headerparser

Release 0.5.0

John T. Wodder II

2023 Oct 04

CONTENTS

1 Input Format **3**

2 Parser **5**

3 Scanner **11**
- 3.1 Scanner Class . . . 11
- 3.2 Functions . . . 12
- 3.3 Deprecated Functions . . . 13

4 Utilities **17**

5 Exceptions **19**
- 5.1 Parser Errors . . . 19
- 5.2 Scanner Errors . . . 20

6 Changelog **21**
- 6.1 v0.5.0 (2023-10-04) . . . 21
- 6.2 v0.4.0 (2019-05-29) . . . 21
- 6.3 v0.3.0 (2018-10-12) . . . 22
- 6.4 v0.2.0 (2018-02-14) . . . 22
- 6.5 v0.1.0 (2017-03-17) . . . 22

7 Installation **23**

8 Examples **25**

9 Indices and tables **27**

Python Module Index **29**

Index **31**

GitHub | PyPI | Documentation | Issues | *Changelog*

CHAPTER

ONE

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: *headerparser* 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.

  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?
```

(continues on next page)

(continued from previous page)

```
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() 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
```

CHAPTER

TWO

PARSER

class headerparser.**HeaderParser**(*normalizer: Callable[[str], Any] | None = None*, *body: bool | None = None*, ***kwargs: Any*)

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** – Passed to the *Scanner* constructor

add_additional(*enable: bool = True*, ***kwargs: Any*) → None

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*.

Changed 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: str*, **altnames: str*, ***kwargs: Any*) → None

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.)

Changed 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(*data: str | Iterable[str]*) → *NormalizedDict*

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 string, filehandle, or sequence of lines and return a dictionary of the header fields (possibly with body attached). If `data` 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.

Changed in version 0.5.0: `data` can now be a string.

Parameters
iterable – a string, 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_next_stanza(*iterator: Iterator[str]*) → *NormalizedDict*

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*.

Deprecated since version 0.5.0: Instead combine *Scanner.scan_next_stanza()* with *parse_stream()*

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: str*) → tuple[*NormalizedDict*, str]

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*.

Deprecated since version 0.5.0: Instead combine *Scanner.scan_next_stanza()* with *parse_stream()*

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(*data: str | Iterable[str]*) → Iterator[*NormalizedDict*]

New in version 0.4.0.

Parse zero or more stanzas of RFC 822-style header fields from the given string, 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*.

Changed in version 0.5.0: `data` can now be a string.

Parameters
data – a string, 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: Iterable[Iterable[tuple[str, str]]]*) → Iterator[*NormalizedDict*]

New in version 0.4.0.

Parse an iterable of iterables of `(name, value)` pairs as returned by *scan_stanzas()* 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: str*) → Iterator[*NormalizedDict*]

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*.

Deprecated since version 0.5.0: Use *parse_stanzas()* instead.

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: Iterable[tuple[str | None, str]]*) → *NormalizedDict*

Process a sequence of `(name, value)` pairs as returned by *scan()* 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: str*) → *NormalizedDict*

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)

Deprecated since version 0.5.0: Use *parse()* instead.

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

CHAPTER

THREE

SCANNER

The *Scanner* class and related functions perform basic parsing of RFC 822-style header fields, splitting formatted input up into sequences of `(name, value)` pairs without any further validation or transformation.

Each pair returned by a scanner method or function represents an individual header field. 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"`.

3.1 Scanner Class

class headerparser.**Scanner**(*data: str | Iterable[str]*, *, *separator_regex: str | Pattern[str] | None = re.compile('[\\t]*:[\\t]*')*, *skip_leading_newlines: bool | None = False*)

New in version 0.5.0.

A class for scanning text for RFC 822-style header fields. Each method processes some portion of the input yet unscanned; the *scan()*, *scan_stanzas()*, and *get_unscanned()* methods process the entirety of the remaining input, while the *scan_next_stanza()* method only processes up through the first blank line.

Parameters

- **data** – The text to scan. This may be a string, a text-file-like object, or an iterable of lines. If it is a string, it will be broken into lines on CR, LF, and CR LF boundaries.
- **separator_regex** – A regex (as a `str` or compiled regex object) defining the name-value separator; defaults to `[ \t]*:[ \t]*`. When the regex is found in 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** (*bool*) – 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.

get_unscanned() → str

Return all of the input that has not yet been processed. After calling this method, calling any method again on the same *Scanner* instance will raise *ScannerEOFError*.

Raises
ScannerEOFError – if all of the input has already been consumed

`scan()` → Iterator[Tuple[str | None, str]]

Scan the remaining input for RFC 822-style header fields and return a generator of `(name, value)` pairs for each header field encountered, 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 the input, then no body pair is yielded. If the empty line is the last line in the input, the body will be the empty string. If the empty line is the *first* line in the input 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.

Raises

- *ScannerError* – if the header section is malformed
- *ScannerEOFError* – if all of the input has already been consumed

`scan_next_stanza()` → Iterator[tuple[str, str]]

Scan the remaining input 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. (If `skip_leading_newlines` is true, the function only stops on a blank line after a non-blank line.)

Raises

- *ScannerError* – if the header section is malformed
- *ScannerEOFError* – if all of the input has already been consumed

`scan_stanzas()` → Iterator[list[tuple[str, str]]]

Scan the remaining input 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.

Raises

- *ScannerError* – if the header section is malformed
- *ScannerEOFError* – if all of the input has already been consumed

3.2 Functions

`headerparser.scan`(*data: str | Iterable[str]*, *, *separator_regex: str | Pattern[str] | None = None*, *skip_leading_newlines: bool = False*) → Iterator[Tuple[str | None, str]]

New in version 0.4.0.

Scan a string, 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.

If `data` is a string, it will be broken into lines on CR, LF, and CR LF boundaries.

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 `data`, then no body pair is yielded. If the empty line is the last line in `data`, the body will be the empty string. If the empty

line is the *first* line in `data` 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.

Changed in version 0.5.0: `data` can now be a string.

Parameters

- **data** – a string, text-file-like object, or iterable of strings representing lines of input
- **kwargs** – Passed to the *Scanner* constructor

Return type
generator of pairs of strings

Raises
ScannerError – if the header section is malformed

`headerparser.`**scan_stanzas**(*data: str | Iterable[str]*, *, *separator_regex: str | Pattern[str] | None = None*, *skip_leading_newlines: bool = False*) → Iterator[list[tuple[str, str]]]

New in version 0.4.0.

Scan a string, 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.

If `data` is a string, it will be broken into lines on CR, LF, and CR LF boundaries.

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.

Changed in version 0.5.0: `data` can now be a string.

Parameters

- **data** – a string, text-file-like object, or iterable of strings representing lines of input
- **kwargs** – Passed to the *Scanner* constructor

Return type
generator of lists of pairs of strings

Raises
ScannerError – if the header section is malformed

3.3 Deprecated Functions

`headerparser.`**scan_string**(*s: str*, *, *separator_regex: str | Pattern[str] | None = None*, *skip_leading_newlines: bool = False*) → Iterator[Tuple[str | None, str]]

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.

Deprecated since version 0.5.0: Use *scan()* instead.

Parameters

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

Return type
generator of pairs of strings

Raises
ScannerError – if the header section is malformed

headerparser.**scan_stanzas_string**(*s: str*, ***, *separator_regex: str | Pattern[str] | None = None*, *skip_leading_newlines: bool = False*) → Iterator[list[tuple[str, str]]]

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.

Deprecated since version 0.5.0: Use *scan_stanzas()* instead

Parameters

- **s** – a string which will be broken into lines on CR, LF, and CR LF boundaries and passed to *scan_stanzas()*
- **kwargs** – Passed to the *Scanner* constructor

Return type
generator of lists of pairs of strings

Raises
ScannerError – if the header section is malformed

headerparser.**scan_next_stanza**(*iterator: Iterator[str]*, ***, *separator_regex: str | Pattern[str] | None = None*, *skip_leading_newlines: bool = False*) → Iterator[tuple[str, str]]

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).

Deprecated since version 0.5.0: Use *Scanner.scan_next_stanza()* instead

Parameters

- **iterator** – a text-file-like object or iterator of strings representing lines of input
- **kwargs** – Passed to the *Scanner* constructor

Return type
generator of pairs of strings

Raises
ScannerError – if the header section is malformed

headerparser.**scan_next_stanza_string**(*s: str*, ***, *separator_regex: str | Pattern[str] | None = None*, *skip_leading_newlines: bool = False*) → tuple[list[tuple[str, str]], str]

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.

Deprecated since version 0.5.0: Use *Scanner.scan_next_stanza()* instead

Parameters

- **s** – a string to scan
- **kwargs** – Passed to the *Scanner* constructor

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

Raises
ScannerError – if the header section is malformed

CHAPTER

FOUR

UTILITIES

class headerparser.**NormalizedDict**(*data: None | Mapping | Iterable[tuple[Any, Any]] = None*, *normalizer: Callable[[Any], Any] | None = None*, *body: str | None = 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: `str | None`

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

copy() → *NormalizedDict*

Create a shallow copy of the mapping

normalized() → *NormalizedDict*

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']
```

(continues on next page)

(continued from previous page)

```
>>> sorted(normdict.normalized())
['bar', 'foo']
```

Return type
NormalizedDict

normalized_dict() → 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: str*) → bool

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: Any*) → Any

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: str*) → str

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

CHAPTER

FIVE

EXCEPTIONS

exception headerparser.**Error**

Bases: Exception

Superclass for all custom exceptions raised by the package

5.1 Parser Errors

exception headerparser.**ParserError**

Bases: *Error*, ValueError

Superclass for all custom exceptions related to errors in parsing

exception headerparser.**BodyNotAllowedError**

Bases: *ParserError*

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

exception headerparser.**DuplicateFieldError**(*name: str*)

Bases: *ParserError*

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

name: str

The name of the duplicated header field

exception headerparser.**FieldTypeError**(*name: str*, *value: str*, *exc_value: BaseException*)

Bases: *ParserError*

Raised when a `type` callable raises an exception

exc_value: BaseException

The exception raised by the `type` callable

name: str

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

value: str

The value on which the `type` callable was called

exception headerparser.**InvalidChoiceError**(*name: str*, *value: Any*)

Bases: *ParserError*

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

name: *str*

The name of the header field

value: *Any*

The invalid value

exception headerparser.**MissingBodyError**

Bases: *ParserError*

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

exception headerparser.**MissingFieldError**(*name: str*)

Bases: *ParserError*

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

name: *str*

The name of the missing header field

exception headerparser.**UnknownFieldError**(*name: str*)

Bases: *ParserError*

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

name: *str*

The name of the unknown header field

5.2 Scanner Errors

exception headerparser.**ScannerError**

Bases: *Error*, ValueError

Superclass for all custom exceptions related to errors in scanning

exception headerparser.**MalformedHeaderError**(*line: str*)

Bases: *ScannerError*

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

line: *str*

The invalid header line

exception headerparser.**UnexpectedFoldingError**(*line: str*)

Bases: *ScannerError*

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

line: *str*

The line containing the unexpected folding (indentation)

exception headerparser.**ScannerEOFError**

Bases: *Error*

Raised when a *Scanner* method is called after all input has been exhausted

CHAPTER

SIX

CHANGELOG

6.1 v0.5.0 (2023-10-04)

- Support Python 3.8 through 3.12
- Drop support for Python 2.7, 3.4, 3.5, and 3.6
- Removed `scan_file()`, `scan_lines()`, `HeaderParser.parse_file()`, and `HeaderParser.parse_lines()` (all deprecated in v0.4.0)
- Type annotations added
- The scanner options to the scanner functions are now keyword-only
- *`scan()`* and *`scan_stanzas()`* can now parse strings directly. As a result, *`scan_string()`* and *`scan_stanzas_string()`* are now deprecated.
- The *`HeaderParser`* methods *`parse()`* and *`parse_stanzas()`* can now parse strings directly. As a result, the *`parse_string()`* and *`parse_stanzas_string()`* methods are now deprecated.
- Added a *`Scanner`* class with methods for scanning a shared input. As a result, the following are now deprecated:
  - *`scan_next_stanza()`*
  - *`scan_next_stanza_string()`*
  - *`HeaderParser.parse_next_stanza()`*
  - *`HeaderParser.parse_next_stanza_string()`*

6.2 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

6.3 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

6.4 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

6.5 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

SEVEN

INSTALLATION

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

```
python3 -m pip install headerparser
```

CHAPTER

EIGHT

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

NINE

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

h
headerparser, ??

INDEX

A

add_additional() (*headerparser.HeaderParser method*), 5
add_field() (*headerparser.HeaderParser method*), 6

B

body (*headerparser.NormalizedDict attribute*), 17
BodyNotAllowedError, 19
BOOL() (*in module headerparser*), 18

C

copy() (*headerparser.NormalizedDict method*), 17

D

DuplicateFieldError, 19

E

Error, 19
exc_value (*headerparser.FieldTypeError attribute*), 19

F

FieldTypeError, 19

G

get_unscanned() (*headerparser.Scanner method*), 11

H

headerparser
    module, 1
HeaderParser (*class in headerparser*), 5

I

InvalidChoiceError, 19

L

line (*headerparser.MalformedHeaderError attribute*), 20
line (*headerparser.UnexpectedFoldingError attribute*), 20
lower() (*in module headerparser*), 18

M

MalformedHeaderError, 20
MissingBodyError, 20
MissingFieldError, 20
module
    headerparser, 1

N

name (*headerparser.DuplicateFieldError attribute*), 19
name (*headerparser.FieldTypeError attribute*), 19
name (*headerparser.InvalidChoiceError attribute*), 19
name (*headerparser.MissingFieldError attribute*), 20
name (*headerparser.UnknownFieldError attribute*), 20
normalized() (*headerparser.NormalizedDict method*), 17
normalized_dict() (*headerparser.NormalizedDict method*), 18
NormalizedDict (*class in headerparser*), 17

P

parse() (*headerparser.HeaderParser method*), 7
parse_next_stanza() (*headerparser.HeaderParser method*), 7
parse_next_stanza_string() (*headerparser.HeaderParser method*), 8
parse_stanzas() (*headerparser.HeaderParser method*), 8
parse_stanzas_stream() (*headerparser.HeaderParser method*), 8
parse_stanzas_string() (*headerparser.HeaderParser method*), 9
parse_stream() (*headerparser.HeaderParser method*), 9
parse_string() (*headerparser.HeaderParser method*), 9
ParserError, 19

R

RFC
    RFC 2822, 3
    RFC 5322, 3
    RFC 822, 3, 22

S

scan() (*headerparser.Scanner method*), 12
scan() (*in module headerparser*), 12
scan_next_stanza() (*headerparser.Scanner method*), 12
scan_next_stanza() (*in module headerparser*), 14
scan_next_stanza_string() (*in module headerparser*), 14
scan_stanzas() (*headerparser.Scanner method*), 12
scan_stanzas() (*in module headerparser*), 13
scan_stanzas_string() (*in module headerparser*), 14
scan_string() (*in module headerparser*), 13
Scanner (*class in headerparser*), 11
ScannerEOFError, 20
ScannerError, 20

U

UnexpectedFoldingError, 20
unfold() (*in module headerparser*), 18
UnknownFieldError, 20

V

value (*headerparser.FieldTypeError attribute*), 19
value (*headerparser.InvalidChoiceError attribute*), 20