linesep

Release 0.5.0

John Thorvald Wodder II

2022 Jun 22

CONTENTS

GitHub | PyPI | Documentation | Issues | *Changelog*

CHAPTER

ONE

CORE FUNCTIONS

1.1 Notes

- Strings, filehandles, and regexes passed to the `*_preceded`, `*_separated`, and `*_terminated` functions may be either binary or text. However, the arguments to a single invocation of a function must be either all binary or all text, and the return type will match.
- Note the following about how the different types of separators are handled at the beginning & end of input:
  - When segments are terminated by a given separator, a separator at the beginning of the input creates an empty leading segment, and a separator at the end of the input simply terminates the last segment.
  - When segments are separated by a given separator, a separator at the beginning of the input creates an empty leading segment, and a separator at the end of the input creates an empty trailing segment.
  - When segments are preceded by a given separator, a separator at the beginning of the input simply starts the first segment, and a separator at the end of the input creates an empty trailing segment.
- Two adjacent separators always create an empty segment between them, unless the separator is a regex that spans both separators at once.

1.2 Splitting Strings

`linesep.`**`split_preceded`**`(`*s: AnyStr*, *sep: Union[AnyStr, Pattern]*, *retain: bool = False*`)` → list[AnyStr]

Split a string `s` into zero or more segments starting with/preceded by the string or compiled regex `sep`. A list of segments is returned; an empty input string will always produce an empty list.

Parameters

- **`s`** – a binary or text string
- **`sep`** – a string or compiled regex that indicates the start of a new segment wherever it occurs
- **`retain`** (*`bool`*) – whether to include the separators at the beginning of each segment

Returns

a list of the segments in `s`

Return type

list of binary or text strings

`linesep.`**`split_separated`**`(`*s: AnyStr*, *sep: Union[AnyStr, Pattern]*, *retain: bool = False*`)` → list[AnyStr]

Split a string `s` into one or more segments separated by the string or compiled regex `sep`. A list of segments is returned; an empty input string will always produce a list with one element, the empty string.

Parameters

- **s** – a binary or text string
- **sep** – a string or compiled regex that indicates the end of one segment and the beginning of another wherever it occurs
- **retain** (*bool*) – When `True`, the segment separators will be included in the output, with the elements of the list alternating between segments and separators, starting with a (possibly empty) segment

Returns
a list of the segments in `s`

Return type
list of binary or text strings

`linesep.`**`split_terminated`**(*s: AnyStr*, *sep: Union[AnyStr, Pattern]*, *retain: bool = False*) → list[AnyStr]

Split a string `s` into zero or more segments terminated by the string or compiled regex `sep`. A list of segments is returned; an empty input string will always produce an empty list.

Parameters

- **s** – a binary or text string
- **sep** – a string or compiled regex that indicates the end of a segment wherever it occurs
- **retain** (*bool*) – whether to include the separators at the end of each segment

Returns
a list of the segments in `s`

Return type
list of binary or text strings

1.3 Joining Strings

`linesep.`**`join_preceded`**(*iterable: Iterable*, *sep: AnyStr*) → AnyStr

Join the elements of `iterable` together, preceding each one with `sep`

Parameters

- **iterable** – an iterable of binary or text strings
- **sep** – a binary or text string

Return type
a binary or text string

`linesep.`**`join_separated`**(*iterable: Iterable*, *sep: AnyStr*) → AnyStr

Join the elements of `iterable` together, separating consecutive elements with `sep`

Parameters

- **iterable** – an iterable of binary or text strings
- **sep** – a binary or text string

Return type
a binary or text string

linesep.**join_terminated**(*iterable: Iterable*, *sep: AnyStr*) → AnyStr

Join the elements of `iterable` together, appending `sep` to each one

Parameters

- **iterable** – an iterable of binary or text strings
- **sep** – a binary or text string

Return type

a binary or text string

1.4 Reading from Filehandles

> **Warning:** Using the `read_*` functions with a variable-length regular expression is unreliable. The only truly foolproof way to split on such regexes is to first read the whole file into memory and then call one of the `split_*` functions. As a result, passing a regular expression separator to a `read_*` function is deprecated starting in version 0.4.0, and support for this will be removed in version 1.0.

linesep.**read_preceded**(*fp: IO*, *sep: Union[AnyStr, Pattern]*, *retain: bool = False*, *chunk_size: int = 512*) → Iterator

Read segments from a file-like object `fp` in which the beginning of each segment is indicated by the string or compiled regex `sep`. A generator of segments is returned; an empty file will always produce an empty generator.

Data is read from the filehandle `chunk_size` characters at a time. If `sep` is a variable-length compiled regex and a separator in the file crosses a chunk boundary, the results are undefined.

Deprecated since version 0.4.0: Passing a regular expression as a separator is deprecated, and support will be removed in version 1.0.

Parameters

- **fp** – a binary or text file-like object
- **sep** – a string or compiled regex that indicates the start of a new segment wherever it occurs
- **retain** (*bool*) – whether to include the separators at the beginning of each segment
- **chunk_size** (*int*) – how many bytes or characters to read from `fp` at a time

Returns

a generator of the segments in `fp`

Return type

generator of binary or text strings

linesep.**read_separated**(*fp: IO*, *sep: Union[AnyStr, Pattern]*, *retain: bool = False*, *chunk_size: int = 512*) → Iterator

Read segments from a file-like object `fp` in which segments are separated by the string or compiled regex `sep`. A generator of segments is returned; an empty file will always produce a generator with one element, the empty string.

Data is read from the filehandle `chunk_size` characters at a time. If `sep` is a variable-length compiled regex and a separator in the file crosses a chunk boundary, the results are undefined.

Deprecated since version 0.4.0: Passing a regular expression as a separator is deprecated, and support will be removed in version 1.0.

Parameters

- **fp** – a binary or text file-like object
- **sep** – a string or compiled regex that indicates the end of one segment and the beginning of another wherever it occurs
- **retain** (*bool*) – When `True`, the segment separators will be included in the output, with the elements of the generator alternating between segments and separators, starting with a (possibly empty) segment
- **chunk_size** (*int*) – how many bytes or characters to read from `fp` at a time

Returns
a generator of the segments in `fp`

Return type
generator of binary or text strings

`linesep.`**`read_terminated`**(*fp: IO*, *sep: Union[AnyStr, Pattern]*, *retain: bool = False*, *chunk_size: int = 512*) → Iterator

Read segments from a file-like object `fp` in which the end of each segment is indicated by the string or compiled regex `sep`. A generator of segments is returned; an empty file will always produce an empty generator.

Data is read from the filehandle `chunk_size` characters at a time. If `sep` is a variable-length compiled regex and a separator in the file crosses a chunk boundary, the results are undefined.

Deprecated since version 0.4.0: Passing a regular expression as a separator is deprecated, and support will be removed in version 1.0.

Parameters

- **fp** – a binary or text file-like object
- **sep** – a string or compiled regex that indicates the end of a segment wherever it occurs
- **retain** (*bool*) – whether to include the separators at the end of each segment
- **chunk_size** (*int*) – how many bytes or characters to read from `fp` at a time

Returns
a generator of the segments in `fp`

Return type
generator of binary or text strings

1.5 Writing to Filehandles

`linesep.`**`write_preceded`**(*fp: IO*, *iterable: Iterable*, *sep: AnyStr*) → None

Write the elements of `iterable` to the filehandle `fp`, preceding each one with `sep`

Parameters

- **fp** – a binary or text file-like object
- **iterable** – an iterable of binary or text strings
- **sep** – a binary or text string

Returns
`None`

`linesep.`**`write_separated`**(*fp: IO*, *iterable: Iterable*, *sep: AnyStr*) → None

Write the elements of `iterable` to the filehandle `fp`, separating consecutive elements with `sep`

Parameters

- **`fp`** – a binary or text file-like object
- **`iterable`** – an iterable of binary or text strings
- **`sep`** – a binary or text string

Returns

`None`

`linesep.`**`write_terminated`**(*fp: IO*, *iterable: Iterable*, *sep: AnyStr*) → None

Write the elements of `iterable` to the filehandle `fp`, appending `sep` to each one

Parameters

- **`fp`** – a binary or text file-like object
- **`iterable`** – an iterable of binary or text strings
- **`sep`** – a binary or text string

Returns

`None`

CHAPTER

TWO

SPLITTER CLASSES

linesep provides a set of classes (called *splitters*) for splitting strings in chunks, inspired by the IncrementalEncoder and IncrementalDecoder classes of the codecs module. Input is fed to a splitter instance one piece at a time, and the segments split from the input so far are (depending on the methods used) either returned immediately or else retrieveable from the splitter afterwards. This is useful when you have a data source that is neither a string nor a filehandle.

If the input is in the form of an iterable, a splitter can be used to iterate over it and yield each segment:

```
>>> import linesep
>>> splitter = linesep.SeparatedSplitter("|", retain=True)
>>> input_data = ["one|two|thr", "ee|four|", "five||six"]
>>> for item in splitter.itersplit(input_data):
...     print(repr(item))
...
'one'
'|'
'two'
'|'
'three'
'|'
'four'
'|'
'five'
'|'
''
'|'
'six'
```

Alternatively, input can be provided to the splitter one piece at a time by passing it to the *split()* method, which returns all newly-split off items:

```
>>> splitter = linesep.TerminatedSplitter("\0", retain=False)
>>> splitter.split("foo\0bar\0baz")
['foo', 'bar']
>>> splitter.split("\0quux\0gnusto\0", final=True)
['baz', 'quux', 'gnusto']
```

At a lower level, input can be provided to the *feed()* method, and the output can be retrieved with *get()* or *getall()*:

```
>>> splitter = linesep.UniversalNewlineSplitter(retain=True, translate=True)
>>> splitter.feed("foo\nbar\r\nbaz")
```

(continues on next page)

(continued from previous page)

```
>>> splitter.nonempty
True
>>> splitter.get()
'foo\n'
>>> splitter.nonempty
True
>>> splitter.get()
'bar\n'
>>> splitter.nonempty
False
>>> splitter.get()
Traceback (most recent call last):
    ...
SplitterEmptyError: No items available in splitter
>>> splitter.close()
>>> splitter.nonempty
True
>>> splitter.get()
'baz'
>>> splitter.nonempty
False
```

Like the `*_preceded`, `*_separated`, and `*_terminated` functions, strings passed to splitters may be either binary or text. However, the input to a single instance of a splitter must be either all binary or all text, and the output type will match.

2.1 Splitters

class linesep.**Splitter**

New in version 0.4.0.

Abstract base class for all splitters. The abstract methods are an implementation detail; this class is exported only for `isinstance()` and typing purposes and should not be subclassed by users.

`Splitter` and its subclasses are generic in `AnyStr`; i.e., they should be written in type annotations as `SplitterClass[AnyStr]`, `SplitterClass[str]`, or `SplitterClass[bytes]`, as appropriate.

feed(*data: AnyStr*) → None

Split input `data`. Any segments or separators extracted can afterwards be retrieved by calling *`get()`* or *`getall()`*.

Raises

`SplitterClosedError` – if *`close()`* has already been called on this splitter

get() → AnyStr

Retrieve the next unfetched item that has been split from the input.

Raises

`SplitterEmptyError` – if there are no items currently available

property nonempty: bool

Whether a subsequent call to *`get()`* would return an item

getall() → list[AnyStr]

Retrieve all unfetched items that have been split from the input

split(*data: AnyStr*, *final: bool = False*) → list[AnyStr]

Split input `data` and return all items thus extracted. Set `final` to `True` if this is the last chunk of input.

Note that, if a previous call to *feed()* was not followed by enough calls to *get()* to retrieve all items, any items left over from the previous round of input will be prepended to the list returned by this method.

Raises

SplitterClosedError – if *close()* has already been called on this splitter

close() → None

Indicate to the splitter that the end of input has been reached. No further calls to *feed()* or *split()* may be made after calling this method unless *reset()* or *setstate()* is called in between.

Depending on the internal state, calling this method may cause more segments or separators to be split from unprocessed input; be sure to fetch them with *get()* or *getall()*.

property closed: bool

Whether *close()* has been called on this splitter

reset() → None

Reset the splitter to its initial state, as though a new instance with the same parameters were constructed

getstate() → *SplitterState*

Retrieve a representation of the splitter's current state

setstate(*state:* SplitterState) → None

Restore the state of the splitter to the what it was when the corresponding *getstate()* call was made

itersplit(*iterable: Iterable*) → Iterator

Feed each element of `iterable` as input to the splitter and yield each item produced.

None of the splitter's other methods should be called while iterating over the yielded values.

The splitter's state is saved & reset before processing the iterable, and the saved state is restored at the end. If you break out of the resulting iterator early, the splitter will be in an undefined state unless & until you reset it.

async **aitersplit**(*aiterable: AsyncIterable*) → AsyncIterator

Like *itersplit()*, but for asynchronous iterators

class linesep.**ParagraphSplitter**(*retain: bool = False*, *translate: bool = True*)

New in version 0.5.0.

A splitter that splits segments terminated by one or more blank lines (i.e., lines containing only a line ending), where lines are terminated by the ASCII newline sequences `"\n"`, `"\r\n"`, and `"\r"`.

Parameters

- **retain** (*bool*) – Whether to include the trailing newlines in split items (`True`) or discard them (`False`, default)
- **translate** (*bool*) – Whether to convert all newlines (both trailing and internal) to `"\n"` (`True`, default) or leave them as-is (`False`)

class linesep.**PrecededSplitter**(*separator: AnyStr*, *retain: bool = False*)

New in version 0.4.0.

A splitter that splits segments preceded by a given string.

A separator at the beginning of the input simply starts the first segment, and a separator at the end of the input creates an empty trailing segment. Two adjacent separators always create an empty segment between them.

Parameters

- **separator** (*AnyStr*) – The string to split the input on
- **retain** (*bool*) – Whether to include the separators in split items (`True`) or discard them (`False`, default)

Raises

`ValueError` – if `separator` is an empty string

`class linesep.SeparatedSplitter(`*separator: AnyStr*, *retain: bool = False*`)`

New in version 0.4.0.

A splitter that splits segments separated by a given string.

A separator at the beginning of the input creates an empty leading segment, and a separator at the end of the input creates an empty trailing segment. Two adjacent separators always create an empty segment between them.

Note that, when `retain` is true, separators are returned as separate items, alternating with segments (unlike *`TerminatedSplitter`* and *`PrecededSplitter`*, where separators are appended/prepended to the segments). In a list returned by `split()` or `getall()`, the segments will be the items at the even indices (starting at 0), and the separators will be at the odd indices (assuming you're calling `get()` the right amount of times and not leaving any output unfetched).

Parameters

- **separator** (*AnyStr*) – The string to split the input on
- **retain** (*bool*) – Whether to include the separators in split items (`True`) or discard them (`False`, default)

Raises

`ValueError` – if `separator` is an empty string

`class linesep.TerminatedSplitter(`*separator: AnyStr*, *retain: bool = False*`)`

New in version 0.4.0.

A splitter that splits segments terminated by a given string.

A separator at the beginning of the input creates an empty leading segment, and a separator at the end of the input simply terminates the last segment. Two adjacent separators always create an empty segment between them.

Parameters

- **separator** (*AnyStr*) – The string to split the input on
- **retain** (*bool*) – Whether to include the separators in split items (`True`) or discard them (`False`, default)

Raises

`ValueError` – if `separator` is an empty string

`class linesep.UnicodeNewlineSplitter(`*retain: bool = False*, *translate: bool = True*`)`

New in version 0.5.0.

A splitter that splits segments terminated by the same set of line endings as recognized by the `str.splitlines()` method. Note that, unlike other splitters, this class is not generic and is only usable on `str` values, not `bytes`.

Parameters

- **retain** (*bool*) – Whether to include the newlines in split items (True) or discard them (False, default)
- **translate** (*bool*) – Whether to convert all retained newlines to "\n" (True, default) or leave them as-is (False)

class linesep.**UniversalNewlineSplitter**(*retain: bool = False*, *translate: bool = True*)

New in version 0.4.0.

A splitter that splits segments terminated by the ASCII newline sequences "\n", "\r\n", and "\r".

Parameters

- **retain** (*bool*) – Whether to include the newlines in split items (True) or discard them (False, default)
- **translate** (*bool*) – Whether to convert all retained newlines to "\n" (True, default) or leave them as-is (False)

2.2 Utilities

linesep.**get_newline_splitter**(*newline: Optional[str] = None*, *retain: bool = False*) → *Splitter*[str]

New in version 0.4.0.

Return a splitter for splitting on newlines following the same rules as the newline option to open().

Specifically:

- If newline is None, a splitter that splits on all ASCII newlines and converts them to "\n" is returned.
- If newline is "" (the empty string), a splitter that splits on all ASCII newlines and leaves them as-is is returned.
- If newline is "\n", "\r\n", or "\r", a splitter that splits on the given string is returned.
- If newline is any other value, a ValueError is raised.

Note that this function is limited to splitting on strs and does not support bytes.

Parameters

retain (*bool*) – Whether the returned splitter should include the newlines in split items (True) or discard them (False, default)

class linesep.**SplitterState**

New in version 0.4.0.

A representation of the internal state of a splitter, returned by *getstate()*. This can be passed to *setstate()* to restore the spitter's internal state to what it was previously.

A given *SplitterState* should only be passed to the *setstate()* method of a splitter of the same class and with the same constructor arguments as the splitter that produced the *SplitterState*; otherwise, the behavior is undefined.

Instances of this class should be treated as opaque objects and should not be inspected, nor should any observed property be relied upon to be the same in future library versions.

exception linesep.**SplitterClosedError**

Bases: ValueError

New in version 0.4.0.

Raised when *feed()* or *split()* is called on a splitter after its *close()* method is called `exception linesep.SplitterEmptyError`

Bases: `Exception`

New in version 0.4.0.

Raised when `get()` is called on a splitter that does not have any unfetched items to return

CHAPTER THREE

MISCELLANEOUS FUNCTIONS

`linesep.ascii_splitlines(s: str, keepends: bool = False) → list[str]`

New in version 0.3.0.

Like `str.splitlines()`, except it only treats `"\n"`, `"\r\n"`, and `"\r"` as line endings

`linesep.read_paragraphs(fp: Iterable[str]) → Iterator[str]`

New in version 0.3.0.

Read a text filehandle or other iterable of lines (with trailing line endings retained) paragraph by paragraph. Each paragraph is terminated by one or more blank lines (i.e., lines containing only a line ending). Trailing and embedded line endings in each paragraph are retained.

Only `"\n"`, `"\r\n"`, and `"\r"` are recognized as line endings.

`linesep.split_paragraphs(s: str) → list[str]`

New in version 0.3.0.

Split a string into paragraphs, each one terminated by one or more blank lines (i.e., lines containing only a line ending). Trailing and embedded line endings in each paragraph are retained.

Only `"\n"`, `"\r\n"`, and `"\r"` are recognized as line endings.

CHAPTER

FOUR

CHANGELOG

4.1 v0.5.0 (2022-06-22)

- Added *UnicodeNewlineSplitter* for incremental splitting on Unicode line ending sequences
- Added *ParagraphSplitter* for incremental splitting on blank lines

4.2 v0.4.0 (2022-06-17)

- Passing a regular expression separator to a `read_*()` function is now deprecated, and support will be removed in version 1.0.
- Added *TerminatedSplitter*, *PrecededSplitter*, *SeparatedSplitter*, & *UniversalNewlineSplitter* classes and *get_newline_splitter()* function for incremental splitting of strings in chunks
- Drop support for Python 3.6

4.3 v0.3.1 (2022-05-31)

- Support Python 3.10
- Refine return type annotation on *read_paragraphs()*

4.4 v0.3.0 (2020-12-02)

- Added *ascii_splitlines()*, *read_paragraphs()*, and *split_paragraphs()* functions

4.5 v0.2.0 (2020-11-28)

- Now support only Python 3.6 and up (tested through 3.9) and PyPy3
- Add type annotations
- Renamed the “size” parameter of the read_* functions to chunk_size
- Add API documentation to README

4.6 v0.1.1 (2017-05-29)

- Remove a PendingDeprecationWarning generated in newer versions of Python 3.5 and 3.6
- More testing infrastructure
- PyPy now supported

4.7 v0.1.0 (2017-01-16)

Initial release

linesep provides basic functions & classes for reading, writing, splitting, & joining text with custom separators that can occur either before, between, or after the segments they separate.

CHAPTER FIVE

INSTALLATION

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

```
python3 -m pip install linesep
```

CHAPTER

SIX

EXAMPLES

Reading sections separated by a "---" line:

```
with open('text.txt') as fp:
    for entry in linesep.read_separated(fp, '\n---\n'):
        ...
```

Parsing output from `find -print0`:

```
find = subprocess.Popen(
    ['find', '/', '-some', '-complicated', '-condition', '-print0'],
    stdout=subprocess.PIPE,
)
for filepath in linesep.read_terminated(find.stdout, '\0'):
    ...
```

A poor man's JSON Text Sequence parser:

```
for entry in linesep.read_preceded(fp, '\x1E'):
    try:
        obj = json.loads(entry)
    except ValueError:
        pass
    else:
        yield obj
```

Read from a text file one paragraph at a time:

```
with open("my-novel.txt") as fp:
    for paragraph in linesep.read_paragraphs(fp):
        ...
```

Split input from an `anyio.TextReceiveStream` on newlines:

```
async with anyio.TextReceiveStream( ... ) as stream:
    splitter = linesep.UnicodeNewlineSplitter()
    async for line in splitter.aitersplit(stream):
        print(line)
```

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

INDEX