

---

# PyTapable

Vidhu

Jun 21, 2020



## CONTENTS:

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Inline Hooks</b>                       | <b>1</b>  |
| 1.1      | Usage . . . . .                           | 1         |
| 1.1.1    | Inline . . . . .                          | 1         |
| 1.1.2    | In a Class . . . . .                      | 1         |
| 1.2      | Inline Hooks Documentation . . . . .      | 2         |
| 1.2.1    | Hook . . . . .                            | 2         |
| 1.2.2    | HookableMixin . . . . .                   | 2         |
| <b>2</b> | <b>Functional Hooks</b>                   | <b>3</b>  |
| 2.1      | Usage on Class Instance methods . . . . . | 3         |
| 2.1.1    | How it works . . . . .                    | 3         |
| 2.1.2    | Inheritance . . . . .                     | 4         |
| 2.2      | Functional Hooks Documentation . . . . .  | 4         |
| 2.2.1    | FunctionalHook . . . . .                  | 4         |
| 2.2.2    | CreateHook . . . . .                      | 5         |
| 2.2.3    | HookableMixin . . . . .                   | 5         |
| 2.2.4    | HookMapping . . . . .                     | 5         |
| 2.2.5    | create_hook_name . . . . .                | 6         |
| 2.2.6    | create_hook_names . . . . .               | 6         |
| <b>3</b> | <b>Interceptors</b>                       | <b>7</b>  |
| <b>4</b> | <b>API Documentation</b>                  | <b>9</b>  |
| 4.1      | HookInterceptor . . . . .                 | 9         |
| 4.2      | Tap . . . . .                             | 9         |
| <b>5</b> | <b>Recipes</b>                            | <b>11</b> |
| 5.1      | Timing how long taps take . . . . .       | 11        |
| <b>6</b> | <b>Contributing</b>                       | <b>13</b> |
| <b>7</b> | <b>Installation</b>                       | <b>15</b> |
| <b>8</b> | <b>Introduction</b>                       | <b>17</b> |
| 8.1      | Type of Hooks . . . . .                   | 17        |
| 8.2      | Context . . . . .                         | 17        |
| <b>9</b> | <b>Quick Start</b>                        | <b>19</b> |
| 9.1      | InlineHooks . . . . .                     | 19        |
| 9.2      | FunctionalHooks . . . . .                 | 19        |

|                              |           |
|------------------------------|-----------|
| <b>10 Indices and tables</b> | <b>21</b> |
| <b>Index</b>                 | <b>23</b> |

## INLINE HOOKS

Inline hooks are defined inline with business logic code. They are manually triggered with user defined arguments which are received by callback functions

### 1.1 Usage

#### 1.1.1 Inline

```
from pytapable import Hook

# 1. Create our hook
my_hook = Hook()

# 2. Define a function to execute when hook triggers
def my_callback(context, fn_args, fn_kwargs):
    print(f"Hook says: {fn_kwargs['greeting']}")

# 3. Tap into our hook
my_hook.tap('My Tap Name', my_callback)

# 4. Trigger our hook
my_hook.call(greeting="Hi Callback")

>>> "Hook says: Hi Callback"
```

#### 1.1.2 In a Class

```
from pytapable import HookableMixin, create_hook_name

# 1. Class extends `HookableMixin`
class Car(HookableMixin):
    HOOK_ON_MOVE = create_hook_name('on_move')

    def __init__(self):
        super(Car, self).__init__()

    # 2. Define the hook
    self.hooks.add_hook(Hook(HOOK_ON_MOVE))

    def move(self, speed=10):
```

(continues on next page)

(continued from previous page)

```
# 3. Trigger the hook
self.hooks[HOOK_ON_MOVE].call(speed=speed)
return f"Moving at {speed}Mph"
```

---

**Note:** When using inline hooks in a class, its useful to have the class extend the `HookableMixin` class and create the hooks in the `self.hooks` dictionary. This allows for other classes to inherit hooks.

---

## 1.2 Inline Hooks Documentation

### 1.2.1 Hook

**class** `pytapable.Hook` (*name=None, interceptor=None*)

**call** (*\*\*kwargs*)

Triggers the hook which executes all the taps with a arguments passed in args, kwargs and a context dict

---

**Note:** Only named args are supported

---

```
# Arguments to a callback

"fn_kwargs": **kwargs

context = {
    'hook': Hook,
    'tap': Tap,
}
```

**tap** (*name, fn*)

Creates a *Tap* for this hook

#### Parameters

- **name** (*basestring*) – name of the tapable
- **fn** (*Callable*) – callable to execute when hook is triggered

### 1.2.2 HookableMixin

**class** `pytapable.HookableMixin` (*\*args, \*\*kwargs*)

Mixin which instantiates all the decorated class methods. This is needed for decorated class methods

Instantiates an instance property `self.hook` which is a *HookMapping*

## FUNCTIONAL HOOKS

Functional hooks are hooks which wrap a function. They fire before and after the execution of a function automatically. They are created using decorators on the function.

### 2.1 Usage on Class Instance methods

```
from pytapable import CreateHook, HookableMixin, create_hook_name

# 1. Class extends `HookableMixin` to initialize hooks on instance
class Car(HookableMixin):
    HOOK_ON_MOVE = create_hook_name('on_move')

    # 2. Mark this method as hookable
    @CreateHook(name=HOOK_ON_MOVE)
    def move(self, speed=10):
        return f"Moving at {speed}Mph"

c = Car()
c.hooks[Car.HOOK_ON_MOVE].tap(
    'log_metric_speed',
    lambda context, fn_kwargs, fn_output, is_before: ...,
    before=False
)
```

#### 2.1.1 How it works

When a method is decorated using the `CreateHook()` decorator, the wrapped function is marked. The class must extend the `HookableMixin` class. This is necessary because when the `Car` class is initialized, the hookable mixin goes through all the marked methods and constructs a `FunctionalHook`` for each of them.

These newly created hooks are stored on the `instance.hooks` attribute which is defined by the `HookableMixin` class. `instance.hooks` is a super class of a dict `HookMapping`

## Callback Arguments

- Arguments passed to callbacks from a *FunctionalHook* are predefined unlike *Hook* (inline hook). See *FunctionalHook*
- Hooked function args are converted to kwargs so a call to a hooked function like `obj.fn(1, 2, c=3, d=4)` will be passed to the callback as `fn_kwargs={'a': 1, 'b': 2, 'c': 3, 'd': 4}`. See *CreateHook*

## 2.1.2 Inheritance

*HookableMixin* allows you to inherit hooks from other classes that implement the *HookableMixin*

```
class MyClass(HookableMixin):  
  
    def __init__(self):  
        super(MyClass, self).__init__()  
        self.car = Car()  
  
        self.hooks.inherit_hooks(self.car)  
  
my_class = MyClass()  
my_class.hooks[Car.HOOK_ON_MOVE].tap(...)
```

## 2.2 Functional Hooks Documentation

### 2.2.1 FunctionalHook

**class** `pytapable.FunctionalHook` (*name=None, interceptor=None*)

Functional hooks are created when *CreateHook* is used to decorate a class function. When a functional hook is tapped, a *FunctionalTap* is created. Look at *FunctionalHook.call()* to see how taps are called

**call** (*fn\_kwargs, is\_before, fn\_output=None*)

Triggers all taps installed on this hook.

Taps receive predefined arguments (*context, fn\_args, fn\_output*)

```
# Arguments to a callback  
  
fn_kwargs: **kwargs  
  
fn_output = Optional[Any]  
  
context = {  
    'hook': FunctionalHook,  
    'tap': FunctionalTap,  
    'is_before': is_before  
}
```

#### Parameters

- **fn\_kwargs** (*dict*) – The kwargs the hooked function was called with. \*args should be converted to \*\*kwargs. See *utils.merge\_args\_to\_kwargs*



- **is\_before** (*bool*) – True if the hook is being called after the hooked function has executed
- **fn\_output** (*Optional[Any]*) – The return value of the hooked function if any. None otherwise

**tap** (*name, fn, before=True, after=True*)  
Creates a `FunctionalTap` for this hook

**Parameters**

- **name** (*str*) – Name of the tap
- **fn** (*Callable*) – This will be called when the hook triggers
- **before** (*bool*) – If true, this tap will be called *before* the hooked function executes
- **after** (*bool*) – If true, this tap will be called *after* the hooked function executes

## 2.2.2 CreateHook

`@pytapable.CreateHook` (*name, interceptor=None*)  
Decorator used for creating Hooks on instance methods. It takes in a name and optionally an instance of a `HookInterceptor`.

---

**Note:** This decorator doesn't actually create the hook. It just annotates the method. The hooks are created by the `HookableMixin` upon instantiation

---

---

**Note:** The wrapped function may be called with different combinations of positional and named args which would make it difficult for the callback function owner to know whether to read values from *\*args* or *\*\*kwargs*. We instead convert all positional args to named args to remove any ambiguity

See `utils.merge_args_to_kwargs()` for implementation details

---

## 2.2.3 HookableMixin

**class** `pytapable.HookableMixin` (*\*args, \*\*kwargs*)  
Mixin which instantiates all the decorated class methods. This is needed for decorated class methods  
Instantiates an instance property `self.hook` which is a `HookMapping`

## 2.2.4 HookMapping

**class** `pytapable.HookMapping`  
A dict like object with helper methods to inherit hooks and add hooks

**add\_hook** (*hook*)  
Adds the passed in hook to the hooks mapping dict

**Parameters** **hook** (*BaseHook*) – Hook to add to the mapping

**inherit\_hooks** (*hookable\_instance*)  
Given an instance which extends the `HookableMixin` class, inherits all hooks from it to expose it on top level

**Parameters** `hookable_instance` (`HookableMixin`) – Instance from which to inherit hooks

## 2.2.5 `create_hook_name`

`pytapable.create_hook_name(name=)`

Utility to create a unique hook name. Optionally takes in a name. The output string is the name prefixed with a UUID. This is useful to prevent collision in hook names when one class with hooks inherits hooks from another class

### Example

```
>>> create_hook_name()
>>> '7087eefc-8e94-4f0a-b7d3-453062bb7a34'
>>> create_hook_name('my_hook')
>>> '7087eefc-8e94-4f0a-b7d3-453062bb7a34:my_hook'
```

**Parameters** `name` (*Optional[str]*) – Name of the hook

## 2.2.6 `create_hook_names`

`pytapable.create_hook_names(*names)`

Useful shortcut to create multiple unique hook names in one statement

### Example

```
>>> HOOK_MY, HOOK_UNIQUE, HOOK_HOOK = create_hook_names('my', 'unique', 'hook')
>>> HOOK_ONE, HOOK_TWO, HOOK_THREE = create_hook_names(*range(3))
```

**Parameters** `*names` – Argument of hook names

**Returns** iterable which can be deconstructed across constants.

**Return type** Iterable

## INTERCEPTORS

Interceptors can be thought of as hooks for your hooks.

```
from pytapable import Hook, HookInterceptor

class TapLogger(HookInterceptor):

    def create(self, hook):
        print(f"Hook {hook.name} initialized")

    def register(self, context, tap):
        print(f"Hook being tapping is '{context['hook']}'")
        print(f"Hook being tapped by '{tap.name}'")
        return tap

tap_logger = TapLogger()

my_hook = Hook(interceptor=tap_logger)

>>> Hook XYZ initialized

my_hook.tap('My Tap', my_callback)

>>> Hook being tapped by 'My Tap'
```

They are a mechanism for you to intercept whenever one of your hooks are tapped into or when a hook was created

The return value of the `register` method must be a tap. If you need to modify the behavior of the callback in the tap, this is the place to do it.



## API DOCUMENTATION

### 4.1 HookInterceptor

**class** pytapable.HookInterceptor

Interceptors allow you to intercept actions that are being performed on hooks and optionally modify it

**create** (*hook*)

Triggered when a hook has been initialized

**Parameters** **hook** (*BaseHook*) – The hook that was created

**register** (*context, tap*)

Triggered for each added tap and allows you to modify the tap

```
context = {  
    'hook': BaseHook, # The hook that's being tapped  
}
```

**Parameters**

- **context** (*dict*) –
- **tap** (*Tap*) – The Tap that is going to be installed on the hook

**Returns** The Tap to install on the hook

**Return type** *modified\_tap* (*Tap*)

### 4.2 Tap

**class** pytapable.Tap (*name, fn*)

A Tap is an object created when you tap into a hook. It holds a reference to the function you want to execute when the hook triggers



**RECIPES**

Here's a list of interesting ways to use hooks. If you find an interesting way to use hooks that's worthy of sharing, please contribute by making a pull request!

## 5.1 Timing how long taps take

We don't always know what is tapping our hooks. This might make monitoring challenging especially if there are SLOs to be met.

Here we wrap callbacks in statsd timers using an interceptor and log them against a key derived from the stats' name

```
class InterceptorHookTimer (HookInterceptor):  
  
    def register(self, context, tap):  
        tap.fn = statsd.timer(f"interceptor.${tap.name}") (tap.fn)  
        return tap
```





## CONTRIBUTING

Contributions are what make the open source community such an amazing place to be learn, inspire, and create. Any contributions you make are **greatly appreciated**.

1. Fork the Project
2. Create your Feature Branch (`git checkout -b feature/AmazingFeature`)
3. Commit your Changes (`git commit -m 'Add some AmazingFeature'`)
4. Push to the Branch (`git push origin feature/AmazingFeature`)
5. Open a Pull Request

To tests on your changes locally, run:

```
$ pip install -r test_requirements.txt
$ tox .
```

This will run your changes on python-2 and python-3

Documentation for any new changes to APIs are a must. We use [Sphinx](#) and to build the documentation locally, run:

```
$ cd docs/
$ make html
# or on windows
$ make.bat html
```



## INSTALLATION

```
$ pip install pytapable
```



## INTRODUCTION

PyTapable provides a way to attach hooks into your application. Its a way to implement event listeners with side effects. This is particularly useful to maintain service boundaries in your application, allow plugable in your libraries which you users can extend upon etc

Lets first clarify the terminologies used in this library

**Hook** A Hook is an object which maintains a list of taps that have been installed/registered with it

**Tap** A Tap is an object which holds a reference to the function which is to be run when a hook is *executed*.

**tapping** Tapping into a hook is the act of registering a callable with a hook

**callbacks** Consumer defined function/callable which is executed in response to a hook being triggered

### 8.1 Type of Hooks

There are two types of hooks provided in this library

- **FunctionalHook:** Functional hooks are hooks which wrap a function. They fire before and after the execution of a function automatically. They are created using decorators on the function. See [Functional Hooks](#)
- **InlineHook:** Inline hooks are created and triggered manually. They are used in the body of functions and modules See [Inline Hooks](#)

The parameters your [callbacks](#) are called with differ based on which hooks there were called from. Callbacks from a functional hook contain the function's arguments and return value (if available) whereas callbacks from an inline hook contain parameters defined by the caller

### 8.2 Context

A context dict is also passed to the callback function which contain various metadata. This is covered in more detail in the API doc for [FunctionalHook](#) and [Hook](#)



## QUICK START

Here's an example on how to use *InlineHooks* and *FunctionalHooks*

### 9.1 InlineHooks

```
from pytapable import Hook

# 1. Create our hook
my_hook = Hook()

# 2. Define a function to execute when hook triggers
def my_callback(context, fn_kwargs):
    print(f"Hook says: {fn_kwargs['greeting']}")

# 3. Tap into our hook
my_hook.tap('My Tap Name', my_callback)

# 4. Trigger our hook
my_hook.call(greeting="Hi Callback")

>>> "Hook says: Hi Callback"
```

### 9.2 FunctionalHooks

Lets define out class with a hookable instance method

```
from pytapable import CreateHook, HookableMixin, create_hook_name

# 1. Class extends `HookableMixin` to initialize hooks on instance
class Car(HookableMixin):
    HOOK_ON_MOVE = create_hook_name('on_move')

    # 2. Mark this method as hookable
    @CreateHook(name=HOOK_ON_MOVE)
    def move(self, speed=10):
        return f"Moving at {speed}Mph"
```

and then tap into the hook

```
def log_metric_speed(context, fn_kwargs, fn_output):
    kmph_speed = fn_kwargs['speed'] * 1.61
    print(f"The car is moving at {kmph_speed} kmph")

c = Car()

# 3. Tap into our hook. before=False means callback will
#     execute after hooked function returns
c.hooks[Car.HOOK_ON_MOVE].tap(
    'log_metric_speed',
    log_metric_speed,
    before=False
)

# 4. Hook is automatically triggered
c.move(speed=10)

# Positional args are converted to named arguments
c.move(10)

>>> "The car is moving at 16.1 kmph"
```



## INDICES AND TABLES

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



## INDEX

### A

`add_hook()` (*pytapable.HookMapping* method), 5

### C

`call()` (*pytapable.FunctionalHook* method), 4

`call()` (*pytapable.Hook* method), 2

callbacks, 17

`create()` (*pytapable.HookInterceptor* method), 9

`create_hook_name()` (*in module pytapable*), 6

`create_hook_names()` (*in module pytapable*), 6

`CreateHook()` (*in module pytapable*), 5

### F

*FunctionalHook* (class in *pytapable*), 4

### H

Hook, 17

*Hook* (class in *pytapable*), 2

*HookableMixin* (class in *pytapable*), 5

*HookInterceptor* (class in *pytapable*), 9

*HookMapping* (class in *pytapable*), 5

### I

`inherit_hooks()` (*pytapable.HookMapping*  
method), 5

### R

`register()` (*pytapable.HookInterceptor* method), 9

### T

Tap, 17

*Tap* (class in *pytapable*), 9

`tap()` (*pytapable.FunctionalHook* method), 5

`tap()` (*pytapable.Hook* method), 2

tapping, 17