mtoolbox Documentation

Release 0.1.6

Maik Messerschmidt

Aug 13, 2017

Contents

1 README **3**
- 1.1 Installation . . . 3
- 1.2 Documentation . . . 3
- 1.3 License . . . 3

2 Metaprogramming **5**
- 2.1 Autoname . . . 5
- 2.2 ILists . . . 8
- 2.3 Instance Logging . . . 10

3 License **13**

4 Indices and tables **15**

Python Module Index **17**

A collection of (meta-)programming tools for Python.

Contents:

CHAPTER 1

README

mtoolbox - A collection of (meta-)programming tools for Python.

<https://github.com/messersm/mtoolbox>

Installation

mtoolbox is available in the Python Package Index (PyPi). Simply install it with: `pip install mtoolbox`

You can also download the package directly from:

<https://pypi.python.org/pypi/mtoolbox>

Documentation

Documentation is available at: <http://mtoolbox.readthedocs.io/en/latest/>.

License

CHAPTER 2

Metaprogramming

Autoname

Access an object's name as a property

Autoname is a data-descriptor, which automatically looks up the name under which the object on which the descriptor is accessed is known by.

Import the descriptor using `from mtoolbox.autoname import Autoname`.

Example:

```
>>> class Object(object):
...     name = Autoname()
>>> obj1 = Object()
>>> obj1.name
'obj1'
>>> obj2 = Object()
>>> obj2.name
'obj2'
```

By default Autoname will return the outer-most name that was defined for the object:

```
>>> class Object(object):
...     name = Autoname()
>>> def func(anobject):
...     return anobject.name
>>> o = Object()
>>> func(o)
'o'
```

You can change this behaviour by using the 'inner' keyword:

```
>>> class Object(object):
...     name = Autoname(inner=True)
>>> o = Object()
```

```
>>> def func(anobject):
...     return anobject.name
>>> func(o)
'anobject'
```

Note: Please be aware, that getting the inner-most name, is not what you want in most cases:

```
>>> class Object(object):
...     name = Autoname(inner=True)
...     def printname(self):
...         print(self.name)
>>> o = Object()
>>> o.printname()
self
```

When in automatic mode (see the class documentation below) the descriptor will always return a name, that is in some callframe dictionary. If you delete a name, it will use another one, that is still in use:

```
>>> class Object(object):
...     name = Autoname()
>>> o = Object()
>>> o.name
'o'
>>> g = o
>>> del o
>>> g.name
'g'
```

This can be helped a bit by using the 'bind' keyword argument and calling <object>.name with the name that should be used first:

```
>>> class Object(object):
...     name = Autoname(bind=True)
>>> o = Object()
>>> o.name
'o'
>>> g = o
>>> del o
>>> g.name
'o'
```

Warning: Defining multiple names for an object in the same call frame (which is easily said the same level of indention in your program) will cause undetermined behaviour, depending on the Python interpreter:

```
>>> class Object(object):
...     name = Autoname()
>>> o = Object()
>>> g = o
>>> o.name in ['o', 'g']
True
```

class mtoolbox.autoname.**Autoname**(*initval=True*, *inner=False*, *bind=False*)

Bases: object

Create a new Autoname descriptor

Parameters

- **initval** (*str, bool, None*) – The initial name
- **inner** (*bool*) – Return the inner-most name of the object (or not)
- **bind** (*bool*) – Bind the descriptor to the first name it returns

Returns An Autoname instance

Return type *Autoname*

__get__(*theobject*, *objtype*)

Return the name of theobject or None

Returns the name of the object

Return type str or None

Usage:

```
>>> class Object(object):
...     name = Autoname()
>>> obj = Object()
>>> obj.name
'obj'
>>> obj.name = 'another name'
>>> obj.name
'another name'
```

__set__(*theobject*, *val*)

Set the name of the theobject

Parameters

- **theobject** (*object*) – The object to which's class the descriptor is attached to
- **val** (*str, bool or None*) – Sets the name to depending on the type: str sets the name to this str. False or None sets the name to None. True sets the name to automatically lookup.

Returns None

Raises TypeError if type(val) is invalid

Usage:

```
>>> class Object(object):
...     name = Autoname()
>>> o = Object()
>>> o.name = 'k'
>>> o.name
'k'
>>> o.name = True
>>> o.name
'o'
>>> o.name = False
>>> str(o.name)
'None'
>>> o.name = 4
```

```
Traceback (most recent call last):
...
TypeError: Autoname must be set to str, bool, NoneType
```

ILists

Module to provide an 'intelligent' list class

The IList class translates attribute access to the items it holds:

`l.<name> == IList([obj.<name> for obj in l])`

Import the IList class using `from mtoolbox.ilist import IList`.

Example

```
>>> l = IList([complex(3, 4), complex(6)])
>>> l.real
[3.0, 6.0]
```

You can also use callable attributes of your objects:

```
>>> l = IList([complex(3, 4), complex(6)])
>>> l
[(3+4j), (6+0j)]
>>> l.conjugate()
[(3-4j), (6-0j)]
```

You can add callbacks, for appending and removing objects. These callbacks must accept two positional arguments - the list and the object. The callbacks are called _after_ executing append or remove:

```
>>> def on_append(l, x):
...     print("Adding %s to %s." % (x, l))
>>> def on_remove(l, x):
...     print("Removing %s from %s." % (x, l))
>>> l = IList(on_append=on_append, on_remove=on_remove)
>>> l.append(3)
Adding 3 to [3].
>>> l.remove(3)
Removing 3 from [].
>>> def invalid_callback(l):
...     print(l)
>>> l = IList(on_append=invalid_callback)
Traceback (most recent call last):
...
TypeError: on_append and on_remove must accept 2 positional arguments
>>> l = IList(on_append=3)
Traceback (most recent call last):
...
TypeError: on_append and on_remove must accept 2 positional arguments
```

```
>>> def valid_callback(l=[], x=5):
...     pass
>>> def valid_callback2(l, x=5, y=3):
```

```
...     pass
>>> l = IList(on_append=valid_callback, on_remove=valid_callback2)
```

Be aware, that only attribute names, that are not used by the list class are overwritten, so if list implemented a attribute name, you can't use it in this way. The following code doesn't work, because list implements '__add__' (so the result is NOT [4, 5] as one could expect):

```
>>> l = IList([1, 2])
>>> l + 3
Traceback (most recent call last):
...
TypeError: can only concatenate list (not "int") to list
```

If you wish to access attributes with these names, you can use *IList.getattr()* (see method documentation).

You can also apply any function to the items of an IList by calling *IList.apply()* (see method documentation).

class mtoolbox.ilist.**IList**(*iterable=None*, *on_append=None*, *on_remove=None*)

Bases: list

'intelligent' list object

Parameters

- **iterable** (*iterable*) – The
- **on_append** (*callable*) – callback(list, item) for append()
- **on_remove** (*callable*) – callback(list, item) for remove()

Returns An *IList* instance

Return type *IList*

Note: Both callbacks must accept two positional arguments

---

append(*obj*)

Add obj to IList

Parameters **obj** (*object*) – object to append to list

Returns None

Usage:

```
>>> l = IList()
>>> l
[]
>>> l.append(3)
>>> l
[3]
```

apply(*func*, **args*, ***kwargs*)

Apply func to the items of this IList

Parameters

- **func** (*callable*) – function to apply to this IList's items
- **args** (*iterable*) – additional arguments for func

- **kwargs** (*dict*) – additional keyword arguments for func

Returns An *IList* instance

Return type *IList*

Usage:

```
>>> def f(x, pow=2):
...     return x**pow
>>> l = IList([0, 1, 2, 3, 4, 5])
>>> l.apply(f, pow=3)
[0, 1, 8, 27, 64, 125]
```

getattr(*name*)

Parameters **name** (*str*) – name of the items attributes to access

Returns An IList instance

Return type *IList*

Usage:

```
>>> l = IList([3, 5, 4])
>>> l.getattr('__add__')(2)
[5, 7, 6]
```

remove(*obj*)

Remove obj from IList

Parameters **obj** (*object*) – object to remove from list

Returns None

Usage:

```
>>> l = IList([8])
>>> l
[8]
>>> l.remove(8)
>>> l
[]
>>> l.remove(6)
Traceback (most recent call last):
...
ValueError: list.remove(x): x not in list
```

Instance Logging

log object instantiation of (almost) all python classes

Import the module using `from mtoolbox import instancelog`

Note: You have to run `enable()` BEFORE importing the module, which has classes you would like to log. The reason for this is, that the name `object` from the `__builtin__` namespace has to point to the object class overwrite of the instancelog module, when a new class is defined. For the same reason builtin objects will never be logged.

Example:

In this example `MyClass1` objects will not be logged, while `MyClass2` objects will be logged:

```
>>> from . import instancelog
>>> class MyClass1(object):
...     def __repr__(self):
...         return 'MyClass1 object'
>>> instancelog.enable()
>>> class MyClass2(object):
...     def __repr__(self):
...         return 'MyClass2 object'
>>> objlist = []
>>> def my_callback(obj, cls, args, kwargs):
...   objlist.append(obj)
>>> instancelog.callbacks.append(my_callback)
>>> obj1 = MyClass1()
>>> obj2 = MyClass2()
>>> print(objlist)
[MyClass2 object]
```

class `mtoolbox.instancelog.`**`Object`**

Bases: `object`

Class to replace (builtin) object

`mtoolbox.instancelog.`**`disable`**`()`

Disable the logging of objects.

`mtoolbox.instancelog.`**`enable`**`()`

Enable the logging of objects.

CHAPTER 3

License

```
MIT License

Copyright (c) 2016-2017 Maik Messerschmidt

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

mtoolbox.autoname, 5
mtoolbox.ilist, 8
mtoolbox.instancelog, 10

Index

Symbols

__get__() (mtoolbox.autoname.Autoname method), 7
__set__() (mtoolbox.autoname.Autoname method), 7

A

append() (mtoolbox.ilist.IList method), 9
apply() (mtoolbox.ilist.IList method), 9
Autoname (class in mtoolbox.autoname), 6

D

disable() (in module mtoolbox.instancelog), 11

E

enable() (in module mtoolbox.instancelog), 11

G

getattr() (mtoolbox.ilist.IList method), 10

I

IList (class in mtoolbox.ilist), 9

M

mtoolbox.autoname (module), 5
mtoolbox.ilist (module), 8
mtoolbox.instancelog (module), 10

O

Object (class in mtoolbox.instancelog), 11

R

remove() (mtoolbox.ilist.IList method), 10