.chardistribution.**GB2312DistributionAnalysis**

Bases: *CharDistributionAnalysis*

get_order(*byte_str: bytes | bytearray*) → int

class chardet.chardistribution.**JOHABDistributionAnalysis**

Bases: *CharDistributionAnalysis*

get_order(*byte_str: bytes | bytearray*) → int

class chardet.chardistribution.**SJISDistributionAnalysis**

Bases: *CharDistributionAnalysis*

get_order(*byte_str: bytes | bytearray*) → int

chardet.charsetgroupprober module

class chardet.charsetgroupprober.**CharSetGroupProber**(*lang_filter: LanguageFilter = LanguageFilter.NONE*)

Bases: *CharSetProber*

property charset_name: str | None

feed(*byte_str: bytes | bytearray*) → ProbingState

get_confidence() → float

property language: str | None

reset() → None

chardet.charsetprober module

class chardet.charsetprober.**CharSetProber**(*lang_filter: LanguageFilter = LanguageFilter.NONE*)

Bases: object

SHORTCUT_THRESHOLD = 0.95

property charset_name: str | None

feed(*byte_str: bytes | bytearray*) → ProbingState

static filter_high_byte_only(*buf: bytes | bytearray*) → bytes

static filter_international_words(*buf: bytes | bytearray*) → bytearray

We define three types of bytes: alphabet: english alphabets [a-zA-Z] international: international characters [□-ÿ] marker: everything else [^a-zA-Z□-ÿ] The input buffer can be thought to contain a series of words delimited by markers. This function works to filter all words that contain at least one international character. All contiguous sequences of markers are replaced by a single space ascii character. This filter applies to all scripts which do not use English characters.

get_confidence() → float

property language: str | None

static remove_xml_tags(*buf: bytes | bytearray*) → bytes

Returns a copy of buf that retains only the sequences of English alphabet and high byte characters that are not between <> characters. This filter can be applied to all scripts which contain both English characters and extended ASCII characters, but is currently only used by Latin1Prober.

reset() → None

property state: ProbingState

chardet.codingstatemachine module

class chardet.codingstatemachine.**CodingStateMachine**(*sm: dict*)

Bases: object

A state machine to verify a byte sequence for a particular encoding. For each byte the detector receives, it will feed that byte to every active state machine available, one byte at a time. The state machine changes its state based on its previous state and the byte it receives. There are 3 states in a state machine that are of interest to an auto-detector:

START state: This is the state to start with, or a legal byte sequence
(i.e. a valid code point) for character has been identified.

ME state: This indicates that the state machine identified a byte sequence
that is specific to the charset it is designed for and that there is no other possible encoding which can contain this byte sequence. This will to lead to an immediate positive answer for the detector.

ERROR state: This indicates the state machine identified an illegal byte
sequence for that encoding. This will lead to an immediate negative answer for this encoding. Detector will exclude this encoding from consideration from here on.

get_coding_state_machine() → str

get_current_charlen() → int

property language: str

next_state(*c: int*) → int

reset() → None

chardet.compat module

chardet.constants module

chardet.cp949prober module

class chardet.cp949prober.**CP949Prober**

Bases: *MultiByteCharSetProber*

property charset_name: str

property language: str

chardet.escprober module

class chardet.escprober.**EscCharSetProber**(*lang_filter: LanguageFilter = LanguageFilter.NONE*)

Bases: *CharSetProber*

This CharSetProber uses a "code scheme" approach for detecting encodings, whereby easily recognizable escape or shift sequences are relied on to identify these encodings.

property charset_name: str | None

feed(*byte_str: bytes | bytearray*) → ProbingState

get_confidence() → float

property language: str | None

reset() → None

chardet.escsm module

chardet.eucjpprober module

class chardet.eucjpprober.**EUCJPProber**

Bases: *MultiByteCharSetProber*

property charset_name: str

feed(*byte_str: bytes | bytearray*) → ProbingState

get_confidence() → float

property language: str

reset() → None

chardet.euckrfreq module

chardet.euckrprober module

class chardet.euckrprober.**EUCKRProber**

Bases: *MultiByteCharSetProber*

property charset_name: str

property language: str

chardet.euctwfreq module

chardet.euctwprober module

class chardet.euctwprober.**EUCTWProber**

Bases: *MultiByteCharSetProber*

property charset_name: str

property language: str

chardet.gb2312freq module

chardet.gb2312prober module

class chardet.gb2312prober.**GB2312Prober**

Bases: *MultiByteCharSetProber*

property charset_name: str

property language: str

chardet.hebrewprober module

class chardet.hebrewprober.**HebrewProber**

Bases: *CharSetProber*

FINAL_KAF = 234

FINAL_MEM = 237

FINAL_NUN = 239

FINAL_PE = 243

FINAL_TSADI = 245

LOGICAL_HEBREW_NAME = 'windows-1255'

MIN_FINAL_CHAR_DISTANCE = 5

MIN_MODEL_DISTANCE = 0.01

NORMAL_KAF = 235

NORMAL_MEM = 238

NORMAL_NUN = 240

NORMAL_PE = 244

NORMAL_TSADI = 246

SPACE = 32

VISUAL_HEBREW_NAME = 'ISO-8859-8'

property charset_name: str

feed(*byte_str: bytes | bytearray*) → ProbingState

is_final(*c: int*) → bool

is_non_final(*c: int*) → bool

property language: str

reset() → None

set_model_probers(*logical_prober:* SingleByteCharSetProber, *visual_prober:* SingleByteCharSetProber) → None

property state: ProbingState

chardet.jisfreq module

chardet.jpcntx module

class chardet.jpcntx.**EUCJPContextAnalysis**

Bases: *JapaneseContextAnalysis*

get_order(*byte_str: bytes | bytearray*) → Tuple[int, int]

class chardet.jpcntx.**JapaneseContextAnalysis**

Bases: object

DONT_KNOW = -1

ENOUGH_REL_THRESHOLD = 100

MAX_REL_THRESHOLD = 1000

MINIMUM_DATA_THRESHOLD = 4

NUM_OF_CATEGORY = 6

feed(*byte_str: bytes | bytearray*, *num_bytes: int*) → None

get_confidence() → float

get_order(*_: bytes | bytearray*) → Tuple[int, int]

got_enough_data() → bool

reset() → None

class chardet.jpcntx.**SJISContextAnalysis**

Bases: *JapaneseContextAnalysis*

property charset_name: str

get_order(*byte_str: bytes | bytearray*) → Tuple[int, int]

chardet.langbulgarianmodel module

chardet.langcyrillicmodel module

chardet.langgreekmodel module

chardet.langhebrewmodel module

chardet.langhungarianmodel module

chardet.langthaimodel module

chardet.latin1prober module

class chardet.latin1prober.**Latin1Prober**

Bases: *CharSetProber*

property charset_name: str

feed(*byte_str: bytes | bytearray*) → ProbingState

get_confidence() → float

property language: str

reset() → None

chardet.mbcharsetprober module

class chardet.mbcharsetprober.**MultiByteCharSetProber**(*lang_filter: LanguageFilter = LanguageFilter.NONE*)

Bases: *CharSetProber*

feed(*byte_str: bytes | bytearray*) → ProbingState

get_confidence() → float

reset() → None

chardet.mbcsgroupprober module

class chardet.mbcsgroupprober.**MBCSGroupProber**(*lang_filter: LanguageFilter = LanguageFilter.NONE*)

Bases: *CharSetGroupProber*

chardet.mbcssm module

chardet.sbcharsetprober module

class chardet.sbcharsetprober.**SingleByteCharSetModel**(*charset_name*, *language*, *char_to_order_map*, *language_model*, *typical_positive_ratio*, *keep_ascii_letters*, *alphabet*)

Bases: NamedTuple

alphabet: str

Alias for field number 6

char_to_order_map: Dict[int, int]

Alias for field number 2

charset_name: str

Alias for field number 0

keep_ascii_letters: bool

Alias for field number 5

language: str

Alias for field number 1

language_model: Dict[int, Dict[int, int]]
Alias for field number 3

typical_positive_ratio: float
Alias for field number 4

class chardet.sbcharsetprober.**SingleByteCharSetProber**(*model:* SingleByteCharSetModel, *is_reversed: bool = False*, *name_prober:* CharSetProber | *None = None*)

Bases: *CharSetProber*

NEGATIVE_SHORTCUT_THRESHOLD = 0.05

POSITIVE_SHORTCUT_THRESHOLD = 0.95

SAMPLE_SIZE = 64

SB_ENOUGH_REL_THRESHOLD = 1024

property charset_name: str | None

feed(*byte_str: bytes* | *bytearray*) → ProbingState

get_confidence() → float

property language: str | None

reset() → None

chardet.sbcsgroupprober module

class chardet.sbcsgroupprober.**SBCSGroupProber**
Bases: *CharSetGroupProber*

chardet.sjisprober module

class chardet.sjisprober.**SJISProber**
Bases: *MultiByteCharSetProber*

property charset_name: str

feed(*byte_str: bytes* | *bytearray*) → ProbingState

get_confidence() → float

property language: str

reset() → None

chardet.universaldetector module

Module containing the UniversalDetector detector class, which is the primary class a user of `chardet` should use.

author
Mark Pilgrim (initial port to Python)

author
Shy Shalom (original C code)

author
Dan Blanchard (major refactoring for 3.0)

author
Ian Cordasco

class chardet.universaldetector.**UniversalDetector**(*lang_filter: LanguageFilter = LanguageFilter.ALL*, *should_rename_legacy: bool = False*)

Bases: `object`

The `UniversalDetector` class underlies the `chardet.detect` function and coordinates all of the different charset probers.

To get a `dict` containing an encoding and its confidence, you can simply run:

```
u = UniversalDetector()
u.feed(some_bytes)
u.close()
detected = u.result
```

ESC_DETECTOR = re.compile(b'(\x1b|~{)')

HIGH_BYTE_DETECTOR = re.compile(b'[\x80-\xff]')

ISO_WIN_MAP = {'iso-8859-1': 'Windows-1252', 'iso-8859-13': 'Windows-1257', 'iso-8859-2': 'Windows-1250', 'iso-8859-5': 'Windows-1251', 'iso-8859-6': 'Windows-1256', 'iso-8859-7': 'Windows-1253', 'iso-8859-8': 'Windows-1255', 'iso-8859-9': 'Windows-1254'}

LEGACY_MAP = {'ascii': 'Windows-1252', 'euc-kr': 'CP949', 'gb2312': 'GB18030', 'iso-8859-1': 'Windows-1252', 'iso-8859-9': 'Windows-1254', 'tis-620': 'ISO-8859-11', 'utf-16le': 'UTF-16'}

MINIMUM_THRESHOLD = 0.2

WIN_BYTE_DETECTOR = re.compile(b'[\x80-\x9f]')

property charset_probers: List[*CharSetProber*]

close() → dict

Stop analyzing the current document and come up with a final prediction.

Returns
The `result` attribute, a `dict` with the keys *encoding*, *confidence*, and *language*.

feed(*byte_str: bytes | bytearray*) → None

Takes a chunk of a document and feeds it through all of the relevant charset probers.

After calling `feed`, you can check the value of the `done` attribute to see if you need to continue feeding the `UniversalDetector` more data, or if it has made a prediction (in the `result` attribute).

Note: You should always call `close` when you're done feeding in your document if `done` is not already `True`.

property has_win_bytes: bool

property input_state: int

reset() → None

Reset the UniversalDetector and all of its probers back to their initial states. This is called by `__init__`, so you only need to call this directly in between analyses of different documents.

chardet.utf8prober module

class chardet.utf8prober.**UTF8Prober**

Bases: *CharSetProber*

ONE_CHAR_PROB = 0.5

property charset_name: str

feed(*byte_str: bytes | bytearray*) → ProbingState

get_confidence() → float

property language: str

reset() → None

Module contents

class chardet.**UniversalDetector**(*lang_filter: LanguageFilter = LanguageFilter.ALL*, *should_rename_legacy: bool = False*)

Bases: `object`

The `UniversalDetector` class underlies the `chardet.detect` function and coordinates all of the different charset probers.

To get a `dict` containing an encoding and its confidence, you can simply run:

```
u = UniversalDetector()
u.feed(some_bytes)
u.close()
detected = u.result
```

ESC_DETECTOR = re.compile(b'(\x1b|~{)')

HIGH_BYTE_DETECTOR = re.compile(b'[\x80-\xff]')

```
ISO_WIN_MAP = {'iso-8859-1': 'Windows-1252', 'iso-8859-13':
'Windows-1257', 'iso-8859-2': 'Windows-1250', 'iso-8859-5':
'Windows-1251', 'iso-8859-6': 'Windows-1256', 'iso-8859-7':
'Windows-1253', 'iso-8859-8': 'Windows-1255', 'iso-8859-9':
'Windows-1254'}

LEGACY_MAP = {'ascii': 'Windows-1252', 'euc-kr': 'CP949', 'gb2312':
'GB18030', 'iso-8859-1': 'Windows-1252', 'iso-8859-9': 'Windows-1254',
'tis-620': 'ISO-8859-11', 'utf-16le': 'UTF-16'}

MINIMUM_THRESHOLD = 0.2

WIN_BYTE_DETECTOR = re.compile(b'[\x80-\x9f]')

property charset_probers: List[CharSetProber]
```

close() → dict

Stop analyzing the current document and come up with a final prediction.

Returns
The `result` attribute, a `dict` with the keys *encoding*, *confidence*, and *language*.

feed(*byte_str: bytes | bytearray*) → None

Takes a chunk of a document and feeds it through all of the relevant charset probers.

After calling `feed`, you can check the value of the `done` attribute to see if you need to continue feeding the `UniversalDetector` more data, or if it has made a prediction (in the `result` attribute).

Note: You should always call `close` when you're done feeding in your document if `done` is not already `True`.

```
property has_win_bytes: bool

property input_state: int
```

reset() → None

Reset the UniversalDetector and all of its probers back to their initial states. This is called by `__init__`, so you only need to call this directly in between analyses of different documents.

```
result: dict
```

chardet.**detect**(*byte_str: bytes | bytearray*, *should_rename_legacy: bool = False*) → dict

Detect the encoding of the given byte string.

Parameters

- **byte_str** (`bytes` or `bytearray`) – The byte sequence to examine.
- **should_rename_legacy** (`bool`) – Should we rename legacy encodings to their more modern equivalents?

chardet.**detect_all**(*byte_str: bytes | bytearray*, *ignore_threshold: bool = False*, *should_rename_legacy: bool = False*) → List[dict]

Detect all the possible encodings of the given byte string.

Parameters

- **byte_str** (`bytes` or `bytearray`) – The byte sequence to examine.

- **ignore_threshold** (bool) – Include encodings that are below UniversalDetector.MINIMUM_THRESHOLD in results.
- **should_rename_legacy** (bool) – Should we rename legacy encodings to their more modern equivalents?

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

C

INDEX

D

E

F

G

H

I

J

K

L

M

N