venusian Documentation

Release 3.1.0

Pylons Project

Nov 09, 2023

CONTENTS

Venusian is a library which allows you to defer the action of decorators. Instead of taking actions when a function, method, or class decorator is executed at import time, you can defer the action until a separate "scan" phase.

This library is most useful for framework authors. It is compatible with CPython versions 3.7+. It is also known to work on PyPy (Compatible with Python 3.7+).

Note: The name "Venusian" is a riff on a library named *Martian* (which had its genesis in the *Grok* web framework), from which the idea for Venusian was stolen. Venusian is similar to Martian, but it offers less functionality, making it slightly simpler to use.

CHAPTER

ONE

OVERVIEW

Offering a decorator that wraps a function, method, or class can be a convenience to your framework's users. But the very purpose of a decorator makes it likely to impede testability of the function or class it decorates: use of a decorator often prevents the function it decorates from being called with the originally passed arguments, or a decorator may modify the return value of the decorated function. Such modifications to behavior are "hidden" in the decorator code itself.

For example, let's suppose your framework defines a decorator function named `jsonify` which can wrap a function that returns an arbitrary Python data structure and renders it to a JSON serialization:

```
import json

def jsonify(wrapped):
    def json_wrapper(request):
        result = wrapped(request)
        dumped = json.dumps(result)
        return dumped
    return json_wrapper
```

Let's also suppose a user has written an application using your framework, and he has imported your jsonify decorator function, and uses it to decorate an application function:

```
from theframework import jsonify

@jsonify
def logged_in(request):
   return {'result':'Logged in'}
```

As a result of an import of the module containing the `logged_in` function, a few things happen:

- The user's `logged_in` function is replaced by the `json_wrapper` function.
- The only reference left to the original `logged_in` function is inside the frame stack of the call to the `jsonify` decorator.

This means, from the perspective of the application developer that the original `logged_in` function has effectively "disappeared" when it is decorated with your `jsonify` decorator. Without bothersome hackery, it can no longer be imported or retrieved by its original author.

More importantly, it also means that if the developer wants to unit test the `logged_in` function, he'll need to do so only indirectly: he'll need to call the `json_wrapper` wrapper decorator function and test that the json returned by the function contains the expected values. This will often imply using the `json.loads` function to turn the result of the function *back* into a Python dictionary from the JSON representation serialized by the decorator.

If the developer is a stickler for unit testing, however, he'll want to test *only* the function he has actually defined, not the wrapper code implied by the decorator your framework has provided. This is the very definition of unit testing (testing a "unit" without any other integration with other code). In this case, it is also more convenient for him to be able to test the function without the decorator: he won't need to use the `json.loads` function to turn the result back into a dictionary to make test assertions against. It's likely such a developer will try to find ways to get at the original function for testing purposes.

To do so, he might refactor his code to look like this:

```
from theframework import jsonify

@jsonify
def logged_in(request):
   return _logged_in(request)

def _logged_in(request):
   return {'result':'Logged in'}
```

Then in test code he might import only the `_logged_in` function instead of the decorated `logged_in` function for purposes of unit testing. In such a scenario, the conscientious unit testing app developer has to define two functions for each decorated function. If you're thinking "that looks pretty tedious", you're right.

To give the intrepid tester an "out", you might be tempted as a framework author to leave a reference to the original function around somewhere that the unit tester can import and use only for testing purposes. You might modify the `jsonify` decorator like so in order to do that:

```
import json
def jsonify(wrapped):
    def json_wrapper(request):
        result = wrapped(request)
        dumped = json.dumps(result)
        return dumped
    json_wrapper.original_function = wrapped
    return json_wrapper
```

The line `json_wrapper.original_function = wrapped` is the interesting one above. It means that the application developer has a chance to grab a reference to his original function:

```
from myapp import logged_in
result = logged_in.original_func(None)
self.assertEqual(result['result'], 'Logged in')
```

That works. But it's just a little weird. Since the `jsonify` decorator function has been imported by the developer from a module in your framework, the developer probably shouldn't really need to know how it works. If he needs to read its code, or understand documentation about how the decorator functions for testing purposes, your framework *might* be less valuable to him on some level. This is arguable, really. If you use some consistent pattern like this for all your decorators, it might be a perfectly reasonable solution.

However, what if the decorators offered by your framework were passive until activated explicitly? This is the promise of using Venusian within your decorator implementations. You may use Venusian within your decorators to associate a wrapped function, class, or method with a callback. Then you can return the originally wrapped function. Instead of your decorators being "active", the callback associated with the decorator is passive until a "scan" is initiated.

CHAPTER

TWO

USING VENUSIAN

The most basic use of Venusian within a decorator implementation is demonstrated below.

```
import venusian

def jsonify(wrapped):
    def callback(scanner, name, ob):
        print('jsonified')
    venusian.attach(wrapped, callback)
    return wrapped
```

As you can see, this decorator actually calls into venusian, but then simply returns the wrapped object. Effectively this means that this decorator is "passive" when the module is imported.

Usage of the decorator:

```
from theframework import jsonify

@jsonify
def logged_in(request):
    return {'result':'Logged in'}
```

Note that when we import and use the function, the fact that it is decorated with the `jsonify` decorator is immaterial. Our decorator doesn't actually change its behavior.

```
>>> from theapp import logged_in
>>> logged_in(None)
{'result':'Logged in'}
>>>
```

This is the intended result. During unit testing, the original function can be imported and tested despite the fact that it has been wrapped with a decorator.

However, we can cause something to happen when we invoke a *scan*.

```
import venusian
import theapp

scanner = venusian.Scanner()
scanner.scan(theapp)
```

Above we've imported a module named `theapp`. The `logged_in` function which we decorated with our `jsonify` decorator lives in this module. We've also imported the *venusian* module, and we've created an instance of the *venusian.*

Scanner class. Once we've created the instance of *venusian.Scanner*, we invoke its *venusian.Scanner.scan()* method, passing the `theapp` module as an argument to the method.

Here's what happens as a result of invoking the *venusian.Scanner.scan()* method:

1. Every object defined at module scope within the `theapp` Python module will be inspected to see if it has had a Venusian callback attached to it.
2. For every object that *does* have a Venusian callback attached to it, the callback is called.

We could have also passed the `scan` method a Python *package* instead of a module. This would recursively import each module in the package (as well as any modules in subpackages), looking for callbacks.

Note: During scan, the only Python files that are processed are Python *source* (`.py`) files. Compiled Python files (`.pyc`, `.pyo` files) without a corresponding source file are ignored.

In our case, because the callback we defined within the `jsonify` decorator function prints `jsonified` when it is invoked, which means that the word `jsonified` will be printed to the console when we cause *venusian.Scanner.scan()* to be invoked. How is this useful? It's not! At least not yet. Let's create a more realistic example.

Let's change our `jsonify` decorator to perform a more useful action when a scan is invoked by changing the body of its callback.

```
import venusian

def jsonify(wrapped):
    def callback(scanner, name, ob):
        def jsonified(request):
            result = wrapped(request)
            return json.dumps(result)
        scanner.registry.add(name, jsonified)
    venusian.attach(wrapped, callback)
    return wrapped
```

Now if we invoke a scan, we'll get an error:

```
import venusian
import theapp

scanner = venusian.Scanner()
scanner.scan(theapp)

AttributeError: Scanner has no attribute 'registry'.
```

The *venusian.Scanner* class constructor accepts any key-value pairs; for each key/value pair passed to the scanner's constructor, an attribute named after the key which points at the value is added to the scanner instance. So when you do:

```
import venusian
scanner = venusian.Scanner(a=1)
```

Thereafter, `scanner.a` will equal the integer 1.

Any number of key-value pairs can be passed to a scanner. The purpose of being able to pass arbitrary key/value pairs to a scanner is to allow cooperating decorator callbacks to access these values: each callback is passed the `scanner` constructed when a scan is invoked.

Let's fix our example by creating an object named `registry` that we'll pass to our scanner's constructor:

```
import venusian
import theapp

class Registry(object):
    def __init__(self):
        self.registered = []

    def add(self, name, ob):
        self.registered.append((name, ob))

registry = Registry()
scanner = venusian.Scanner(registry=registry)
scanner.scan(theapp)
```

At this point, we have a system which, during a scan, for each object that is wrapped with a Venusian-aware decorator, a tuple will be appended to the `registered` attribute of a `Registry` object. The first element of the tuple will be the decorated object's name, the second element of the tuple will be a "truly" decorated object. In our case, this will be a jsonify-decorated callable.

Our framework can then use the information in the registry to decide which view function to call when a request comes in.

Venusian callbacks must accept three arguments:

`scanner`

This will be the instance of the scanner that has had its `scan` method invoked.

`name`

This is the module-level name of the object being decorated.

`ob`

This is the object being decorated if it's a function or an instance; if the object being decorated is a *method*, however, this value will be the *class*.

If you consider that the decorator and the scanner can cooperate, and can perform arbitrary actions together, you can probably imagine a system where a registry will be populated that informs some higher-level system (such as a web framework) about the available decorated functions.

CHAPTER

THREE

SCAN CATEGORIES

Because an application may use two separate Venusian-using frameworks, Venusian allows for the concept of "scan categories".

The *venusian.attach()* function accepts an additional argument named `category`.

For example:

```
import venusian

def jsonify(wrapped):
    def callback(scanner, name, ob):
        def jsonified(request):
            result = wrapped(request)
            return json.dumps(result)
        scanner.registry.add(name, jsonified)
    venusian.attach(wrapped, callback, category='myframework')
    return wrapped
```

Note the `category='myframework'` argument in the call to *venusian.attach()*. This tells Venusian to attach the callback to the wrapped object under the specific scan category `myframework`. The default scan category is `None`.

Later, during *venusian.Scanner.scan()*, a user can choose to activate all the decorators associated only with a particular set of scan categories by passing a `categories` argument. For example:

```
import venusian
scanner = venusian.Scanner(a=1)
scanner.scan(theapp, categories=('myframework',))
```

The default `categories` argument is `None`, which means activate all Venusian callbacks during a scan regardless of their category.

CHAPTER

FOUR

ONERROR SCAN CALLBACK

New in version 1.0.

By default, when Venusian scans a package, it will propagate all exceptions raised while attempting to import code. You can use an `onerror` callback argument to *venusian.Scanner.scan()* to change this behavior.

The `onerror` argument should either be `None` or a callback function which behaves the same way as the `onerror` callback function described in http://docs.python.org/library/pkgutil.html#pkgutil.walk_packages .

Here's an example `onerror` callback that ignores all `ImportError` exceptions:

```
import sys
def onerror(name):
    if not issubclass(sys.exc_info()[0], ImportError):
        raise # reraise the last exception
```

Here's how we'd use this callback:

```
import venusian
scanner = venusian.Scanner()
scanner.scan(theapp, onerror=onerror)
```

The `onerror` callback should execute `raise` at some point if any exception is to be propagated, otherwise it can simply return. The `name` passed to `onerror` is the module or package dotted name that could not be imported due to an exception.

CHAPTER

FIVE

IGNORE SCAN ARGUMENT

New in version 1.0a3.

The `ignore` to `scan` allows you to ignore certain modules, packages, or global objects during a scan. It should be a sequence containing strings and/or callables that will be used to match against the full dotted name of each object encountered during the scanning process. If the ignore value you provide matches a package name, global objects contained by that package as well any submodules and subpackages of the package (and any global objects contained by them) will be ignored. If the ignore value you provide matches a module name, any global objects in that module will be ignored. If the ignore value you provide matches a global object that lives in a package or module, only that particular global object will be ignored.

The sequence can contain any of these three types of objects:

- A string representing a full dotted name. To name an object by dotted name, use a string representing the full dotted name. For example, if you want to ignore the `my.package` package and any of its subobjects during the scan, pass `ignore=['my.package']`. If the string matches a global object (e.g. `ignore=['my.package.MyClass']`), only that object will be ignored and the rest of the objects in the module or package that contains the object will be processed.
- A string representing a relative dotted name. To name an object relative to the `package` passed to this method, use a string beginning with a dot. For example, if the `package` you've passed is imported as `my.package`, and you pass `ignore=['.mymodule']`, the `my.package.mymodule` module and any of its subobjects will be omitted during scan processing. If the string matches a global object (e.g. `ignore=['my.package.MyClass']`), only that object will be ignored and the rest of the objects in the module or package that contains the object will be processed.
- A callable that accepts a full dotted name string of an object as its single positional argument and returns `True` or `False`. If the callable returns `True` or anything else truthy, the module, package, or global object is ignored, if it returns `False` or anything else falsy, it is not ignored. If the callable matches a package name, the package as well as any of that package's submodules and subpackages (recursively) will be ignored. If the callable matches a module name, that module and any of its contained global objects will be ignored. If the callable matches a global object name, only that object name will be ignored. For example, if you want to skip all packages, modules, and global objects that have a full dotted name that ends with the word "tests", you can use `ignore=[re.compile('tests$').search]`.

Here's an example of how we might use the `ignore` argument to `scan` to ignore an entire package (and any of its submodules and subpackages) by absolute dotted name:

```
import venusian
scanner = venusian.Scanner()
scanner.scan(theapp, ignore=['theapp.package'])
```

Here's an example of how we might use the `ignore` argument to `scan` to ignore an entire package (and any of its submodules and subpackages) by relative dotted name (`theapp.package`):

```
import venusian
scanner = venusian.Scanner()
scanner.scan(theapp, ignore=['.package'])
```

Here's an example of how we might use the `ignore` argument to `scan` to ignore a particular class object:

```
import venusian
scanner = venusian.Scanner()
scanner.scan(theapp, ignore=['theapp.package.MyClass'])
```

Here's an example of how we might use the `ignore` argument to `scan` to ignore any module, package, or global object that has a name which ends with the string `tests`:

```
import re
import venusian
scanner = venusian.Scanner()
scanner.scan(theapp, ignore=[re.compile('tests$').search])
```

You can mix and match the three types in the list. For example, `scanner.scan(my, ignore=['my.package', '.someothermodule', re.compile('tests$').search])` would cause `my.package` (and all its submodules and subobjects) to be ignored, `my.someothermodule` to be ignored, and any modules, packages, or global objects found during the scan that have a full dotted path that ends with the word `tests` to be ignored beneath the `my` package.

Packages and modules matched by any ignore in the list will not be imported, and their top-level code will not be run as a result.

CHAPTER

SIX

LIMITATIONS AND AUDIENCE

Venusian is not really a tool that is maximally useful to an application developer. It would be a little silly to use it every time you needed a decorator. Instead, it's most useful for framework authors, in order to be able to say to their users "the frobozz decorator doesn't change the output of your function at all" in documentation. This is a lot easier than telling them how to test methods/functions/classes decorated by each individual decorator offered by your frameworks.

CHAPTER

SEVEN

API DOCUMENTATION / GLOSSARY

7.1 API Documentation for Venusian

class venusian.**Scanner**(***kw*)

scan(*package*, *categories=None*, *onerror=None*, *ignore=None*)

Scan a Python package and any of its subpackages. All top-level objects will be considered; those marked with venusian callback attributes related to `category` will be processed.

The `package` argument should be a reference to a Python package or module object.

The `categories` argument should be sequence of Venusian callback categories (each category usually a string) or the special value `None` which means all Venusian callback categories. The default is `None`.

The `onerror` argument should either be `None` or a callback function which behaves the same way as the `onerror` callback function described in http://docs.python.org/library/pkgutil.html#pkgutil.walk_packages . By default, during a scan, Venusian will propagate all errors that happen during its code importing process, including `ImportError`. If you use a custom `onerror` callback, you can change this behavior.

Here's an example `onerror` callback that ignores `ImportError`:

```
import sys
def onerror(name):
    if not issubclass(sys.exc_info()[0], ImportError):
        raise # reraise the last exception
```

The `name` passed to `onerror` is the module or package dotted name that could not be imported due to an exception.

New in version 1.0: the `onerror` callback

The `ignore` argument allows you to ignore certain modules, packages, or global objects during a scan. It should be a sequence containing strings and/or callables that will be used to match against the full dotted name of each object encountered during a scan. The sequence can contain any of these three types of objects:

- A string representing a full dotted name. To name an object by dotted name, use a string representing the full dotted name. For example, if you want to ignore the `my.package` package *and any of its subobjects or subpackages* during the scan, pass `ignore=['my.package']`.
- A string representing a relative dotted name. To name an object relative to the `package` passed to this method, use a string beginning with a dot. For example, if the `package` you've passed is imported as `my.package`, and you pass `ignore=['.mymodule']`, the `my.package.mymodule` mymodule *and any of its subobjects or subpackages* will be omitted during scan processing.

- A callable that accepts a full dotted name string of an object as its single positional argument and returns `True` or `False`. For example, if you want to skip all packages, modules, and global objects with a full dotted path that ends with the word "tests", you can use `ignore=[re.compile('tests$').search]`. If the callable returns `True` (or anything else truthy), the object is ignored, if it returns `False` (or anything else falsy) the object is not ignored. *Note that unlike string matches, ignores that use a callable don't cause submodules and subobjects of a module or package represented by a dotted name to also be ignored, they match individual objects found during a scan, including packages, modules, and global objects*.

You can mix and match the three types of strings in the list. For example, if the package being scanned is `my`, `ignore=['my.package', '.someothermodule', re.compile('tests$').search]` would cause `my.package` (and all its submodules and subobjects) to be ignored, `my.someothermodule` to be ignored, and any modules, packages, or global objects found during the scan that have a full dotted name that ends with the word `tests` to be ignored.

Note that packages and modules matched by any ignore in the list will not be imported, and their top-level code will not be run as a result.

A string or callable alone can also be passed as `ignore` without a surrounding list.

New in version 1.0a3: the `ignore` argument

class venusian.**AttachInfo**(***kw*)

An instance of this class is returned by the *venusian.attach()* function. It has the following attributes:

`scope`

One of `exec`, `module`, `class`, `function call` or `unknown` (each a string). This is the scope detected while executing the decorator which runs the attach function.

`module`

The module in which the decorated function was defined.

`locals`

A dictionary containing decorator frame's f_locals.

`globals`

A dictionary containing decorator frame's f_globals.

`category`

The `category` argument passed to `attach` (or `None`, the default).

`codeinfo`

A tuple in the form `(filename, lineno, function, sourceline)` representing the context of the venusian decorator used. Eg. `('/home/chrism/projects/venusian/tests/test_advice.py', 81, 'testCallInfo', 'add_handler(foo, bar)')`

venusian.**attach**(*wrapped*, *callback*, *category=None*, *name=None*)

Attach a callback to the wrapped object. It will be found later during a scan. This function returns an instance of the *venusian.AttachInfo* class.

`category` should be `None` or a string representing a decorator category name.

`name` should be `None` or a string representing a subcategory within the category. This will be used by the `lift` class decorator to determine if decorations of a method should be inherited or overridden.

class venusian.**lift**(*categories=None*)

A class decorator which 'lifts' superclass venusian configuration decorations into subclasses. For example:

```
from venusian import lift
from somepackage import venusian_decorator

class Super(object):
    @venusian_decorator()
    def boo(self): pass

    @venusian_decorator()
    def hiss(self): pass

    @venusian_decorator()
    def jump(self): pass

@lift()
class Sub(Super):
    def boo(self): pass

    def hiss(self): pass

    @venusian_decorator()
    def smack(self): pass
```

The above configuration will cause the callbacks of seven venusian decorators. The ones attached to Super.boo, Super.hiss, and Super.jump *plus* ones attached to Sub.boo, Sub.hiss, Sub.hump and Sub.smack.

If a subclass overrides a decorator on a method, its superclass decorators will be ignored for the subclass. That means that in this configuration:

```
from venusian import lift
from somepackage import venusian_decorator

class Super(object):
    @venusian_decorator()
    def boo(self): pass

    @venusian_decorator()
    def hiss(self): pass

@lift()
class Sub(Super):

    def boo(self): pass

    @venusian_decorator()
    def hiss(self): pass
```

Only four, not five decorator callbacks will be run: the ones attached to Super.boo and Super.hiss, the inherited one of Sub.boo and the non-inherited one of Sub.hiss. The inherited decorator on Super.hiss will be ignored for the subclass.

The `lift` decorator takes a single argument named 'categories'. If supplied, it should be a tuple of category names. Only decorators in this category will be lifted if it is suppled.

class venusian.**onlyliftedfrom**

A class decorator which marks a class as 'only lifted from'. Decorations made on methods of the class won't have

their callbacks called directly, but classes which inherit from only-lifted-from classes which also use the `lift` class decorator will use the superclass decoration callbacks.

For example:

```
from venusian import lift, onlyliftedfrom
from somepackage import venusian_decorator

@onlyliftedfrom()
class Super(object):
    @venusian_decorator()
    def boo(self): pass

    @venusian_decorator()
    def hiss(self): pass

@lift()
class Sub(Super):

    def boo(self): pass

    def hiss(self): pass
```

Only two decorator callbacks will be run: the ones attached to Sub.boo and Sub.hiss. The inherited decorators on Super.boo and Super.hiss will be not be registered.

7.2 Glossary

Grok
A Zope-based *web framework <http://grok.zope.org>*.

Martian
The package venusian was inspired by, part of the *Grok* project.

scan
Walk a module or package executing callbacks defined by venusian-aware decorators along the way.

CHAPTER

EIGHT

INDICES AND TABLES

- *Glossary*
- modindex
- search

PYTHON MODULE INDEX

INDEX