javaproperties

Release 0.9.0.dev1

John T. Wodder II

2024 Feb 05

CONTENTS

GitHub | PyPI | Documentation | Issues | *Changelog*

CHAPTER

ONE

SIMPLE LINE-ORIENTED `.PROPERTIES` FORMAT

1.1 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 \\
```

Unicode characters outside the Basic Multilingual Plane can be represented by a pair of `\uXXXX` escape sequences encoding the corresponding UTF-16 surrogate pair.

If a backslash is followed by character other than those listed above, the backslash is discarded.

An example simple line-oriented `.properties` file:

```
#This is a comment.
foo=bar
baz: quux
gnusto cleesh
snowman = \u2603
goat = \ud83d\udc10
novalue
host\:port=127.0.0.1\:80
```

This corresponds to the Python `dict`:

```
{
    "foo": "bar",
    "baz": "quux",
    "gnusto": "cleesh",
    "snowman": "",
    "goat": "",
    "novalue": "",
    "host:port": "127.0.0.1:80",
}
```

1.2 File Encoding

Although the *load()* and *loads()* functions accept arbitrary Unicode characters in their input, by default the *dump()* and *dumps()* functions limit the characters in their output as follows:

- When `ensure_ascii` is `True` (the default), *dump()* and *dumps()* output keys & values in pure ASCII; non-ASCII and unprintable characters are escaped with the escape sequences listed above. When `ensure_ascii` is `False`, the functions instead pass all non-ASCII characters through as-is; unprintable characters are still escaped.
- When `ensure_ascii_comments` is `None` (the default), *dump()* and *dumps()* output the `comments` argument (if set) using only Latin-1 (ISO-8859-1) characters; all other characters are escaped. When `ensure_ascii_comments` is `True`, the functions instead escape all non-ASCII characters in `comments`. When `ensure_ascii_comments` is `False`, the functions instead pass all characters in `comments` through as-is.
  – Note that, in order to match the behavior of Java's `Properties` class, unprintable ASCII characters in `comments` are always passed through as-is rather than escaped.
  – Newlines inside `comments` are not escaped, but a `#` is inserted after every one not already followed by a `#` or `!`.

When writing properties to a file, you must either (a) open the file using an encoding that supports all of the characters in the formatted output or else (b) open the file using the *'javapropertiesreplace' error handler* defined by this module. The latter option allows one to write valid simple-format properties files in any encoding without having to worry about whether the properties or comment contain any characters not representable in the encoding.

1.3 Functions

`javaproperties.`**`dump`**(*props: Mapping[str, str] | Iterable[tuple[str, str]], fp: TextIO, separator: str = '=', comments: str | None = None, timestamp: None | bool | float | datetime = True, sort_keys: bool = False, ensure_ascii: bool = True, ensure_ascii_comments: bool | None = None*) → None

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

Changed in version 0.6.0: `ensure_ascii` and `ensure_ascii_comments` parameters added

Parameters

- **`props`** – A mapping or iterable of `(key, value)` pairs to write to `fp`. All keys and values in `props` must be `str` values. If `sort_keys` is `False`, the entries are output in iteration order.

- **fp** (*TextIO*) – A file-like object to write the values of `props` to. It must have been opened as a text file.
- **separator** (*str*) – The string to use for separating keys & values. Only `" "`, `"="`, and `":"` (possibly with added whitespace) should ever be used as the separator.
- **comments** (*Optional[str]*) – 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
- **ensure_ascii** (*bool*) – if true, all non-ASCII characters will be replaced with `\uXXXX` escape sequences in the output; if false, non-ASCII characters will be passed through as-is
- **ensure_ascii_comments** (*Optional[bool]*) – if true, all non-ASCII characters in `comments` will be replaced with `\uXXXX` escape sequences in the output; if `None`, only non-Latin-1 characters will be escaped; if false, no characters will be escaped

Returns

`None`

`javaproperties.`**`dumps`**(*props: Mapping[str, str] | Iterable[tuple[str, str]], separator: str = '=', comments: str | None = None, timestamp: None | bool | float | datetime = True, sort_keys: bool = False, ensure_ascii: bool = True, ensure_ascii_comments: bool | None = None*) → str

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

Changed in version 0.6.0: `ensure_ascii` and `ensure_ascii_comments` parameters added

Parameters

- **props** – A mapping or iterable of `(key, value)` pairs to serialize. All keys and values in `props` must be `str` values. If `sort_keys` is `False`, the entries are output in iteration order.
- **separator** (*str*) – The string to use for separating keys & values. Only `" "`, `"="`, and `":"` (possibly with added whitespace) should ever be used as the separator.
- **comments** (*Optional[str]*) – 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
- **ensure_ascii** (*bool*) – if true, all non-ASCII characters will be replaced with `\uXXXX` escape sequences in the output; if false, non-ASCII characters will be passed through as-is

- **`ensure_ascii_comments`** (*`Optional[bool]`*) – if true, all non-ASCII characters in `comments` will be replaced with `\uXXXX` escape sequences in the output; if `None`, only non-Latin-1 characters will be escaped; if false, no characters will be escaped

Return type
text string

`javaproperties.load`(*fp: IO*) → dict[str, str]

`javaproperties.load`(*fp: IO*, *object_pairs_hook: type[T]*) → T

`javaproperties.load`(*fp: IO*, *object_pairs_hook: Callable[[Iterator[tuple[str, str]]], T]*) → T

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`.

Changed in version 0.5.0: Invalid `\uXXXX` escape sequences will now cause an *`InvalidUEscapeError`* to be raised

Parameters

- **`fp`** (*`IO`*) – 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`

Raises
`InvalidUEscapeError` – if an invalid `\uXXXX` escape sequence occurs in the input

`javaproperties.loads`(*s: str | bytes*) → dict[str, str]

`javaproperties.loads`(*s: str | bytes*, *object_pairs_hook: type[T]*) → T

`javaproperties.loads`(*s: str | bytes*, *object_pairs_hook: Callable[[Iterator[tuple[str, str]]], T]*) → T

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`.

Changed in version 0.5.0: Invalid `\uXXXX` escape sequences will now cause an *`InvalidUEscapeError`* to be raised

Parameters

- **`s`** (*`Union[str,bytes]`*) – 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`

Raises

InvalidUEscapeError – if an invalid `\uXXXX` escape sequence occurs in the input

CHAPTER

TWO

XML .PROPERTIES FORMAT

2.1 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>
```

An example XML `.properties` file:

```
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
<comment>This is a comment.</comment>
<entry key="foo">bar</entry>
<entry key="snowman"></entry>
<entry key="goat"></entry>
<entry key="host:port">127.0.0.1:80</entry>
</properties>
```

This corresponds to the Python `dict`:

```
{
    "foo": "bar",
    "snowman": "",
    "goat": "",
    "host:port": "127.0.0.1:80",
}
```

2.2 Functions

`javaproperties.`**`dump_xml`**(*props: Mapping[str, str] | Iterable[tuple[str, str]], fp: BinaryIO, comment: str | None = None, encoding: str = 'UTF-8', sort_keys: bool = False*) → None

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 `str` values. If `sort_keys` is `False`, the entries are output in iteration order.
- **`fp`** (*`BinaryIO`*) – a file-like object to write the values of `props` to
- **`comment`** (*`Optional[str]`*) – if non-`None`, `comment` will be output as a `<comment>` element before the `<entry>` elements
- **`encoding`** (*`str`*) – 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: Mapping[str, str] | Iterable[tuple[str, str]], comment: str | None = None, sort_keys: bool = False*) → str

Convert a series `props` of key-value pairs to a `str` 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 `str` values. If `sort_keys` is `False`, the entries are output in iteration order.
- **`comment`** (*`Optional[str]`*) – 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
str

`javaproperties.`**`load_xml`**(*fp: IO*) → dict[str, str]

`javaproperties.`**`load_xml`**(*fp: IO, object_pairs_hook: type[T]*) → T

`javaproperties.`**`load_xml`**(*fp: IO, object_pairs_hook: Callable[[Iterator[tuple[str, str]]], T]*) → T

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`.

Parameters

- **fp** (*IO*) – 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: AnyStr*`)` → dict[str, str]

`javaproperties.`**`loads_xml`**`(`*fp: IO*, *object_pairs_hook: type[T]*`)` → T

`javaproperties.`**`loads_xml`**`(`*s: AnyStr*, *object_pairs_hook: Callable[[Iterator[tuple[str, str]]], T]*`)` → T

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`.

Parameters

- **s** (*Union[str, bytes]*) – 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

THREE

PROPERTIES CLASS

class javaproperties.**Properties**(*data: None | Mapping[str, str] | Iterable[tuple[str, str]] = None*, *defaults: Properties | None = None*)

A port of Java 8's `java.util.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 `str` values.

Two *`Properties`* instances compare equal iff both their key-value pairs and *`defaults`* attributes are equal. When comparing a *`Properties`* instance to any other type of mapping, only the key-value pairs are considered.

Changed in version 0.5.0: *`Properties`* instances can now compare equal to `dict`s and other mapping types

Parameters

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

`copy`() → *Properties*

New in version 0.5.0.

Create a shallow copy of the mapping. The copy's *`defaults`* attribute will be the same instance as the original's *`defaults`*.

`defaults`

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

`getProperty`(*key: str*, *defaultValue: T | None = None*) → str | T | None

Fetch the value associated with the key `key` in the *`Properties`* instance. 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`** (*`str`*) – the key to look up the value of
- **`defaultValue`** (*`Any`*) – the value to return if `key` is not found in the *`Properties`* instance

Return type

str (if `key` was found)

load(*inStream: IO*) → None

Update the *Properties* instance 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.

Changed in version 0.5.0: Invalid `\uXXXX` escape sequences will now cause an *InvalidUEscapeError* to be raised

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

Returns
None

Raises
InvalidUEscapeError – if an invalid `\uXXXX` escape sequence occurs in the input

loadFromXML(*inStream: IO*) → None

Update the *Properties* instance 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.

Parameters
inStream (*IO*) – 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() → Iterator[str]

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

Return type
Iterator[str]

setProperty(*key: str*, *value: str*) → None

Equivalent to `self[key] = value`

store(*out: TextIO*, *comments: str* | *None = None*) → None

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

Parameters

- **out** (*TextIO*) – 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** (*Optional[str]*) – If non-None, `comments` will be written to `out` as a comment before any other content

Returns
None

storeToXML(*out: BinaryIO*, *comment: str* | *None = None*, *encoding: str = 'UTF-8'*) → None

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

Parameters

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

Returns
None

stringPropertyNames() → set[str]

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

Return type
set[str]

CHAPTER

FOUR

PROPERTIESFILE CLASS

class javaproperties.**PropertiesFile**(*mapping: None | Mapping[str, str] | Iterable[tuple[str, str]] = None, **kwargs: str*)

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() → *PropertiesFile*

Create a copy of the mapping, including formatting information

dump(*fp: TextIO, separator: str = '=', ensure_ascii: bool = True*) → None

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 and `ensure_ascii` setting. All key-value pairs are output in the order they were defined, with new keys added to the end.

Changed in version 0.8.0: `ensure_ascii` parameter added

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** (*TextIO*) – 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** (*str*) – The string to use for separating new or modified keys & values. Only `" "`, `"="`, and `":"` (possibly with added whitespace) should ever be used as the separator.
- **ensure_ascii** (*bool*) – if true, all non-ASCII characters in new or modified key-value pairs will be replaced with `\uXXXX` escape sequences in the output; if false, non-ASCII characters will be passed through as-is

Returns

`None`

dumps(*separator: str = '='*, *ensure_ascii: bool = True*) → str

Convert the mapping to a `str` 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 and `ensure_ascii` setting. All key-value pairs are output in the order they were defined, with new keys added to the end.

Changed in version 0.8.0: `ensure_ascii` parameter added

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** (*str*) – The string to use for separating new or modified keys & values. Only `" "`, `"="`, and `":"` (possibly with added whitespace) should ever be used as the separator.
- **ensure_ascii** (*bool*) – if true, all non-ASCII characters in new or modified key-value pairs will be replaced with `\uXXXX` escape sequences in the output; if false, non-ASCII characters will be passed through as-is

Return type

str

property header_comment: str | None

New in version 0.7.0.

The concatenated values of all comments at the top of the file, up to (but not including) the first key-value pair or timestamp comment, whichever comes first. The comments are returned with comment markers and the whitespace leading up to them removed, with line endings changed to `\n`, and with the line ending on the final comment (if any) removed. Blank/all-whitespace lines among the comments are ignored.

The header comment can be changed by assigning to this property. Assigning a string `s` causes everything before the first key-value pair or timestamp comment to be replaced by the output of `to_comment(s)`. Assigning `None` causes the header comment to be deleted (also achievable with `del pf.header_comment`).

```
>>> pf = PropertiesFile.loads('''\
... #This is a comment.
...   ! This is also a comment.
```

(continues on next page)

(continued from previous page)

```
... #Tue Feb 25 19:13:27 EST 2020
... key = value
... zebra: apple
... ''')
>>> pf.header_comment
'This is a comment.\n This is also a comment.'
>>> pf.header_comment = 'New comment'
>>> print(pf.dumps(), end='')
#New comment
#Tue Feb 25 19:13:27 EST 2020
key = value
zebra: apple
>>> del pf.header_comment
>>> pf.header_comment is None
True
>>> print(pf.dumps(), end='')
#Tue Feb 25 19:13:27 EST 2020
key = value
zebra: apple
```

classmethod load(*fp: IO*) → *PropertiesFile*

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.

Changed in version 0.5.0: Invalid `\uXXXX` escape sequences will now cause an `InvalidUEscapeError` to be raised

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

Return type
PropertiesFile

Raises
`InvalidUEscapeError` – if an invalid `\uXXXX` escape sequence occurs in the input

classmethod loads(*s: AnyStr*) → *PropertiesFile*

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.

Changed in version 0.5.0: Invalid `\uXXXX` escape sequences will now cause an `InvalidUEscapeError` to be raised

Parameters
s (*Union[str, bytes]*) – the string from which to read the `.properties` document

Return type
PropertiesFile

Raises
`InvalidUEscapeError` – if an invalid `\uXXXX` escape sequence occurs in the input

property timestamp: str | None

New in version 0.7.0.

The value of the timestamp comment, with the comment marker, any whitespace leading up to it, and the trailing newline removed. The timestamp comment is the first comment that appears to be a valid timestamp as produced by Java 8's `Date.toString()` and that does not come after any key-value pairs; if there is no such comment, the value of this property is `None`.

The timestamp can be changed by assigning to this property. Assigning a string `s` replaces the timestamp comment with the output of `to_comment(s)`; no check is made as to whether the result is a valid timestamp comment. Assigning `None` or `False` causes the timestamp comment to be deleted (also achievable with `del pf.timestamp`). Assigning any other value `x` replaces the timestamp comment with the output of `to_comment(java_timestamp(x))`.

```
>>> pf = PropertiesFile.loads('''\
... #This is a comment.
... #Tue Feb 25 19:13:27 EST 2020
... key = value
... zebra: apple
... ''')
>>> pf.timestamp
'Tue Feb 25 19:13:27 EST 2020'
>>> pf.timestamp = 1234567890
>>> pf.timestamp
'Fri Feb 13 18:31:30 EST 2009'
>>> print(pf.dumps(), end='')
#This is a comment.
#Fri Feb 13 18:31:30 EST 2009
key = value
zebra: apple
>>> del pf.timestamp
>>> pf.timestamp is None
True
>>> print(pf.dumps(), end='')
#This is a comment.
key = value
zebra: apple
```

CHAPTER

FIVE

LOW-LEVEL UTILITIES

javaproperties.**escape**(*field: str*, *ensure_ascii: bool = True*) → str

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).

Changed in version 0.6.0: `ensure_ascii` parameter added

Parameters

- **field** (*str*) – the string to escape
- **ensure_ascii** (*bool*) – if true, all non-ASCII characters will be replaced with `\uXXXX` escape sequences in the output; if false, non-ASCII characters will be passed through as-is

Return type

str

javaproperties.**java_timestamp**(*timestamp: None | bool | float | datetime = True*) → str

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

str

javaproperties.**join_key_value**(*key: str*, *value: str*, *separator: str = '='*, *ensure_ascii: bool = True*) → str

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'
```

Changed in version 0.6.0: `ensure_ascii` parameter added

Parameters

- **`key`** (*`str`*) – the key
- **`value`** (*`str`*) – the value
- **`separator`** (*`str`*) – the string to use for separating the key & value. Only `" "`, `"="`, and `":"` (possibly with added whitespace) should ever be used as the separator.
- **`ensure_ascii`** (*`bool`*) – if true, all non-ASCII characters will be replaced with `\\uXXXX` escape sequences in the output; if false, non-ASCII characters will be passed through as-is

Return type

str

`javaproperties.`**`to_comment`**(*comment: str*, *ensure_ascii: bool | None = None*) → str

Convert a string to a `.properties` file comment. Non-Latin-1 or non-ASCII characters in the string may be escaped using `\uXXXX` escapes (depending on the value of `ensure_ascii`), 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?'
```

Changed in version 0.6.0: `ensure_ascii` parameter added

Parameters

- **`comment`** (*`str`*) – the string to convert to a comment
- **`ensure_ascii`** (*`Optional[bool]`*) – if true, all non-ASCII characters will be replaced with `\uXXXX` escape sequences in the output; if `None`, only non-Latin-1 characters will be escaped; if false, no characters will be escaped

Return type

str

`javaproperties.`**`unescape`**(*field: str*) → str

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.)

Changed in version 0.5.0: Invalid `\uXXXX` escape sequences will now cause an *`InvalidUEscapeError`* to be raised

Parameters

`field` (*`str`*) – the string to decode

Return type

str

Raises
InvalidUEscapeError – if an invalid `\uXXXX` escape sequence occurs in the input

exception `javaproperties.`**`InvalidUEscapeError`**(*escape: str*)

Bases: `ValueError`

New in version 0.5.0.

Raised when an invalid `\uXXXX` escape sequence (i.e., a `\u` not immediately followed by four hexadecimal digits) is encountered in a simple line-oriented `.properties` file

`escape: str`

The invalid `\uXXXX` escape sequence encountered

5.1 Low-Level Parsing

`javaproperties.`**`parse`**(*src: IO | str | bytes*) → Iterator[*PropertiesElement*]

Parse the given data as a simple line-oriented `.properties` file and return a generator of *PropertiesElement* objects representing the key-value pairs (as *KeyValue* objects), comments (as *Comment* objects), and blank lines (as *Whitespace* objects) in the input in order of occurrence.

If the same key appears multiple times in the input, a separate *KeyValue* object is emitted for each entry.

`src` may be a text string, a bytes string, or a text or binary filehandle/file-like object supporting the `readline` method (with or without universal newlines enabled). Bytes input is decoded as Latin-1.

Changed in version 0.5.0: Invalid `\uXXXX` escape sequences will now cause an *InvalidUEscapeError* to be raised

Changed in version 0.7.0: *parse()* now accepts strings as input, and it now returns a generator of custom objects instead of triples of strings

Parameters
src (*string or file-like object*) – the `.properties` document

Return type
Iterator[*PropertiesElement*]

Raises
InvalidUEscapeError – if an invalid `\uXXXX` escape sequence occurs in the input

class `javaproperties.`**`PropertiesElement`**(*source: str*)

New in version 0.7.0.

Superclass of objects returned by *parse()*

`source: str`

The raw, unmodified input line (including trailing newlines)

`property source_stripped: str`

Like *source*, but with the final trailing newline and line continuation (if any) removed

class `javaproperties.`**`Comment`**(*source: str*)

New in version 0.7.0.

Subclass of *PropertiesElement* representing a comment

is_timestamp() → bool

Returns `True` iff the comment's value appears to be a valid timestamp as produced by Java 8's `Date.toString()`

`property` **source_stripped**: `str`

Like `source`, but with the final trailing newline (if any) removed

`property` **value**: `str`

Returns the contents of the comment, with the comment marker, any whitespace leading up to it, and the trailing newline removed

`class` `javaproperties.`**KeyValue**(*key: str*, *value: str*, *source: str*)

New in version 0.7.0.

Subclass of *`PropertiesElement`* representing a key-value entry

key: `str`

The entry's key, after processing escape sequences

value: `str`

The entry's value, after processing escape sequences

`class` `javaproperties.`**Whitespace**(*source: str*)

New in version 0.7.0.

Subclass of *`PropertiesElement`* representing a line that is either empty or contains only whitespace (and possibly some line continuations)

5.2 Custom Encoding Error Handler

New in version 0.6.0.

Importing *`javaproperties`* causes a custom error handler, `'javapropertiesreplace'`, to be automatically defined that can then be supplied as the *errors* argument to `str.encode`, `open`, or similar encoding operations in order to cause all unencodable characters to be replaced by `\uXXXX` escape sequences (with non-BMP characters converted to surrogate pairs first).

This is useful, for example, when calling `javaproperties.dump(obj, fp, ensure_ascii=False)` where `fp` has been opened using an encoding that does not contain all Unicode characters (e.g., Latin-1); in such a case, if `errors='javapropertiesreplace'` is supplied when opening `fp`, then any characters in a key or value of `obj` that exist outside `fp`'s character set will be safely encoded as `.properties` file format-compatible escape sequences instead of raising an error.

Note that the hexadecimal value used in a `\uXXXX` escape sequences is always based on the source character's codepoint value in Unicode regardless of the target encoding:

```
>>> # Here we see one character encoded to the byte 0x00f0 (because that's
>>> # how the target encoding represents it) and a completely different
>>> # character encoded as the escape sequence \u00f0 (because that's its
>>> # value in Unicode):
>>> 'apple: \uF8FF; edh: \xF0'.encode('mac_roman', 'javapropertiesreplace')
b'apple: \xf0; edh: \\u00f0'
```

`javaproperties.`**javapropertiesreplace_errors**(*e: UnicodeError*) → tuple[str, int]

New in version 0.6.0.

Implements the `'javapropertiesreplace'` error handling (for text encodings only): unencodable characters are replaced by `\uXXXX` escape sequences (with non-BMP characters converted to surrogate pairs first)

CHAPTER

SIX

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

SEVEN

CHANGELOG

7.1 v0.9.0 (in development)

- Drop support for Python 3.6
- Support Python 3.11 and 3.12
- Migrated from setuptools to hatch

7.2 v0.8.1 (2021-10-05)

- Fix a typing issue in Python 3.9
- Support Python 3.10

7.3 v0.8.0 (2020-11-28)

- Drop support for Python 2.7, 3.4, and 3.5
- Support Python 3.9
- `ensure_ascii` parameter added to *`PropertiesFile.dump()`* and *`PropertiesFile.dumps()`*
- **Bugfix**: When parsing XML input, empty `<entry>` tags now produce an empty string as a value, not `None`
- Added type annotations
- *`Properties`* and *`PropertiesFile`* no longer raise `TypeError` when given a non-string key or value, as type correctness is now expected to be enforced through static type checking
- The *`PropertiesElement`* classes returned by *`parse()`* are no longer subclasses of `namedtuple`, but they can still be iterated over to retrieve their fields like a tuple

7.4 v0.7.0 (2020-03-09)

- `parse()` now accepts strings as input
- **Breaking**: `parse()` now returns a generator of custom objects instead of triples of strings
- Gave `PropertiesFile` a settable `timestamp` property
- Gave `PropertiesFile` a settable `header_comment` property
- Handle unescaping surrogate pairs on narrow Python builds

7.5 v0.6.0 (2020-02-28)

- Include changelog in the Read the Docs site
- Support Python 3.8
- When dumping a value that begins with more than one space, only escape the first space in order to better match Java's behavior
- Gave `dump()`, `dumps()`, `escape()`, and `join_key_value()` an `ensure_ascii` parameter for optionally not escaping non-ASCII characters in output
- Gave `dump()` and `dumps()` an `ensure_ascii_comments` parameter for controlling what characters in the `comments` parameter are escaped
- Gave `to_comment()` an `ensure_ascii` parameter for controlling what characters are escaped
- Added a custom encoding error handler `'javapropertiesreplace'` that encodes invalid characters as `\uXXXX` escape sequences

7.6 v0.5.2 (2019-04-08)

- Added an example of each format to the format descriptions in the docs
- Fix building in non-UTF-8 environments

7.7 v0.5.1 (2018-10-25)

- **Bugfix**: `java_timestamp()` now properly handles naïve `datetime` objects with `fold=1`
- Include installation instructions, examples, and GitHub links in the Read the Docs site

7.8 v0.5.0 (2018-09-18)

- **Breaking**: Invalid `\uXXXX` escape sequences now cause an *InvalidUEscapeError* to be raised
- *Properties* instances can now compare equal to `dict`s and other mapping types
- Gave *Properties* a `copy` method
- Drop support for Python 2.6 and 3.3
- Fixed a `DeprecationWarning` in Python 3.7

7.9 v0.4.0 (2017-04-22)

- Split off the command-line programs into a separate package, `javaproperties-cli`

7.10 v0.3.0 (2017-04-13)

- Added the *PropertiesFile* class for preserving comments in files [#1]
- The `ordereddict` package is now required under Python 2.6

7.11 v0.2.1 (2017-03-20)

- **Bugfix** to **javaproperties** command: Don't die horribly on missing non-ASCII keys
- PyPy now supported

7.12 v0.2.0 (2016-11-14)

- Added a **javaproperties** command for basic command-line manipulating of `.properties` files
- Gave **json2properties** a `--separator` option
- Gave **json2properties** and **properties2json** `--encoding` options
- Exported the *java_timestamp()* function
- *to_comment()* now converts CR LF and CR line endings inside comments to LF
- Some minor documentation improvements

7.13 v0.1.0 (2016-10-02)

Initial release

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.util.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`.

CHAPTER EIGHT

INSTALLATION

`javaproperties` requires Python 3.7 or higher. Just use pip for Python 3 (You have pip, right?) to install it:

```
python3 -m pip install javaproperties
```

CHAPTER NINE

EXAMPLES

Dump some keys & values (output order not guaranteed):

```
>>> properties = {"key": "value", "host:port": "127.0.0.1:80", "snowman": "", "goat": ""}
>>> print(javaproperties.dumps(properties))
#Mon Sep 26 14:57:44 EDT 2016
key=value
goat=\ud83d\udc10
host\:port=127.0.0.1\:80
snowman=\u2603
```

Load some keys & values:

```
>>> javaproperties.loads('''
... #Mon Sep 26 14:57:44 EDT 2016
... key = value
... goat: \\ud83d\\udc10
... host\\:port=127.0.0.1:80
... #foo = bar
... snowman
... ''')
{'goat': '', 'host:port': '127.0.0.1:80', 'key': 'value', 'snowman': ''}
```

Dump some properties to a file and read them back in again:

```
>>> with open('example.properties', 'w', encoding='latin-1') as fp:
...     javaproperties.dump(properties, fp)
...
>>> with open('example.properties', 'r', encoding='latin-1') as fp:
...     javaproperties.load(fp)
...
{'goat': '', 'host:port': '127.0.0.1:80', 'key': 'value', 'snowman': ''}
```

Sort the properties you're dumping:

```
>>> print(javaproperties.dumps(properties, sort_keys=True))
#Mon Sep 26 14:57:44 EDT 2016
goat=\ud83d\udc10
host\:port=127.0.0.1\:80
key=value
snowman=\u2603
```

Turn off the timestamp:

```
>>> print(javaproperties.dumps(properties, timestamp=None))
key=value
goat=\ud83d\udc10
host\:port=127.0.0.1\:80
snowman=\u2603
```

Use your own timestamp (automatically converted to local time):

```
>>> print(javaproperties.dumps(properties, timestamp=1234567890))
#Fri Feb 13 18:31:30 EST 2009
key=value
goat=\ud83d\udc10
host\:port=127.0.0.1\:80
snowman=\u2603
```

Dump as XML:

```
>>> print(javaproperties.dumps_xml(properties))
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
<entry key="key">value</entry>
<entry key="goat"></entry>
<entry key="host:port">127.0.0.1:80</entry>
<entry key="snowman"></entry>
</properties>
```

New in v0.6.0: Dump Unicode characters as-is instead of escaping them:

```
>>> print(javaproperties.dumps(properties, ensure_ascii=False))
#Tue Feb 25 19:13:27 EST 2020
key=value
goat=
host\:port=127.0.0.1\:80
snowman=
```

CHAPTER

TEN

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

INDEX