Pytoolkit

Dennis Muth

Feb 21, 2021

CONTENTS

CHAPTER

ONE

PURPOSE

Pretty much every utility stuff you could imagine is already written down in some python package. But almost every time these stuff is part of a heavyweight framework with a lots of dependencies. And on top they do it in a slightly different way - you would do it - because it is tailored to their needs.

This is why I made up this *Yet Another Utility Package* called *pytoolkit42*. I want this *toolkit* to be

- lightweight and
- easy to use

Right now it does not have the answer to every problem you will encounter, but it serves well to solve problems I stumble upon on a regular basis.

The collection of functions, classes, decorator, mixins will grow when I come up with a solution that seems to be of use for everybody else out there.

But to be honest: I try to make this package as generic as possible so it might be useful to you, but I cannot promise. Nevertheless you are encouraged to make PRs.

CHAPTER

TWO

INSTALLATION

Installation is done by pip. Simple and straightforward:

```
pip install pytoolkit42
```

CHAPTER

THREE

TOOLS

3.1 Basic

class pytoolkit.basics.**classproperty**(*caching: Union[Callable[[type], Any], bool] = False*)

Make class methods look like read-only class properties. Writing to that class-property will not do what you expect ;-)

If *caching* is set to *True* will only invoke the getter method once and cache the result. This makes sense if your property is computed once and after that never changed.

Examples

```
>>> class Foo:
...     _instance = 5
...     @classproperty
...     def foo(cls):
...         return cls._instance
...
...     @classproperty(caching=True)
...     def bar(cls):
...         return cls._instance
```

```
>>> Foo.foo, Foo.bar
(5, 5)
>>> Foo._instance = 15
>>> Foo.foo, Foo.bar  # Due to caching Foo.bar still returns 5
(15, 5)
```

```
>>> Foo.foo = 4242  # Setting the classproperty is not allowed
>>> Foo.foo, Foo.bar, Foo._instance
(4242, 5, 15)
```

pytoolkit.basics.**field_mro**(*clazz: Any*, *field: str*) → Set[str]

Goes up the mro (method resolution order) of the given class / instance and returns the union of values for a given class field.

Parameters

- **clazz** (*Any*) – The class to inspect.
- **field** (*str*) – The field to collect the values.

Returns Returns a compiled set of values for the given field for each class in the class hierarchy of the passed class or instance.

Return type Set[str]

Example

```
>>> class Root:
...     __myfield__ = 'root'
>>> class A(Root):
...     __myfield__ = ['a', 'common']
>>> class B(Root):
...     __myfield__ = ['b', 'common']
>>> class Final(A, B):
...     __myfield__ = 'final'
```

```
>>> field_mro(Final, '__myfield__') == {'root', 'a', 'b', 'common', 'final'}
True
>>> field_mro(A, '__myfield__') == {'root', 'a', 'common'}
True
>>> field_mro(Final(), '__myfield__') == {'root', 'a', 'b', 'common', 'final'}
True
```

3.2 Check

pytoolkit.check.**is_iterable_but_no_str**(*candidate: Any*) → bool

Checks if the given *candidate* is an *iterable* but not a *str* instance

Parameters **candidate** (*Any*) – The candidate to test.

Returns Returns *True* if the given *candidate* is an *iterable* but no *str*; otherwise *False*.

Return type bool

Example

```
>>> is_iterable_but_no_str(['a'])
True
>>> is_iterable_but_no_str('a')
False
>>> is_iterable_but_no_str(None)
False
```

pytoolkit.check.**is_real_float**(*candidate: Any*) → bool

Checks if the given *candidate* is a real *float*. An *integer* will return *False*.

Parameters **candidate** (*Any*) – The candidate to test.

Returns Returns *True* if the *candidate* is a real float; otherwise *False*.

Return type bool

Examples

```
>>> is_real_float(1.1)
True
>>> is_real_float(1.0)
False
>>> is_real_float(object())
False
>>> is_real_float(1)
False
>>> is_real_float("str")
False
```

3.3 Converter

pytoolkit.converter.**listify**(*item_or_items: Union[Any, Iterable[Any]]*) → Optional[List[Any]]

Makes a list out of the given item or items.

Parameters **item_or_items** (*Any*) – A single value or an iterable.

Returns Returns the given argument as a list. If the argument is already a list the identity will be returned unaltered.

Return type List[Any]

Examples

```
>>> listify(1)
[1]
>>> listify('str')
['str']
>>> listify(('i', 'am', 'a', 'tuple'))
['i', 'am', 'a', 'tuple']
>>> print(listify(None))
None
```

```
>>> # An instance of dict is used as is
>>> listify({'foo': 'bar'})
[{'foo': 'bar'}]
```

```
>>> # An instance of lists is unchanged
>>> l = ['i', 'am', 'a', 'list']
>>> l_res = listify(l)
>>> l_res
['i', 'am', 'a', 'list']
>>> l_res is l
True
```

pytoolkit.converter.**try_parse_bool**(*value: Any*, *default: Optional[bool] = None*) → Optional[bool]

Tries to parse the given value as a boolean. If the parsing is unsuccessful the default will be returned. A special case is *None*: It will always return the default value.

Parameters

- **value** (*Any*) – Value to parse.
- **default** (*bool, optional*) – The value to return in case the conversion is not successful.

Returns If the conversion is successful the converted representation of value; otherwise the default.

Return type (bool, optional)

Examples

```
>>> try_parse_bool(1)
True
>>> try_parse_bool('true')
True
>>> try_parse_bool('T')
True
>>> try_parse_bool('F')
False
>>> try_parse_bool(False)
False
>>> print(try_parse_bool('unknown', default=None))
None
>>> try_parse_bool(None, default=True)  # Special case
True
>>> try_parse_bool(1.0)
True
>>> try_parse_bool(0.99)
True
>>> try_parse_bool(0.0)
False
>>> try_parse_bool(lambda x: False, default=True)  # Will not be invoked
True
```

3.4 Environment

pytoolkit.env.**modify_environ**(**remove: str*, ***update: str*) → Iterator[None]

Temporarily updates the *os.environ* dictionary in-place and resets it to the original state when finished.

The *os.environ* dictionary is updated in-place so that the modification is sure to work in most situations.

Parameters

- **remove** (*str*) – Environment variables to remove from the environment scope.
- **update** (*str*) – Dictionary of environment variables and values to add if it does not exist or update it's value.

Examples

```
>>> import os
>>> os.environ['THIS_IS_SOME_DOCTEST'] = "42"
>>> print(os.environ['THIS_IS_SOME_DOCTEST'])
42
```

```
>>> with modify_environ("THIS_IS_SOME_DOCTEST", Test='abc'):
...     print(os.environ.get('Test'))
...     print(os.environ.get('THIS_IS_SOME_DOCTEST'))
abc
None
```

```
>>> print(os.environ.get('Test'))
None
>>> print(os.environ.get("THIS_IS_SOME_DOCTEST"))
42
```

3.5 Hashing

pytoolkit.hashing.**is_hashable**(*candidate: Any*) → bool

Determines whether the *candidate* can be hashed or not.

Parameters **candidate** (*Any*) – The candidate to test if it is hashable.

Returns *True* if *candidate* is hashable; otherwise *False*.

Return type bool

Examples

```
>>> is_hashable("i am")
True
>>> is_hashable({"I am": "not"})
False
```

pytoolkit.hashing.**make_hashable**(*obj: Any*) → Any

Converts a non-hashable instance into a hashable representation. Will take care of nested objects (like in iterables, dictionaries) as well. Will not detect a recursion and the function will fail in that case.

Parameters **obj** (*Any*) – The object to convert to a hashable object.

Returns Returns a hashable representation of the passed argument.

Return type Any

Examples

```
>>> make_hashable("unchanged")
'unchanged'
>>> make_hashable((1, 2, 3))
frozenset({1, 2, 3})
>>> make_hashable({1: {2: [3, 4, 5]}})
frozenset({(1, frozenset({(2, frozenset({3, 4, 5}))})),})
```

3.6 Mixins

class pytoolkit.mixins.**LogMixin**

Adds a logger property to the class to provide easy access to a configured logging instance to use.

Example

```
>>> class NeedsLogger(LogMixin):
...     def do(self, message):
...         self.logger.info(message)
>>> dut = NeedsLogger()
>>> dut.do('Instance logging')
>>> NeedsLogger.logger.info("Class logging")
```

class pytoolkit.mixins.**ReprMixin**

Adds a *__repr__* and a *__str__* method to the instance. You can control the fields to show via the *__REPR_FIELDS__* class field.

Examples

```
>>> class A(ReprMixin):
...     __REPR_FIELDS__ = ['a']
...     def __init__(self):
...         self.a = 13
```

```
>>> class B(A):
...     __REPR_FIELDS__ = 'b'
...     def __init__(self):
...         super().__init__()
...         self.b = 42
```

```
>>> repr(A())
'A(a=13)'
>>> repr(B())
'B(a=13, b=42)'
>>> repr(B()) == str(B())
True
```

3.7 Network

pytoolkit.net.**is_local**(*server: str*, *allow_ipv6: bool = False*) → bool

Checks if the given server (name or ip address) is actually a local one.

Parameters

- **server** (*str*) – The server name or ip to check.
- **allow_ipv6** (*bool*) – If True the local ipv6 ip address is checked too.

Returns Returns *True* if the given server is local; otherwise *False*.

Return type bool

Examples

```
>>> is_local('www.google.de')
False
>>> is_local('LOCALHOST')
True
>>> is_local('127.0.0.1')
True
>>> is_local('0.0.0.0')
True
>>> is_local('::1')
False
>>> is_local('::1', allow_ipv6=True)
True
```

3.8 Transform

pytoolkit.transform.**bps_mbps**(*val: float*) → float

Converts bits per second (bps) into megabits per second (mbps).

Parameters **val** (*float*) – The value in bits per second to convert.

Returns Returns val in megabits per second.

Return type float

Examples

```
>>> bps_mbps(1000000)
1.0
>>> bps_mbps(1129000)
1.13
```

pytoolkit.transform.**camel_to_snake**(*camel_str: str*) → str

Converts camelCase to snake_case.

https://stackoverflow.com/questions/1175208/elegant-python-function-to-convert-camelcase-to-snake-case

Parameters **camel_str** (*str*) – The camelCase string to convert to snake_case.

Returns Returns the snake_case representation of the passed camelCase string.

Return type str

```
>>> camel_to_snake('CamelCase')
'camel_case'
>>> camel_to_snake('CamelCamelCase')
'camel_camel_case'
>>> camel_to_snake('Camel2Camel2Case')
'camel2_camel2_case'
>>> camel_to_snake('getHTTPResponseCode')
'get_http_response_code'
>>> camel_to_snake('get2HTTPResponseCode')
'get2_http_response_code'
>>> camel_to_snake('HTTPResponseCode')
'http_response_code'
>>> camel_to_snake('HTTPResponseCodeXYZ')
'http_response_code_xyz'
```

pytoolkit.transform.**transform_dict**(*dct: Dict[Any, Any]*, *key_fun: Optional[Callable[[Any], Any]] = None*, *val_fun: Optional[Callable[[Any], Any]] = None*, *recursive: bool = False*) → Dict[Any, Any]

Transforms keys and/or values of the given dictionary by applying the given functions.

Parameters

- **dct** (*dict*) – The dictionary to transform.
- **key_fun** (*TransformDictFun*) – The function to apply to all dictionary keys. If not passed the keys will be unaltered.
- **val_fun** (*TransformDictFun*) – The function to apply to all dictionary values. If not passed the values will be unaltered.
- **recursive** (*bool*) – If True will recursively go down any encountered dict; otherwise will only transform the first level of the dict.

Returns Returns a new dictionary by applying the key and/or value function to the given dictionary. If both transformation functions are not supplied the passed dictionary will be returned unaltered.

Return type dict

Examples

```
>>> dct = {"CamelCase": "gnaaa", "foo_oool": 42}
>>> (transform_dict(dct, key_fun=camel_to_snake) ==
...     {"camel_case": "gnaaa", "foo_oool": 42})
True
```

```
>>> transform_dict(dct, val_fun=str) == {"CamelCase": "gnaaa", "foo_oool": "42"}
True
```

```
>>> (transform_dict(dct, key_fun=camel_to_snake, val_fun=str) ==
...     {"camel_case": "gnaaa", "foo_oool": "42"})
True
```

```
>>> res = transform_dict(dct, None, None)
>>> print(res)
{'CamelCase': 'gnaaa', 'foo_oool': 42}
>>> res is dct
True
```

```
>>> dct_ = {1: {11: 'snakeCase', 12: 'snake_case'}, 2: 22}
>>> (transform_dict(dct_, str, camel_to_snake, True) ==
...     {'1': {'11': 'snake_case', '12': 'snake_case'}, '2': '22'})
True
```

CHAPTER

FOUR

CHANGELOG

0.1.2

- Implements a *recursive* flag for *transform.transform_dict* to go down and transform an encountered dict as well

0.1.1

- Documentation fixes

0.1.0

- First version

INDEX