

---

# **decoutils Documentation**

***Release 0.0.4***

**Liyu Gong**

**Jan 04, 2019**



---

## Contents:

---

|          |                            |           |
|----------|----------------------------|-----------|
| <b>1</b> | <b>Introduction</b>        | <b>1</b>  |
| 1.1      | Rationale . . . . .        | 1         |
| 1.2      | Installation . . . . .     | 2         |
| 1.3      | Simple example . . . . .   | 2         |
| 1.4      | License . . . . .          | 2         |
| 1.5      | Documentation . . . . .    | 3         |
| <b>2</b> | <b>API Reference</b>       | <b>5</b>  |
| <b>3</b> | <b>ChangeLog</b>           | <b>7</b>  |
| 3.1      | 0.0.3 . . . . .            | 7         |
| 3.2      | 0.0.2 . . . . .            | 7         |
| 3.3      | 0.0.1 . . . . .            | 7         |
| <b>4</b> | <b>Indices and tables</b>  | <b>9</b>  |
|          | <b>Python Module Index</b> | <b>11</b> |



*decouils*: Utilities for writing decorators

## 1.1 Rationale

Python *decorators* are very useful. However, writing decorators with arguments is not straightforward. Take the following function as an example:

```
register_plugin(plugin, arg1=1):  
    print('registering ', plugin.__name__, ' with arg1=', arg1)
```

This function registers a input function somewhere in the system. If this function could work as a decorator, that will be convinient. In order to make it a decorator, we just need it to return the input plugin trivially:

```
register_plugin(plugin, arg1=1):  
    print('registering ', plugin.__name__, ' with arg1=', arg1)  
    return plugin  
  
@register_plugin  
def plugin1(): pass
```

It is pretty easy so far. What if we want *register\_plugin* works as a decorator with arguments? That's to say, we want to use it as:

```
@register_plugin(arg1=2)  
def plugin2(): pass
```

In order to accomplish this goal, we need to wrap the function *register\_plugin* so that it return a decorated plugin if the first input is a callable object, otherwise return a decorator with arguments. The *decouils.decorator\_with\_args* is intend to abstract that wrapping, so that we can reuse it.

## 1.2 Installation

### 1.2.1 Install from PyPI

```
pip install decoutils
```

### 1.2.2 Install from Anaconda

```
conda install -c liyugong decoutils
```

## 1.3 Simple example

Basically, *decorator\_with\_args* enables a ordinary decorator function to be a decorator with arguments.

```
@decorator_with_args
def register_plugin(plugin, arg1=1):
    print('registering ', plugin.__name__, ' with arg1=', arg1)
    return plugin

@register_plugin(arg1=10)
def plugin1(): pass
```

Moreover, *decorator\_with\_args* itself is also a decorator with arguments: one argument *return\_original* which can be used to convert a non-decorator function to be a decorator

```
@decorator_with_args
def register_plugin(plugin, arg1=1):
    print('registering ', plugin.__name__, ' with arg1=', arg1)
    # Note here the function does not return the plugin, so it cannot work as a
    ↪ decorator originally

@register_plugin(arg1=10)
def plugin1(): pass
```

*decorator\_with\_args* can also convert a function to decorator whose decorating target is not its first argument, e.g.

```
decorator_with_args(target_pos=1)
def register_plugin(arg1, plugin):
    return plugin

@register_plugin(100) # plugin2 will be registered with arg1=100
def plugin2(): pass
```

*return\_original* control whether the resultant decorator return the original plugin, or the result of function *register\_plugin*.

## 1.4 License

The *decoutils* package is released under the [MIT License](#)

## 1.5 Documentation

<https://decouils.readthedocs.io>



`decoutils.decorator_with_args(*args, **kwargs)`

Enable a function to work with a decorator with arguments

### Parameters

- **func** (*callable*) – The input function.
- **return\_original** (*bool*) – Whether the resultant decorator returns the decorating target unchanged. If True, will return the target unchanged. Otherwise, return the returned value from *func*. Default to False. This is useful for converting a non-decorator function to a decorator. See examples below.

**Returns** a decorator with arguments.

**Return type** callable

Examples:

```
>>> @decorator_with_args
... def register_plugin(plugin, arg1=1):
...     print('Registering '+plugin.__name__+' with arg1='+str(arg1))
...     return plugin # note register_plugin is an ordinary decorator
>>> @register_plugin(arg1=10)
... def plugin1(): pass
Registering plugin1 with arg1=10
```

```
>>> @decorator_with_args(return_original=True)
... def register_plugin_xx(plugin, arg1=1):
...     print('Registering '+plugin.__name__+' with arg1='+str(arg1))
...     # Note register_plugin_xxx does not return plugin, so it cannot
...     # be used as a decorator directly before applying
...     # decorator_with_args.
>>> @register_plugin_xx(arg1=10)
... def plugin1(): pass
Registering plugin1 with arg1=10
>>> plugin1()
```

```
>>> @decorator_with_args(return_original=True)
... def register_plugin_xxx(plugin, arg1=1): pass
```

```
>>> # use result decorator as a function
>>> register_plugin_xxx(plugin=plugin1, arg1=10)
<function plugin1...>
```

```
>>> @decorator_with_args(return_original=True, target_pos=1)
... def register_plugin_xxxx(arg1, plugin, arg2=10):
...     print('Registering '+plugin.__name__+' with arg1='+str(arg1))
>>> @register_plugin_xxxx(100)
... def plugin2(): pass
Registering plugin2 with arg1=100
```

## CHAPTER 3

---

### ChangeLog

---

#### 3.1 0.0.3

- Add support for *target\_pos*

#### 3.2 0.0.2

Fix small problems in README.md

#### 3.3 0.0.1

First release, contains *decorator\_with\_args()* only.



## CHAPTER 4

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`



**d**

`decouils`, 5



## D

`decorator_with_args()` (in module `decoutils`), [5](#)  
`decoutils` (module), [5](#)