FQN Decorators Documentation

Mattias Sluis

Dec 07, 2018

Contents

Contents:

CHAPTER 1

Installation

At the command line:

```
$ pip install fqn-decorators
```

CHAPTER 2

Usage

Topics

2.1 Advanced Usage

Note: It is possible to decorate static and class methods but you have to ensure that the order of decorators is right. The *@staticmethod* and *@classmethod* decorators should always be on top.:

```
@staticmethod
@my_decorator
def my_static_method():
    pass
```

Warning: The fully qualified name of a method cannot be properly determined for static methods and class methods.

Warning: The fully qualified name of a method or function cannot be properly determined in case they are already decorated. This only applies to decorators that aren't using *Decorator*

Decorators can be used in three different ways.:

```
@my_decorator
@my_decorator()
@my_decorator(my_argument=True)
def my_function():
   pass
```

Decorators can be used on all callables so you can decorator functions, (new style) classes and methods.:

```
@my_decorator
def my_function():
    pass

@my_decorator
class MyClass(object):
    @my_decorator
    def my_method():
        pass
```

2.1.1 Combining decorators

Combining decorators is as simple as just stacking them on a function definition.

Important: The *before()* and *after()* methods of the decorators are in different orders. In the example below the *before()* methods of step2 and step1 are executed and then the method itself. The *after()* method is called for step1 and then step2 after the method is executed. So the call stack becomes

- step2.before()
- step1.before()
- my_process()
- step1.after()
- step2.after()

```
@step2
@step1
def my_process():
    pass
```

If you want to create a decorator that combines decorators you can do that like this:

```
class process(decorators.Decorator):
    """Combines step1 and step2 in a single decorator"""

    def before(self):
        self.func = step2(step1(self.func))
```

2.1.2 Non-keyworded decorators

Although not supported out of the box, it is possible to create decorators with non-keyworded or positional arguments:

```
import fqn_decorators

class arg_decorator(fqn_decorators.Decorator):

    def __init__(self, func=None, *args, **kwargs):
        self._args = args
        super(arg_decorator, self).__init__(func, **kwargs)

    def __call__(self, *args, **kwargs):
        if not self.func:
            # Decorator initialized without providing the function
            return self.__class__(args[0], *self._args, **self.params)
        return super(arg_decorator, self).__call__(*args, **kwargs)

    def __get__(self, obj, type=None):
        return self.__class__(self.func.__get__(obj, type), *self._args, **self.
↪params)

    def before(self):
        print self._args


@arg_decorator(None, 1, 2)
def my_function():
    pass

>>>my_function()
(1, 2)
```

2.2 Introduction

By extending the *Decorator* class you can create simple decorators. Implement the *before()* and/or *after()* methods to perform actions before or after execution of the decorated item. The *before()* method can access the arguments of the decorated item by changing the *args* and *kwargs* attributes. The *after()* method can access or change the result using the *result* attribute. The *exception()* method can be used for do something with an Exception that has been raised. In all three methods the *fqn* and *func* attributes are available.

2.3 Simple decorator

Create a simple decorator:

```
import fqn_decorators
import time

class time_it(fqn_decorators.Decorator):

    def before(self):
        self.start = time.time()
```

(continues on next page)

(continued from previous page)

```
    def after(self):
        duration = time.time() - self.start
        print("{0} took {1} seconds".format(self.fqn, duration))


@time_it
def my_function():
    time.sleep(1)

>>>my_function()
__main__.my_function took 1.00293397903 seconds
```

2.4 Decorator with arguments

It is also very easy to create a decorator with arguments.

Note: It is not possible to create decorators with *non-keyworded* arguments. To create a decorator that supports non-keyworded arguments see the *Advanced Usage* section.

Example:

```
import fqn_decorators
import time

class threshold(fqn_decorators.Decorator):

    def before(self):
        self.start = time.time()

    def after(self):
        duration = time.time() - self.start
        treshold = self.params.get('threshold')
        if threshold and duration > threshold:
            raise Exception('Execution took longer than the threshold')


@threshold(threshold=2)
def my_function():
    time.sleep(3)

>>> my_function()
Exception: Execution took longer than the threshold
```

2.5 Async Decorator

There's also support for decorating coroutines (or any awaitable), for Python >=3.5 only.

The implementation is the same as with the sync version, just inherit from `AsyncDecorator` instead.

Example:

```
import asyncio
import time
from fqn_decorators.async import AsyncDecorator

class time_it_async(AsyncDecorator):

    def before(self):
        self.start = time.time()

    def after(self):
        duration = time.time() - self.start
        print("{0} took {1} seconds".format(self.fqn, duration))

@time_it_async
async def coro():
    await asyncio.sleep(1)

>>> loop = asyncio.get_event_loop()
>>> loop.run_until_complete(coro())
__main__.coro took 1.001493215560913 seconds
```

CHAPTER 3

API

Contents:

3.1 fqn_decorators

3.1.1 fqn_decorators.async module

3.1.2 fqn_decorators.decorators module

class fqn_decorators.decorators.**ChainedDecorator**(*func=None*, *decorators=None*, ***params*)

Bases: *fqn_decorators.decorators.Decorator*

Simple decorator which allows you to combine regular decorators and decorators based on Decorator. It will preserve the FQN for those based on Decorator.

before()

Allow performing an action before the function is called.

class fqn_decorators.decorators.**Decorator**(*func=None*, ***params*)

Bases: object

A base class to easily create decorators.

after()

Allow performing an action after the function is called.

args = None

The non-keyworded arguments with which the callable will be called

before()

Allow performing an action before the function is called.

exc_info = None

Exception information in case of an exception

exception()
Allow exception processing (note that the exception will still be raised after processing.

fqn = None
The fully qualified name of the callable

func = None
The callable that is being decorated

get_fqn()
Allow overriding the fqn and also change functionality to determine fqn.

kwargs = None
The keyword arguments with which the callable will be called

params = None
The keyword arguments provided to the decorator on init

result = None
The result of the execution of the callable

fqn_decorators.decorators.**chained_decorator**
alias of *fqn_decorators.decorators.ChainedDecorator*

fqn_decorators.decorators.**get_fqn**(*obj*)
This function tries to determine the fully qualified name (FQN) of the callable that is provided. It only works for classes, methods and functions. It is unable to properly determine the FQN of class instances, static methods and class methods.

CHAPTER 4

License

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

f

Index