# DecMore

*Release 1.1.2*

**Daniel Trivelli**

**Mar 01, 2024**

# CONTENTS

CHAPTER

ONE

# DECORATORS THAT WILL BE USEFUL FOR BOTH DAILY DEVELOPMENT AND APPLICATION NEEDS.

CHAPTER

# TWO

# CONTENTS

## 2.1 Getting started

### 2.1.1 Requirements

- Python (3.7, 3.8, 3.9, 3.10)

### 2.1.2 Installation

DecMore can be installed with pip:

```
pip install decmore
```

### 2.1.3 Usage

To use the decorators, simply import the library into the desired `.py` file and place them in the functions or classes:

```python
from decmore import CheckTypes

@CheckTypes()
def test_function(var1: str, var2: list, var: int):
    ...
```

## 2.2 Base

The Base decorator is only meant to help other useful decorators, so it should not be used as a decorator for your code. If you want to understand more about it locate the file `default.py` With its implementation we can tell the decorators what properties, features and changes they should support

```python
class BaseDecorator(object):
    allowed_params = []
    allowed_methods = []
    disallowed_methods = []
    class_injection = True
    instance: Callable | None = None
    _traced_methods = {}
    is_class = False
```

### 2.2.1 `allowed_params`

This property, which is in WIP, will allow you to restrict the parameters passed to each decorator

### 2.2.2 `allowed_methods`

This property is sent directly by the user when instantiating a decorator that can be used in classes that tells which methods of that class can be overridden to run with the decorator It is an `empty list` by default.

### 2.2.3 `disallowed_methods`

This property is sent directly by the user when instantiating a decorator that can be used in classes that tells which methods of that class cannot be overridden to be executed with the decorator It is an empty list by default and is changed when the Base decorator overwrites functions. Here are the functions that are included in this property:

- __class__
- __delattr__
- __dict__
- __dir__
- __doc__
- __eq__
- __format__
- __ge__
- __getattribute__
- __gt__
- __hash__
- __init__
- __init_subclass__
- __le__
- __lt__
- __module__
- __ne__
- __new__
- __reduce__
- __reduce_ex__
- __repr__
- __setattr__
- __sizeof__
- __str__
- __subclasshook__

- __weakref__
- __getstate__

If you want some to be overwritten, add them to the `allowed_methods` property

### 2.2.4 `class_injection`

This property is sent through the child decorator to say whether it can be placed under a class, thus injecting a radar function that overrides the other functions of that class so that we can trigger the decorator in each of them. This property is `True` by default.

### 2.2.5 `instance`

This property is only for controlling the actions of each decorator.

### 2.2.6 `_traced_methods`

This property helps the Base decorator in injecting the radar function.

### 2.2.7 `is_class`

This property is changed when the Base decorator analyzes the object to be updated and allows the injection of the radar function to happen automatically

## 2.3 CheckTypes

The CheckTypes decorator helps the developer to make sure that the variables passed to the function conform to the required types. This decorator only works on functions, when instantiated under a class it will return an error saying that it supports functions and that it can be instantiated in static functions inside the class

### 2.3.1 Usage

```python
from decmore import CheckTypes

@CheckTypes()
def func(var1: str, var2: list, var3: list | tuple):
    ...


class klass:
    def __init__(self):
        ...

    @CheckTypes()
    @staticmethod
    def static(var1: list | tuple):
        ...
```

## 2.4 Profiler

The Profiler decorator helps the developer to analyze the performance of his code by showing on the console, in order of time, which functions and lines took longer to execute. Accepts to be instantiated in classes and can receive the `allowed_methods` and `disallowed_methods` parameters.

```
from time import sleep
from decmore import Profiler


@Profiler()
def func():
    sleep(10)


@Profiler(allowed_methods=['__init__'], disallowed_methods=['post'])
class klass:
    def __init__(self):
        sleep(1)

    def get(self):
        ...

    def post(self):
        ...
```

## 2.5 ToThreads

The ToThreads decorator divides up the work for a set number of threads when it is instantiated. Receive `amount` and `return_expected` parameters. This decorator only works on functions, when instantiated under a class it will return an error saying that it supports functions and that it can be instantiated in static functions inside the class

Parameters:

- **amount:**
  Number of threads that will be created and executed
- **return_expected:**
  If the function returns something, this parameter should be changed to True, since the default value is False.

```
from time import sleep
from decmore import ToThreads


@ToThreads(amount=2, return_expected=True)
def test_threads(v):
    return [x * 10 for x in v]
```

## 2.6 Cache

The Cache decorator saves the return of the function so that it is not executed if the parameters result in the same return. This decorator only works on functions, when instantiated under a class it will return an error saying that it supports functions and that it can be instantiated in static functions inside the class

```
from time import sleep
from decmore import ToThreads


@Cache()
def test_threads(v):
    return [x * 10 for x in v]
```

CHAPTER

# THREE

## INDICES AND TABLES

- genindex
- modindex