humanfriendly

Release 3.2

May 17, 2017

Contents

The functions and classes in the *humanfriendly* package can be used to make text interfaces more user friendly. Some example features:

- Parsing and formatting numbers, file sizes, pathnames and timespans in simple, human friendly formats.
- Easy to use timers for long running operations, with human friendly formatting of the resulting timespans.
- Prompting the user to select a choice from a list of options by typing the option's number or a unique substring of the option.
- Terminal interaction including text styling (ANSI escape sequences), user friendly rendering of usage messages and querying the terminal for its size.

The *humanfriendly* package is currently tested on Python 2.6, 2.7, 3.4, 3.5, 3.6 and PyPy (2.7).

CHAPTER 1

Getting started

It's very simple to start using the *humanfriendly* package:

```
>>> import humanfriendly
>>> user_input = raw_input("Enter a readable file size: ")
Enter a readable file size: 16G
>>> num_bytes = humanfriendly.parse_size(user_input)
>>> print num_bytes
16000000000
>>> print "You entered:", humanfriendly.format_size(num_bytes)
You entered: 16 GB
>>> print "You entered:", humanfriendly.format_size(num_bytes, binary=True)
You entered: 14.9 GiB
```

CHAPTER 2

Command line

Usage: *humanfriendly [OPTIONS]*

Human friendly input/output (text formatting) on the command line based on the Python package with the same name.

Supported options:

Option	Description
`-c`, `--run-command`	Execute an external command (given as the positional arguments) and render a spinner and timer while the command is running. The exit status of the command is propagated.
`--format-table`	Read tabular data from standard input (each line is a row and each whitespace separated field is a column), format the data as a table and print the resulting table to standard output. See also the `--delimiter` option.
`-d`, `--delimiter=VALUE`	Change the delimiter used by `--format-table` to `VALUE` (a string). By default all whitespace is treated as a delimiter.
`-l`, `--format-length=LENGTH`	Convert a length count (given as the integer or float `LENGTH`) into a human readable string and print that string to standard output.
`-n`, `--format-number=VALUE`	Format a number (given as the integer or floating point number `VALUE`) with thousands separators and two decimal places (if needed) and print the formatted number to standard output.
`-s`, `--format-size=BYTES`	Convert a byte count (given as the integer `BYTES`) into a human readable string and print that string to standard output.
`-t`, `--format-timespan=SECONDS`	Convert a number of seconds (given as the floating point number `SECONDS`) into a human readable timespan and print that string to standard output.
`--parse-size=VALUE`	Parse a human readable data size (given as the string `VALUE`) and print the number of bytes to standard output.
`--parse-length=VALUE`	Parse a human readable length (given as the string `VALUE`) and print the number of metres to standard output.
`-h`, `--help`	Show this message and exit.

CHAPTER 3

A note about size units

When I originally published the *humanfriendly* package I went with binary multiples of bytes (powers of two). It was pointed out several times that this was a poor choice (see issue #4 and pull requests #8 and #9) and thus the new default became decimal multiples of bytes (powers of ten):

Unit	Binary value	Decimal value
KB	1024	1000
MB	1048576	1000000
GB	1073741824	1000000000
TB	1099511627776	1000000000000
etc		

The option to use binary multiples of bytes remains by passing the keyword argument *binary=True* to the format_size() and parse_size() functions.

CHAPTER 4

Contact

The latest version of *humanfriendly* is available on PyPI and GitHub. The documentation is hosted on Read the Docs. For bug reports please create an issue on GitHub. If you have questions, suggestions, etc. feel free to send me an e-mail at peter@peterodding.com.

CHAPTER 5

License

CHAPTER 6

API documentation

The following documentation is based on the source code of version 3.2 of the *humanfriendly* package.

A note about backwards compatibility

The *humanfriendly* package started out as a single `humanfriendly` module. Eventually this module grew to a size that necessitated splitting up the code into multiple modules (see e.g. `tables`, `terminal`, `text` and `usage`). Most of the functionality that remains in the `humanfriendly` module will eventually be moved to submodules as well (as time permits and a logical subdivision of functionality presents itself to me).

While moving functionality around like this my goal is to always preserve backwards compatibility. For example if a function is moved to a submodule an import of that function is added in the main module so that backwards compatibility with previously written import statements is preserved.

If backwards compatibility of documented functionality has to be broken then the major version number will be bumped. So if you're using the *humanfriendly* package in your project, make sure to at least pin the major version number in order to avoid unexpected surprises.

The `humanfriendly` module

The main module of the *humanfriendly* package.

class `humanfriendly.`**`SizeUnit`**(*divider*, *symbol*, *name*)

`__getnewargs__`()
Return self as a plain tuple. Used by copy and pickle.

`__getstate__`()
Exclude the OrderedDict from pickling

static **`__new__`**(*_cls*, *divider*, *symbol*, *name*)
Create new instance of SizeUnit(divider, symbol, name)

`__repr__`()
Return a nicely formatted representation string

`_asdict`()
Return a new OrderedDict which maps field names to their values

classmethod **`_make`**(*iterable*, *new=<built-in method __new__ of type object at 0x906d60>*, *len=<built-in function len>*)
Make a new SizeUnit object from a sequence or iterable

`_replace`(*_self*, ***kwds*)
Return a new SizeUnit object replacing specified fields with new values

`divider`
Alias for field number 0

`name`
Alias for field number 2

`symbol`
Alias for field number 1

class `humanfriendly.`**`CombinedUnit`**(*decimal*, *binary*)

`__getnewargs__`()
Return self as a plain tuple. Used by copy and pickle.

`__getstate__`()
Exclude the OrderedDict from pickling

static **`__new__`**(*_cls*, *decimal*, *binary*)
Create new instance of CombinedUnit(decimal, binary)

`__repr__`()
Return a nicely formatted representation string

`_asdict`()
Return a new OrderedDict which maps field names to their values

classmethod _make(*iterable*, *new=<built-in method __new__ of type object at 0x906d60>*, *len=<built-in function len>*)
Make a new CombinedUnit object from a sequence or iterable

_replace(*_self*, ***kwds*)
Return a new CombinedUnit object replacing specified fields with new values

binary
Alias for field number 1

decimal
Alias for field number 0

humanfriendly.**coerce_boolean**(*value*)
Coerce any value to a boolean.

Parameters **value** – Any Python value. If the value is a string:

- The strings '1', 'yes', 'true' and 'on' are coerced to `True`.
- The strings '0', 'no', 'false' and 'off' are coerced to `False`.
- Other strings raise an exception.

Other Python values are coerced using `bool()`.

Returns A proper boolean value.

Raises `exceptions.ValueError` when the value is a string but cannot be coerced with certainty.

humanfriendly.**format_size**(*num_bytes*, *keep_width=False*, *binary=False*)
Format a byte count as a human readable file size.

Parameters

- **num_bytes** – The size to format in bytes (an integer).
- **keep_width** – `True` if trailing zeros should not be stripped, `False` if they can be stripped.
- **binary** – `True` to use binary multiples of bytes (base-2), `False` to use decimal multiples of bytes (base-10).

Returns The corresponding human readable file size (a string).

This function knows how to format sizes in bytes, kilobytes, megabytes, gigabytes, terabytes and petabytes. Some examples:

```
>>> from humanfriendly import format_size
>>> format_size(0)
'0 bytes'
>>> format_size(1)
'1 byte'
>>> format_size(5)
'5 bytes'
> format_size(1000)
'1 KB'
> format_size(1024, binary=True)
'1 KiB'
>>> format_size(1000 ** 3 * 4)
'4 GB'
```

humanfriendly.**parse_size**(*size*, *binary=False*)

Parse a human readable data size and return the number of bytes.

Parameters

- **size** – The human readable file size to parse (a string).
- **binary** – `True` to use binary multiples of bytes (base-2) for ambiguous unit symbols and names, `False` to use decimal multiples of bytes (base-10).

Returns The corresponding size in bytes (an integer).

Raises *InvalidSize* when the input can't be parsed.

This function knows how to parse sizes in bytes, kilobytes, megabytes, gigabytes, terabytes and petabytes. Some examples:

```
>>> from humanfriendly import parse_size
>>> parse_size('42')
42
>>> parse_size('13b')
13
>>> parse_size('5 bytes')
5
>>> parse_size('1 KB')
1000
>>> parse_size('1 kilobyte')
1000
>>> parse_size('1 KiB')
1024
>>> parse_size('1 KB', binary=True)
1024
>>> parse_size('1.5 GB')
1500000000
>>> parse_size('1.5 GB', binary=True)
1610612736
```

humanfriendly.**format_length**(*num_metres*, *keep_width=False*)

Format a metre count as a human readable length.

Parameters

- **num_metres** – The length to format in metres (float / integer).
- **keep_width** – `True` if trailing zeros should not be stripped, `False` if they can be stripped.

Returns The corresponding human readable length (a string).

This function supports ranges from nanometres to kilometres.

Some examples:

```
>>> from humanfriendly import format_length
>>> format_length(0)
'0 metres'
>>> format_length(1)
'1 metre'
>>> format_length(5)
'5 metres'
>>> format_length(1000)
'1 km'
```

```
>>> format_length(0.004)
'4 mm'
```

humanfriendly.**parse_length**(*length*)

Parse a human readable length and return the number of metres.

Parameters **length** – The human readable length to parse (a string).

Returns The corresponding length in metres (a float).

Raises *InvalidLength* when the input can't be parsed.

Some examples:

```
>>> from humanfriendly import parse_length
>>> parse_length('42')
42
>>> parse_length('1 km')
1000
>>> parse_length('5mm')
0.005
>>> parse_length('15.3cm')
0.153
```

humanfriendly.**format_number**(*number*, *num_decimals=2*)

Format a number as a string including thousands separators.

Parameters

- **number** – The number to format (a number like an `int`, `long` or `float`).
- **num_decimals** – The number of decimals to render (2 by default). If no decimal places are required to represent the number they will be omitted regardless of this argument.

Returns The formatted number (a string).

This function is intended to make it easier to recognize the order of size of the number being formatted.

Here's an example:

```
>>> from humanfriendly import format_number
>>> print(format_number(6000000))
6,000,000
> print(format_number(6000000000.42))
6,000,000,000.42
> print(format_number(6000000000.42, num_decimals=0))
6,000,000,000
```

humanfriendly.**round_number**(*count*, *keep_width=False*)

Round a floating point number to two decimal places in a human friendly format.

Parameters

- **count** – The number to format.
- **keep_width** – `True` if trailing zeros should not be stripped, `False` if they can be stripped.

Returns The formatted number as a string. If no decimal places are required to represent the number, they will be omitted.

The main purpose of this function is to be used by functions like *format_length()*, *format_size()* and *format_timespan()*.

Here are some examples:

```
>>> from humanfriendly import round_number
>>> round_number(1)
'1'
>>> round_number(math.pi)
'3.14'
>>> round_number(5.001)
'5'
```

humanfriendly.**format_timespan**(*num_seconds*, *detailed=False*, *max_units=3*)

Format a timespan in seconds as a human readable string.

Parameters

- **num_seconds** – Number of seconds (integer or float).
- **detailed** – If `True` milliseconds are represented separately instead of being represented as fractional seconds (defaults to `False`).
- **max_units** – The maximum number of units to show in the formatted time span (an integer, defaults to three).

Returns The formatted timespan as a string.

Some examples:

```
>>> from humanfriendly import format_timespan
>>> format_timespan(0)
'0 seconds'
>>> format_timespan(1)
'1 second'
>>> import math
>>> format_timespan(math.pi)
'3.14 seconds'
>>> hour = 60 * 60
>>> day = hour * 24
>>> week = day * 7
>>> format_timespan(week * 52 + day * 2 + hour * 3)
'1 year, 2 days and 3 hours'
```

humanfriendly.**parse_timespan**(*timespan*)

Parse a "human friendly" timespan into the number of seconds.

Parameters **value** – A string like `5h` (5 hours), `10m` (10 minutes) or `42s` (42 seconds).

Returns The number of seconds as a floating point number.

Raises *InvalidTimespan* when the input can't be parsed.

Note that the *parse_timespan()* function is not meant to be the "mirror image" of the *format_timespan()* function. Instead it's meant to allow humans to easily and succinctly specify a timespan with a minimal amount of typing. It's very useful to accept easy to write time spans as e.g. command line arguments to programs.

The time units (and abbreviations) supported by this function are:

•ms, millisecond, milliseconds

•s, sec, secs, second, seconds

•m, min, mins, minute, minutes

- h, hour, hours
- d, day, days
- w, week, weeks
- y, year, years

Some examples:

```
>>> from humanfriendly import parse_timespan
>>> parse_timespan('42')
42.0
>>> parse_timespan('42s')
42.0
>>> parse_timespan('1m')
60.0
>>> parse_timespan('1h')
3600.0
>>> parse_timespan('1d')
86400.0
```

humanfriendly.**parse_date**(*datestring*)

Parse a date/time string into a tuple of integers.

Parameters **datestring** – The date/time string to parse.

Returns A tuple with the numbers `(year, month, day, hour, minute, second)` (all numbers are integers).

Raises *InvalidDate* when the date cannot be parsed.

Supported date/time formats:

- `YYYY-MM-DD`
- `YYYY-MM-DD HH:MM:SS`

Note: If you want to parse date/time strings with a fixed, known format and *parse_date()* isn't useful to you, consider `time.strptime()` or `datetime.datetime.strptime()`, both of which are included in the Python standard library. Alternatively for more complex tasks consider using the date/time parsing module in the dateutil package.

Examples:

```
>>> from humanfriendly import parse_date
>>> parse_date('2013-06-17')
(2013, 6, 17, 0, 0, 0)
>>> parse_date('2013-06-17 02:47:42')
(2013, 6, 17, 2, 47, 42)
```

Here's how you convert the result to a number (Unix time):

```
>>> from humanfriendly import parse_date
>>> from time import mktime
>>> mktime(parse_date('2013-06-17 02:47:42') + (-1, -1, -1))
1371430062.0
```

And here's how you convert it to a `datetime.datetime` object:

```
>>> from humanfriendly import parse_date
>>> from datetime import datetime
>>> datetime(*parse_date('2013-06-17 02:47:42'))
datetime.datetime(2013, 6, 17, 2, 47, 42)
```

Here's an example that combines *format_timespan()* and *parse_date()* to calculate a human friendly timespan since a given date:

```
>>> from humanfriendly import format_timespan, parse_date
>>> from time import mktime, time
>>> unix_time = mktime(parse_date('2013-06-17 02:47:42') + (-1, -1, -1))
>>> seconds_since_then = time() - unix_time
>>> print(format_timespan(seconds_since_then))
1 year, 43 weeks and 1 day
```

humanfriendly.**format_path**(*pathname*)

Shorten a pathname to make it more human friendly.

Parameters **pathname** – An absolute pathname (a string).

Returns The pathname with the user's home directory abbreviated.

Given an absolute pathname, this function abbreviates the user's home directory to `~/` in order to shorten the pathname without losing information. It is not an error if the pathname is not relative to the current user's home directory.

Here's an example of its usage:

```
>>> from os import environ
>>> from os.path import join
>>> vimrc = join(environ['HOME'], '.vimrc')
>>> vimrc
'/home/peter/.vimrc'
>>> from humanfriendly import format_path
>>> format_path(vimrc)
'~/.vimrc'
```

humanfriendly.**parse_path**(*pathname*)

Convert a human friendly pathname to an absolute pathname.

Expands leading tildes using `os.path.expanduser()` and environment variables using `os.path.expandvars()` and makes the resulting pathname absolute using `os.path.abspath()`.

Parameters **pathname** – A human friendly pathname (a string).

Returns An absolute pathname (a string).

class humanfriendly.**Timer**(*start_time=None*, *resumable=False*)

Easy to use timer to keep track of long during operations.

__init__(*start_time=None*, *resumable=False*)

Remember the time when the *Timer* was created.

Parameters

- **start_time** – The start time (a float, defaults to the current time).
- **resumable** – Create a resumable timer (defaults to `False`).

When *start_time* is given *Timer* uses `time.time()` as a clock source, otherwise it uses *humanfriendly.compat.monotonic()*.

__enter__()
Start or resume counting elapsed time.

Returns The *Timer* object.

Raises ValueError when the timer isn't resumable.

__exit__(*exc_type=None*, *exc_value=None*, *traceback=None*)
Stop counting elapsed time.

Raises ValueError when the timer isn't resumable.

elapsed_time
Get the number of seconds counted so far.

rounded
Human readable timespan rounded to seconds (a string).

__str__()
Show the elapsed time since the *Timer* was created.

class humanfriendly.**Spinner**(*label=None*, *total=0*, *stream=<open file '<stderr>', mode 'w'>*, *interactive=None*, *timer=None*, *hide_cursor=True*)
Show a spinner on the terminal as a simple means of feedback to the user.

The *Spinner* class shows a "spinner" on the terminal to let the user know that something is happening during long running operations that would otherwise be silent (leaving the user to wonder what they're waiting for). Below are some visual examples that should illustrate the point.

Simple spinners:

Here's a screen capture that shows the simplest form of spinner:

The following code was used to create the spinner above:

```
import itertools
import time
from humanfriendly import Spinner

with Spinner(label="Downloading") as spinner:
    for i in itertools.count():
        # Do something useful here.
        time.sleep(0.1)
        # Advance the spinner.
        spinner.step()
```

Spinners that show elapsed time:

Here's a spinner that shows the elapsed time since it started:

The following code was used to create the spinner above:

```
import itertools
import time
from humanfriendly import Spinner, Timer

with Spinner(label="Downloading", timer=Timer()) as spinner:
    for i in itertools.count():
        # Do something useful here.
        time.sleep(0.1)
```

```
        # Advance the spinner.
        spinner.step()
```

Spinners that show progress:

Here's a spinner that shows a progress percentage:

The following code was used to create the spinner above:

```
import itertools
import random
import time
from humanfriendly import Spinner, Timer

with Spinner(label="Downloading", total=100) as spinner:
    progress = 0
    while progress < 100:
        # Do something useful here.
        time.sleep(0.1)
        # Advance the spinner.
        spinner.step(progress)
        # Determine the new progress value.
        progress += random.random() * 5
```

If you want to provide user feedback during a long running operation but it's not practical to periodically call the *step()* method consider using *AutomaticSpinner* instead.

As you may already have noticed in the examples above, *Spinner* objects can be used as context managers to automatically call *clear()* when the spinner ends. This helps to make sure that if the text cursor is hidden its visibility is restored before the spinner ends (even if an exception interrupts the spinner).

__init__(*label=None*, *total=0*, *stream=<open file '<stderr>', mode 'w'>*, *interactive=None*, *timer=None*, *hide_cursor=True*)

Initialize a spinner.

Parameters

- **label** – The label for the spinner (a string, defaults to `None`).
- **total** – The expected number of steps (an integer).
- **stream** – The output stream to show the spinner on (defaults to `sys.stderr`).
- **interactive** – If this is `False` then the spinner doesn't write to the output stream at all. It defaults to the return value of `stream.isatty()`.
- **timer** – A *Timer* object (optional). If this is given the spinner will show the elapsed time according to the timer.
- **hide_cursor** – If `True` (the default) the text cursor is hidden as long as the spinner is active.

step(*progress=0*, *label=None*)

Advance the spinner by one step and redraw it.

Parameters

- **progress** – The number of the current step, relative to the total given to the *Spinner* constructor (an integer, optional). If not provided the spinner will not show progress.

- **label** – The label to use while redrawing (a string, optional). If not provided the label given to the *Spinner* constructor is used instead.

This method advances the spinner by one step without starting a new line, causing an animated effect which is very simple but much nicer than waiting for a prompt which is completely silent for a long time.

Note: This method uses time based rate limiting to avoid redrawing the spinner too frequently. If you know you're dealing with code that will call *step()* at a high frequency, consider using *sleep()* to avoid creating the equivalent of a busy loop that's rate limiting the spinner 99% of the time.

---

sleep()
Sleep for a short period before redrawing the spinner.

This method is useful when you know you're dealing with code that will call *step()* at a high frequency. It will sleep for the interval with which the spinner is redrawn (less than a second). This avoids creating the equivalent of a busy loop that's rate limiting the spinner 99% of the time.

This method doesn't redraw the spinner, you still have to call *step()* in order to do that.

clear()
Clear the spinner.

The next line which is shown on the standard output or error stream after calling this method will overwrite the line that used to show the spinner. Also the visibility of the text cursor is restored.

__enter__()
Enable the use of spinners as context managers.

Returns The *Spinner* object.

__exit__(*exc_type=None*, *exc_value=None*, *traceback=None*)
Clear the spinner when leaving the context.

class humanfriendly.**AutomaticSpinner**(*label*, *show_time=True*)
Show a spinner on the terminal that automatically starts animating.

This class shows a spinner on the terminal (just like *Spinner* does) that automatically starts animating. This class should be used as a context manager using the `with` statement. The animation continues for as long as the context is active.

AutomaticSpinner provides an alternative to *Spinner* for situations where it is not practical for the caller to periodically call *step()* to advance the animation, e.g. because you're performing a blocking call and don't fancy implementing threading or subprocess handling just to provide some user feedback.

This works using the `multiprocessing` module by spawning a subprocess to render the spinner while the main process is busy doing something more useful. By using the `with` statement you're guaranteed that the subprocess is properly terminated at the appropriate time.

__init__(*label*, *show_time=True*)
Initialize an automatic spinner.

Parameters

- **label** – The label for the spinner (a string).
- **show_time** – If this is `True` (the default) then the spinner shows elapsed time.

__enter__()
Enable the use of automatic spinners as context managers.

__exit__(*exc_type=None*, *exc_value=None*, *traceback=None*)
Enable the use of automatic spinners as context managers.

exception humanfriendly.**InvalidDate**
Raised when a string cannot be parsed into a date.

For example:

```
>>> from humanfriendly import parse_date
>>> parse_date('2013-06-XY')
Traceback (most recent call last):
  File "humanfriendly.py", line 206, in parse_date
    raise InvalidDate, msg % datestring
humanfriendly.InvalidDate: Invalid date! (expected 'YYYY-MM-DD' or 'YYYY-MM-DD␣
↪HH:MM:SS' but got: '2013-06-XY')
```

exception humanfriendly.**InvalidSize**
Raised when a string cannot be parsed into a file size.

For example:

```
>>> from humanfriendly import parse_size
>>> parse_size('5 Z')
Traceback (most recent call last):
  File "humanfriendly/__init__.py", line 267, in parse_size
    raise InvalidSize(msg % (size, tokens))
humanfriendly.InvalidSize: Failed to parse size! (input '5 Z' was tokenized as [5,
↪ 'Z'])
```

exception humanfriendly.**InvalidLength**
Raised when a string cannot be parsed into a length.

For example:

```
>>> from humanfriendly import parse_length
>>> parse_length('5 Z')
Traceback (most recent call last):
  File "humanfriendly/__init__.py", line 267, in parse_length
    raise InvalidLength(msg % (length, tokens))
humanfriendly.InvalidLength: Failed to parse length! (input '5 Z' was tokenized␣
↪as [5, 'Z'])
```

exception humanfriendly.**InvalidTimespan**
Raised when a string cannot be parsed into a timespan.

For example:

```
>>> from humanfriendly import parse_timespan
>>> parse_timespan('1 age')
Traceback (most recent call last):
  File "humanfriendly/__init__.py", line 419, in parse_timespan
    raise InvalidTimespan(msg % (timespan, tokens))
humanfriendly.InvalidTimespan: Failed to parse timespan! (input '1 age' was␣
↪tokenized as [1, 'age'])
```

The humanfriendly.cli module

Usage: *humanfriendly [OPTIONS]*

Human friendly input/output (text formatting) on the command line based on the Python package with the same name.

Supported options:

Option	Description
`-c`, `--run-command`	Execute an external command (given as the positional arguments) and render a spinner and timer while the command is running. The exit status of the command is propagated.
`--format-table`	Read tabular data from standard input (each line is a row and each whitespace separated field is a column), format the data as a table and print the resulting table to standard output. See also the `--delimiter` option.
`-d`, `--delimiter=VALUE`	Change the delimiter used by `--format-table` to `VALUE` (a string). By default all whitespace is treated as a delimiter.
`-l`, `--format-length=LENGTH`	Convert a length count (given as the integer or float `LENGTH`) into a human readable string and print that string to standard output.
`-n`, `--format-number=VALUE`	Format a number (given as the integer or floating point number `VALUE`) with thousands separators and two decimal places (if needed) and print the formatted number to standard output.
`-s`, `--format-size=BYTES`	Convert a byte count (given as the integer `BYTES`) into a human readable string and print that string to standard output.
`-t`, `--format-timespan=SECONDS`	Convert a number of seconds (given as the floating point number `SECONDS`) into a human readable timespan and print that string to standard output.
`--parse-size=VALUE`	Parse a human readable data size (given as the string `VALUE`) and print the number of bytes to standard output.
`--parse-length=VALUE`	Parse a human readable length (given as the string `VALUE`) and print the number of metres to standard output.
`-h`, `--help`	Show this message and exit.

`humanfriendly.cli.`**`main`**`()`
Command line interface for the `humanfriendly` program.

`humanfriendly.cli.`**`run_command`**`(`*`command_line`*`)`
Run an external command and show a spinner while the command is running.

`humanfriendly.cli.`**`print_formatted_length`**`(`*`value`*`)`
Print a human readable length.

`humanfriendly.cli.`**`print_formatted_number`**`(`*`value`*`)`
Print large numbers in a human readable format.

`humanfriendly.cli.`**`print_formatted_size`**`(`*`value`*`)`
Print a human readable size.

`humanfriendly.cli.`**`print_formatted_table`**`(`*`delimiter`*`)`
Read tabular data from standard input and print a table.

`humanfriendly.cli.`**`print_formatted_timespan`**`(`*`value`*`)`
Print a human readable timespan.

`humanfriendly.cli.`**`print_parsed_length`**`(`*`value`*`)`
Parse a human readable length and print the number of metres.

`humanfriendly.cli.`**`print_parsed_size`**`(`*`value`*`)`
Parse a human readable data size and print the number of bytes.

The `humanfriendly.compat` module

Compatibility with Python 2 and 3.

This module exposes aliases and functions that make it easier to write Python code that is compatible with Python 2 and Python 3.

humanfriendly.compat.**basestring**
Alias for `basestring()` (in Python 2) or `str` (in Python 3). See also *`is_string()`*.

humanfriendly.compat.**interactive_prompt**
Alias for `raw_input()` (in Python 2) or `input()` (in Python 3).

humanfriendly.compat.**StringIO**
Alias for `StringIO.StringIO` (in Python 2) or `io.StringIO` (in Python 3).

humanfriendly.compat.**unicode**
Alias for `unicode()` (in Python 2) or `str` (in Python 3). See also *`coerce_string()`*.

humanfriendly.compat.**monotonic**
Alias for `time.monotonic()` (in Python 3.3 and higher) or *monotonic.monotonic()* (a conditional dependency on older Python versions).

class humanfriendly.compat.**unicode**(*object=''*) → unicode object
unicode(string[, encoding[, errors]]) -> unicode object

Create a new Unicode object from the given encoded string. encoding defaults to the current default string encoding. errors can be 'strict', 'replace' or 'ignore' and defaults to 'strict'.

capitalize() → unicode
Return a capitalized version of S, i.e. make the first character have upper case and the rest lower case.

center(*width*[, *fillchar*]) → unicode
Return S centered in a Unicode string of length width. Padding is done using the specified fill character (default is a space)

count(*sub*[, *start*[, *end*]]) → int
Return the number of non-overlapping occurrences of substring sub in Unicode string S[start:end]. Optional arguments start and end are interpreted as in slice notation.

decode([*encoding*[, *errors*]]) → string or unicode
Decodes S using the codec registered for encoding. encoding defaults to the default encoding. errors may be given to set a different error handling scheme. Default is 'strict' meaning that encoding errors raise a UnicodeDecodeError. Other possible values are 'ignore' and 'replace' as well as any other name registered with codecs.register_error that is able to handle UnicodeDecodeErrors.

encode([*encoding*[, *errors*]]) → string or unicode
Encodes S using the codec registered for encoding. encoding defaults to the default encoding. errors may be given to set a different error handling scheme. Default is 'strict' meaning that encoding errors raise a UnicodeEncodeError. Other possible values are 'ignore', 'replace' and 'xmlcharrefreplace' as well as any other name registered with codecs.register_error that can handle UnicodeEncodeErrors.

endswith(*suffix*[, *start*[, *end*]]) → bool
Return True if S ends with the specified suffix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. suffix can also be a tuple of strings to try.

expandtabs([*tabsize*]) → unicode
Return a copy of S where all tab characters are expanded using spaces. If tabsize is not given, a tab size of 8 characters is assumed.

find(*sub*[, *start*[, *end*]]) → int
Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

format(**args*, ***kwargs*) → unicode
Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

index(*sub*[, *start*[, *end*]]) → int
Like S.find() but raise ValueError when the substring is not found.

isalnum() → bool
Return True if all characters in S are alphanumeric and there is at least one character in S, False otherwise.

isalpha() → bool
Return True if all characters in S are alphabetic and there is at least one character in S, False otherwise.

isdecimal() → bool
Return True if there are only decimal characters in S, False otherwise.

isdigit() → bool
Return True if all characters in S are digits and there is at least one character in S, False otherwise.

islower() → bool
Return True if all cased characters in S are lowercase and there is at least one cased character in S, False otherwise.

isnumeric() → bool
Return True if there are only numeric characters in S, False otherwise.

isspace() → bool
Return True if all characters in S are whitespace and there is at least one character in S, False otherwise.

istitle() → bool
Return True if S is a titlecased string and there is at least one character in S, i.e. upper- and titlecase characters may only follow uncased characters and lowercase characters only cased ones. Return False otherwise.

isupper() → bool
Return True if all cased characters in S are uppercase and there is at least one cased character in S, False otherwise.

join(*iterable*) → unicode
Return a string which is the concatenation of the strings in the iterable. The separator between elements is S.

ljust(*width*[, *fillchar*]) → int
Return S left-justified in a Unicode string of length width. Padding is done using the specified fill character (default is a space).

lower() → unicode
Return a copy of the string S converted to lowercase.

lstrip([*chars*]) → unicode
Return a copy of the string S with leading whitespace removed. If chars is given and not None, remove characters in chars instead. If chars is a str, it will be converted to unicode before stripping

partition(*sep) -> (head*, *sep*, *tail*)
Search for the separator sep in S, and return the part before it, the separator itself, and the part after it. If the separator is not found, return S and two empty strings.

replace(*old*, *new*[, *count*]) → unicode
Return a copy of S with all occurrences of substring old replaced by new. If the optional argument count is given, only the first count occurrences are replaced.

rfind(*sub*[, *start*[, *end*]]) → int
Return the highest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

rindex(*sub*[, *start*[, *end*]]) → int
Like S.rfind() but raise ValueError when the substring is not found.

rjust(*width*[, *fillchar*]) → unicode
Return S right-justified in a Unicode string of length width. Padding is done using the specified fill character (default is a space).

rpartition(*sep) -> (head*, *sep*, *tail*)
Search for the separator sep in S, starting at the end of S, and return the part before it, the separator itself, and the part after it. If the separator is not found, return two empty strings and S.

rsplit([*sep*[, *maxsplit*]]) → list of strings
Return a list of the words in S, using sep as the delimiter string, starting at the end of the string and working to the front. If maxsplit is given, at most maxsplit splits are done. If sep is not specified, any whitespace string is a separator.

rstrip([*chars*]) → unicode
Return a copy of the string S with trailing whitespace removed. If chars is given and not None, remove characters in chars instead. If chars is a str, it will be converted to unicode before stripping

split([*sep*[, *maxsplit*]]) → list of strings
Return a list of the words in S, using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done. If sep is not specified or is None, any whitespace string is a separator and empty strings are removed from the result.

splitlines(*keepends=False*) → list of strings
Return a list of the lines in S, breaking at line boundaries. Line breaks are not included in the resulting list unless keepends is given and true.

startswith(*prefix*[, *start*[, *end*]]) → bool
Return True if S starts with the specified prefix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. prefix can also be a tuple of strings to try.

strip([*chars*]) → unicode
Return a copy of the string S with leading and trailing whitespace removed. If chars is given and not None, remove characters in chars instead. If chars is a str, it will be converted to unicode before stripping

swapcase() → unicode
Return a copy of S with uppercase characters converted to lowercase and vice versa.

title() → unicode
Return a titlecased version of S, i.e. words start with title case characters, all remaining cased characters have lower case.

translate(*table*) → unicode
Return a copy of the string S, where all characters have been mapped through the given translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, Unicode strings or None. Unmapped characters are left untouched. Characters mapped to None are deleted.

upper() → unicode
Return a copy of S converted to uppercase.

zfill(*width*) → unicode
Pad a numeric string S with zeros on the left, to fill a field of the specified width. The string S is never

truncated.

class humanfriendly.compat.**basestring**
Type basestring cannot be instantiated; it is the base for str and unicode.

humanfriendly.compat.**interactive_prompt**()
raw_input([prompt]) -> string

Read a string from standard input. The trailing newline is stripped. If the user hits EOF (Unix: Ctl-D, Windows: Ctl-Z+Return), raise EOFError. On Unix, GNU readline is used if enabled. The prompt string, if given, is printed without a trailing newline before reading.

class humanfriendly.compat.**StringIO**(*buf=''*)
class StringIO([buffer])

When a StringIO object is created, it can be initialized to an existing string by passing the string to the constructor. If no string is given, the StringIO will start empty.

The StringIO object can accept either Unicode or 8-bit strings, but mixing the two may take some care. If both are used, 8-bit strings that cannot be interpreted as 7-bit ASCII (that use the 8th bit) will cause a UnicodeError to be raised when getvalue() is called.

close()
Free the memory buffer.

flush()
Flush the internal buffer

getvalue()
Retrieve the entire contents of the "file" at any time before the StringIO object's close() method is called.

The StringIO object can accept either Unicode or 8-bit strings, but mixing the two may take some care. If both are used, 8-bit strings that cannot be interpreted as 7-bit ASCII (that use the 8th bit) will cause a UnicodeError to be raised when getvalue() is called.

isatty()
Returns False because StringIO objects are not connected to a tty-like device.

next()
A file object is its own iterator, for example iter(f) returns f (unless f is closed). When a file is used as an iterator, typically in a for loop (for example, for line in f: print line), the next() method is called repeatedly. This method returns the next input line, or raises StopIteration when EOF is hit.

read(*n=-1*)
Read at most size bytes from the file (less if the read hits EOF before obtaining size bytes).

If the size argument is negative or omitted, read all data until EOF is reached. The bytes are returned as a string object. An empty string is returned when EOF is encountered immediately.

readline(*length=None*)
Read one entire line from the file.

A trailing newline character is kept in the string (but may be absent when a file ends with an incomplete line). If the size argument is present and non-negative, it is a maximum byte count (including the trailing newline) and an incomplete line may be returned.

An empty string is returned only when EOF is encountered immediately.

Note: Unlike stdio's fgets(), the returned string contains null characters ('0') if they occurred in the input.

readlines(*sizehint=0*)
Read until EOF using readline() and return a list containing the lines thus read.

If the optional sizehint argument is present, instead of reading up to EOF, whole lines totalling approximately sizehint bytes (or more to accommodate a final whole line).

seek(*pos*, *mode=0*)
Set the file's current position.

The mode argument is optional and defaults to 0 (absolute file positioning); other values are 1 (seek relative to the current position) and 2 (seek relative to the file's end).

There is no return value.

tell()
Return the file's current position.

truncate(*size=None*)
Truncate the file's size.

If the optional size argument is present, the file is truncated to (at most) that size. The size defaults to the current position. The current file position is not changed unless the position is beyond the new file size.

If the specified size exceeds the file's current size, the file remains unchanged.

write(*s*)
Write a string to the file.

There is no return value.

writelines(*iterable*)
Write a sequence of strings to the file. The sequence can be any iterable object producing strings, typically a list of strings. There is no return value.

(The name is intended to match readlines(); writelines() does not add line separators.)

humanfriendly.compat.**coerce_string**(*value*)
Coerce any value to a Unicode string (`unicode()` in Python 2 and `str` in Python 3).

Parameters **value** – The value to coerce.

Returns The value coerced to a Unicode string.

humanfriendly.compat.**monotonic**()
time() -> floating point number

Return the current time in seconds since the Epoch. Fractions of a second may be present if the system clock provides them.

humanfriendly.compat.**is_string**(*value*)
Check if a value is a `basestring()` (in Python 2) or `str` (in Python 3) object.

Parameters **value** – The value to check.

Returns `True` if the value is a string, `False` otherwise.

humanfriendly.compat.**is_unicode**(*value*)
Check if a value is a `unicode()` (in Python 2) or `str` (in Python 3) object.

Parameters **value** – The value to check.

Returns `True` if the value is a Unicode string, `False` otherwise.

The `humanfriendly.prompts` module

Interactive terminal prompts.

The *prompts* module enables interaction with the user (operator) by asking for confirmation (*prompt_for_confirmation()*) and asking to choose from a list of options (*prompt_for_choice()*). It works by rendering interactive prompts on the terminal.

humanfriendly.prompts.**MAX_ATTEMPTS = 10**
The number of times an interactive prompt is shown on invalid input (an integer).

humanfriendly.prompts.**prompt_for_confirmation**(*question*, *default=None*, *padding=True*)
Prompt the user for confirmation.

Parameters

- **question** – The text that explains what the user is confirming (a string).
- **default** – The default value (a boolean) or `None`.
- **padding** – Refer to the documentation of *prompt_for_input()*.

Returns

- If the user enters 'yes' or 'y' then `True` is returned.
- If the user enters 'no' or 'n' then `False` is returned.
- If the user doesn't enter any text or standard input is not connected to a terminal (which makes it impossible to prompt the user) the value of the keyword argument `default` is returned (if that value is not `None`).

Raises

- Any exceptions raised by *retry_limit()*.
- Any exceptions raised by *prompt_for_input()*.

When *default* is `False` and the user doesn't enter any text an error message is printed and the prompt is repeated:

```
>>> prompt_for_confirmation("Are you sure?")

Are you sure? [y/n]

Error: Please enter 'yes' or 'no' (there's no default choice).

Are you sure? [y/n]
```

The same thing happens when the user enters text that isn't recognized:

```
>>> prompt_for_confirmation("Are you sure?")

Are you sure? [y/n] about what?

Error: Please enter 'yes' or 'no' (the text 'about what?' is not recognized).

Are you sure? [y/n]
```

humanfriendly.prompts.**prompt_for_choice**(*choices*, *default=None*, *padding=True*)
Prompt the user to select a choice from a group of options.

Parameters

- **choices** – A sequence of strings with available options.
- **default** – The default choice if the user simply presses Enter (expected to be a string, defaults to `None`).

- **padding** – Refer to the documentation of *prompt_for_input()*.

Returns The string corresponding to the user's choice.

Raises

- ValueError if *choices* is an empty sequence.
- Any exceptions raised by *retry_limit()*.
- Any exceptions raised by *prompt_for_input()*.

When no options are given an exception is raised:

```
>>> prompt_for_choice([])
Traceback (most recent call last):
  File "humanfriendly/prompts.py", line 148, in prompt_for_choice
    raise ValueError("Can't prompt for choice without any options!")
ValueError: Can't prompt for choice without any options!
```

If a single option is given the user isn't prompted:

```
>>> prompt_for_choice(['only one choice'])
'only one choice'
```

Here's what the actual prompt looks like by default:

```
>>> prompt_for_choice(['first option', 'second option'])

  1. first option
  2. second option

 Enter your choice as a number or unique substring (Control-C aborts): second

'second option'
```

If you don't like the whitespace (empty lines and indentation):

```
>>> prompt_for_choice(['first option', 'second option'], padding=False)
 1. first option
 2. second option
Enter your choice as a number or unique substring (Control-C aborts): first
'first option'
```

humanfriendly.prompts.**prompt_for_input**(*question*, *default=None*, *padding=True*, *strip=True*)

Prompt the user for input (free form text).

Parameters

- **question** – An explanation of what is expected from the user (a string).
- **default** – The return value if the user doesn't enter any text or standard input is not connected to a terminal (which makes it impossible to prompt the user).
- **padding** – Render empty lines before and after the prompt to make it stand out from the surrounding text? (a boolean, defaults to True)
- **strip** – Strip leading/trailing whitespace from the user's reply?

Returns The text entered by the user (a string) or the value of the *default* argument.

Raises

- `KeyboardInterrupt` when the program is interrupted while the prompt is active, for example because the user presses Control-C.
- `EOFError` when reading from standard input fails, for example because the user presses Control-D or because the standard input stream is redirected (only if *default* is `None`).

`humanfriendly.prompts.`**`prepare_prompt_text`**(*prompt_text*, ***options*)

Wrap a text to be rendered as an interactive prompt in ANSI escape sequences.

Parameters

- **`prompt_text`** – The text to render on the prompt (a string).
- **`options`** – Any keyword arguments are passed on to *`ansi_wrap()`*.

Returns The resulting prompt text (a string).

ANSI escape sequences are only used when the standard output stream is connected to a terminal. When the standard input stream is connected to a terminal any escape sequences are wrapped in "readline hints".

`humanfriendly.prompts.`**`prepare_friendly_prompts`**()

Make interactive prompts more user friendly.

The prompts presented by `raw_input()` (in Python 2) and `input()` (in Python 3) are not very user friendly by default, for example the cursor keys (←, ↑, → and ↓) and the `Home` and `End` keys enter characters instead of performing the action you would expect them to. By simply importing the `readline` module these prompts become much friendlier (as mentioned in the Python standard library documentation).

This function is called by the other functions in this module to enable user friendly prompts.

`humanfriendly.prompts.`**`retry_limit`**()

Allow the user to provide valid input up to *`MAX_ATTEMPTS`* times.

Raises *`TooManyInvalidReplies`* when an interactive prompt receives repeated invalid input (*`MAX_ATTEMPTS`*).

This function returns a generator for interactive prompts that want to repeat on invalid input without getting stuck in infinite loops.

exception `humanfriendly.prompts.`**`TooManyInvalidReplies`**

Raised by interactive prompts when they've received too many invalid inputs.

The `humanfriendly.sphinx` module

Customizations for and integration with the Sphinx documentation generator.

The *`humanfriendly.sphinx`* module uses the Sphinx extension API to customize the process of generating Sphinx based Python documentation. The most relevant functions to take a look at are *`setup()`*, *`enable_special_methods()`* and *`enable_usage_formatting()`*.

`humanfriendly.sphinx.`**`setup`**(*app*)

Enable all of the provided Sphinx customizations.

Parameters **`app`** – The Sphinx application object.

The *`setup()`* function makes it easy to enable all of the Sphinx customizations provided by the *`humanfriendly.sphinx`* module with the least amount of code. All you need to do is to add the module name to the `extensions` variable in your `conf.py` file:

```
# Sphinx extension module names.
extensions = [
    'sphinx.ext.autodoc',
    'sphinx.ext.doctest',
    'sphinx.ext.intersphinx',
    'humanfriendly.sphinx',
]
```

When Sphinx sees the *humanfriendly.sphinx* name it will import the module and call its *setup()* function.

At the time of writing this just calls *enable_special_methods()* and *enable_usage_formatting()*, but of course more functionality may be added at a later stage. If you don't like that idea you may be better of calling the individual functions from your own `setup()` function.

humanfriendly.sphinx.**enable_special_methods**(*app*)

Enable documenting "special methods" using the autodoc extension.

Parameters **app** – The Sphinx application object.

This function connects the *special_methods_callback()* function to `autodoc-skip-member` events.

humanfriendly.sphinx.**special_methods_callback**(*app*, *what*, *name*, *obj*, *skip*, *options*)

Enable documenting "special methods" using the autodoc extension.

Refer to *enable_special_methods()* to enable the use of this function (you probably don't want to call *special_methods_callback()* directly).

This function implements a callback for `autodoc-skip-member` events to include documented "special methods" (method names with two leading and two trailing underscores) in your documentation. The result is similar to the use of the `special-members` flag with one big difference: Special methods are included but other types of members are ignored. This means that attributes like `__weakref__` will always be ignored (this was my main annoyance with the `special-members` flag).

The parameters expected by this function are those defined for Sphinx event callback functions (i.e. I'm not going to document them here :-).

humanfriendly.sphinx.**enable_usage_formatting**(*app*)

Reformat human friendly usage messages to reStructuredText.

Parameters **app** – The Sphinx application object (as given to `setup()`).

This function connects the *usage_message_callback()* function to `autodoc-process-docstring` events.

humanfriendly.sphinx.**usage_message_callback**(*app*, *what*, *name*, *obj*, *options*, *lines*)

Reformat human friendly usage messages to reStructuredText.

Refer to *enable_usage_formatting()* to enable the use of this function (you probably don't want to call *usage_message_callback()* directly).

This function implements a callback for `autodoc-process-docstring` that reformats module docstrings using *render_usage()* so that Sphinx doesn't mangle usage messages that were written to be human readable instead of machine readable. Only module docstrings whose first line starts with *USAGE_MARKER* are reformatted.

The parameters expected by this function are those defined for Sphinx event callback functions (i.e. I'm not going to document them here :-).

The `humanfriendly.text` module

Simple text manipulation functions.

The *`text`* module contains simple functions to manipulate text:

- The *`concatenate()`* and *`pluralize()`* functions make it easy to generate human friendly output.
- The *`format()`*, *`compact()`* and *`dedent()`* functions provide a clean and simple to use syntax for composing large text fragments with interpolated variables.
- The *`tokenize()`* function parses simple user input.

humanfriendly.text.**concatenate**(*items*)

Concatenate a list of items in a human friendly way.

Parameters **items** – A sequence of strings.

Returns A single string.

```
>>> from humanfriendly.text import concatenate
>>> concatenate(["eggs", "milk", "bread"])
'eggs, milk and bread'
```

humanfriendly.text.**format**(*text*, **args*, ***kw*)

Format a string using the string formatting operator and/or `str.format()`.

Parameters

- **text** – The text to format (a string).
- **args** – Any positional arguments are interpolated into the text using the string formatting operator (`%`). If no positional arguments are given no interpolation is done.
- **kw** – Any keyword arguments are interpolated into the text using the `str.format()` function. If no keyword arguments are given no interpolation is done.

Returns The text with any positional and/or keyword arguments interpolated (a string).

The implementation of this function is so trivial that it seems silly to even bother writing and documenting it. Justifying this requires some context :-).

Why format() instead of the string formatting operator?

For really simple string interpolation Python's string formatting operator is ideal, but it does have some strange quirks:

•When you switch from interpolating a single value to interpolating multiple values you have to wrap them in tuple syntax. Because *`format()`* takes a variable number of arguments it always receives a tuple (which saves me a context switch :-). Here's an example:

```
>>> from humanfriendly.text import format
>>> # The string formatting operator.
>>> print('the magic number is %s' % 42)
the magic number is 42
>>> print('the magic numbers are %s and %s' % (12, 42))
the magic numbers are 12 and 42
>>> # The format() function.
>>> print(format('the magic number is %s', 42))
the magic number is 42
>>> print(format('the magic numbers are %s and %s', 12, 42))
the magic numbers are 12 and 42
```

- When you interpolate a single value and someone accidentally passes in a tuple your code raises a `TypeError`. Because *`format()`* takes a variable number of arguments it always receives a tuple so this can never happen. Here's an example:

```
>>> # How expecting to interpolate a single value can fail.
>>> value = (12, 42)
>>> print('the magic value is %s' % value)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: not all arguments converted during string formatting
>>> # The following line works as intended, no surprises here!
>>> print(format('the magic value is %s', value))
the magic value is (12, 42)
```

Why format() instead of the str.format() method?

When you're doing complex string interpolation the `str.format()` function results in more readable code, however I frequently find myself adding parentheses to force evaluation order. The *`format()`* function avoids this because of the relative priority between the comma and dot operators. Here's an example:

```
>>> "{adjective} example" + " " + "(can't think of anything less {adjective})".
↪format(adjective='silly')
"{adjective} example (can't think of anything less silly)"
>>> ("{adjective} example" + " " + "(can't think of anything less {adjective})").
↪format(adjective='silly')
"silly example (can't think of anything less silly)"
>>> format("{adjective} example" + " " + "(can't think of anything less
↪{adjective})", adjective='silly')
"silly example (can't think of anything less silly)"
```

The *`compact()`* and *`dedent()`* functions are wrappers that combine *`format()`* with whitespace manipulation to make it easy to write nice to read Python code.

`humanfriendly.text.`**`compact`**(*text*, **args*, ***kw*)

Compact whitespace in a string.

Trims leading and trailing whitespace, replaces runs of whitespace characters with a single space and interpolates any arguments using *`format()`*.

Parameters

- **`text`** – The text to compact (a string).
- **`args`** – Any positional arguments are interpolated using *`format()`*.
- **`kw`** – Any keyword arguments are interpolated using *`format()`*.

Returns The compacted text (a string).

Here's an example of how I like to use the *`compact()`* function, this is an example from a random unrelated project I'm working on at the moment:

```
raise PortDiscoveryError(compact("""
    Failed to discover port(s) that Apache is listening on!
    Maybe I'm parsing the wrong configuration file? ({filename})
""", filename=self.ports_config))
```

The combination of *`compact()`* and Python's multi line strings allows me to write long text fragments with interpolated variables that are easy to write, easy to read and work well with Python's whitespace sensitivity.

humanfriendly.text.**dedent**(*text*, **args*, ***kw*)
Dedent a string (remove common leading whitespace from all lines).

Removes common leading whitespace from all lines in the string using `textwrap.dedent()`, removes leading and trailing empty lines using *trim_empty_lines()* and interpolates any arguments using *format()*.

Parameters

- **text** – The text to dedent (a string).
- **args** – Any positional arguments are interpolated using *format()*.
- **kw** – Any keyword arguments are interpolated using *format()*.

Returns The dedented text (a string).

The *compact()* function's documentation contains an example of how I like to use the *compact()* and *dedent()* functions. The main difference is that I use *compact()* for text that will be presented to the user (where whitespace is not so significant) and *dedent()* for data file and code generation tasks (where newlines and indentation are very significant).

humanfriendly.text.**trim_empty_lines**(*text*)
Trim leading and trailing empty lines from the given text.

Parameters **text** – The text to trim (a string).

Returns The trimmed text (a string).

humanfriendly.text.**is_empty_line**(*text*)
Check if a text is empty or contains only whitespace.

Parameters **text** – The text to check for "emptiness" (a string).

Returns `True` if the text is empty or contains only whitespace, `False` otherwise.

humanfriendly.text.**split_paragraphs**(*text*)
Split a string into paragraphs (one or more lines delimited by an empty line).

Parameters **text** – The text to split into paragraphs (a string).

Returns A list of strings.

humanfriendly.text.**join_lines**(*text*)
Remove "hard wrapping" from the paragraphs in a string.

Parameters **text** – The text to reformat (a string).

Returns The text without hard wrapping (a string).

This function works by removing line breaks when the last character before a line break and the first character after the line break are both non-whitespace characters. This means that common leading indentation will break *join_lines()* (in that case you can use *dedent()* before calling *join_lines()*).

humanfriendly.text.**pluralize**(*count*, *singular*, *plural=None*)
Combine a count with the singular or plural form of a word.

If the plural form of the word is not provided it is obtained by concatenating the singular form of the word with the letter "s". Of course this will not always be correct, which is why you have the option to specify both forms.

Parameters

- **count** – The count (a number).
- **singular** – The singular form of the word (a string).
- **plural** – The plural form of the word (a string or `None`).

Returns The count and singular/plural word concatenated (a string).

humanfriendly.text.**split**(*text*, *delimiter=', '*)

Split a comma-separated list of strings.

Parameters

- **text** – The text to split (a string).
- **delimiter** – The delimiter to split on (a string).

Returns A list of zero or more nonempty strings.

Here's the default behavior of Python's built in `str.split()` function:

```
>>> 'foo,bar, baz,'.split(',')
['foo', 'bar', ' baz', '']
```

In contrast here's the default behavior of the *split()* function:

```
>>> from humanfriendly.text import split
>>> split('foo,bar, baz,')
['foo', 'bar', 'baz']
```

Here is an example that parses a nested data structure (a mapping of logging level names to one or more styles per level) that's encoded in a string so it can be set as an environment variable:

```
>>> from pprint import pprint
>>> encoded_data = 'debug=green;warning=yellow;error=red;critical=red,bold'
>>> parsed_data = dict((k, split(v, ',')) for k, v in (split(kv, '=') for kv in␣
↪split(encoded_data, ';')))
>>> pprint(parsed_data)
{'debug': ['green'],
 'warning': ['yellow'],
 'error': ['red'],
 'critical': ['red', 'bold']}
```

humanfriendly.text.**tokenize**(*text*)

Tokenize a text into numbers and strings.

Parameters **text** – The text to tokenize (a string).

Returns A list of strings and/or numbers.

This function is used to implement robust tokenization of user input in functions like *parse_size()* and *parse_timespan()*. It automatically coerces integer and floating point numbers, ignores whitespace and knows how to separate numbers from strings even without whitespace. Some examples to make this more concrete:

```
>>> from humanfriendly.text import tokenize
>>> tokenize('42')
[42]
>>> tokenize('42MB')
[42, 'MB']
>>> tokenize('42.5MB')
[42.5, 'MB']
>>> tokenize('42.5 MB')
[42.5, 'MB']
```

The `humanfriendly.tables` module

Functions that render ASCII tables.

Some generic notes about the table formatting functions in this module:

- These functions were not written with performance in mind (*at all*) because they're intended to format tabular data to be presented on a terminal. If someone were to run into a performance problem using these functions, they'd be printing so much tabular data to the terminal that a human wouldn't be able to digest the tabular data anyway, so the point is moot :-).
- These functions ignore ANSI escape sequences (at least the ones generated by the *terminal* module) in the calculation of columns widths. On reason for this is that column names are highlighted in color when connected to a terminal. It also means that you can use ANSI escape sequences to highlight certain column's values if you feel like it (for example to highlight deviations from the norm in an overview of calculated values).

humanfriendly.tables.**format_smart_table**(*data*, *column_names*)

Render tabular data using the most appropriate representation.

Parameters

- **data** – An iterable (e.g. a `tuple()` or `list`) containing the rows of the table, where each row is an iterable containing the columns of the table (strings).
- **column_names** – An iterable of column names (strings).

Returns The rendered table (a string).

If you want an easy way to render tabular data on a terminal in a human friendly format then this function is for you! It works as follows:

- If the input data doesn't contain any line breaks the function *format_pretty_table()* is used to render a pretty table. If the resulting table fits in the terminal without wrapping the rendered pretty table is returned.
- If the input data does contain line breaks or if a pretty table would wrap (given the width of the terminal) then the function *format_robust_table()* is used to render a more robust table that can deal with data containing line breaks and long text.

humanfriendly.tables.**format_pretty_table**(*data*, *column_names=None*, *horizontal_bar='-'*, *vertical_bar='|'*)

Render a table using characters like dashes and vertical bars to emulate borders.

Parameters

- **data** – An iterable (e.g. a `tuple()` or `list`) containing the rows of the table, where each row is an iterable containing the columns of the table (strings).
- **column_names** – An iterable of column names (strings).
- **horizontal_bar** – The character used to represent a horizontal bar (a string).
- **vertical_bar** – The character used to represent a vertical bar (a string).

Returns The rendered table (a string).

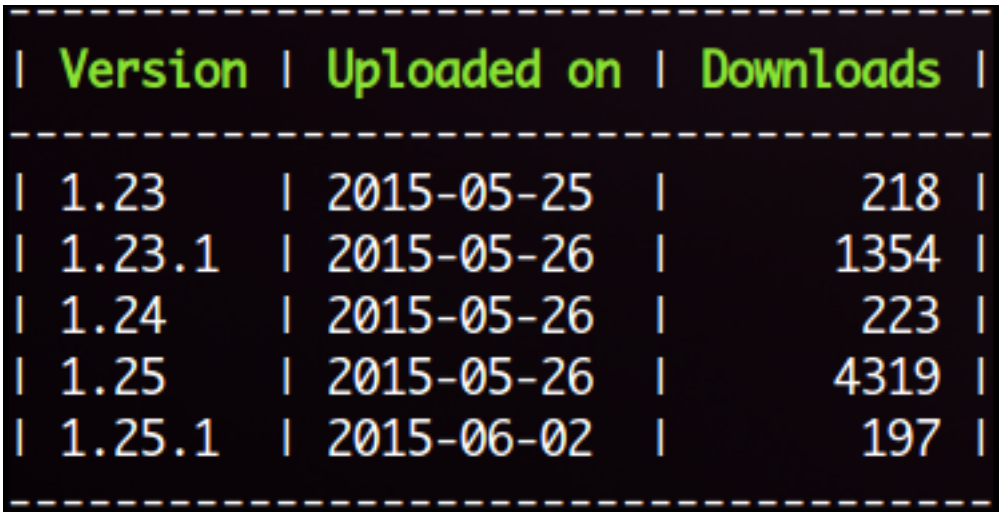
Here's an example:

```
>>> from humanfriendly.tables import format_pretty_table
>>> column_names = ['Version', 'Uploaded on', 'Downloads']
>>> humanfriendly_releases = [
... ['1.23', '2015-05-25', '218'],
... ['1.23.1', '2015-05-26', '1354'],
```

```
... ['1.24', '2015-05-26', '223'],
... ['1.25', '2015-05-26', '4319'],
... ['1.25.1', '2015-06-02', '197'],
... ]
>>> print(format_pretty_table(humanfriendly_releases, column_names))
-------------------------------------
| Version | Uploaded on | Downloads |
-------------------------------------
| 1.23    | 2015-05-25  |       218 |
| 1.23.1  | 2015-05-26  |      1354 |
| 1.24    | 2015-05-26  |       223 |
| 1.25    | 2015-05-26  |      4319 |
| 1.25.1  | 2015-06-02  |       197 |
-------------------------------------
```

Notes about the resulting table:

- If a column contains numeric data (integer and/or floating point numbers) in all rows (ignoring column names of course) then the content of that column is right-aligned, as can be seen in the example above. The idea here is to make it easier to compare the numbers in different columns to each other.
- The column names are highlighted in color so they stand out a bit more (see also *HIGHLIGHT_COLOR*). The following screen shot shows what that looks like (my terminals are always set to white text on a black background):



humanfriendly.tables.**format_robust_table**(*data*, *column_names*)

Render tabular data with one column per line (allowing columns with line breaks).

Parameters

- **data** – An iterable (e.g. a `tuple()` or `list`) containing the rows of the table, where each row is an iterable containing the columns of the table (strings).
- **column_names** – An iterable of column names (strings).

Returns The rendered table (a string).

Here's an example:

```
>>> from humanfriendly.tables import format_robust_table
>>> column_names = ['Version', 'Uploaded on', 'Downloads']
>>> humanfriendly_releases = [
```

```
... ['1.23', '2015-05-25', '218'],
... ['1.23.1', '2015-05-26', '1354'],
... ['1.24', '2015-05-26', '223'],
... ['1.25', '2015-05-26', '4319'],
... ['1.25.1', '2015-06-02', '197'],
... ]
>>> print(format_robust_table(humanfriendly_releases, column_names))
-----------------------
Version: 1.23
Uploaded on: 2015-05-25
Downloads: 218
-----------------------
Version: 1.23.1
Uploaded on: 2015-05-26
Downloads: 1354
-----------------------
Version: 1.24
Uploaded on: 2015-05-26
Downloads: 223
-----------------------
Version: 1.25
Uploaded on: 2015-05-26
Downloads: 4319
-----------------------
Version: 1.25.1
Uploaded on: 2015-06-02
Downloads: 197
-----------------------
```

The column names are highlighted in bold font and color so they stand out a bit more (see *HIGHLIGHT_COLOR*).

The `humanfriendly.terminal` module

Interaction with UNIX terminals.

The *terminal* module makes it easy to interact with UNIX terminals and format text for rendering on UNIX terminals. If the terms used in the documentation of this module don't make sense to you then please refer to the Wikipedia article on ANSI escape sequences for details about how ANSI escape sequences work.

`humanfriendly.terminal.`**`ANSI_CSI`** = **'\x1b['**
The ANSI "Control Sequence Introducer" (a string).

`humanfriendly.terminal.`**`ANSI_SGR`** = **'m'**
The ANSI "Select Graphic Rendition" sequence (a string).

`humanfriendly.terminal.`**`ANSI_ERASE_LINE`** = **'\x1b[K'**
The ANSI escape sequence to erase the current line (a string).

`humanfriendly.terminal.`**`ANSI_RESET`** = **'\x1b[0m'**
The ANSI escape sequence to reset styling (a string).

`humanfriendly.terminal.`**`ANSI_COLOR_CODES`** = **{'blue': 4, 'yellow': 3, 'green': 2, 'cyan': 6, 'white': 7, 'magenta': 5, 'r**
A dictionary with (name, number) pairs of portable color codes. Used by *ansi_style()* to generate ANSI escape sequences that change font color.

humanfriendly.terminal.**ANSI_TEXT_STYLES = {'inverse': 7, 'strike_through': 9, 'faint': 2, 'underline': 4, 'bold': 1}**
A dictionary with (name, number) pairs of text styles (effects). Used by *ansi_style()* to generate ANSI escape sequences that change text styles. Only widely supported text styles are included here.

humanfriendly.terminal.**CLEAN_OUTPUT_PATTERN = <_sre.SRE_Pattern object>**
A compiled regular expression used to separate significant characters from other text.

This pattern is used by *clean_terminal_output()* to split terminal output into regular text versus backspace, carriage return and line feed characters and ANSI 'erase line' escape sequences.

humanfriendly.terminal.**DEFAULT_LINES = 25**
The default number of lines in a terminal (an integer).

humanfriendly.terminal.**DEFAULT_COLUMNS = 80**
The default number of columns in a terminal (an integer).

humanfriendly.terminal.**DEFAULT_ENCODING = 'UTF-8'**
The output encoding for Unicode strings.

humanfriendly.terminal.**HIGHLIGHT_COLOR = 'green'**
The color used to highlight important tokens in formatted text (e.g. the usage message of the `humanfriendly` program). If the environment variable `$HUMANFRIENDLY_HIGHLIGHT_COLOR` is set it determines the value of *HIGHLIGHT_COLOR*.

humanfriendly.terminal.**output**(*text*, **args*, ***kw*)
Print a formatted message to the standard output stream.

For details about argument handling please refer to *format()*.

Renders the message using *format()* and writes the resulting string (followed by a newline) to `sys.stdout` using *auto_encode()*.

humanfriendly.terminal.**message**(*text*, **args*, ***kw*)
Print a formatted message to the standard error stream.

For details about argument handling please refer to *format()*.

Renders the message using *format()* and writes the resulting string (followed by a newline) to `sys.stderr` using *auto_encode()*.

humanfriendly.terminal.**warning**(*text*, **args*, ***kw*)
Show a warning message on the terminal.

For details about argument handling please refer to *format()*.

Renders the message using *format()* and writes the resulting string (followed by a newline) to `sys.stderr` using *auto_encode()*.

If `sys.stderr` is connected to a terminal that supports colors, *ansi_wrap()* is used to color the message in a red font (to make the warning stand out from surrounding text).

humanfriendly.terminal.**auto_encode**(*stream*, *text*, **args*, ***kw*)
Reliably write Unicode strings to the terminal.

Parameters

- **stream** – The file-like object to write to (a value like `sys.stdout` or `sys.stderr`).
- **text** – The text to write to the stream (a string).
- **args** – Refer to *format()*.
- **kw** – Refer to *format()*.

Renders the text using *format()* and writes it to the given stream. If an UnicodeEncodeError is encountered in doing so, the text is encoded using *DEFAULT_ENCODING* and the write is retried. The reasoning behind this rather blunt approach is that it's preferable to get output on the command line in the wrong encoding then to have the Python program blow up with a UnicodeEncodeError exception.

humanfriendly.terminal.**ansi_strip**(*text*, *readline_hints=True*)

Strip ANSI escape sequences from the given string.

Parameters

- **text** – The text from which ANSI escape sequences should be removed (a string).
- **readline_hints** – If True then *readline_strip()* is used to remove readline hints from the string.

Returns The text without ANSI escape sequences (a string).

humanfriendly.terminal.**ansi_style**(***kw*)

Generate ANSI escape sequences for the given color and/or style(s).

Parameters

- **color** – The name of a color (one of the strings 'black', 'red', 'green', 'yellow', 'blue', 'magenta', 'cyan' or 'white') or None (the default) which means no escape sequence to switch color will be emitted.
- **readline_hints** – If True then *readline_wrap()* is applied to the generated ANSI escape sequences (the default is False).
- **kw** – Any additional keyword arguments are expected to match an entry in the *ANSI_TEXT_STYLES* dictionary. If the argument's value evaluates to True the respective style will be enabled.

Returns The ANSI escape sequences to enable the requested text styles or an empty string if no styles were requested.

Raises ValueError when an invalid color name is given.

humanfriendly.terminal.**ansi_width**(*text*)

Calculate the effective width of the given text (ignoring ANSI escape sequences).

Parameters **text** – The text whose width should be calculated (a string).

Returns The width of the text without ANSI escape sequences (an integer).

This function uses *ansi_strip()* to strip ANSI escape sequences from the given string and returns the length of the resulting string.

humanfriendly.terminal.**ansi_wrap**(*text*, ***kw*)

Wrap text in ANSI escape sequences for the given color and/or style(s).

Parameters

- **text** – The text to wrap (a string).
- **kw** – Any keyword arguments are passed to *ansi_style()*.

Returns

The result of this function depends on the keyword arguments:

- If *ansi_style()* generates an ANSI escape sequence based on the keyword arguments, the given text is prefixed with the generated ANSI escape sequence and suffixed with *ANSI_RESET*.

- If *ansi_style()* returns an empty string then the text given by the caller is returned unchanged.

humanfriendly.terminal.**readline_wrap**(*expr*)
Wrap an ANSI escape sequence in readline hints.

Parameters **text** – The text with the escape sequence to wrap (a string).

Returns The wrapped text.

humanfriendly.terminal.**readline_strip**(*expr*)
Remove readline hints from a string.

Parameters **text** – The text to strip (a string).

Returns The stripped text.

humanfriendly.terminal.**clean_terminal_output**(*text*)
Clean up the terminal output of a command.

Parameters **text** – The raw text with special characters (a Unicode string).

Returns A list of Unicode strings (one for each line).

This function emulates the effect of backspace (0x08), carriage return (0x0D) and line feed (0x0A) characters and the ANSI 'erase line' escape sequence on interactive terminals. It's intended to clean up command output that was originally meant to be rendered on an interactive terminal and that has been captured using e.g. the script program[1] or the `pty` module[2].

Some caveats about the use of this function:

- Strictly speaking the effect of carriage returns cannot be emulated outside of an actual terminal due to the interaction between overlapping output, terminal widths and line wrapping. The goal of this function is to sanitize noise in terminal output while preserving useful output. Think of it as a useful and pragmatic but possibly lossy conversion.
- The algorithm isn't smart enough to properly handle a pair of ANSI escape sequences that open before a carriage return and close after the last carriage return in a linefeed delimited string; the resulting string will contain only the closing end of the ANSI escape sequence pair. Tracking this kind of complexity requires a state machine and proper parsing.

humanfriendly.terminal.**connected_to_terminal**(*stream=None*)
Check if a stream is connected to a terminal.

Parameters **stream** – The stream to check (a file-like object, defaults to `sys.stdout`).

Returns `True` if the stream is connected to a terminal, `False` otherwise.

See also *terminal_supports_colors()*.

humanfriendly.terminal.**terminal_supports_colors**(*stream=None*)
Check if a stream is connected to a terminal that supports ANSI escape sequences.

Parameters **stream** – The stream to check (a file-like object, defaults to `sys.stdout`).

Returns `True` if the terminal supports ANSI escape sequences, `False` otherwise.

This function is inspired by the implementation of django.core.management.color.supports_color().

humanfriendly.terminal.**find_terminal_size**()
Determine the number of lines and columns visible in the terminal.

[1] My coloredlogs package supports the `coloredlogs --to-html` command which uses script to fool a subprocess into thinking that it's connected to an interactive terminal (in order to get it to emit ANSI escape sequences).

[2] My capturer package uses the `pty` module to fool the current process and subprocesses into thinking they are connected to an interactive terminal (in order to get them to emit ANSI escape sequences).

Returns A tuple of two integers with the line and column count.

The result of this function is based on the first of the following three methods that works:

1. First *find_terminal_size_using_ioctl()* is tried,
2. then *find_terminal_size_using_stty()* is tried,
3. finally *DEFAULT_LINES* and *DEFAULT_COLUMNS* are returned.

Note: The *find_terminal_size()* function performs the steps above every time it is called, the result is not cached. This is because the size of a virtual terminal can change at any time and the result of *find_terminal_size()* should be correct.

Pre-emptive snarky comment: It's possible to cache the result of this function and use `signal.SIGWINCH` to refresh the cached values!

Response: As a library I don't consider it the role of the *humanfriendly.terminal* module to install a process wide signal handler ...

`humanfriendly.terminal.`**`find_terminal_size_using_ioctl`**(*stream*)

Find the terminal size using `fcntl.ioctl()`.

Parameters **`stream`** – A stream connected to the terminal (a file object with a `fileno` attribute).

Returns A tuple of two integers with the line and column count.

Raises This function can raise exceptions but I'm not going to document them here, you should be using *find_terminal_size()*.

Based on an implementation found on StackOverflow.

`humanfriendly.terminal.`**`find_terminal_size_using_stty`**()

Find the terminal size using the external command `stty size`.

Parameters **`stream`** – A stream connected to the terminal (a file object).

Returns A tuple of two integers with the line and column count.

Raises This function can raise exceptions but I'm not going to document them here, you should be using *find_terminal_size()*.

`humanfriendly.terminal.`**`usage`**(*usage_text*)

Print a human friendly usage message to the terminal.

Parameters **`text`** – The usage message to print (a string).

This function does two things:

1. If `sys.stdout` is connected to a terminal (see *connected_to_terminal()*) then the usage message is formatted using *format_usage()*.
2. The usage message is shown using a pager (see *show_pager()*).

`humanfriendly.terminal.`**`show_pager`**(*formatted_text*, *encoding='UTF-8'*)

Print a large text to the terminal using a pager.

Parameters

- **`formatted_text`** – The text to print to the terminal (a string).
- **`encoding`** – The name of the text encoding used to encode the formatted text if the formatted text is a Unicode string (a string, defaults to *DEFAULT_ENCODING*).

When *connected_to_terminal()* returns `True` a pager is used to show the text on the terminal, otherwise the text is printed directly without invoking a pager.

The use of a pager helps to avoid the wall of text effect where the user has to scroll up to see where the output began (not very user friendly).

Refer to *get_pager_command()* for details about the command line that's used to invoke the pager.

humanfriendly.terminal.**get_pager_command**(*text=None*)

Get the command to show a text on the terminal using a pager.

Parameters **text** – The text to print to the terminal (a string).

Returns A list of strings with the pager command and arguments.

The use of a pager helps to avoid the wall of text effect where the user has to scroll up to see where the output began (not very user friendly).

If the given text contains ANSI escape sequences the command `less --RAW-CONTROL-CHARS` is used, otherwise the environment variable `$PAGER` is used (if `$PAGER` isn't set less is used).

When the selected pager is less, the following options are used to make the experience more user friendly:

- `--quit-if-one-screen` causes less to automatically exit if the entire text can be displayed on the first screen. This makes the use of a pager transparent for smaller texts (because the operator doesn't have to quit the pager).
- `--no-init` prevents less from clearing the screen when it exits. This ensures that the operator gets a chance to review the text (for example a usage message) after quitting the pager, while composing the next command.

The `humanfriendly.usage` module

Parsing and reformatting of usage messages.

The *usage* module parses and reformats usage messages:

- The *format_usage()* function takes a usage message and inserts ANSI escape sequences that highlight items of special significance like command line options, meta variables, etc. The resulting usage message is (intended to be) easier to read on a terminal.
- The *render_usage()* function takes a usage message and rewrites it to reStructuredText suitable for inclusion in the documentation of a Python package. This provides a DRY solution to keeping a single authoritative definition of the usage message while making it easily available in documentation. As a cherry on the cake it's not just a pre-formatted dump of the usage message but a nicely formatted reStructuredText fragment.
- The remaining functions in this module support the two functions above.

Usage messages in general are free format of course, however the functions in this module assume a certain structure from usage messages in order to successfully parse and reformat them, refer to *parse_usage()* for details.

humanfriendly.usage.**USAGE_MARKER = 'Usage:'**

The string that starts the first line of a usage message.

humanfriendly.usage.**format_usage**(*usage_text*)

Highlight special items in a usage message.

Parameters **usage_text** – The usage message to process (a string).

Returns The usage message with special items highlighted.

This function highlights the following special items:

- The initial line of the form "Usage: ..."
- Short and long command line options
- Environment variables
- Meta variables (see *find_meta_variables()*)

All items are highlighted in the color defined by *HIGHLIGHT_COLOR*.

humanfriendly.usage.**find_meta_variables**(*usage_text*)

Find the meta variables in the given usage message.

Parameters **usage_text** – The usage message to parse (a string).

Returns A list of strings with any meta variables found in the usage message.

When a command line option requires an argument, the convention is to format such options as `--option=ARG`. The text `ARG` in this example is the meta variable.

humanfriendly.usage.**parse_usage**(*text*)

Parse a usage message by inferring its structure (and making some assumptions :-).

Parameters **text** – The usage message to parse (a string).

Returns

A tuple of two lists:

1. A list of strings with the paragraphs of the usage message's "introduction" (the paragraphs before the documentation of the supported command line options).
2. A list of strings with pairs of command line options and their descriptions: Item zero is a line listing a supported command line option, item one is the description of that command line option, item two is a line listing another supported command line option, etc.

Usage messages in general are free format of course, however *parse_usage()* assume a certain structure from usage messages in order to successfully parse them:

- The usage message starts with a line `Usage: ...` that shows a symbolic representation of the way the program is to be invoked.
- After some free form text a line `Supported options:` (surrounded by empty lines) precedes the documentation of the supported command line options.
- The command line options are documented as follows:

```
-v, --verbose

  Make more noise.
```

  So all of the variants of the command line option are shown together on a separate line, followed by one or more paragraphs describing the option.

- There are several other minor assumptions, but to be honest I'm not sure if anyone other than me is ever going to use this functionality, so for now I won't list every intricate detail :-).

If you're curious anyway, refer to the usage message of the *humanfriendly* package (defined in the *humanfriendly.cli* module) and compare it with the usage message you see when you run `humanfriendly --help` and the generated usage message embedded in the readme.

Feel free to request more detailed documentation if you're interested in using the *humanfriendly.usage* module outside of the little ecosystem of Python packages that I have been building over the past years.

humanfriendly.usage.**render_usage**(*text*)

Reformat a command line program's usage message to reStructuredText.

Parameters **text** – The plain text usage message (a string).

Returns The usage message rendered to reStructuredText (a string).

humanfriendly.usage.**inject_usage**(*module_name*)

Use cog to inject a usage message into a reStructuredText file.

Parameters **module_name** – The name of the module whose __doc__ attribute is the source of the usage message (a string).

This simple wrapper around *render_usage()* makes it very easy to inject a reformatted usage message into your documentation using cog. To use it you add a fragment like the following to your *.rst file:

```
.. [[[cog
.. from humanfriendly.usage import inject_usage
.. inject_usage('humanfriendly.cli')
.. ]]]
.. [[[end]]]
```

The lines in the fragment above are single line reStructuredText comments that are not copied to the output. Their purpose is to instruct cog where to inject the reformatted usage message. Once you've added these lines to your *.rst file, updating the rendered usage message becomes really simple thanks to cog:

```
$ cog.py -r README.rst
```

This will inject or replace the rendered usage message in your README.rst file with an up to date copy.

humanfriendly.usage.**import_module**(*name*, *package=None*)

Import a module.

The 'package' argument is required when performing a relative import. It specifies the package to use as the anchor point from which to resolve the relative import to an absolute import.

Python Module Index

h

Index

Symbols

A

B

C

D

O

P

R

S

T

U

W

Z