

---

# headerparser

*Release 0.4.0*

2019 May 29



---

## Contents

---

<b>1</b>	<b>Input Format</b>	<b>3</b>
<b>2</b>	<b>Parser</b>	<b>5</b>
<b>3</b>	<b>Scanner</b>	<b>11</b>
3.1	Deprecated Functions . . . . .	13
3.2	Scanner Options . . . . .	14
<b>4</b>	<b>Utilities</b>	<b>15</b>
<b>5</b>	<b>Exceptions</b>	<b>17</b>
5.1	Parser Errors . . . . .	17
5.2	Scanner Errors . . . . .	18
<b>6</b>	<b>Changelog</b>	<b>19</b>
6.1	v0.4.0 (2019-05-29) . . . . .	19
6.2	v0.3.0 (2018-10-12) . . . . .	19
6.3	v0.2.0 (2018-02-14) . . . . .	20
6.4	v0.1.0 (2017-03-17) . . . . .	20
<b>7</b>	<b>Installation</b>	<b>21</b>
<b>8</b>	<b>Examples</b>	<b>23</b>
<b>9</b>	<b>Indices and tables</b>	<b>25</b>
	<b>Python Module Index</b>	<b>27</b>



[GitHub](#) | [PyPI](#) | [Documentation](#) | [Issues](#) | *[Changelog](#)*



# CHAPTER 1

---

## Input Format

---

`headerparser` accepts a syntax that is intended to be a simplified superset of the Internet Message (e-mail) Format specified in **RFC 822**, **RFC 2822**, and **RFC 5322**. Specifically:

- Everything in the input up to (but not including) the first blank line (i.e., a line containing only a line ending) constitutes a *stanza* or *header section*. Everything after the first blank line is a free-form *message body*. If there are no blank lines, the entire input is used as the header section, and there is no body.

---

**Note:** By default, blank lines at the beginning of a document are interpreted as the ending of a zero-length stanza. Such blank lines can instead be ignored by setting the `skip_leading_newlines` *scanner option* to true.

---

- A stanza or header section is composed of zero or more *header fields*. A header field is composed of one or more lines, with all lines after the first beginning with a space or tab. Additionally, the first line must contain a colon (optionally surrounded by whitespace); everything before the colon is the *header field name*, while everything after (including subsequent lines) is the *header field value*.

---

**Note:** Name-value separators other than a colon can be used by setting the `separator_regex` *scanner option* appropriately.

---

---

**Note:** This format only recognizes CR, LF, and CR LF sequences as line endings.

---

An example:

```
Key: Value
Foo: Bar
Bar:Whitespace around the colon is optional
Baz : Very optional
Long-Field: This field has a very long value, so I'm going to split it
        across multiple lines.
```

(continues on next page)

(continued from previous page)

```
The above line is all whitespace. This counts as line folding, and so
we're still in the "Long Field" value, but the RFCs consider such lines
obsolete, so you should avoid using them.
.
One alternative to an all-whitespace line is a line with just indentation
and a period. Debian package description fields use this.
Foo: Wait, I already defined a value for this key. What happens now?
What happens now: It depends on whether the `multiple` option for the "Foo"
    field was set in the HeaderParser.
If multiple=True: The "Foo" key in the dictionary returned by
    HeaderParser.parse_string() would map to a list of all of Foo's values
If multiple=False: A ParserError is raised
If multiple=False but there's only one "Foo" anyway:
    The "Foo" key in the result dictionary would map to just a single string.
Compare this to: the standard library's `email` package, which accepts
    multi-occurrence fields, but *which* occurrence Message.__getitem__
    returns is unspecified!

Are we still in the header: no
    There was a blank line above, so we're now in the body, which isn't
    processed for headers.
Good thing, too, because this isn't a valid header line.
```

On the other hand, this is not a valid RFC 822-style document:

```
An indented first line -- without a "Name:" line before it!
A header line without a colon isn't good, either.
Does this make up for the above: no
```



**class** `headerparser.HeaderParser` (*normalizer=None, body=None, \*\*kwargs*)

A parser for RFC 822-style header sections. Define the fields the parser should recognize with the `add_field()` method, configure handling of unrecognized fields with `add_additional()`, and then parse input with `parse()` or another `parse_*` () method.

#### Parameters

- **normalizer** (*callable*) – By default, the parser will consider two field names to be equal iff their lowercased forms are equal. This can be overridden by setting `normalizer` to a custom callable that takes a field name and returns a “normalized” name for use in equality testing. The normalizer will also be used when looking up keys in the `NormalizedDict` instances returned by the parser’s `parse_*` () methods.
- **body** (*bool*) – whether the parser should allow or forbid a body after the header section; `True` means a body is required, `False` means a body is prohibited, and `None` (the default) means a body is optional
- **kwargs** – *scanner options*

**add\_additional** (*enable=True, \*\*kwargs*)

Specify how the parser should handle fields in the input that were not previously registered with `add_field`. By default, unknown fields will cause the `parse_*` methods to raise an `UnknownFieldError`, but calling this method with `enable=True` (the default) will change the parser’s behavior so that all unregistered fields are processed according to the options in `**kwargs`. (If no options are specified, the additional values will just be stored in the result dictionary.)

If this method is called more than once, only the settings from the last call will be used.

Note that additional field values are always stored in the result dictionary using their field name as the key, and two fields are considered the same (for the purposes of `multiple`) iff their names are the same after normalization. Customization of the dictionary key and field name can only be done through `add_field`.

New in version 0.2.0: `action` argument added

#### Parameters

- **enable** (*bool*) – whether the parser should accept input fields that were not registered with `add_field`; setting this to `False` disables additional fields and restores the parser’s default behavior
- **multiple** (*bool*) – If `True`, each additional header field will be allowed to occur more than once in the input, and each field’s values will be stored in a list. If `False` (the default), a `DuplicateFieldError` will be raised if an additional field occurs more than once in the input.
- **unfold** (*bool*) – If `True` (default `False`), additional field values will be “unfolded” (i.e., line breaks will be removed and whitespace around line breaks will be converted to a single space) before applying `type`
- **type** (*callable*) – a callable to apply to additional field values before storing them in the result dictionary
- **choices** (*iterable*) – A sequence of values which additional fields are allowed to have. If `choices` is defined, all additional field values in the input must have one of the given values (after applying `type`) or else an `InvalidChoiceError` is raised.
- **action** (*callable*) – A callable to invoke whenever the field is encountered in the input. The callable will be passed the current dictionary of header fields, the field’s name, and the field’s value (after processing with `type` and `unfold` and checking against `choices`). The callable replaces the default behavior of storing the field’s values in the result dictionary, and so the callable must explicitly store the values if desired.

**Returns** `None`

**Raises** `ValueError` –

- if `enable` is true and a previous call to `add_field` used a custom `dest`
- if `choices` is an empty sequence

**add\_field** (*name*, *\*altnames*, *\*\*kwargs*)

Define a header field for the parser to parse. During parsing, if a field is encountered whose name (*modulo* normalization) equals either `name` or one of the `altnames`, the field’s value will be processed according to the options in `**kwargs`. (If no options are specified, the value will just be stored in the result dictionary.)

New in version 0.2.0: `action` argument added

#### Parameters

- **name** (*string*) – the primary name for the field, used in error messages and as the default value of `dest`
- **altnames** (*strings*) – field name synonyms
- **dest** – The key in the result dictionary in which the field’s value(s) will be stored; defaults to `name`. When additional headers are enabled (see `add_additional`), `dest` must equal (after normalization) one of the field’s names.
- **required** (*bool*) – If `True` (default `False`), the `parse_*` methods will raise a `MissingFieldError` if the field is not present in the input
- **default** – The value to associate with the field if it is not present in the input. If no default value is specified, the field will be omitted from the result dictionary if it is not present in the input. `default` cannot be set when the field is required. `type`, `unfold`, and `action` will not be applied to the default value, and the default value need not belong to `choices`.

- **multiple** (*bool*) – If `True`, the header field will be allowed to occur more than once in the input, and all of the field’s values will be stored in a list. If `False` (the default), a `DuplicateFieldError` will be raised if the field occurs more than once in the input.
- **unfold** (*bool*) – If `True` (default `False`), the field value will be “unfolded” (i.e., line breaks will be removed and whitespace around line breaks will be converted to a single space) before applying `type`
- **type** (*callable*) – a callable to apply to the field value before storing it in the result dictionary
- **choices** (*iterable*) – A sequence of values which the field is allowed to have. If `choices` is defined, all occurrences of the field in the input must have one of the given values (after applying `type`) or else an `InvalidChoiceError` is raised.
- **action** (*callable*) – A callable to invoke whenever the field is encountered in the input. The callable will be passed the current dictionary of header fields, the field’s `name`, and the field’s value (after processing with `type` and `unfold` and checking against `choices`). The callable replaces the default behavior of storing the field’s values in the result dictionary, and so the callable must explicitly store the values if desired. When `action` is defined for a field, `dest` cannot be.

**Returns** `None`

**Raises**

- **ValueError** –
  - if another field with the same `name` or `dest` was already defined
  - if `dest` is not one of the field’s names and `add_additional` is enabled
  - if `default` is defined and `required` is `true`
  - if `choices` is an empty sequence
  - if both `dest` and `action` are defined
- **TypeError** – if `name` or one of the `altnames` is not a string

**parse** (*iterable*)

New in version 0.4.0.

Parse an RFC 822-style header field section (possibly followed by a message body) from the contents of the given filehandle or sequence of lines and return a dictionary of the header fields (possibly with body attached). If `iterable` is an iterable of `str`, newlines will be appended to lines in multiline header fields where not already present but will not be inserted where missing inside the body.

**Parameters** `iterable` – a text-file-like object or iterable of lines to parse

**Return type** `NormalizedDict`

**Raises**

- **ParserError** – if the input fields do not conform to the field definitions declared with `add_field` and `add_additional`
- **ScannerError** – if the header section is malformed

**parse\_file** (*fp*)

Parse an RFC 822-style header field section (possibly followed by a message body) from the contents of the given filehandle and return a dictionary of the header fields (possibly with body attached)

Deprecated since version 0.4.0: Use `parse()` instead.

**Parameters** **fp** (*file-like object*) – the file to parse

**Return type** *NormalizedDict*

**Raises**

- **ParserError** – if the input fields do not conform to the field definitions declared with *add\_field* and *add\_additional*
- **ScannerError** – if the header section is malformed

**parse\_lines** (*iterable*)

Parse an RFC 822-style header field section (possibly followed by a message body) from the given sequence of lines and return a dictionary of the header fields (possibly with body attached). Newlines will be inserted where not already present in multiline header fields but will not be inserted inside the body.

Deprecated since version 0.4.0: Use *parse()* instead.

**Parameters** **iterable** (*iterable of strings*) – a sequence of lines comprising the text to parse

**Return type** *NormalizedDict*

**Raises**

- **ParserError** – if the input fields do not conform to the field definitions declared with *add\_field* and *add\_additional*
- **ScannerError** – if the header section is malformed

**parse\_next\_stanza** (*iterator*)

New in version 0.4.0.

Parse a RFC 822-style header field section from the contents of the given filehandle or iterator of lines and return a dictionary of the header fields. Input processing stops at the end of the header section, leaving the rest of the iterator unconsumed. As a message body is not consumed, calling this method when *body* is true will produce a *MissingBodyError*.

**Parameters** **iterator** – a text-file-like object or iterator of lines to parse

**Return type** *NormalizedDict*

**Raises**

- **ParserError** – if the input fields do not conform to the field definitions declared with *add\_field* and *add\_additional*
- **ScannerError** – if a header section is malformed

**parse\_next\_stanza\_string** (*s*)

New in version 0.4.0.

Parse a RFC 822-style header field section from the given string and return a pair of a dictionary of the header fields and the rest of the string. As a message body is not consumed, calling this method when *body* is true will produce a *MissingBodyError*.

**Parameters** **s** (*string*) – the text to parse

**Return type** pair of *NormalizedDict* and a string

**Raises**

- **ParserError** – if the input fields do not conform to the field definitions declared with *add\_field* and *add\_additional*
- **ScannerError** – if a header section is malformed

**parse\_stanzas** (*iterable*)

New in version 0.4.0.

Parse zero or more stanzas of RFC 822-style header fields from the given filehandle or sequence of lines and return a generator of dictionaries of header fields.

All of the input is treated as header sections, not message bodies; as a result, calling this method when `body` is true will produce a `MissingBodyError`.

**Parameters** **iterable** – a text-file-like object or iterable of lines to parse

**Return type** generator of *NormalizedDict*

**Raises**

- **ParserError** – if the input fields do not conform to the field definitions declared with *add\_field* and *add\_additional*
- **ScannerError** – if a header section is malformed

**parse\_stanzas\_stream** (*fields*)

New in version 0.4.0.

Parse an iterable of iterables of (name, value) pairs as returned by *scan\_stanzas()* or *scan\_stanzas\_string()* and return a generator of dictionaries of header fields. This is a low-level method that you will usually not need to call.

**Parameters** **fields** – an iterable of iterables of pairs of strings

**Return type** generator of *NormalizedDict*

**Raises**

- **ParserError** – if the input fields do not conform to the field definitions declared with *add\_field* and *add\_additional*
- **ScannerError** – if a header section is malformed

**parse\_stanzas\_string** (*s*)

New in version 0.4.0.

Parse zero or more stanzas of RFC 822-style header fields from the given string and return a generator of dictionaries of header fields.

All of the input is treated as header sections, not message bodies; as a result, calling this method when `body` is true will produce a `MissingBodyError`.

**Parameters** **s** (*string*) – the text to parse

**Return type** generator of *NormalizedDict*

**Raises**

- **ParserError** – if the input fields do not conform to the field definitions declared with *add\_field* and *add\_additional*
- **ScannerError** – if a header section is malformed

**parse\_stream** (*fields*)

Process a sequence of (name, value) pairs as returned by *scan()* or *scan\_string()* and return a dictionary of header fields (possibly with body attached). This is a low-level method that you will usually not need to call.

**Parameters** **fields** (*iterable of pairs of strings*) – a sequence of (name, value) pairs representing the input fields

**Return type** *NormalizedDict*

**Raises**

- **ParserError** – if the input fields do not conform to the field definitions declared with *add\_field* and *add\_additional*
- **ValueError** – if the input contains more than one body pair

**parse\_string**(*s*)

Parse an RFC 822-style header field section (possibly followed by a message body) from the given string and return a dictionary of the header fields (possibly with body attached)

**Parameters** *s* (*string*) – the text to parse

**Return type** *NormalizedDict*

**Raises**

- **ParserError** – if the input fields do not conform to the field definitions declared with *add\_field* and *add\_additional*
- **ScannerError** – if the header section is malformed

Scanner functions perform basic parsing of RFC 822-style header fields, splitting them up into sequences of (name, value) pairs without any further validation or transformation.

In each pair, the first element (the header field name) is the substring up to but not including the first whitespace-padded colon (or other delimiter specified by `separator_regex`) in the first source line of the header field. The second element (the header field value) is a single string, the concatenation of one or more lines, starting with the substring after the first colon in the first source line, with leading whitespace on lines after the first preserved; the ending of each line is converted to `'\n'` (added if there is no line ending in the actual input), and the last line of the field value has its trailing line ending (if any) removed.

---

**Note:** “Line ending” here means a CR, LF, or CR LF sequence. Unicode line separators are not treated as line endings and are not trimmed or converted to `'\n'`.

---

The various functions differ in how they behave once the end of the header section is encountered:

- `scan()` and `scan_string()` gather up everything after the header section and (if there is anything) yield it as a (None, body) pair
- `scan_next_stanza()` and `scan_next_stanza_string()` stop processing input at the end of the header section; `scan_next_stanza()` leaves the unprocessed input in the iterator, while `scan_next_stanza_string()` returns the rest of the input alongside the header fields
- `scan_stanzas()` and `scan_stanzas_string()` expect their input to consist entirely of multiple blank-line-terminated header sections, all of which are processed

The `scan()`, `scan_next_stanza()`, and `scan_stanzas()` functions take as input an iterable of strings (e.g., a text file object) and treat each string as a single line, regardless of whether it ends with a line ending or not (or even whether it contains a line ending in the middle of the string).

The `scan_string()`, `scan_next_stanza_string()`, and `scan_stanzas_string()` functions take as input a single string which is then broken into lines on CR, LF, and CR LF boundaries and then processed as a list of strings.

`headerparser.scan(iterable, **kwargs)`  
New in version 0.4.0.

Scan a text-file-like object or iterable of lines for RFC 822-style header fields and return a generator of (name, value) pairs for each header field in the input, plus a (None, body) pair representing the body (if any) after the header section.

All lines after the first blank line are concatenated & yielded as-is in a (None, body) pair. (Note that body lines which do not end with a line terminator will not have one appended.) If there is no empty line in *iterable*, then no body pair is yielded. If the empty line is the last line in *iterable*, the body will be the empty string. If the empty line is the *first* line in *iterable* and the *skip\_leading\_newlines* option is false (the default), then all other lines will be treated as part of the body and will not be scanned for header fields.

**Parameters**

- **iterable** – a text-file-like object or iterable of strings representing lines of input
- **kwargs** – *scanner options*

**Return type** generator of pairs of strings

**Raises** *ScannerError* – if the header section is malformed

`headerparser.scan_string(s, **kwargs)`

Scan a string for RFC 822-style header fields and return a generator of (name, value) pairs for each header field in the input, plus a (None, body) pair representing the body (if any) after the header section.

See *scan()* for more information on the exact behavior of the scanner.

**Parameters**

- **s** – a string which will be broken into lines on CR, LF, and CR LF boundaries and passed to *scan()*
- **kwargs** – *scanner options*

**Return type** generator of pairs of strings

**Raises** *ScannerError* – if the header section is malformed

`headerparser.scan_next_stanza(iterator, **kwargs)`

New in version 0.4.0.

Scan a text-file-like object or iterator of lines for RFC 822-style header fields and return a generator of (name, value) pairs for each header field in the input. Input processing stops as soon as a blank line is encountered, leaving the rest of the iterator unconsumed (If *skip\_leading\_newlines* is true, the function only stops on a blank line after a non-blank line).

**Parameters**

- **iterator** – a text-file-like object or iterator of strings representing lines of input
- **kwargs** – *scanner options*

**Return type** generator of pairs of strings

**Raises** *ScannerError* – if the header section is malformed

`headerparser.scan_next_stanza_string(s, **kwargs)`

New in version 0.4.0.

Scan a string for RFC 822-style header fields and return a pair (fields, extra) where *fields* is a list of (name, value) pairs for each header field in the input up to the first blank line and *extra* is everything after the first blank line (If *skip\_leading\_newlines* is true, the dividing point is instead the first blank line after a non-blank line); if there is no appropriate blank line in the input, *extra* is the empty string.

**Parameters**



- **s** – a string to scan
- **kwargs** – *scanner options*

**Return type** pair of a list of pairs of strings and a string

**Raises** *ScannerError* – if the header section is malformed

`headerparser.scan_stanzas(iterable, **kwargs)`  
New in version 0.4.0.

Scan a text-file-like object or iterable of lines for zero or more stanzas of RFC 822-style header fields and return a generator of lists of (name, value) pairs, where each list represents a stanza of header fields in the input.

The stanzas are terminated by blank lines. Consecutive blank lines between stanzas are treated as a single blank line. Blank lines at the end of the input are discarded without creating a new stanza.

#### Parameters

- **iterable** – a text-file-like object or iterable of strings representing lines of input
- **kwargs** – *scanner options*

**Return type** generator of lists of pairs of strings

**Raises** *ScannerError* – if the header section is malformed

`headerparser.scan_stanzas_string(s, **kwargs)`  
New in version 0.4.0.

Scan a string for zero or more stanzas of RFC 822-style header fields and return a generator of lists of (name, value) pairs, where each list represents a stanza of header fields in the input.

The stanzas are terminated by blank lines. Consecutive blank lines between stanzas are treated as a single blank line. Blank lines at the end of the input are discarded without creating a new stanza.

#### Parameters

- **s** – a string which will be broken into lines on CR, LF, and CR LF boundaries and passed to `scan_stanzas()`
- **kwargs** – *scanner options*

**Return type** generator of lists of pairs of strings

**Raises** *ScannerError* – if the header section is malformed

## 3.1 Deprecated Functions

`headerparser.scan_file(fp, **kwargs)`

Scan a file for RFC 822-style header fields and return a generator of (name, value) pairs for each header field in the input, plus a (None, body) pair representing the body (if any) after the header section.

See `scan()` for more information on the exact behavior of the scanner.

Deprecated since version 0.4.0: Use `scan()` instead.

#### Parameters

- **fp** – A file-like object than can be iterated over to produce lines to pass to `scan()`. Opening the file in universal newlines mode is recommended.
- **kwargs** – *scanner options*

**Return type** generator of pairs of strings

**Raises** *ScannerError* – if the header section is malformed

`headerparser.scan_lines` (*fp*, *\*\*kwargs*)

Scan an iterable of lines for RFC 822-style header fields and return a generator of (*name*, *value*) pairs for each header field in the input, plus a (*None*, *body*) pair representing the body (if any) after the header section.

See *scan()* for more information on the exact behavior of the scanner.

Deprecated since version 0.4.0: Use *scan()* instead.

**Parameters**

- **iterable** – an iterable of strings representing lines of input
- **kwargs** – *scanner options*

**Return type** generator of pairs of strings

**Raises** *ScannerError* – if the header section is malformed

## 3.2 Scanner Options

The following keyword arguments can be passed to *HeaderParser* and the scanner functions in order to configure scanning behavior:

**separator\_regex=r'[\t]\*:[\t]\*'** A regex (as a *str* or compiled regex object) defining the name-value separator. When the regex matches a line, everything before the matched substring becomes the field name, and everything after becomes the first line of the field value. Note that the regex must match any surrounding whitespace in order for it to be trimmed from the key & value.

**skip\_leading\_newlines=False** If *True*, blank lines at the beginning of the input will be discarded. If *False*, a blank line at the beginning of the input marks the end of an empty header section.

New in version 0.3.0: *separator\_regex*, *skip\_leading\_newlines*

**class** `headerparser.NormalizedDict` (*data=None, normalizer=None, body=None*)

A generalization of a case-insensitive dictionary. *NormalizedDict* takes a callable (the “normalizer”) that is applied to any key passed to its `__getitem__`, `__setitem__`, or `__delitem__` method, and the result of the call is then used for the actual lookup. When iterating over a *NormalizedDict*, each key is returned as the “pre-normalized” form passed to `__setitem__` the last time the key was set (but see *normalized()* below). Aside from this, *NormalizedDict* behaves like a normal *MutableMapping* class.

If a normalizer is not specified upon instantiation, a default will be used that converts strings to lowercase and leaves everything else unchanged, so *NormalizedDict* defaults to yet another case-insensitive dictionary.

Two *NormalizedDict* instances compare equal iff their normalizers, bodies, and *normalized\_dict()* return values are equal. When comparing a *NormalizedDict* to any other type of mapping, the other mapping is first converted to a *NormalizedDict* using the same normalizer.

#### Parameters

- **data** (*mapping*) – a mapping or iterable of (*key*, *value*) pairs with which to initialize the instance
- **normalizer** (*callable*) – A callable to apply to keys before looking them up; defaults to *lower*. The callable MUST be idempotent (i.e., `normalizer(x)` must equal `normalizer(normalizer(x))` for all inputs) or else bad things will happen to your dictionary.
- **body** (string or *None*) – initial value for the *body* attribute

**body = None**

This is where *HeaderParser* stores the message body (if any) accompanying the header section represented by the mapping

**copy()**

Create a shallow copy of the mapping

**normalized()**

Return a copy of the instance such that iterating over it will return normalized keys instead of the keys passed to `__setitem__`

```
>>> normdict = NormalizedDict()
>>> normdict['Foo'] = 23
>>> normdict['bar'] = 42
>>> sorted(normdict)
['Foo', 'bar']
>>> sorted(normdict.normalized())
['bar', 'foo']
```

**Return type** *NormalizedDict*

**normalized\_dict()**

Convert to a *dict* with all keys normalized. (A *dict* with non-normalized keys can be obtained with `dict(normdict)`.)

**Return type** *dict*

`headerparser.BOOL(s)`

Convert boolean-like strings to *bool* values. The strings 'yes', 'y', 'on', 'true', and '1' are converted to *True*, and the strings 'no', 'n', 'off', 'false', and '0' are converted to *False*. The conversion is case-insensitive and ignores leading & trailing whitespace. Any value that cannot be converted to a *bool* results in a *ValueError*.

**Parameters** *s* (*string*) – a boolean-like string to convert to a *bool*

**Return type** *bool*

**Raises** *ValueError* – if *s* is not one of the values listed above

`headerparser.lower(s)`

New in version 0.2.0.

Convert *s* to lowercase by calling its `lower()` method if it has one; otherwise, return *s* unchanged

`headerparser.unfold(s)`

New in version 0.2.0.

Remove folding whitespace from a string by converting line breaks (and any whitespace adjacent to line breaks) to a single space and removing leading & trailing whitespace.

```
>>> unfold('This is a \n folded string.\n')
'This is a folded string.'
```

**Parameters** *s* (*string*) – a string to unfold

**Return type** *string*

**exception** `headerparser.errors.Error`

Bases: `Exception`

Superclass for all custom exceptions raised by the package

### 5.1 Parser Errors

**exception** `headerparser.errors.ParserError`

Bases: `headerparser.errors.Error`, `ValueError`

Superclass for all custom exceptions related to errors in parsing

**exception** `headerparser.errors.BodyNotAllowedError`

Bases: `headerparser.errors.ParserError`

Raised when `body=False` and the parser encounters a message body

**exception** `headerparser.errors.DuplicateFieldError` (*name*)

Bases: `headerparser.errors.ParserError`

Raised when a header field not marked as multiple occurs two or more times in the input

**name = None**

The name of the duplicated header field

**exception** `headerparser.errors.FieldTypeError` (*name*, *value*, *exc\_value*)

Bases: `headerparser.errors.ParserError`

Raised when a type callable raises an exception

**exc\_value = None**

The exception raised by the type callable

**name = None**

The name of the header field for which the type callable was called

**value = None**

The value on which the type callable was called

**exception** `headerparser.errors.InvalidChoiceError` (*name*, *value*)

Bases: `headerparser.errors.ParserError`

Raised when a header field is given a value that is not one of its allowed choices

**name = None**

The name of the header field

**value = None**

The invalid value

**exception** `headerparser.errors.MissingBodyError`

Bases: `headerparser.errors.ParserError`

Raised when `body=True` but there is no message body in the input

**exception** `headerparser.errors.MissingFieldError` (*name*)

Bases: `headerparser.errors.ParserError`

Raised when a header field marked as required is not present in the input

**name = None**

The name of the missing header field

**exception** `headerparser.errors.UnknownFieldError` (*name*)

Bases: `headerparser.errors.ParserError`

Raised when an unknown header field is encountered and additional header fields are not enabled

**name = None**

The name of the unknown header field

## 5.2 Scanner Errors

**exception** `headerparser.errors.ScannerError`

Bases: `headerparser.errors.Error`, `ValueError`

Superclass for all custom exceptions related to errors in scanning

**exception** `headerparser.errors.MalformedHeaderError` (*line*)

Bases: `headerparser.errors.ScannerError`

Raised when the scanner encounters an invalid header line, i.e., a line without either a colon or leading whitespace

**line = None**

The invalid header line

**exception** `headerparser.errors.UnexpectedFoldingError` (*line*)

Bases: `headerparser.errors.ScannerError`

Raised when the scanner encounters a folded (indented) line that is not preceded by a valid header line

**line = None**

The line containing the unexpected folding (indentation)

### 6.1 v0.4.0 (2019-05-29)

- Added a `scan()` function combining the behavior of `scan_file()` and `scan_lines()`, which are now deprecated
- Gave `HeaderParser` a `parse()` method combining the behavior of `parse_file()` and `parse_lines()`, which are now deprecated
- Added `scan_next_stanza()` and `scan_next_stanza_string()` functions for scanning & consuming input only up to the end of the first header section
- Added `scan_stanzas()` and `scan_stanzas_string()` functions for scanning input composed entirely of multiple stanzas/header sections
- Gave `HeaderParser` `parse_next_stanza()` and `parse_next_stanza_string()` methods for parsing & consuming input only up to the end of the first header section
- Gave `HeaderParser` `parse_stanzas()` and `parse_stanzas_string()` methods for parsing input composed entirely of multiple stanzas/header sections

### 6.2 v0.3.0 (2018-10-12)

- Drop support for Python 3.3
- Gave `HeaderParser` and the scanner functions options for configuring scanning behavior:
  - `separator_regex`
  - `skip_leading_newlines`
- Fixed a `DeprecationWarning` in Python 3.7

## 6.3 v0.2.0 (2018-02-14)

- *NormalizedDict*'s default normalizer (exposed as the *lower()* function) now passes non-strings through unchanged
- *HeaderParser* instances can now be compared for non-identity equality
- *HeaderParser.add\_field()* and *HeaderParser.add\_additional()* now take an optional *action* argument for customizing the parser's behavior when a field is encountered
- Made the *unfold()* function public

## 6.4 v0.1.0 (2017-03-17)

Initial release

*headerparser* parses key-value pairs in the style of **RFC 822** (e-mail) headers and converts them into case-insensitive dictionaries with the trailing message body (if any) attached. Fields can be converted to other types, marked required, or given default values using an API based on the standard library's *argparse* module. (Everyone loves *argparse*, right?) Low-level functions for just scanning header fields (breaking them into sequences of key-value pairs without any further processing) are also included.



## CHAPTER 7

---

### Installation

---

Just use `pip` (You have `pip`, right?) to install `headerparser` and its dependencies:

```
pip install headerparser
```



## CHAPTER 8

---

### Examples

---

Define a parser:

```
>>> import headerparser
>>> parser = headerparser.HeaderParser()
>>> parser.add_field('Name', required=True)
>>> parser.add_field('Type', choices=['example', 'demonstration', 'prototype'],
↳ default='example')
>>> parser.add_field('Public', type=headerparser.BOOL, default=False)
>>> parser.add_field('Tag', multiple=True)
>>> parser.add_field('Data')
```

Parse some headers and inspect the results:

```
>>> msg = parser.parse_string('''\
... Name: Sample Input
... Public: yes
... tag: doctest, examples,
...   whatever
... TAG: README
...
... Wait, why I am using a body instead of the "Data" field?
... ''')
>>> sorted(msg.keys())
['Name', 'Public', 'Tag', 'Type']
>>> msg['Name']
'Sample Input'
>>> msg['Public']
True
>>> msg['Tag']
['doctest, examples,\n  whatever', 'README']
>>> msg['TYPE']
'example'
>>> msg['Data']
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```

...
KeyError: 'data'
>>> msg.body
'Wait, why I am using a body instead of the "Data" field?\n'

```

Fail to parse headers that don't meet your requirements:

```

>>> parser.parse_string('Type: demonstration')
Traceback (most recent call last):
...
headerparser.errors.MissingFieldError: Required header field 'Name' is not present
>>> parser.parse_string('Name: Bad type\nType: other')
Traceback (most recent call last):
...
headerparser.errors.InvalidChoiceError: 'other' is not a valid choice for 'Type'
>>> parser.parse_string('Name: unknown field\nField: Value')
Traceback (most recent call last):
...
headerparser.errors.UnknownFieldError: Unknown header field 'Field'

```

Allow fields you didn't even think of:

```

>>> parser.add_additional()
>>> msg = parser.parse_string('Name: unknown field\nField: Value')
>>> msg['Field']
'Value'

```

Just split some headers into names & values and worry about validity later:

```

>>> for field in headerparser.scan_string(''\
... Name: Scanner Sample
... Unknown headers: no problem
... Unparsed-Boolean: yes
... CaSe-SeNsItIvE-rEsUlTs: true
... Whitespace around colons: optional
... Whitespace around colons : I already said it's optional.
... That means you have the _option_ to use as much as you want!
...
... And there's a body, too, I guess.
... ''): print(field)
('Name', 'Scanner Sample')
('Unknown headers', 'no problem')
('Unparsed-Boolean', 'yes')
('CaSe-SeNsItIvE-rEsUlTs', 'true')
('Whitespace around colons', 'optional')
('Whitespace around colons', "I already said it's optional.\n That means you have_
↳the _option_ to use as much as you want!")
(None, "And there's a body, too, I guess.\n")

```

## CHAPTER 9

---

### Indices and tables

---

- `genindex`
- `search`



## h

[headerparser](#), 1





## A

`add_additional()` (*headerparser.HeaderParser* method), 5

`add_field()` (*headerparser.HeaderParser* method), 6

## B

`body` (*headerparser.NormalizedDict* attribute), 15

`BodyNotAllowedError`, 17

`BOOL()` (in module *headerparser*), 16

## C

`copy()` (*headerparser.NormalizedDict* method), 15

## D

`DuplicateFieldError`, 17

## E

`Error`, 17

`exc_value` (*headerparser.errors.FieldTypeError* attribute), 17

## F

`FieldTypeError`, 17

## H

`HeaderParser` (class in *headerparser*), 5

*headerparser* (module), 1

## I

`InvalidChoiceError`, 18

## L

`line` (*headerparser.errors.MalformedHeaderError* attribute), 18

`line` (*headerparser.errors.UnexpectedFoldingError* attribute), 18

`lower()` (in module *headerparser*), 16

## M

`MalformedHeaderError`, 18

`MissingBodyError`, 18

`MissingFieldError`, 18

## N

`name` (*headerparser.errors.DuplicateFieldError* attribute), 17

`name` (*headerparser.errors.FieldTypeError* attribute), 17

`name` (*headerparser.errors.InvalidChoiceError* attribute), 18

`name` (*headerparser.errors.MissingFieldError* attribute), 18

`name` (*headerparser.errors.UnknownFieldError* attribute), 18

`normalized()` (*headerparser.NormalizedDict* method), 15

`normalized_dict()` (*headerparser.NormalizedDict* method), 16

`NormalizedDict` (class in *headerparser*), 15

## P

`parse()` (*headerparser.HeaderParser* method), 7

`parse_file()` (*headerparser.HeaderParser* method), 7

`parse_lines()` (*headerparser.HeaderParser* method), 8

`parse_next_stanza()` (*headerparser.HeaderParser* method), 8

`parse_next_stanza_string()` (*headerparser.HeaderParser* method), 8

`parse_stanzas()` (*headerparser.HeaderParser* method), 8

`parse_stanzas_stream()` (*headerparser.HeaderParser* method), 9

`parse_stanzas_string()` (*headerparser.HeaderParser* method), 9

`parse_stream()` (*headerparser.HeaderParser* method), 9

`parse_string()` (*headerparser.HeaderParser*  
    *method*), 10  
`ParserError`, 17

## R

`RFC`  
    `RFC 2822`, 3  
    `RFC 5322`, 3  
    `RFC 822`, 3, 20

## S

`scan()` (*in module headerparser*), 11  
`scan_file()` (*in module headerparser*), 13  
`scan_lines()` (*in module headerparser*), 14  
`scan_next_stanza()` (*in module headerparser*), 12  
`scan_next_stanza_string()` (*in module header-*  
    *parser*), 12  
`scan_stanzas()` (*in module headerparser*), 13  
`scan_stanzas_string()` (*in module header-*  
    *parser*), 13  
`scan_string()` (*in module headerparser*), 12  
`ScannerError`, 18

## U

`UnexpectedFoldingError`, 18  
`unfold()` (*in module headerparser*), 16  
`UnknownFieldError`, 18

## V

`value` (*headerparser.errors.FieldTypeError* attribute),  
    17  
`value` (*headerparser.errors.InvalidChoiceError* at-  
    tribute), 18