PyTapable

Vidhu

Jun 14, 2020

CONTENTS:

CHAPTER

ONE

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, greeting):
   print(f"Hook says: {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[HOOK_ON_MOVE] = Hook()

   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 inheriting hooks.

1.2 Inline Hooks Documentation

1.2.1 Hook

class pytapable.**Hook**(*interceptor=None*)

call(**args*, ***kwargs*)
Triggers the hook which executes all the taps with arguments passed as is

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

inherit_hooks(*hookable_instance*)
Given an instance which extends the *HookableMixin* class, inherits all hooks from it to expose it on top level

References to the inherited hooks are added in the `self.hooks` dict

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

CHAPTER

TWO

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 fn_args, fn_output, context: ...,
   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

Arguments passed to callables from a *FunctionalHook* are predefined unlike *Hook* (inline hook). Refer to the documentation below to understand the arguments

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.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**(*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_args*, *is_before*, *fn_output=None*)

Triggers all taps installed on this hook.

Taps receive predefined arguments *(fn_args, fn_output, context)*

```
fn_args = {
  "args": *args,
  "kwargs": **kwargs
}

fn_output = Optional[Any]

context = {
    'hook_type': FunctionalHook.HOOK_TYPE,
    'hook_type_label': self.label,
    'tap_name': tap.name,
    'is_before': is_before
}
```

Parameters

- **fn_args** (*dict*) – The arguments the hooked function was called with. *fn_args: { args: Tuple, kwargs: Dict }*
- **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* when the class is instantiated

---

2.2.3 HookableMixin

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

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

inherit_hooks(*hookable_instance*)

Given an instance which extends the *HookableMixin* class, inherits all hooks from it to expose it on top level

References to the inherited hooks are added in the `self.hooks` dict

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

2.2.4 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

2.2.5 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

CHAPTER

THREE

INTERCEPTORS

Interceptors can be thought of as hooks for your hooks.

```
from pytapable import Hook, HookInterceptor


class TapLogger(HookInterceptor):

    def register(self, tap):
        print(f"Hook being tapped by '${tap.name}'")
        return tap

tap_logger = TapLogger()

my_hook = Hook(interceptor=tap_logger)
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.

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.

3.1 API Documentation

3.1.1 HookInterceptor

class pytapable.**HookInterceptor**

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

register(*tap*)

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

Parameters **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*)

3.1.2 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

CHAPTER

FOUR

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!

4.1 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, tap):
        tap.fn = statsd.timer(f"interceptor.${tap.name}")(tap.fn)
        return tap
```

CHAPTER

FIVE

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
```

CHAPTER

SIX

INSTALLATION

```
$ pip install pytapable
```

CHAPTER

SEVEN

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

7.1 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 parameter 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

7.2 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*

CHAPTER

EIGHT

QUICK START

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

8.1 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, greeting):
   print(f"Hook says: {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"
```

8.2 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_args, fn_output):
   kmph_speed = fn_args['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(10)

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

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX