
Python Utils

Release 3.9.1

Rick van Hattem

Nov 26, 2024

CONTENTS

1	Useful Python Utils	3
1.1	Links	3
1.2	Security contact information	3
1.3	Requirements for installing:	3
1.4	Installation:	3
1.5	Quickstart	4
1.6	Examples	4
1.6.1	Retrying until timeout	4
1.6.2	Formatting of timestamps, dates and times	4
1.6.3	Converting your test from camel-case to underscores:	5
1.6.4	Attribute setting decorator. Very useful for the Django admin	5
1.6.5	Scaling numbers between ranges	5
1.6.6	Get the screen/window/terminal size in characters:	6
1.6.7	Extracting numbers from nearly every string:	6
1.6.8	Doing a global import of all the modules in a package programmatically:	6
1.6.9	Automatically named logger for classes:	6
2	python_utils package	9
2.1	Submodules	9
2.2	python_utils.decorators module	9
2.3	python_utils.converters module	11
2.4	python_utils.formatters module	15
2.5	python_utils.import_ module	17
2.6	python_utils.logger module	18
2.7	python_utils.terminal module	19
2.8	python_utils.time module	19
2.9	Module contents	22
3	Indices and tables	39
	Python Module Index	41
	Index	43

Contents:

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 Security contact information

To report a security vulnerability, please use the [Tidelift security contact](#). Tidelift will coordinate the fix and disclosure.

1.3 Requirements for installing:

For the Python 3+ release (i.e. v3.0.0 or higher) there are no requirements. For the Python 2 compatible version (v2.x.x) the *six* package is needed.

1.4 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
```

1.5 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.

1.6 Examples

Automatically converting a generator to a list, dict or other collections using a decorator:

```
>>> @decorators.listify()
... def generate_list():
...     yield 1
...     yield 2
...     yield 3
...
>>> generate_list()
[1, 2, 3]

>>> @listify(collection=dict)
... def dict_generator():
...     yield 'a', 1
...     yield 'b', 2
...
>>> dict_generator()
{'a': 1, 'b': 2}
```

1.6.1 Retrying until timeout

To easily retry a block of code with a configurable timeout, you can use the *time.timeout_generator*:

```
>>> for i in time.timeout_generator(10):
...     try:
...         # Run your code here
...     except Exception as e:
...         # Handle the exception
```

1.6.2 Formatting of timestamps, dates and times

Easy formatting of timestamps and calculating the time since:

```
>>> time.format_time('1')
'0:00:01'
>>> time.format_time(1.234)
'0:00:01'
>>> time.format_time(1)
```

(continues on next page)

(continued from previous page)

```
'0:00:01'
>>> time.format_time(datetime.datetime(2000, 1, 2, 3, 4, 5, 6))
'2000-01-02 03:04:05'
>>> time.format_time(datetime.date(2000, 1, 2))
'2000-01-02'
>>> time.format_time(datetime.timedelta(seconds=3661))
'1:01:01'
>>> time.format_time(None)
'--:--:--'

>>> formatters.timesince(now)
'just now'
>>> formatters.timesince(now - datetime.timedelta(seconds=1))
'1 second ago'
>>> formatters.timesince(now - datetime.timedelta(seconds=2))
'2 seconds ago'
>>> formatters.timesince(now - datetime.timedelta(seconds=60))
'1 minute ago'
```

1.6.3 Converting your test from camel-case to underscores:

```
>>> camel_to_underscore('SpamEggsAndBacon')
'spam_eggs_and_bacon'
```

1.6.4 Attribute setting decorator. Very useful for the Django admin

A convenient decorator to set function attributes using a decorator:

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

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

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

This can be very useful for the Django admin as it allows you to have all metadata in one place.

1.6.5 Scaling numbers between ranges

```
>>> converters.remap(500, old_min=0, old_max=1000, new_min=0, new_max=100)
50

# Or with decimals:
>>> remap(decimal.Decimal('250.0'), 0.0, 1000.0, 0.0, 100.0)
Decimal('25.0')
```

1.6.6 Get the screen/window/terminal size in characters:

```
>>> terminal.get_terminal_size()
(80, 24)
```

That method supports IPython and Jupyter as well as regular shells, using *blessings* and other modules depending on what is available.

1.6.7 Extracting numbers from nearly every string:

```
>>> converters.to_int('spam15eggs')
15
>>> converters.to_int('spam')
0
>>> number = converters.to_int('spam', default=1)
1
```

1.6.8 Doing a global import of all the modules in a package programmatically:

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')
```

1.6.9 Automatically named logger for classes:

Or add a correctly 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')

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

Alternatively loguru is also supported. It is largely a drop-in replacement for the logging module which is a bit more convenient to configure:

First install the extra loguru package:

```
pip install 'python-utils[loguru]'
```

```
class MyClass(Logurud):  
    ...
```

Now you can use the *Logurud* class to make functions such as *self.info()* available. The benefit of this approach is that you can add extra context or options to you specific loguru instance (i.e. *self.logger*):

Convenient type aliases and some commonly used types:

```
# For type hinting scopes such as locals/globals/vars  
Scope = Dict[str, Any]  
OptionalScope = O[Scope]  
  
# Note that Number is only useful for extra clarity since float  
# will work for both int and float in practice.  
Number = U[int, float]  
DecimalNumber = U[Number, decimal.Decimal]  
  
# To accept an exception or list of exceptions  
ExceptionType = Type[Exception]  
ExceptionsType = U[Tuple[ExceptionType, ...], ExceptionType]  
  
# Matching string/bytes types:  
StringTypes = U[str, bytes]
```


PYTHON_UTILS PACKAGE

2.1 Submodules

2.2 `python_utils.decorators` module

This module provides various utility decorators for Python functions and methods.

The decorators include:

1. `set_attributes`: Sets attributes on functions and classes.
2. `listify`: Converts any generator to a list or other collection.
3. `sample`: Limits calls to a function based on a sample rate.
4. `wraps_classmethod`: Wraps classmethods with type info from a regular method.

Each decorator is designed to enhance the functionality of Python functions and methods in a simple and reusable manner.

```
python_utils.decorators.listify(collection:  
    ~typing.Callable[~typing.Iterable[~python_utils.decorators._T]],  
    ~typing.Collection[~python_utils.decorators._T]) = <class 'list'>,  
    allow_empty: bool = True) → Callable[[Callable[[...], Iterable[_T] |  
    None]], Callable[[...], Collection[_T]]]
```

Convert any generator to a list or other type of collection.

```
>>> @listify()  
... def generator():  
...     yield 1  
...     yield 2  
...     yield 3
```

```
>>> generator()  
[1, 2, 3]
```

```
>>> @listify()  
... def empty_generator():  
...     pass
```

```
>>> empty_generator()  
[]
```

```
>>> @listify(allow_empty=False)
... def empty_generator_not_allowed():
...     pass
```

```
>>> empty_generator_not_allowed()
Traceback (most recent call last):
...
TypeError: ... `allow_empty` is `False`
```

```
>>> @listify(collection=set)
... def set_generator():
...     yield 1
...     yield 1
...     yield 2
```

```
>>> set_generator()
{1, 2}
```

```
>>> @listify(collection=dict)
... def dict_generator():
...     yield 'a', 1
...     yield 'b', 2
```

```
>>> dict_generator()
{'a': 1, 'b': 2}
```

`python_utils.decorators.sample(sample_rate: float) → Callable[[Callable[[_P], _T]], Callable[[_P], _T] | None]`

Limit calls to a function based on given sample rate. Number of calls to the function will be roughly equal to `sample_rate` percentage.

Usage:

```
>>> @sample(0.5)
... def demo_function(*args, **kwargs):
...     return 1
```

Calls to `demo_function` will be limited to 50% approximatly.

`python_utils.decorators.set_attributes(**kwargs: Any) → Callable[[...], Any]`

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'
```

```
python_utils.decorators.wraps_classmethod(wrapped: Callable[[Concatenate[_S, _P]], _T]) →
Callable[[Callable[[Concatenate[Any, _P]], _T]],
Callable[[Concatenate[_S, _P]], _T]]
```

Like *functools.wraps*, but for wrapping classmethods with the type info from a regular method.

2.3 python_utils.converters module

This module provides utility functions for type conversion.

Functions:

- `to_int`: Convert a string to an integer with optional regular expression

matching. - `to_float`: Convert a string to a float with optional regular expression matching. - `to_unicode`: Convert objects to Unicode strings. - `to_str`: Convert objects to byte strings. - `scale_1024`: Scale a number down to a suitable size based on powers of 1024. - `remap`: Remap a value from one range to another.

```
python_utils.converters.remap(value: Decimal, old_min: Decimal | float, old_max: Decimal | float,
                              new_min: Decimal | float, new_max: Decimal | float) → Decimal
```

```
python_utils.converters.remap(value: Decimal | float, old_min: Decimal, old_max: Decimal | float,
                              new_min: Decimal | float, new_max: Decimal | float) → Decimal
```

```
python_utils.converters.remap(value: Decimal | float, old_min: Decimal | float, old_max: Decimal,
                              new_min: Decimal | float, new_max: Decimal | float) → Decimal
```

```
python_utils.converters.remap(value: Decimal | float, old_min: Decimal | float, old_max: Decimal | float,
                              new_min: Decimal, new_max: Decimal | float) → Decimal
```

```
python_utils.converters.remap(value: Decimal | float, old_min: Decimal | float, old_max: Decimal | float,
                              new_min: Decimal | float, new_max: Decimal) → Decimal
```

```
python_utils.converters.remap(value: float, old_min: float, old_max: float, new_min: float, new_max: float)
                              → float
```

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
>>> remap(decimal.Decimal('250.0'), 0.0, 1000.0, 0.0, 100.0)
Decimal('25.0')
```

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 equivalent in that dB range

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

I added using `decimal.Decimal` so floating point math errors can be avoided. Here is an example of a floating point math error `>>> 0.1 + 0.1 + 0.1 0.30000000000000004`

If floating point remaps need to be done my suggestion is to pass at least one parameter as a *decimal.Decimal*. This will ensure that the output from this function is accurate. I left passing *floats* for backwards compatibility and there is no conversion done from float to *decimal.Decimal* unless one of the passed parameters has a type of *decimal.Decimal*. This will ensure that any existing code that uses this function will work exactly how it has in the past.

Some edge cases to test `>>> remap(1, 0, 0, 1, 2)` Traceback (most recent call last): ... `ValueError: Input range (0-0) is empty`

```
>>> remap(1, 1, 2, 0, 0)
Traceback (most recent call last):
...
ValueError: Output range (0-0) is empty
```

Args:

`value` (int, float, `decimal.Decimal`): Value to be converted. `old_min` (int, float, `decimal.Decimal`): Minimum of the range for the

value that has been passed.

`old_max` (int, float, `decimal.Decimal`): Maximum of the range for the

value that has been passed.

`new_min` (int, float, `decimal.Decimal`): The minimum of the new range. `new_max` (int, float, `decimal.Decimal`): The maximum of the new range.

Returns: int, float, `decimal.Decimal`: Value that has been re-ranged. If

any of the parameters passed is a *decimal.Decimal*, all of the parameters will be converted to *decimal.Decimal*. The same thing also happens if one of the parameters is a *float*. Otherwise, all parameters will get converted into an *int*. Technically, you can pass a *str* of an integer and it will get converted. The returned value type will be *decimal.Decimal* if any of the passed parameters are *decimal.Decimal*, the return type will be *float* if any of the passed parameters are a *float*, otherwise the returned type will be *int*.

`python_utils.converters.scale_1024(x: int | float, n_prefixes: int) → Tuple[int | float, int | float]`

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_: str, default: int = 0, exception:
    ~typing.Tuple[~typing.Type[Exception], ...] | ~typing.Type[Exception] =
    (<class 'ValueError'>, <class 'TypeError'>), regexp: ~typing.Pattern[str]
    | str | ~typing.Literal[True] | None = None) → int | float
```

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

When trying to convert the exceptions given in the exception parameter are automatically caught 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_: str | None = None, default: int = 0, exception:
    ~typing.Tuple[~typing.Type[Exception], ...] | ~typing.Type[Exception] =
    (<class 'ValueError'>, <class 'TypeError'>), regexp: ~typing.Pattern[str] |
    str | ~typing.Literal[True] | None = None) → int
```

Convert the given input to an integer or return default.

When trying to convert the exceptions given in the exception parameter are automatically caught 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')
1
>>> to_int('')
0
>>> to_int()
0
>>> 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)
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
TypeError: unknown argument for regexp parameter: 123
```

`python_utils.converters.to_str(input_: str | bytes, encoding: str = 'utf-8', errors: str = 'replace')` → bytes

Convert objects to string, encodes to the given encoding.

Return type

str

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

`python_utils.converters.to_unicode(input_: str | bytes, encoding: str = 'utf-8', errors: str = 'replace')` → str

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

Return type

str

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

2.4 python_utils.formatters module

This module provides utility functions for formatting strings and dates.

Functions:

camel_to_underscore(name: str) -> str:

Convert camel case style naming to underscore/snake case style naming.

apply_recursive(function: Callable[[str], str], data: OptionalScope = None,

****kwargs: Any) -> OptionalScope:**

Apply a function to all keys in a scope recursively.

timesince(dt: Union[datetime.datetime, datetime.timedelta],

default: str = 'just now') -> str:

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

`python_utils.formatters.apply_recursive(function: Callable[[str], str], data: Dict[str, Any] | None = None, **kwargs: Any) → Dict[str, Any] | None`

Apply a function to all keys in a scope recursively.

```
>>> apply_recursive(camel_to_underscore, {'SpamEggsAndBacon': 'spam'})
{'spam_eggs_and_bacon': 'spam'}
>>> apply_recursive(
...     camel_to_underscore,
...     {
...         'SpamEggsAndBacon': {
...             'SpamEggsAndBacon': 'spam',
...         }
...     },
... )
{'spam_eggs_and_bacon': {'spam_eggs_and_bacon': 'spam'}}
```

```
>>> a = {'a_b_c': 123, 'def': {'DeF': 456}}
>>> b = apply_recursive(camel_to_underscore, a)
>>> b
{'a_b_c': 123, 'def': {'de_f': 456}}
```

```
>>> apply_recursive(camel_to_underscore, None)
```

`python_utils.formatters.camel_to_underscore(name: str) → str`

Convert camel case style naming to underscore/snake case 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: datetime | timedelta, default: str = 'just now') → str`

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))
```

(continues on next page)

(continued from previous page)

```
'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))
'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

This module provides utilities for importing modules and handling exceptions.

Classes:

DummyError(Exception):

A custom exception class used as a default for exception handling.

Functions:

import_global(name, modules=None, exceptions=DummyError, locals_=None, globals_=None, level=-1): Imports the requested items into the global scope, with support for relative imports and custom exception handling.

exception python_utils.import_.**DummyError**

Bases: `Exception`

A custom exception class used as a default for exception handling.

`python_utils.import_.DummyException`

alias of `DummyError`

```
python_utils.import_.import_global(name: str, modules: ~typing.List[str] | None = None, exceptions:
    ~typing.Tuple[~typing.Type[Exception], ...] | ~typing.Type[Exception]
    = <class 'python_utils.import_.DummyError'>, locals_:
    ~typing.Dict[str, ~typing.Any] | None = None, globals_:
    ~typing.Dict[str, ~typing.Any] | None = None, level: int = -1) → Any
```

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 exceptions (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

This module provides a base class `LoggerBase` and a derived class `Logged` for adding logging capabilities to classes. The `LoggerBase` class expects a `logger` attribute to be a `logging.Logger` or compatible instance and provides methods for logging at various levels. The `Logged` class automatically adds a named logger to the class.

Classes:

LoggerBase:

A base class that adds logging utilities to a class.

Logged:

A derived class that automatically adds a named logger to a class.

Example:

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

```
>>> my_class = MyClass()
>>> my_class.debug('debug')
>>> my_class.info('info')
>>> my_class.warning('warning')
>>> my_class.error('error')
>>> my_class.exception('exception')
>>> my_class.log(0, 'log')
```

`class python_utils.logger.Logged(*args: Any, **kwargs: Any)`

Bases: `LoggerBase`

Class which automatically adds a named logger to your class when inheriting.

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

```
>>> my_class._Logged__get_name('spam')
'spam'
```

logger: `Logger`

2.7 python_utils.terminal module

This module provides functions to get the terminal size across different platforms.

Functions:

`get_terminal_size`: Get the current size of the terminal. `_get_terminal_size_windows`: Get terminal size on Windows. `_get_terminal_size_tput`: Get terminal size using *tput*. `_get_terminal_size_linux`: Get terminal size on Linux.

Usage example:

```
>>> width, height = get_terminal_size()
```

`python_utils.terminal.get_terminal_size()` → `tuple[int, int]`

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

This module provides utility functions for handling time-related operations.

Functions: - `timedelta_to_seconds`: Convert a `timedelta` to seconds with microseconds as fraction.

- `delta_to_seconds`: Convert a `timedelta` or numeric interval to seconds.
- `delta_to_seconds_or_none`: Convert a `timedelta` to seconds or return `None`.
- `format_time`: Format a timestamp (`timedelta`, `datetime`, or seconds) to a string.
- `timeout_generator`: Generate items from an iterable until a timeout is reached.
- `aio_timeout_generator`: Asynchronously generate items from an iterable until a timeout is reached.
- `aio_generator_timeout_detector`: Detect if an `async` generator has not yielded an element for a set amount of time.
- `aio_generator_timeout_detector_decorator`: Decorator for `aio_generator_timeout_detector`.

```
async python_utils.time.aio_generator_timeout_detector(generator: ~typing.AsyncGenerator[~python_utils.time._T, None], timeout: ~datetime.timedelta | int | float | None = None, total_timeout: ~datetime.timedelta | int | float | None = None, on_timeout: ~typing.Callable[[~typing.AsyncGenerator[~python_utils.time._T, None], ~datetime.timedelta | int | float | None, ~datetime.timedelta | int | float | None, BaseException], ~typing.Any] | None = <function reraise>, **on_timeout_kwargs: ~typing.Mapping[str, ~typing.Any]) → AsyncGenerator[_T, None]
```

This function is used to detect if an asyncio generator has not yielded an element for a set amount of time.

The *on_timeout* argument is called with the *generator*, *timeout*, *total_timeout*, *exception* and the extra ***kwargs* to this function as arguments. If *on_timeout* is not specified, the exception is reraised. If *on_timeout* is *None*, the exception is silently ignored and the generator will finish as normal.

```
python_utils.time.aio_generator_timeout_detector_decorator(timeout: ~datetime.timedelta | int | float | None = None, total_timeout: ~datetime.timedelta | int | float | None = None, on_timeout: ~typing.Callable[[~typing.AsyncGenerator[~typing.Any, None], ~datetime.timedelta | int | float | None, ~datetime.timedelta | int | float | None, BaseException], ~typing.Any] | None = <function reraise>, **on_timeout_kwargs: ~typing.Mapping[str, ~typing.Any]) → Callable[[Callable[[_P], AsyncGenerator[_T, None]]], Callable[[_P], AsyncGenerator[_T, None]]]
```

A decorator wrapper for *aio_generator_timeout_detector*.

```
async python_utils.time.aio_timeout_generator(timeout: ~datetime.timedelta | int | float, interval: ~datetime.timedelta | int | float = datetime.timedelta(seconds=1), iterable: ~typing.AsyncIterable[~python_utils.time._T] | ~typing.Callable[...], ~typing.AsyncIterable[~python_utils.time._T]) = <function acount>, interval_multiplier: float = 1.0, maximum_interval: ~datetime.timedelta | int | float | None = None) → AsyncGenerator[_T, None]
```

Async generator that walks through the given async iterable (a counter by default) until the float_timeout is reached with a configurable float_interval between items.

The *interval_exponent* automatically increases the float_timeout with each run. Note that if the float_interval is less than 1, 1/interval_exponent will be used so the float_interval is always growing. To double the float_interval with each run, specify 2.

Doctests and asyncio are not friends, so no examples. But this function is effectively the same as the *timeout_generator* but it uses *async for* instead.

`python_utils.time.delta_to_seconds(interval: timedelta | int | float) → int | float`

Convert a timedelta to seconds.

```
>>> delta_to_seconds(datetime.timedelta(seconds=1))
1
>>> delta_to_seconds(datetime.timedelta(seconds=1, microseconds=1))
1.000001
>>> delta_to_seconds(1)
1
>>> delta_to_seconds('whatever')
Traceback (most recent call last):
...
TypeError: Unknown type ...
```

`python_utils.time.delta_to_seconds_or_none(interval: timedelta | int | float | None) → int | float | None`

Convert a timedelta to seconds or return None.

`python_utils.time.format_time(timestamp: timedelta | date | datetime | str | int | float | None, precision: timedelta = datetime.timedelta(seconds=1)) → str`

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)
Traceback (most recent call last):
...
TypeError: Unknown type ...
```

`python_utils.time.timedelta_to_seconds(delta: timedelta) → int | float`

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'
```

(continues on next page)

(continued from previous page)

```
>>> '%.6f' % timedelta_to_seconds(timedelta(microseconds=1))
'0.000001'
```

`python_utils.time.timeout_generator`(*timeout*: ~datetime.timedelta | int | float, *interval*: ~datetime.timedelta | int | float = datetime.timedelta(seconds=1), *iterable*: ~typing.Iterable[~python_utils.time._T] | ~typing.Callable[[], ~typing.Iterable[~python_utils.time._T]] = <class 'itertools.count'>, *interval_multiplier*: float = 1.0, *maximum_interval*: ~datetime.timedelta | int | float | None = None) → Iterable[_T]

Generator that walks through the given iterable (a counter by default) until the float_timeout is reached with a configurable float_interval between items.

This can be used to limit the time spent on a slow operation. This can be useful for testing slow APIs so you get a small sample of the data in a reasonable amount of time.

```
>>> for i in timeout_generator(0.1, 0.06):
...     # Put your slow code here
...     print(i)
0
1
2
>>> timeout = datetime.timedelta(seconds=0.1)
>>> interval = datetime.timedelta(seconds=0.06)
>>> for i in timeout_generator(timeout, interval, itertools.count()):
...     print(i)
0
1
2
>>> for i in timeout_generator(1, interval=0.1, iterable='ab'):
...     print(i)
a
b
```

```
>>> timeout = datetime.timedelta(seconds=0.1)
>>> interval = datetime.timedelta(seconds=0.06)
>>> for i in timeout_generator(timeout, interval, interval_multiplier=2):
...     print(i)
0
1
2
```

2.9 Module contents

This module initializes the `python_utils` package by importing various submodules and functions.

Submodules:

aio converters decorators formatters generators **import_** logger terminal time types

Functions:

account remap scale_1024 to_float to_int to_str to_unicode listify set_attributes raise_exception reraise camel_to_underscore timesince abatcher batcher import_global get_terminal_size aio_generator_timeout_detector aio_generator_timeout_detector_decorator aio_timeout_generator delta_to_seconds delta_to_seconds_or_none format_time timedelta_to_seconds timeout_generator

Classes:

CastedDict LazyCastedDict UniqueList Logged LoggerBase

```
class python_utils.CastedDict(key_cast: Callable[[...], KT] | None = None, value_cast: Callable[[...], VT] |
                             None = None, *args: Mapping[KT, VT] | Iterable[Tuple[KT, VT]] |
                             Iterable[Mapping[KT, VT]] | _typeshed.SupportsKeysAndGetItem[KT, VT],
                             **kwargs: VT)
```

Bases: CastedDictBase[KT, VT]

Custom dictionary that casts keys and values to the specified typing.

Note that you can specify the types for mypy and type hinting with: CastedDictint, int

```
>>> d: CastedDict[int, int] = CastedDict(int, int)
>>> d[1] = 2
>>> d['3'] = '4'
>>> d.update({'5': '6'})
>>> d.update([('7', '8')])
>>> d
{1: 2, 3: 4, 5: 6, 7: 8}
>>> list(d.keys())
[1, 3, 5, 7]
>>> list(d)
[1, 3, 5, 7]
>>> list(d.values())
[2, 4, 6, 8]
>>> list(d.items())
[(1, 2), (3, 4), (5, 6), (7, 8)]
>>> d[3]
4
```

```
# Casts are optional and can be disabled by passing None as the cast >>> d = CastedDict() >>> d[1] = 2 >>>
d['3'] = '4' >>> d.update({'5': '6'}) >>> d.update([('7', '8')]) >>> d {1: 2, '3': '4', '5': '6', '7': '8'}
```

```
class python_utils.LazyCastedDict(key_cast: Callable[[...], KT] | None = None, value_cast: Callable[[...],
VT] | None = None, *args: Mapping[KT, VT] | Iterable[Tuple[KT, VT]] |
| Iterable[Mapping[KT, VT]] | _typeshed.SupportsKeysAndGetItem[KT,
VT], **kwargs: VT)
```

Bases: CastedDictBase[KT, VT]

Custom dictionary that casts keys and lazily casts values to the specified typing. Note that the values are cast only when they are accessed and are not cached between executions.

Note that you can specify the types for mypy and type hinting with: LazyCastedDictint, int

```
>>> d: LazyCastedDict[int, int] = LazyCastedDict(int, int)
>>> d[1] = 2
>>> d['3'] = '4'
>>> d.update({'5': '6'})
>>> d.update([('7', '8')])
>>> d
{1: 2, 3: '4', 5: '6', 7: '8'}
>>> list(d.keys())
[1, 3, 5, 7]
>>> list(d)
[1, 3, 5, 7]
```

(continues on next page)

(continued from previous page)

```
>>> list(d.values())
[2, 4, 6, 8]
>>> list(d.items())
[(1, 2), (3, 4), (5, 6), (7, 8)]
>>> d[3]
4
```

```
# Casts are optional and can be disabled by passing None as the cast >>> d = LazyCastedDict() >>> d[1] = 2
>>> d['3'] = '4' >>> d.update({'5': '6'}) >>> d.update([('7', '8')]) >>> d {1: 2, '3': '4', '5': '6', '7': '8'} >>>
list(d.keys()) [1, '3', '5', '7'] >>> list(d.values()) [2, '4', '6', '8']
```

```
>>> list(d.items())
[(1, 2), ('3', '4'), ('5', '6'), ('7', '8')]
>>> d['3']
'4'
```

items() → Generator[Tuple[KT, VT], None, None]

Returns a generator of the dictionary's items, casting the values if a value cast callable is provided.

Yields:

types.Generator[types.Tuple[KT, VT], None, None]: A generator of the dictionary's items.

values() → Generator[VT, None, None]

Returns a generator of the dictionary's values, casting the values if a value cast callable is provided.

Yields:

types.Generator[VT, None, None]: A generator of the dictionary's values.

class python_utils.**Logged**(*args: Any, **kwargs: Any)

Bases: *LoggerBase*

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

```
>>> my_class._Logged__get_name('spam')
'spam'
```

logger: *Logger*

class python_utils.LoggerBase

Bases: ABC

Class which automatically adds logging utilities to your class when interiting. Expects *logger* to be a logging.Logger or compatible instance.

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

```
>>> class MyClass(LoggerBase):
...     logger = logging.getLogger(__name__)
...
...     def __init__(self):
...         Logged.__init__(self)
```

```
>>> my_class = MyClass()
>>> my_class.debug('debug')
>>> my_class.info('info')
>>> my_class.warning('warning')
>>> my_class.error('error')
>>> my_class.exception('exception')
>>> my_class.log(0, 'log')
```

classmethod critical(*msg: object, *args: object, exc_info: bool | Tuple[Type[BaseException], BaseException, TracebackType | None] | Tuple[None, None, None] | BaseException | None = None, stack_info: bool = False, stacklevel: int = 1, extra: Mapping[str, object] | None = None*) → None

classmethod debug(*msg: object, *args: object, exc_info: bool | Tuple[Type[BaseException], BaseException, TracebackType | None] | Tuple[None, None, None] | BaseException | None = None, stack_info: bool = False, stacklevel: int = 1, extra: Mapping[str, object] | None = None*) → None

classmethod error(*msg: object, *args: object, exc_info: bool | Tuple[Type[BaseException], BaseException, TracebackType | None] | Tuple[None, None, None] | BaseException | None = None, stack_info: bool = False, stacklevel: int = 1, extra: Mapping[str, object] | None = None*) → None

classmethod exception(*msg: object, *args: object, exc_info: bool | Tuple[Type[BaseException], BaseException, TracebackType | None] | Tuple[None, None, None] | BaseException | None = None, stack_info: bool = False, stacklevel: int = 1, extra: Mapping[str, object] | None = None*) → None

classmethod info(*msg: object, *args: object, exc_info: bool | Tuple[Type[BaseException], BaseException, TracebackType | None] | Tuple[None, None, None] | BaseException | None = None, stack_info: bool = False, stacklevel: int = 1, extra: Mapping[str, object] | None = None*) → None

classmethod log(*level: int, msg: object, *args: object, exc_info: bool | Tuple[Type[BaseException], BaseException, TracebackType | None] | Tuple[None, None, None] | BaseException | None = None, stack_info: bool = False, stacklevel: int = 1, extra: Mapping[str, object] | None = None*) → None

logger: Any

```

classmethod warning(msg: object, *args: object, exc_info: bool | Tuple[Type[BaseException],
    BaseException, TracebackType | None] | Tuple[None, None, None] | BaseException |
    None = None, stack_info: bool = False, stacklevel: int = 1, extra: Mapping[str,
    object] | None = None) → None

```

```

class python_utils.UniqueList(*args: HT, on_duplicate: Literal['ignore', 'raise'] = 'ignore')

```

Bases: `List[HT]`

A list that only allows unique values. Duplicate values are ignored by default, but can be configured to raise an exception instead.

```

>>> l = UniqueList(1, 2, 3)
>>> l.append(4)
>>> l.append(4)
>>> l.insert(0, 4)
>>> l.insert(0, 5)
>>> l[1] = 10
>>> l
[5, 10, 2, 3, 4]

```

```

>>> l = UniqueList(1, 2, 3, on_duplicate='raise')
>>> l.append(4)
>>> l.append(4)
Traceback (most recent call last):
...
ValueError: Duplicate value: 4
>>> l.insert(0, 4)
Traceback (most recent call last):
...
ValueError: Duplicate value: 4
>>> 4 in l
True
>>> l[0]
1
>>> l[1] = 4
Traceback (most recent call last):
...
ValueError: Duplicate value: 4

```

append(value: HT) → None

Appends a value to the list, ensuring uniqueness.

Args:

value (HT): The value to append.

Raises:

ValueError: If the value is a duplicate and `on_duplicate` is set to 'raise'.

insert(index: SupportsIndex, value: HT) → None

Inserts a value at the specified index, ensuring uniqueness.

Args:

index (types.SupportsIndex): The index to insert the value at. value (HT): The value to insert.

Raises:

ValueError: If the value is a duplicate and *on_duplicate* is set to 'raise'.

async python_utils.abatcher(*generator: AsyncGenerator[_T, None] | AsyncIterator[_T], batch_size: int | None = None, interval: timedelta | int | float | None = None*) → AsyncGenerator[List[_T], None]

Asyncio generator wrapper that returns items with a given batch size or interval (whichever is reached first).

Args:

generator: The async generator or iterator to batch. *batch_size* (*types.Optional[int]*, optional): The number of items per

batch. Defaults to None.

interval (*types.Optional[types.delta_type]*, optional): **The time**

interval to wait before yielding a batch. Defaults to None.

Yields:

types.AsyncGenerator[types.List[_T], None]: A generator that yields batches of items.

async python_utils.acount(*start: _N = 0, step: _N = 1, delay: float = 0, stop: _N | None = None*) → AsyncIterator[_N]

Asyncio version of `itertools.count()`.

async python_utils.aio_generator_timeout_detector(*generator: ~typing.AsyncGenerator[~python_utils.time._T, None], timeout: ~datetime.timedelta | int | float | None = None, total_timeout: ~datetime.timedelta | int | float | None = None, on_timeout: ~typing.Callable[[~typing.AsyncGenerator[~python_utils.time._T, None], ~datetime.timedelta | int | float | None, ~datetime.timedelta | int | float | None, BaseException], ~typing.Any] | None = <function reraise>, **on_timeout_kwargs: ~typing.Mapping[str, ~typing.Any]*) → AsyncGenerator[_T, None]

This function is used to detect if an asyncio generator has not yielded an element for a set amount of time.

The *on_timeout* argument is called with the *generator*, *timeout*, *total_timeout*, *exception* and the extra ***kwargs* to this function as arguments. If *on_timeout* is not specified, the exception is reraised. If *on_timeout* is *None*, the exception is silently ignored and the generator will finish as normal.

python_utils.aio_generator_timeout_detector_decorator(*timeout: ~datetime.timedelta | int | float | None = None, total_timeout: ~datetime.timedelta | int | float | None = None, on_timeout: ~typing.Callable[[~typing.AsyncGenerator[~typing.Any, None], ~datetime.timedelta | int | float | None, ~datetime.timedelta | int | float | None, BaseException], ~typing.Any] | None = <function reraise>, **on_timeout_kwargs: ~typing.Mapping[str, ~typing.Any]*) → Callable[[Callable[[_P], AsyncGenerator[_T, None]]], Callable[[_P], AsyncGenerator[_T, None]]]

A decorator wrapper for `aio_generator_timeout_detector`.

async python_utils.aio_timeout_generator(*timeout: ~datetime.timedelta | int | float, interval: ~datetime.timedelta | int | float = datetime.timedelta(seconds=1), iterable: ~typing.AsyncIterable[~python_utils.time._T] | ~typing.Callable[...], ~typing.AsyncIterable[~python_utils.time._T]] = <function account>, interval_multiplier: float = 1.0, maximum_interval: ~datetime.timedelta | int | float | None = None*) → AsyncGenerator[_T, None]

Async generator that walks through the given async iterable (a counter by default) until the float_timeout is reached with a configurable float_interval between items.

The interval_exponent automatically increases the float_timeout with each run. Note that if the float_interval is less than 1, 1/interval_exponent will be used so the float_interval is always growing. To double the float_interval with each run, specify 2.

Doctests and asyncio are not friends, so no examples. But this function is effectively the same as the *time-out_generator* but it uses *async for* instead.

python_utils.batcher(*iterable: Iterable[_T], batch_size: int = 10*) → Generator[List[_T], None, None]

Generator wrapper that returns items with a given batch size.

Args:

iterable (types.Iterable[_T]): The iterable to batch. *batch_size* (int, optional): The number of items per batch. Defaults

to 10.

Yields:

types.Generator[types.List[_T], None, None]: A generator that yields batches of items.

python_utils.camel_to_underscore(*name: str*) → str

Convert camel case style naming to underscore/snake case 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.delta_to_seconds(*interval: timedelta | int | float*) → int | float

Convert a timedelta to seconds.

```
>>> delta_to_seconds(datetime.timedelta(seconds=1))
1
>>> delta_to_seconds(datetime.timedelta(seconds=1, microseconds=1))
1.000001
>>> delta_to_seconds(1)
1
```

(continues on next page)

(continued from previous page)

```
>>> delta_to_seconds('whatever')
Traceback (most recent call last):
...
TypeError: Unknown type ...
```

`python_utils.delta_to_seconds_or_none(interval: timedelta | int | float | None) → int | float | None`

Convert a timedelta to seconds or return None.

`python_utils.format_time(timestamp: timedelta | date | datetime | str | int | float | None, precision: timedelta = datetime.timedelta(seconds=1)) → str`

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)
Traceback (most recent call last):
...
TypeError: Unknown type ...
```

`python_utils.get_terminal_size() → tuple[int, int]`

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

`python_utils.import_global(name: str, modules: ~typing.List[str] | None = None, exceptions: ~typing.Tuple[~typing.Type[Exception], ...] | ~typing.Type[Exception] = <class 'python_utils.import_.DummyError'>, locals_: ~typing.Dict[str, ~typing.Any] | None = None, globals_: ~typing.Dict[str, ~typing.Any] | None = None, level: int = -1) → Any`

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 exceptions (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

`python_utils.listify(collection: ~typing.Callable[~typing.Iterable[~python_utils.decorators._T]], ~typing.Collection[~python_utils.decorators._T]) = <class 'list'>, allow_empty: bool = True) → Callable[[Callable[[...], Iterable[_T] | None]], Callable[[...], Collection[_T]]]`

Convert any generator to a list or other type of collection.

```
>>> @listify()
... def generator():
...     yield 1
...     yield 2
...     yield 3
```

```
>>> generator()
[1, 2, 3]
```

```
>>> @listify()
... def empty_generator():
...     pass
```

```
>>> empty_generator()
[]
```

```
>>> @listify(allow_empty=False)
... def empty_generator_not_allowed():
...     pass
```

```
>>> empty_generator_not_allowed()
Traceback (most recent call last):
...
TypeError: ... `allow_empty` is `False`
```

```
>>> @listify(collection=set)
... def set_generator():
...     yield 1
...     yield 1
...     yield 2
```

```
>>> set_generator()
{1, 2}
```

```
>>> @listify(collection=dict)
... def dict_generator():
...     yield 'a', 1
...     yield 'b', 2
```

```
>>> dict_generator()
{'a': 1, 'b': 2}
```

`python_utils.raise_exception(exception_class: Type[Exception], *args: Any, **kwargs: Any) → Callable[[...], None]`

Returns a function that raises an exception of the given type with the given arguments.

```
>>> raise_exception(ValueError, 'spam')('eggs')
Traceback (most recent call last):
...
ValueError: spam
```

`python_utils.remap(value: _TN, old_min: _TN, old_max: _TN, new_min: _TN, new_max: _TN) → _TN`

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
>>> remap(decimal.Decimal('250.0'), 0.0, 1000.0, 0.0, 100.0)
Decimal('25.0')
```

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 equivalent in that dB range

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

I added using `decimal.Decimal` so floating point math errors can be avoided. Here is an example of a floating point math error `>>> 0.1 + 0.1 + 0.1 0.30000000000000004`

If floating point remaps need to be done my suggestion is to pass at least one parameter as a *decimal.Decimal*. This will ensure that the output from this function is accurate. I left passing *floats* for backwards compatibility and there is no conversion done from float to *decimal.Decimal* unless one of the passed parameters has a type of *decimal.Decimal*. This will ensure that any existing code that uses this function will work exactly how it has in the past.

Some edge cases to test `>>> remap(1, 0, 0, 1, 2)` Traceback (most recent call last): ... ValueError: Input range (0-0) is empty

```
>>> remap(1, 1, 2, 0, 0)
Traceback (most recent call last):
...
ValueError: Output range (0-0) is empty
```

Args:

`value` (int, float, `decimal.Decimal`): Value to be converted. `old_min` (int, float, `decimal.Decimal`): Minimum of the range for the

value that has been passed.

`old_max` (int, float, `decimal.Decimal`): Maximum of the range for the

value that has been passed.

`new_min` (int, float, `decimal.Decimal`): The minimum of the new range. `new_max` (int, float, `decimal.Decimal`): The maximum of the new range.

Returns: int, float, decimal.Decimal: Value that has been re-ranged. If

any of the parameters passed is a *decimal.Decimal*, all of the parameters will be converted to *decimal.Decimal*. The same thing also happens if one of the parameters is a *float*. Otherwise, all parameters will get converted into an *int*. Technically, you can pass a *str* of an integer and it will get converted. The returned value type will be *decimal.Decimal* if any of the passed parameters are *decimal.Decimal*, the return type will be *float* if any of the passed parameters are a *float*, otherwise the returned type will be *int*.

`python_utils.reraise(*args: Any, **kwargs: Any) → Any`

Reraises the current exception.

This function seems useless, but it can be useful when you need to pass a callable to another function that raises an exception.

`python_utils.scale_1024(x: int | float, n_prefixes: int) → Tuple[int | float, int | float]`

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.set_attributes(**kwargs: Any) → Callable[[...], Any]`

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'
```

`python_utils.timedelta_to_seconds(delta: timedelta) → int | float`

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'

```

`python_utils.timeout_generator`(*timeout*: ~datetime.timedelta | int | float, *interval*: ~datetime.timedelta | int | float = datetime.timedelta(seconds=1), *iterable*: ~typing.Iterable[~python_utils.time._T] | ~typing.Callable[[], ~typing.Iterable[~python_utils.time._T]] = <class 'itertools.count'>, *interval_multiplier*: float = 1.0, *maximum_interval*: ~datetime.timedelta | int | float | None = None) → Iterable[_T]

Generator that walks through the given iterable (a counter by default) until the float_timeout is reached with a configurable float_interval between items.

This can be used to limit the time spent on a slow operation. This can be useful for testing slow APIs so you get a small sample of the data in a reasonable amount of time.

```

>>> for i in timeout_generator(0.1, 0.06):
...     # Put your slow code here
...     print(i)
0
1
2
>>> timeout = datetime.timedelta(seconds=0.1)
>>> interval = datetime.timedelta(seconds=0.06)
>>> for i in timeout_generator(timeout, interval, itertools.count()):
...     print(i)
0
1
2
>>> for i in timeout_generator(1, interval=0.1, iterable='ab'):
...     print(i)
a
b

```

```

>>> timeout = datetime.timedelta(seconds=0.1)
>>> interval = datetime.timedelta(seconds=0.06)
>>> for i in timeout_generator(timeout, interval, interval_multiplier=2):
...     print(i)
0
1
2

```

`python_utils.timesince`(*dt*: datetime | timedelta, *default*: str = 'just now') → str

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

```

>>> now = datetime.datetime.now()
>>> timesince(now)

```

(continues on next page)

(continued from previous page)

```

'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))
'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'

```

`python_utils.to_float`(*input_*: str, *default*: int = 0, *exception*: ~typing.Tuple[~typing.Type[Exception], ...] | ~typing.Type[Exception] = (<class 'ValueError'>, <class 'TypeError'>), *regexp*: ~typing.Pattern[str] | str | ~typing.Literal[True] | None = None) → int | float

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

When trying to convert the exceptions given in the exception parameter are automatically caught 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')

```

(continues on next page)

(continued from previous page)

```

'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.to_int`(*input_*: *str* | *None* = *None*, *default*: *int* = 0, *exception*:
 ~typing.Tuple[~typing.Type[Exception], ...] | ~typing.Type[Exception] = (<class
 'ValueError'>, <class 'TypeError'>), *regexp*: ~typing.Pattern[*str*] | *str* |
 ~typing.Literal[True] | *None* = *None*) → *int*

Convert the given input to an integer or return default.

When trying to convert the exceptions given in the exception parameter are automatically caught 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')
1

```

(continues on next page)

(continued from previous page)

```

>>> to_int('')
0
>>> to_int()
0
>>> 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.to_str(input_: str | bytes, encoding: str = 'utf-8', errors: str = 'replace') → bytes`

Convert objects to string, encodes to the given encoding.

Return type

str

```

>>> to_str('a')
b'a'
>>> to_str('a')

```

(continues on next page)

(continued from previous page)

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

`python_utils.to_unicode(input_: str | bytes, encoding: str = 'utf-8', errors: str = 'replace') → str`

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

Return type

str

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


INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

- `python_utils`, 22
- `python_utils.converters`, 11
- `python_utils.decorators`, 9
- `python_utils.formatters`, 15
- `python_utils.import_`, 17
- `python_utils.logger`, 18
- `python_utils.terminal`, 19
- `python_utils.time`, 19

A

abatcher() (in module *python_utils*), 27
 account() (in module *python_utils*), 27
 aio_generator_timeout_detector() (in module *python_utils*), 27
 aio_generator_timeout_detector() (in module *python_utils.time*), 19
 aio_generator_timeout_detector_decorator() (in module *python_utils*), 27
 aio_generator_timeout_detector_decorator() (in module *python_utils.time*), 20
 aio_timeout_generator() (in module *python_utils*), 27
 aio_timeout_generator() (in module *python_utils.time*), 20
 append() (*python_utils.UniqueList* method), 26
 apply_recursive() (in module *python_utils.formatters*), 16

B

batcher() (in module *python_utils*), 28

C

camel_to_underscore() (in module *python_utils*), 28
 camel_to_underscore() (in module *python_utils.formatters*), 16
 CastedDict (class in *python_utils*), 23
 critical() (*python_utils.LoggerBase* class method), 25

D

debug() (*python_utils.LoggerBase* class method), 25
 delta_to_seconds() (in module *python_utils*), 28
 delta_to_seconds() (in module *python_utils.time*), 20
 delta_to_seconds_or_none() (in module *python_utils*), 29
 delta_to_seconds_or_none() (in module *python_utils.time*), 21
 DummyError, 17
 DummyException (in module *python_utils.import_*), 17

E

error() (*python_utils.LoggerBase* class method), 25

exception() (*python_utils.LoggerBase* class method), 25

F

format_time() (in module *python_utils*), 29
 format_time() (in module *python_utils.time*), 21

G

get_terminal_size() (in module *python_utils*), 29
 get_terminal_size() (in module *python_utils.terminal*), 19

I

import_global() (in module *python_utils*), 29
 import_global() (in module *python_utils.import_*), 18
 info() (*python_utils.LoggerBase* class method), 25
 insert() (*python_utils.UniqueList* method), 26
 items() (*python_utils.LazyCastedDict* method), 24

L

LazyCastedDict (class in *python_utils*), 23
 listify() (in module *python_utils*), 29
 listify() (in module *python_utils.decorators*), 9
 log() (*python_utils.LoggerBase* class method), 25
 Logged (class in *python_utils*), 24
 Logged (class in *python_utils.logger*), 18
 logger (*python_utils.Logged* attribute), 24
 logger (*python_utils.logger.Logged* attribute), 19
 logger (*python_utils.LoggerBase* attribute), 25
 LoggerBase (class in *python_utils*), 24

M

module
 python_utils, 22
 python_utils.converters, 11
 python_utils.decorators, 9
 python_utils.formatters, 15
 python_utils.import_, 17
 python_utils.logger, 18
 python_utils.terminal, 19
 python_utils.time, 19

P

`python_utils`
 module, 22
`python_utils.converters`
 module, 11
`python_utils.decorators`
 module, 9
`python_utils.formatters`
 module, 15
`python_utils.import_`
 module, 17
`python_utils.logger`
 module, 18
`python_utils.terminal`
 module, 19
`python_utils.time`
 module, 19

R

`raise_exception()` (in module `python_utils`), 30
`remap()` (in module `python_utils`), 31
`remap()` (in module `python_utils.converters`), 11
`reraise()` (in module `python_utils`), 32

S

`sample()` (in module `python_utils.decorators`), 10
`scale_1024()` (in module `python_utils`), 32
`scale_1024()` (in module `python_utils.converters`), 12
`set_attributes()` (in module `python_utils`), 32
`set_attributes()` (in module `python_utils.decorators`), 10

T

`timedelta_to_seconds()` (in module `python_utils`), 32
`timedelta_to_seconds()` (in module `python_utils.time`), 21
`timeout_generator()` (in module `python_utils`), 33
`timeout_generator()` (in module `python_utils.time`), 22
`timesince()` (in module `python_utils`), 33
`timesince()` (in module `python_utils.formatters`), 16
`to_float()` (in module `python_utils`), 34
`to_float()` (in module `python_utils.converters`), 12
`to_int()` (in module `python_utils`), 35
`to_int()` (in module `python_utils.converters`), 13
`to_str()` (in module `python_utils`), 36
`to_str()` (in module `python_utils.converters`), 15
`to_unicode()` (in module `python_utils`), 37
`to_unicode()` (in module `python_utils.converters`), 15

U

`UniqueList` (class in `python_utils`), 26

V

`values()` (`python_utils.LazyCastedDict` method), 24

W

`warning()` (`python_utils.LoggerBase` class method), 25
`wraps_classmethod()` (in module `python_utils.decorators`), 11