javaproperties

Release 0.4.0

2017 Apr 22

Contents

javaproperties provides support for reading & writing Java `.properties` files (both the simple line-oriented format and XML) with a simple API based on the `json` module — though, for recovering Java addicts, it also includes a *`Properties`* class intended to match the behavior of Java 8's `java.net.Properties` as much as is Pythonically possible.

Previous versions of *javaproperties* included command-line programs for basic manipulation of `.properties` files. As of version 0.4.0, these programs have been split off into a separate package, `javaproperties-cli`.

Note: Throughout the following, "text string" means a Unicode character string — `unicode` in Python 2, `str` in Python 3.

CHAPTER 1

Simple Line-Oriented `.properties` Format

Format Overview

The simple line-oriented `.properties` file format consists of a series of key-value string pairs, one (or fewer) per line, with the key & value separated by the first occurrence of an equals sign (=, optionally with surrounding whitespace), a colon (`:`, optionally with surrounding whitespace), or non-leading whitespace. A line without a separator is treated as a key whose value is the empty string. If the same key occurs more than once in a single file, only its last value is used.

Note: Lines are terminated by `\n` (LF), `\r\n` (CR LF), or `\r` (CR).

Note: For the purposes of this format, only the space character (ASCII 0x20), the tab character (ASCII 0x09), and the form feed character (ASCII 0x0C) count as whitespace.

Leading whitespace on a line is ignored, but trailing whitespace (after stripping trailing newlines) is not. Lines whose first non-whitespace character is `#` or `!` (not escaped) are comments and are ignored.

Entries can be extended across multiple lines by ending all but the last line with a backslash; the backslash, the line ending after it, and any leading whitespace on the next line will all be discarded. A backslash at the end of a comment line has no effect. A comment line after a line that ends with a backslash is treated as part of a normal key-value entry, not as a comment.

Occurrences of =, `:`, `#`, `!`, and whitespace inside a key or value are escaped with a backslash. In addition, the following escape sequences are recognized:

```
\t \n \f \r \uXXXX \\
```

If a backslash is followed by any other character, the backslash is dropped.

By default, keys & values in `.properties` files are encoded in ASCII, and comments are encoded in Latin-1; characters outside these ranges, along with any unprintable characters, are escaped with the escape sequences listed

above. Unicode characters outside the Basic Multilingual Plane are first converted to UTF-16 surrogate pairs before escaping with `\uXXXX` escapes.

Functions

`javaproperties.dump`(*props*, *fp*, *separator=u'='*, *comments=None*, *timestamp=True*, *sort_keys=False*)

Write a series of key-value pairs to a file in simple line-oriented `.properties` format.

Parameters

- **`props`** – A mapping or iterable of `(key, value)` pairs to write to `fp`. All keys and values in `props` must be text strings. If `sort_keys` is `False`, the entries are output in iteration order.
- **`fp`** – A file-like object to write the values of `props` to. It must have been opened as a text file with a Latin-1-compatible encoding.
- **`separator`** (*`text string`*) – The string to use for separating keys & values. Only `" "`, `"="`, and `":"` (possibly with added whitespace) should ever be used as the separator.
- **`comments`** (text string or `None`) – if non-`None`, `comments` will be written to `fp` as a comment before any other content
- **`timestamp`** (`None`, `bool`, number, or `datetime.datetime`) – If neither `None` nor `False`, a timestamp in the form of `Mon Sep 02 14:00:54 EDT 2016` is written as a comment to `fp` after `comments` (if any) and before the key-value pairs. If `timestamp` is `True`, the current date & time is used. If it is a number, it is converted from seconds since the epoch to local time. If it is a `datetime.datetime` object, its value is used directly, with naïve objects assumed to be in the local timezone.
- **`sort_keys`** (*`bool`*) – if true, the elements of `props` are sorted lexicographically by key in the output

Returns `None`

`javaproperties.dumps`(*props*, *separator=u'='*, *comments=None*, *timestamp=True*, *sort_keys=False*)

Convert a series of key-value pairs to a text string in simple line-oriented `.properties` format.

Parameters

- **`props`** – A mapping or iterable of `(key, value)` pairs to serialize. All keys and values in `props` must be text strings. If `sort_keys` is `False`, the entries are output in iteration order.
- **`separator`** (*`text string`*) – The string to use for separating keys & values. Only `" "`, `"="`, and `":"` (possibly with added whitespace) should ever be used as the separator.
- **`comments`** (text string or `None`) – if non-`None`, `comments` will be output as a comment before any other content
- **`timestamp`** (`None`, `bool`, number, or `datetime.datetime`) – If neither `None` nor `False`, a timestamp in the form of `Mon Sep 02 14:00:54 EDT 2016` is output as a comment after `comments` (if any) and before the key-value pairs. If `timestamp` is `True`, the current date & time is used. If it is a number, it is converted from seconds since the epoch to local time. If it is a `datetime.datetime` object, its value is used directly, with naïve objects assumed to be in the local timezone.
- **`sort_keys`** (*`bool`*) – if true, the elements of `props` are sorted lexicographically by key in the output

Return type text string

`javaproperties.load`(*fp*, *object_pairs_hook=<type 'dict'>*)

Parse the contents of the `readline`-supporting file-like object `fp` as a simple line-oriented `.properties` file and return a `dict` of the key-value pairs.

`fp` may be either a text or binary filehandle, with or without universal newlines enabled. If it is a binary filehandle, its contents are decoded as Latin-1.

By default, the key-value pairs extracted from `fp` are combined into a `dict` with later occurrences of a key overriding previous occurrences of the same key. To change this behavior, pass a callable as the `object_pairs_hook` argument; it will be called with one argument, a generator of `(key, value)` pairs representing the key-value entries in `fp` (including duplicates) in order of occurrence. *`load`* will then return the value returned by `object_pairs_hook`.

Parameters

- **`fp`** (*`file-like object`*) – the file from which to read the `.properties` document
- **`object_pairs_hook`** (*`callable`*) – class or function for combining the key-value pairs

Return type `dict` of text strings or the return value of `object_pairs_hook`

`javaproperties.loads`(*s*, *object_pairs_hook=<type 'dict'>*)

Parse the contents of the string `s` as a simple line-oriented `.properties` file and return a `dict` of the key-value pairs.

`s` may be either a text string or bytes string. If it is a bytes string, its contents are decoded as Latin-1.

By default, the key-value pairs extracted from `s` are combined into a `dict` with later occurrences of a key overriding previous occurrences of the same key. To change this behavior, pass a callable as the `object_pairs_hook` argument; it will be called with one argument, a generator of `(key, value)` pairs representing the key-value entries in `s` (including duplicates) in order of occurrence. *`loads`* will then return the value returned by `object_pairs_hook`.

Parameters

- **`s`** (*`string`*) – the string from which to read the `.properties` document
- **`object_pairs_hook`** (*`callable`*) – class or function for combining the key-value pairs

Return type `dict` of text strings or the return value of `object_pairs_hook`

CHAPTER 2

XML .properties Format

Format Overview

The XML .properties file format encodes a series of key-value string pairs (and optionally also a comment) as an XML document conforming to the following Document Type Definition (published at <http://java.sun.com/dtd/properties.dtd>):

```
<!ELEMENT properties (comment?, entry*)>
<!ATTLIST properties version CDATA #FIXED "1.0">
<!ELEMENT comment (#PCDATA)>
<!ELEMENT entry (#PCDATA)>
<!ATTLIST entry key CDATA #REQUIRED>
```

Functions

javaproperties.**dump_xml**(*props*, *fp*, *comment=None*, *encoding=u'UTF-8'*, *sort_keys=False*)

Write a series `props` of key-value pairs to a binary filehandle `fp` in the format of an XML properties file. The file will include both an XML declaration and a doctype declaration.

Parameters

- **props** – A mapping or iterable of `(key, value)` pairs to write to `fp`. All keys and values in `props` must be text strings. If `sort_keys` is `False`, the entries are output in iteration order.
- **fp** (*binary file-like object*) – a file-like object to write the values of `props` to
- **comment** (text string or `None`) – if non-`None`, `comment` will be output as a `<comment>` element before the `<entry>` elements
- **encoding** (*string*) – the name of the encoding to use for the XML document (also included in the XML declaration)

- **sort_keys** (*bool*) – if true, the elements of `props` are sorted lexicographically by key in the output

Returns `None`

`javaproperties.`**`dumps_xml`**(*props*, *comment=None*, *sort_keys=False*)

Convert a series `props` of key-value pairs to a text string containing an XML properties document. The document will include a doctype declaration but not an XML declaration.

Parameters

- **props** – A mapping or iterable of `(key, value)` pairs to serialize. All keys and values in `props` must be text strings. If `sort_keys` is `False`, the entries are output in iteration order.
- **comment** (text string or `None`) – if non-`None`, `comment` will be output as a `<comment>` element before the `<entry>` elements
- **sort_keys** (*bool*) – if true, the elements of `props` are sorted lexicographically by key in the output

Return type text string

`javaproperties.`**`load_xml`**(*fp*, *object_pairs_hook=<type 'dict'>*)

Parse the contents of the file-like object `fp` as an XML properties file and return a `dict` of the key-value pairs.

Beyond basic XML well-formedness, *`load_xml`* only checks that the root element is named "`properties`" and that all of its `<entry>` children have `key` attributes. No further validation is performed; if any `<entry>`s happen to contain nested tags, the behavior is undefined.

By default, the key-value pairs extracted from `fp` are combined into a `dict` with later occurrences of a key overriding previous occurrences of the same key. To change this behavior, pass a callable as the `object_pairs_hook` argument; it will be called with one argument, a generator of `(key, value)` pairs representing the key-value entries in `fp` (including duplicates) in order of occurrence. *`load_xml`* will then return the value returned by `object_pairs_hook`.

Note: This uses `xml.etree.ElementTree` for parsing, which does not have decent support for `unicode` input in Python 2. Files containing non-ASCII characters need to be opened in binary mode in Python 2, while Python 3 accepts both binary and text input.

Parameters

- **fp** (*file-like object*) – the file from which to read the XML properties document
- **object_pairs_hook** (*callable*) – class or function for combining the key-value pairs

Return type `dict` or the return value of `object_pairs_hook`

Raises **ValueError** – if the root of the XML tree is not a `<properties>` tag or an `<entry>` element is missing a `key` attribute

`javaproperties.`**`loads_xml`**(*s*, *object_pairs_hook=<type 'dict'>*)

Parse the contents of the string `s` as an XML properties document and return a `dict` of the key-value pairs.

Beyond basic XML well-formedness, *`loads_xml`* only checks that the root element is named "`properties`" and that all of its `<entry>` children have `key` attributes. No further validation is performed; if any `<entry>`s happen to contain nested tags, the behavior is undefined.

By default, the key-value pairs extracted from `s` are combined into a `dict` with later occurrences of a key overriding previous occurrences of the same key. To change this behavior, pass a callable as the `object_pairs_hook` argument; it will be called with one argument, a generator of `(key, value)` pairs representing the key-value entries in `s` (including duplicates) in order of occurrence. *`loads_xml`* will then return the value returned by `object_pairs_hook`.

Note: This uses `xml.etree.ElementTree` for parsing, which does not have decent support for `unicode` input in Python 2. Strings containing non-ASCII characters need to be encoded as bytes in Python 2 (Use either UTF-8 or UTF-16 if the XML document does not contain an encoding declaration), while Python 3 accepts both binary and text input.

Parameters

- **`s`** (*`string`*) – the string from which to read the XML properties document
- **`object_pairs_hook`** (*`callable`*) – class or function for combining the key-value pairs

Return type `dict` or the return value of `object_pairs_hook`

Raises **`ValueError`** – if the root of the XML tree is not a `<properties>` tag or an `<entry>` element is missing a `key` attribute

CHAPTER 3

`Properties` Class

class `javaproperties.`**`Properties`**(*data=None*, *defaults=None*)

A port of Java 8's `java.net.Properties` that tries to match its behavior as much as is Pythonically possible. *`Properties`* behaves like a normal `MutableMapping` class (i.e., you can do `props[key] = value` and so forth), except that it may only be used to store strings (`str` and `unicode` in Python 2; just `str` in Python 3). Attempts to use a non-string object as a key or value will produce a `TypeError`.

Parameters

- **`data`** (mapping or `None`) – A mapping or iterable of `(key, value)` pairs with which to initialize the *`Properties`* object. All keys and values in `data` must be text strings.
- **`defaults`** (*`Properties`* or `None`) – a set of default properties that will be used as fallback for *`getProperty`*

`defaults` = **None**

A *`Properties`* subobject used as fallback for *`getProperty`*. Only *`getProperty`*, *`propertyNames`*, and *`stringPropertyNames`* use this attribute; all other methods (including the standard mapping methods) ignore it.

`getProperty`(*key*, *defaultValue=None*)

Fetch the value associated with the key `key` in the *`Properties`* object. If the key is not present, *`defaults`* is checked, and then *its* *`defaults`*, etc., until either a value for `key` is found or the next *`defaults`* is `None`, in which case `defaultValue` is returned.

Parameters

- **`key`** (*`text string`*) – the key to look up the value of
- **`defaultValue`** – the value to return if `key` is not found in the *`Properties`* object

Return type text string (if `key` was found)

Raises **`TypeError`** – if `key` is not a string

`load`(*inStream*)

Update the *`Properties`* object with the entries in a `.properties` file or file-like object.

`inStream` may be either a text or binary filehandle, with or without universal newlines enabled. If it is a binary filehandle, its contents are decoded as Latin-1.

Parameters **inStream** (*file-like object*) – the file from which to read the `.properties` document

Returns `None`

loadFromXML(*inStream*)

Update the *Properties* object with the entries in the XML properties file `inStream`.

Beyond basic XML well-formedness, *loadFromXML* only checks that the root element is named `properties` and that all of its `entry` children have `key` attributes; no further validation is performed.

Note: This uses `xml.etree.ElementTree` for parsing, which does not have decent support for `unicode` input in Python 2. Files containing non-ASCII characters need to be opened in binary mode in Python 2, while Python 3 accepts both binary and text input.

Parameters **inStream** (*file-like object*) – the file from which to read the XML properties document

Returns `None`

Raises **ValueError** – if the root of the XML tree is not a `<properties>` tag or an `<entry>` element is missing a `key` attribute

propertyNames()

Returns a generator of all distinct keys in the *Properties* object and its *defaults* (and its *defaults*'s *defaults*, etc.) in unspecified order

Return type generator of text strings

setProperty(*key*, *value*)

Equivalent to `self[key] = value`

store(*out*, *comments=None*)

Write the *Properties* object's entries (in unspecified order) in `.properties` format to `out`, including the current timestamp.

Parameters

- **out** – A file-like object to write the properties to. It must have been opened as a text file with a Latin-1-compatible encoding.
- **comments** (text string or `None`) – If non-`None`, `comments` will be written to `out` as a comment before any other content

Returns `None`

storeToXML(*out*, *comment=None*, *encoding='UTF-8'*)

Write the *Properties* object's entries (in unspecified order) in XML properties format to `out`.

Parameters

- **out** (*binary file-like object*) – a file-like object to write the properties to
- **comment** (text string or `None`) – if non-`None`, `comment` will be output as a `<comment>` element before the `<entry>` elements
- **encoding** (*string*) – the name of the encoding to use for the XML document (also included in the XML declaration)

Returns `None`

stringPropertyNames()

Returns a `set` of all keys in the *`Properties`* object and its *`defaults`* (and its *`defaults`*'s *`defaults`*, etc.)

Return type `set` of text strings

CHAPTER 4

PropertiesFile Class

class javaproperties.**PropertiesFile**(*mapping=None*, ***kwargs*)

New in version 0.3.0.

A custom mapping class for reading from, editing, and writing to a `.properties` file while preserving comments & whitespace in the original input.

A *PropertiesFile* instance can be constructed from another mapping and/or iterable of pairs, after which it will act like an `OrderedDict`. Alternatively, an instance can be constructed from a file or string with *PropertiesFile.load()* or *PropertiesFile.loads()*, and the resulting instance will remember the formatting of its input and retain that formatting when written back to a file or string with the *dump()* or *dumps()* method. The formatting information attached to an instance `pf` can be forgotten by constructing another mapping from it via `dict(pf)`, `OrderedDict(pf)`, or even `PropertiesFile(pf)` (Use the *copy()* method if you want to create another *PropertiesFile* instance with the same data & formatting).

When not reading or writing, *PropertiesFile* behaves like a normal `MutableMapping` class (i.e., you can do `props[key] = value` and so forth), except that (a) like `OrderedDict`, key insertion order is remembered and is used when iterating & dumping (and `reversed` is supported), and (b) like *Properties*, it may only be used to store strings and will raise a `TypeError` if passed a non-string object as key or value.

Two *PropertiesFile* instances compare equal iff both their key-value pairs and comment & whitespace lines are equal and in the same order. When comparing a *PropertiesFile* to any other type of mapping, only the key-value pairs are considered, and order is ignored.

PropertiesFile currently only supports reading & writing the simple line-oriented format, not XML.

copy()

Create a copy of the mapping, including formatting information

dump(*fp*, *separator='='*)

Write the mapping to a file in simple line-oriented `.properties` format.

If the instance was originally created from a file or string with *PropertiesFile.load()* or *PropertiesFile.loads()*, then the output will include the comments and whitespace from the original input, and any keys that haven't been deleted or reassigned will retain their original formatting and multiplicity. Key-value pairs that have been modified or added to the mapping will be reformatted

with *join_key_value()* using the given separator. All key-value pairs are output in the order they were defined, with new keys added to the end.

Note: Serializing a *PropertiesFile* instance with the *dump()* function instead will cause all formatting information to be ignored, as *dump()* will treat the instance like a normal mapping.

Parameters

- **fp** – A file-like object to write the mapping to. It must have been opened as a text file with a Latin-1-compatible encoding.
- **separator** (*text string*) – The string to use for separating new or modified keys & values. Only `" "`, `"="`, and `":"` (possibly with added whitespace) should ever be used as the separator.

Returns `None`

dumps (*separator='='*)

Convert the mapping to a text string in simple line-oriented `.properties` format.

If the instance was originally created from a file or string with *PropertiesFile.load()* or *PropertiesFile.loads()*, then the output will include the comments and whitespace from the original input, and any keys that haven't been deleted or reassigned will retain their original formatting and multiplicity. Key-value pairs that have been modified or added to the mapping will be reformatted with *join_key_value()* using the given separator. All key-value pairs are output in the order they were defined, with new keys added to the end.

Note: Serializing a *PropertiesFile* instance with the *dumps()* function instead will cause all formatting information to be ignored, as *dumps()* will treat the instance like a normal mapping.

Parameters **separator** (*text string*) – The string to use for separating new or modified keys & values. Only `" "`, `"="`, and `":"` (possibly with added whitespace) should ever be used as the separator.

Return type text string

classmethod load (*fp*)

Parse the contents of the `readline`-supporting file-like object `fp` as a simple line-oriented `.properties` file and return a *PropertiesFile* instance.

`fp` may be either a text or binary filehandle, with or without universal newlines enabled. If it is a binary filehandle, its contents are decoded as Latin-1.

Parameters **fp** (*file-like object*) – the file from which to read the `.properties` document

Return type *PropertiesFile*

classmethod loads (*s*)

Parse the contents of the string `s` as a simple line-oriented `.properties` file and return a *PropertiesFile* instance.

`s` may be either a text string or bytes string. If it is a bytes string, its contents are decoded as Latin-1.

Parameters **s** (*string*) – the string from which to read the `.properties` document

Return type *PropertiesFile*

CHAPTER 5

Low-Level Utilities

`javaproperties.escape`(*field*)

Escape a string so that it can be safely used as either a key or value in a `.properties` file. All non-ASCII characters, all nonprintable or space characters, and the characters `\ # ! = :` are all escaped using either the single-character escapes recognized by *`unescape`* (when they exist) or `\uXXXX` escapes (after converting non-BMP characters to surrogate pairs).

Parameters **`field`**(*`text string`*) – the string to escape

Return type text string

`javaproperties.java_timestamp`(*timestamp=True*)

New in version 0.2.0.

Returns a timestamp in the format produced by Java 8's `Date.toString()`, e.g.:

```
Mon Sep 02 14:00:54 EDT 2016
```

If `timestamp` is `True` (the default), the current date & time is returned.

If `timestamp` is `None` or `False`, an empty string is returned.

If `timestamp` is a number, it is converted from seconds since the epoch to local time.

If `timestamp` is a `datetime.datetime` object, its value is used directly, with naïve objects assumed to be in the local timezone.

The timestamp is always constructed using the C locale.

Parameters **`timestamp`** (`None`, `bool`, number, or `datetime.datetime`) – the date & time to display

Return type text string

`javaproperties.join_key_value`(*key*, *value*, *separator=u'='*)

Join a key and value together into a single line suitable for adding to a simple line-oriented `.properties` file. No trailing newline is added.

```
>>> join_key_value('possible separators', '= : space')
'possible\\ separators=\\= \\: space'
```

Parameters

- **key** (*text string*) – the key
- **value** (*text string*) – the value
- **separator** (*text string*) – the string to use for separating the key & value. Only `" "`, `"="`, and `":"` (possibly with added whitespace) should ever be used as the separator.

Return type text string

`javaproperties.`**`parse`**(*fp*)

Parse the contents of the `readline`-supporting file-like object `fp` as a simple line-oriented `.properties` file and return a generator of `(key, value, original_lines)` triples for every entry in `fp` (including duplicate keys) in order of occurrence. The third element of each triple is the concatenation of the unmodified lines in `fp` (including trailing newlines) from which the key and value were extracted. The generator also includes comments and blank/all-whitespace lines found in `fp`, one triple per line, with the first two elements of the triples set to `None`. This is the only way to extract comments from a `.properties` file with this library.

`fp` may be either a text or binary filehandle, with or without universal newlines enabled. If it is a binary filehandle, its contents are decoded as Latin-1.

Parameters **fp** (*file-like object*) – the file from which to read the `.properties` document

Return type generator of triples of text strings

`javaproperties.`**`to_comment`**(*comment*)

Convert a string to a `.properties` file comment. All non-Latin-1 characters in the string are escaped using `\uXXXX` escapes (after converting non-BMP characters to surrogate pairs), a `#` is prepended to the string, any CR LF or CR line breaks in the string are converted to LF, and a `#` is inserted after any line break not already followed by a `#` or `!`. No trailing newline is added.

```
>>> to_comment('They say foo=bar,\r\nbut does bar=foo?')
'#They say foo=bar,\n#but does bar=foo?'
```

Parameters **comment** (*text string*) – the string to convert to a comment

Return type text string

`javaproperties.`**`unescape`**(*field*)

Decode escape sequences in a `.properties` key or value. The following escape sequences are recognized:

```
\t \n \f \r \uXXXX \\
```

If a backslash is followed by any other character, the backslash is dropped.

In addition, any valid UTF-16 surrogate pairs in the string after escape-decoding are further decoded into the non-BMP characters they represent. (Invalid & isolated surrogate code points are left as-is.)

Parameters **field** (*text string*) – the string to decode

Return type text string

CHAPTER 6

Command-Line Utilities

As of version 0.4.0, the command-line programs have been split off into a separate package, `javaproperties-cli`, which must be installed separately in order to use them. See the package's documentation for details.

CHAPTER 7

Indices and tables

- genindex
- search

Python Module Index

j

Index