Python Utils Documentation

Release 2.5.5

Rick van Hattem

Jan 26, 2021

Contents

Contents:

CHAPTER 1

Useful Python Utils

Python Utils is a collection of small Python functions and classes which make common patterns shorter and easier. It is by no means a complete collection but it has served me quite a bit in the past and I will keep extending it.

One of the libraries using Python Utils is Django Utils.

Documentation is available at: https://python-utils.readthedocs.org/en/latest/

1.1 Links

- The source: https://github.com/WoLpH/python-utils
- Project page: https://pypi.python.org/pypi/python-utils
- Reporting bugs: https://github.com/WoLpH/python-utils/issues
- Documentation: https://python-utils.readthedocs.io/en/latest/
- My blog: https://wol.ph/

1.2 Requirements for installing:

- *six* any recent version

1.3 Installation:

The package can be installed through *pip* (this is the recommended method):

```
pip install python-utils
```

Or if *pip* is not available, *easy_install* should work as well:

> easy_install python-utils

Or download the latest release from Pypi (https://pypi.python.org/pypi/python-utils) or Github.

Note that the releases on Pypi are signed with my GPG key (https://pgp.mit.edu/pks/lookup?op=vindex&search=0xE81444E9CE1F695D) and can be checked using GPG:

> gpg –verify python-utils-<version>.tar.gz.asc python-utils-<version>.tar.gz

1.4 Quickstart

This module makes it easy to execute common tasks in Python scripts such as converting text to numbers and making sure a string is in unicode or bytes format.

1.5 Examples

To extract a number from nearly every string:

```
from python_utils import converters

number = converters.to_int('spam15eggs')
assert number == 15

number = converters.to_int('spam')
assert number == 0

number = converters.to_int('spam', default=1)
assert number == 1

number = converters.to_float('spam1.234')
```

To do a global import programmatically you can use the *import_global* function. This effectively emulates a *from … import **

```
from python_utils.import_ import import_global

# The following is  the equivalent of `from some_module import *`
import_global('some_module')
```

Or add a correclty named logger to your classes which can be easily accessed:

```
class MyClass(Logged):
    def __init__(self):
        Logged.__init__(self)

my_class = MyClass()

# Accessing the logging method:
my_class.error('error')

# With formatting:
my_class.error('The logger supports %(formatting)s',
               formatting='named parameters')
```

(continues on next page)

(continued from previous page)

```python

# Or to access the actual log function (overwriting the log formatting can
# be done n the log method)
import logging
my_class.log(logging.ERROR, 'log')
```

CHAPTER 2

python_utils package

2.1 Submodules

2.2 python_utils.decorators module

`python_utils.decorators.`**`set_attributes`**`(`*`**kwargs`*`)`

Decorator to set attributes on functions and classes

A common usage for this pattern is the Django Admin where functions can get an optional short_description. To illustrate:

Example from the Django admin using this decorator: https://docs.djangoproject.com/en/3.0/ref/contrib/admin/#django.contrib.admin.ModelAdmin.list_display

Our simplified version:

```
>>> @set_attributes(short_description='Name')
... def upper_case_name(self, obj):
...     return ("%s %s" % (obj.first_name, obj.last_name)).upper()
```

The standard Django version:

```
>>> def upper_case_name(obj):
...     return ("%s %s" % (obj.first_name, obj.last_name)).upper()
```

```
>>> upper_case_name.short_description = 'Name'
```

2.3 python_utils.converters module

`python_utils.converters.`**`remap`**`(`*`value, old_min, old_max, new_min, new_max`*`)`

remap a value from one range into another.

```
>>> remap(500, 0, 1000, 0, 100)
50
>>> remap(250.0, 0.0, 1000.0, 0.0, 100.0)
25.0
>>> remap(-75, -100, 0, -1000, 0)
-750
>>> remap(33, 0, 100, -500, 500)
-170
```

This is a great use case example. Take an AVR that has dB values the minimum being -80dB and the maximum being 10dB and you want to convert volume percent to the equilivint in that dB range

```
>>> remap(46.0, 0.0, 100.0, -80.0, 10.0)
-38.6
```

Some edge cases to test >>> remap(0, 0, 0, 0, 0) 0 >>> remap(0, 0, 0, 1, 0) 1

Parameters

- **value** (*int, float*) – value to be converted
- **old_min** (*int, float*) – minimum of the range for the value that has been passed
- **old_max** (*int, float*) – maximum of the range for the value that has been passed
- **new_min** (*int, float*) – the minimum of the new range
- **new_max** (*int, float*) – the maximum of the new range

Returns value that has been re ranged, if the value is an int floor division is used so the returned value will always be rounded down to the closest whole number.

Return type int, float

python_utils.converters.**scale_1024**(*x*, *n_prefixes*)

Scale a number down to a suitable size, based on powers of 1024.

Returns the scaled number and the power of 1024 used.

Use to format numbers of bytes to KiB, MiB, etc.

```
>>> scale_1024(310, 3)
(310.0, 0)
>>> scale_1024(2048, 3)
(2.0, 1)
>>> scale_1024(0, 2)
(0.0, 0)
>>> scale_1024(0.5, 2)
(0.5, 0)
>>> scale_1024(1, 2)
(1.0, 0)
```

python_utils.converters.**to_float**(*input_*, *default=0*, *exception=(<type 'exceptions.ValueError'>*, *<type 'exceptions.TypeError'>)*, *regexp=None*)

Convert the given *input_* to an integer or return default

When trying to convert the exceptions given in the exception parameter are automatically catched and the default will be returned.

The regexp parameter allows for a regular expression to find the digits in a string. When True it will automatically match any digit in the string. When a (regexp) object (has a search method) is given, that will be used. WHen a string is given, re.compile will be run over it first

The last group of the regexp will be used as value

```
>>> '%.2f' % to_float('abc')
'0.00'
>>> '%.2f' % to_float('1')
'1.00'
>>> '%.2f' % to_float('abc123.456', regexp=True)
'123.46'
>>> '%.2f' % to_float('abc123', regexp=True)
'123.00'
>>> '%.2f' % to_float('abc0.456', regexp=True)
'0.46'
>>> '%.2f' % to_float('abc123.456', regexp=re.compile(r'(\d+\.\d+)'))
'123.46'
>>> '%.2f' % to_float('123.456abc', regexp=re.compile(r'(\d+\.\d+)'))
'123.46'
>>> '%.2f' % to_float('abc123.46abc', regexp=re.compile(r'(\d+\.\d+)'))
'123.46'
>>> '%.2f' % to_float('abc123abc456', regexp=re.compile(r'(\d+(\.\d+|))'))
'123.00'
>>> '%.2f' % to_float('abc', regexp=r'(\d+)')
'0.00'
>>> '%.2f' % to_float('abc123', regexp=r'(\d+)')
'123.00'
>>> '%.2f' % to_float('123abc', regexp=r'(\d+)')
'123.00'
>>> '%.2f' % to_float('abc123abc', regexp=r'(\d+)')
'123.00'
>>> '%.2f' % to_float('abc123abc456', regexp=r'(\d+)')
'123.00'
>>> '%.2f' % to_float('1234', default=1)
'1234.00'
>>> '%.2f' % to_float('abc', default=1)
'1.00'
>>> '%.2f' % to_float('abc', regexp=123)
Traceback (most recent call last):
...
TypeError: unknown argument for regexp parameter
```

`python_utils.converters.`**`to_int`**(*input_, default=0, exception=(<type 'exceptions.ValueError'>, <type 'exceptions.TypeError'>), regexp=None*)

Convert the given input to an integer or return default

When trying to convert the exceptions given in the exception parameter are automatically catched and the default will be returned.

The regexp parameter allows for a regular expression to find the digits in a string. When True it will automatically match any digit in the string. When a (regexp) object (has a search method) is given, that will be used. WHen a string is given, re.compile will be run over it first

The last group of the regexp will be used as value

```
>>> to_int('abc')
0
>>> to_int('1')
```

(continues on next page)

(continued from previous page)

```
1
>>> to_int('abc123')
0
>>> to_int('123abc')
0
>>> to_int('abc123', regexp=True)
123
>>> to_int('123abc', regexp=True)
123
>>> to_int('abc123abc', regexp=True)
123
>>> to_int('abc123abc456', regexp=True)
123
>>> to_int('abc123', regexp=re.compile(r'(\d+)'))
123
>>> to_int('123abc', regexp=re.compile(r'(\d+)'))
123
>>> to_int('abc123abc', regexp=re.compile(r'(\d+)'))
123
>>> to_int('abc123abc456', regexp=re.compile(r'(\d+)'))
123
>>> to_int('abc123', regexp=r'(\d+)')
123
>>> to_int('123abc', regexp=r'(\d+)')
123
>>> to_int('abc', regexp=r'(\d+)')
0
>>> to_int('abc123abc', regexp=r'(\d+)')
123
>>> to_int('abc123abc456', regexp=r'(\d+)')
123
>>> to_int('1234', default=1)
1234
>>> to_int('abc', default=1)
1
>>> to_int('abc', regexp=123)
Traceback (most recent call last):
...
TypeError: unknown argument for regexp parameter: 123
```

python_utils.converters.**to_str**(*input_, encoding=u'utf-8', errors=u'replace'*)

Convert objects to string, encodes to the given encoding

Return type str

```
>>> to_str('a')
b'a'
>>> to_str(u'a')
b'a'
>>> to_str(b'a')
b'a'
>>> class Foo(object): __str__ = lambda s: u'a'
>>> to_str(Foo())
'a'
>>> to_str(Foo)
"<class 'python_utils.converters.Foo'>"
```

python_utils.converters.**to_unicode**(*input_, encoding=u'utf-8', errors=u'replace'*)

Convert objects to unicode, if needed decodes string with the given encoding and errors settings.

Return type unicode

```
>>> to_unicode(b'a')
'a'
>>> to_unicode('a')
'a'
>>> to_unicode(u'a')
'a'
>>> class Foo(object): __str__ = lambda s: u'a'
>>> to_unicode(Foo())
'a'
>>> to_unicode(Foo)
"<class 'python_utils.converters.Foo'>"
```

2.4 python_utils.formatters module

python_utils.formatters.**camel_to_underscore**(*name*)

Convert camel case style naming to underscore style naming

If there are existing underscores they will be collapsed with the to-be-added underscores. Multiple consecutive capital letters will not be split except for the last one.

```
>>> camel_to_underscore('SpamEggsAndBacon')
'spam_eggs_and_bacon'
>>> camel_to_underscore('Spam_and_bacon')
'spam_and_bacon'
>>> camel_to_underscore('Spam_And_Bacon')
'spam_and_bacon'
>>> camel_to_underscore('__SpamAndBacon__')
'__spam_and_bacon__'
>>> camel_to_underscore('__SpamANDBacon__')
'__spam_and_bacon__'
```

python_utils.formatters.**timesince**(*dt*, *default='just now'*)

Returns string representing 'time since' e.g. 3 days ago, 5 hours ago etc.

```
>>> now = datetime.datetime.now()
>>> timesince(now)
'just now'
>>> timesince(now - datetime.timedelta(seconds=1))
'1 second ago'
>>> timesince(now - datetime.timedelta(seconds=2))
'2 seconds ago'
>>> timesince(now - datetime.timedelta(seconds=60))
'1 minute ago'
>>> timesince(now - datetime.timedelta(seconds=61))
'1 minute and 1 second ago'
>>> timesince(now - datetime.timedelta(seconds=62))
'1 minute and 2 seconds ago'
>>> timesince(now - datetime.timedelta(seconds=120))
'2 minutes ago'
>>> timesince(now - datetime.timedelta(seconds=121))
'2 minutes and 1 second ago'
>>> timesince(now - datetime.timedelta(seconds=122))
```

(continues on next page)

(continued from previous page)

```
'2 minutes and 2 seconds ago'
>>> timesince(now - datetime.timedelta(seconds=3599))
'59 minutes and 59 seconds ago'
>>> timesince(now - datetime.timedelta(seconds=3600))
'1 hour ago'
>>> timesince(now - datetime.timedelta(seconds=3601))
'1 hour and 1 second ago'
>>> timesince(now - datetime.timedelta(seconds=3602))
'1 hour and 2 seconds ago'
>>> timesince(now - datetime.timedelta(seconds=3660))
'1 hour and 1 minute ago'
>>> timesince(now - datetime.timedelta(seconds=3661))
'1 hour and 1 minute ago'
>>> timesince(now - datetime.timedelta(seconds=3720))
'1 hour and 2 minutes ago'
>>> timesince(now - datetime.timedelta(seconds=3721))
'1 hour and 2 minutes ago'
>>> timesince(datetime.timedelta(seconds=3721))
'1 hour and 2 minutes ago'
```

2.5 python_utils.import_ module

exception `python_utils.import_.`**`DummyException`**

Bases: `exceptions.Exception`

`python_utils.import_.`**`import_global`**(*name*, *modules=None*, *exceptions=<class 'python_utils.import_.DummyException'>*, *locals_=None*, *globals_=None*, *level=-1*)

Import the requested items into the global scope

WARNING! this method _will_ overwrite your global scope If you have a variable named “path” and you call import_global(‘sys’) it will be overwritten with sys.path

Args: name (str): the name of the module to import, e.g. sys modules (str): the modules to import, use None for everything exception (Exception): the exception to catch, e.g. ImportError *locals_*: the *locals()* method (in case you need a different scope) *globals_*: the *globals()* method (in case you need a different scope) level (int): the level to import from, this can be used for relative imports

2.6 python_utils.logger module

class `python_utils.logger.`**`Logged`**

Bases: `object`

Class which automatically adds a named logger to your class when interiting

Adds easy access to debug, info, warning, error, exception and log methods

```
>>> class MyClass(Logged):
...     def __init__(self):
...         Logged.__init__(self)
>>> my_class = MyClass()
>>> my_class.debug('debug')
>>> my_class.info('info')
```

(continues on next page)

(continued from previous page)

```
>>> my_class.warning('warning')
>>> my_class.error('error')
>>> my_class.exception('exception')
>>> my_class.log(0, 'log')
```

classmethod debug(*msg*, **args*, ***kwargs*)
Log a message with severity 'DEBUG' on the root logger.

classmethod error(*msg*, **args*, ***kwargs*)
Log a message with severity 'ERROR' on the root logger.

classmethod exception(*msg*, **args*, ***kwargs*)
Log a message with severity 'ERROR' on the root logger, with exception information.

classmethod info(*msg*, **args*, ***kwargs*)
Log a message with severity 'INFO' on the root logger.

classmethod log(*lvl*, *msg*, **args*, ***kwargs*)
Log 'msg % args' with the integer severity 'level' on the root logger.

classmethod warning(*msg*, **args*, ***kwargs*)
Log a message with severity 'WARNING' on the root logger.

2.7 python_utils.terminal module

python_utils.terminal.**get_terminal_size**()
Get the current size of your terminal

Multiple returns are not always a good idea, but in this case it greatly simplifies the code so I believe it's justified. It's not the prettiest function but that's never really possible with cross-platform code.

Returns: width, height: Two integers containing width and height

2.8 python_utils.time module

python_utils.time.**format_time**(*timestamp*, *precision=datetime.timedelta(0, 1)*)
Formats timedelta/datetime/seconds

```
>>> format_time('1')
'0:00:01'
>>> format_time(1.234)
'0:00:01'
>>> format_time(1)
'0:00:01'
>>> format_time(datetime.datetime(2000, 1, 2, 3, 4, 5, 6))
'2000-01-02 03:04:05'
>>> format_time(datetime.date(2000, 1, 2))
'2000-01-02'
>>> format_time(datetime.timedelta(seconds=3661))
'1:01:01'
>>> format_time(None)
'--:--:--'
>>> format_time(format_time)  # doctest: +ELLIPSIS
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
    ...
TypeError: Unknown type ...
```

python_utils.time.**timedelta_to_seconds**(*delta*)

Convert a timedelta to seconds with the microseconds as fraction

Note that this method has become largely obsolete with the *timedelta.total_seconds()* method introduced in Python 2.7.

```
>>> from datetime import timedelta
>>> '%d' % timedelta_to_seconds(timedelta(days=1))
'86400'
>>> '%d' % timedelta_to_seconds(timedelta(seconds=1))
'1'
>>> '%.6f' % timedelta_to_seconds(timedelta(seconds=1, microseconds=1))
'1.000001'
>>> '%.6f' % timedelta_to_seconds(timedelta(microseconds=1))
'0.000001'
```

2.9 Module contents

Travis status:

build canceled

Coverage:

coverage 100%

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index