gorilla Documentation

Release 0.4.0

Christopher Crouzet

Apr 17, 2021

Contents

Welcome! If you are just getting started, a recommended first read is the *Overview* as it shortly covers the *why*, *what*, and *how*'s of this library. From there, the *Installation* then the *Tutorial* sections should get you up to speed with the basics required to use it.

Looking how to use a specific function, class, or method? The whole public interface is described in the *API Reference* section.

Please report bugs and suggestions on GitHub.

CHAPTER 1

User's Guide

1.1 Overview

Monkey patching is the process of **modifying module and class attributes at runtime** with the purpose of replacing or extending third-party code.

Although *not* a recommended practice, it is sometimes useful to fix or modify the behaviour of a piece of code from a third-party library, or to extend its public interface while making the additions feel like they are built-in into the library.

The Python language makes monkey patching extremely easy but the advantages of Gorilla are multiple, not only in assuring a **consistent behaviour** on both Python 2 and Python 3 versions, but also in preventing common source of errors, and making the process both **intuitive and convenient** even when faced with *large* numbers of patches to create.

1.1.1 Features

- intuitive and convenient decorator approach to create patches.
- can create patches for all class or module members at once.
- compatible with both Python 2 and Python 3.
- customizable behaviour.

1.1.2 Usage

Thanks to the dynamic nature of Python that makes monkey patching possible, the process happens at runtime without ever having to directly modify the source code of the third-party library:

```
>>> import gorilla
>>> import destination
>>> @gorilla.patches(destination.Class)
... class MyClass(object):
```

(continues on next page)

(continued from previous page)

```
...     def method(self):
...         print("Hello")
...     @classmethod
...     def class_method(cls):
...         print("world!")
```

The code above creates two patches, one for each member of the class `MyClass`, but does not apply them yet. In other words, they define the information required to carry on the operation but are not yet inserted into the specified destination class `destination.Class`.

Such patches created with the decorators can then be automatically retrieved by recursively scanning a package or a module, then applied:

```
>>> import gorilla
>>> import mypackage
>>> patches = gorilla.find_patches([mypackage])
>>> for patch in patches:
...     gorilla.apply(patch)
```

See also:

The *Tutorial* section for more detailed examples and explanations on how to use Gorilla.

1.2 Installation

Gorilla doesn't have any requirement outside of the Python interpreter. Any of the following Python versions is supported: 2.7, 3.3, 3.4, 3.5, and 3.6.

1.2.1 Installing pip

The recommended[1] approach for installing a Python package such as Gorilla is to use `pip`, a package manager for projects written in Python. If `pip` is not already installed on your system, you can do so by following these steps:

1. Download get-pip.py.
2. Run `python get-pip.py` in a shell.

Note: The installation commands described in this page might require `sudo` privileges to run successfully.

1.2.2 System-Wide Installation

Installing globally the most recent version of Gorilla can be done with `pip`:

```
$ pip install gorilla
```

Or using `easy_install` (provided with `setuptools`):

```
$ easy_install gorilla
```

[1] See the Python Packaging User Guide

1.2.3 Virtualenv

If you'd rather make Gorilla only available for your specific project, an alternative approach is to use `virtualenv`. First, make sure that it is installed:

```
$ pip install virtualenv
```

Then, an isolated environment needs to be created for your project before installing Gorilla in there:

```
$ mkdir myproject
$ cd myproject
$ virtualenv env
New python executable in /path/to/myproject/env/bin/python
Installing setuptools, pip, wheel...done.
$ source env/bin/activate
$ pip install gorilla
```

At this point, Gorilla is available for the project `myproject` as long as the virtual environment is activated.

To exit the virtual environment, run:

```
$ deactivate
```

Note: Instead of having to activate the virtual environment, it is also possible to directly use the `env/bin/python`, `env/bin/pip`, and the other executables found in the folder `env/bin`.

Note: For Windows, some code samples might not work out of the box. Mainly, activating `virtualenv` is done by running the command `env\Scripts\activate` instead.

1.2.4 Development Version

To stay cutting edge with the latest development progresses, it is possible to directly retrieve the source from the repository with the help of Git:

```
$ git clone https://github.com/christophercrouzet/gorilla.git
$ cd gorilla
$ pip install --editable .[dev]
```

Note: The `[dev]` part installs additional dependencies required to assist development on Gorilla.

1.3 Tutorial

In the end Gorilla is nothing more than a fancy wrapper around Python's `setattr()` function and thus requires to define patches, represented by the class *`Patch`*, containing the destination object, the attribute name at the destination, and the actual value to set.

The *Patch* class can be used directly if the patching information are only known at runtime, as described in the section *Dynamic Patching*, but otherwise a set of decorators are available to make the whole process more intuitive and convenient.

The recommended approach involving decorators is to be done in two steps:

- create a *single patch* with the *patch()* decorator and/or *multiple patches* using *patches()*.
- *find and apply the patches* through the *find_patches()* and *apply()* functions.

1.3.1 Creating a Single Patch

In order to make a function `my_function()` available from within a third-party module `destination`, the first step is to create a new patch by decorating our function:

```
>>> import gorilla
>>> import destination
>>> @gorilla.patch(destination)
... def my_function():
...     print("Hello world!")
```

This step only creates the *Patch* object containing the patch information but does not inject the function into the destination module just yet. The *apply()* function needs to be called for that to happen, as shown in the section *Finding and Applying the Patches*.

The defaut behaviour is for the patch to inject the function at the destination using the name of the decorated object, that is `'my_function'`. If a different name is desired but changing the function name is not possible, then it can be done via the parameter *name*:

```
>>> import gorilla
>>> import destination
>>> @gorilla.patch(destination, name='better_function')
... def my_function():
...     print("Hello world!")
```

After applying the patch, the function will become accessible through a call to `destination.better_function()`.

A patch's destination can not only be a module as shown above, but also an existing class:

```
>>> import gorilla
>>> import destination
>>> @gorilla.patch(destination.Class)
... def my_method(self):
...     print("Hello")
>>> @gorilla.patch(destination.Class)
... @classmethod
... def my_class_method(cls):
...     print("world!")
```

1.3.2 Creating Multiple Patches at Once

As the number of patches grows, the process of defining a decorator for each individual patch can quickly become cumbersome. Instead, another decorator *patches()* is available to create a batch of patches (tongue-twister challenge: repeat "batch of patches" 10 times):

```
>>> import gorilla
>>> import destination
>>> @gorilla.patches(destination.Class)
... class MyClass(object):
...     def method(self):
...         print("Hello")
...     @classmethod
...     def class_method(cls):
...         print("world")
...     @staticmethod
...     def static_method():
...         print("!")
```

The *patches()* decorator iterates through all the members of the decorated class, by default filtered using the *default_filter()* function, while creating a patch for each of them.

Each patch created in this manner inherits the properties defined by the root decorator but it is still possible to override them using any of the *destination()*, *name()*, *settings()*, and *filter()* modifier decorators:

```
>>> import gorilla
>>> import destination
>>> @gorilla.patches(destination.Class)
... class MyClass(object):
...     @gorilla.name('better_method')
...     def method(self):
...         print("Hello")
...     @gorilla.settings(allow_hit=True)
...     @classmethod
...     def class_method(cls):
...         print("world")
...     @gorilla.filter(False)
...     @staticmethod
...     def static_method():
...         print("!")
```

In the example above, the method's name is overriden to `'better_method'`, the class method is allowed to overwrite an attribute with the same name at the destination, and the static method is to be filtered out during the discovery process described in *Finding and Applying the Patches*, leading to no patch being created for it.

Note: The same operation can also be used to create a patch for each member of a module but, since it is not possible to decorate a module, the function *create_patches()* needs to be directly used instead.

1.3.3 Overwriting Attributes at the Destination

If there was to be an attribute at the patch's destination already existing with the patch's name, then the patching process can optionally override the original attribute after storing a copy of it. This way, the original attribtue remains accessible from within our code with the help of the *get_original_attribute()* function:

```
>>> import gorilla
>>> import destination
>>> settings = gorilla.Settings(allow_hit=True)
>>> @gorilla.patch(destination, settings=settings)
... def function():
...     print("Hello world!")
```

(continues on next page)

(continued from previous page)

```
...     # We're overwriting an existing function here,
...     # preserve its original behaviour.
...     original = gorilla.get_original_attribute(destination, 'function')
...     return original()
```

Note: The default settings of a patch do not allow attributes at the destination to be overwritten. For such a behaviour, the attribute *Settings.allow_hit* needs to be set to `True`.

1.3.4 Stack Ordering

The order in which the decorators are applied *does* matter. The *patch()* decorator can only be aware of the decorators defined below it.

```
>>> import gorilla
>>> import destination
>>> @gorilla.patch(destination.Class)
... @staticmethod
... def my_static_method_1():
...     print("Hello")
>>> @staticmethod
... @gorilla.patch(destination.Class)
... def my_static_method_2():
...     print("world!")
```

Here, only the static method `my_static_method_1()` will be injected as expected with the decorator `staticmethod` while the other one will result in an invalid definition since it will be interpreted as a standard method but doesn't define any parameter referring to the class object such as `self`.

1.3.5 Finding and Applying the Patches

Once that the patches are created with the help of the decorators, the next step is to (recursively) scan the modules and packages to retrieve them. This is easily achieved with the *find_patches()* function.

Finally, each patch can be applied using the *apply()* function.

```
>>> import gorilla
>>> import mypackage
>>> patches = gorilla.find_patches([mypackage])
>>> for patch in patches:
...     gorilla.apply(patch)
```

1.3.6 Dynamic Patching

In the case where patches need to be created dynamically, meaning that the patch source objects and/or destinations are not known until runtime, then it is possible to directly use the *Patch* class.

```
>>> import gorilla
>>> import destination
>>> def my_function():
...     print("Hello world!")
```

(continues on next page)

(continued from previous page)

```
>>> patch = gorilla.Patch(destination, 'better_function', my_function)
>>> gorilla.apply(patch)
```

Note: Special precaution is advised when directly setting the *Patch.obj* attribute. See the warning note in the class *Patch* for more details.

1.4 A Word of Caution

The process of Monkey Patching is at the same time both incredibly powerful *and* dangerous. It makes it easy to improve things on the surface but makes it even easier to cause troubles if done inappropriately.

Mostly, inserting new attributes by prefixing their name to avoid (future?) name clashes is *usually* fine, but **replacing existing attributes should be avoided like the plague** unless you really have to and know what you are doing. That is, if you do not want ending up being fired because you broke everyone else's code.

As a safety measure, Gorilla has its *Settings.allow_hit* attribute set to `False` by default, which raises an exception whenever it detects an attempt at overwriting an existing attribute.

If you still want to go ahead with allowing hits, a second measure enabled by default through the *Settings.store_hit* attribute is to store the overwriten attribute under a different name to have it still accessible using the function *get_original_attribute()*.

But still, avoid it if you can.

You've been warned.

1.5 API Reference

The whole public interface of Gorilla is described here.

All of the library's content is accessible from within the only module `gorilla`.

The classes *Settings*, *Patch*, and the function *apply* form the *core* of the library and cover all the requirements for monkey patching.

For intuitivity and convenience reasons, *decorators* and *utility* functions are also provided.

1.5.1 Core

Settings	Define the patching behaviour.
Patch	Describe all the information required to apply a patch.
apply	Apply a patch.

class gorilla.**Settings**(***kwargs*)

Define the patching behaviour.

allow_hit

A hit occurs when an attribute at the destination already exists with the name given by the patch. If `False`,

the patch process won't allow setting a new value for the attribute by raising an exception. Defaults to `False`.

Type bool

store_hit
If `True` and *allow_hit* is also set to `True`, then any attribute at the destination that is hit is stored under a different name before being overwritten by the patch. Defaults to `True`.

Type bool

__init__(***kwargs*)
Constructor.

Parameters **kwargs** – Keyword arguments, see the attributes.

---

class gorilla.**Patch**(*destination*, *name*, *obj*, *settings=None*)
Describe all the information required to apply a patch.

destination
Patch destination.

Type obj

name
Name of the attribute at the destination.

Type str

obj
Attribute value.

Type obj

settings
Settings. If `None`, the default settings are used.

Type *gorilla.Settings* or None

> **Warning:** It is highly recommended to use the output of the function *get_attribute()* for setting the attribute *obj*. This will ensure that the descriptor protocol is bypassed instead of possibly retrieving attributes invalid for patching, such as bound methods.

__init__(*destination*, *name*, *obj*, *settings=None*)
Constructor.

Parameters

- **destination** (*object*) – See the *destination* attribute.
- **name** (*str*) – See the *name* attribute.
- **obj** (*object*) – See the *obj* attribute.
- **settings** (gorilla.Settings) – See the *settings* attribute.

---

gorilla.**apply**(*patch*, *id='default'*)
Apply a patch.

The patch's *obj* attribute is injected into the patch's *destination* under the patch's *name*.

This is a wrapper around calling `setattr(patch.destination, patch.name, patch.obj)`.

Parameters

- **patch** (gorilla.Patch) – Patch.
- **id** (*str*) – When applying a stack of patches on top of a same attribute, this identifier allows to pinpoint a specific original attribute if needed.

Raises `RuntimeError` – Overwriting an existing attribute is not allowed when the setting *Settings.allow_hit* is set to `True`.

Note: If both the attributes *Settings.allow_hit* and *Settings.store_hit* are `True` but that the target attribute seems to have already been stored, then it won't be stored again to avoid losing the original attribute that was stored the first time around.

gorilla.**revert**(*patch*)
Revert a patch.

Parameters **patch** (gorilla.Patch) – Patch.

Note: This is only possible if the attribute *Settings.store_hit* was set to `True` when applying the patch and overriding an existing attribute.

1.5.2 Decorators

patch	Decorator to create a patch.
patches	Decorator to create a patch for each member of a module or a class.
destination	Modifier decorator to update a patch's destination.
name	Modifier decorator to update a patch's name.
settings	Modifier decorator to update a patch's settings.
filter	Modifier decorator to force the inclusion or exclusion of an attribute.

gorilla.**patch**(*destination*, *name=None*, *settings=None*)
Decorator to create a patch.

The object being decorated becomes the *obj* attribute of the patch.

Parameters

- **destination** (*object*) – Patch destination.
- **name** (*str*) – Name of the attribute at the destination.
- **settings** (gorilla.Settings) – Settings.

Returns The decorated object.

Return type object

See also:

Patch

---

gorilla.**patches**(*destination*, *settings=None*, *traverse_bases=True*, *filter=<function default_filter>*, *recursive=True*, *use_decorators=True*)

Decorator to create a patch for each member of a module or a class.

Parameters

- **destination** (*object*) – Patch destination.
- **settings** (gorilla.Settings) – Settings.
- **traverse_bases** (*bool*) – If the object is a class, the base classes are also traversed.
- **filter** (*function*) – Attributes for which the function returns `False` are skipped. The function needs to define two parameters: `name`, the attribute name, and `obj`, the attribute value. If `None`, no attribute is skipped.
- **recursive** (*bool*) – If `True`, and a hit occurs due to an attribute at the destination already existing with the given name, and both the member and the target attributes are classes, then instead of creating a patch directly with the member attribute value as is, a patch for each of its own members is created with the target as new destination.
- **use_decorators** (*bool*) – Allows to take any modifier decorator into consideration to allow for more granular customizations.

Returns The decorated object.

Return type object

Note: A 'target' differs from a 'destination' in that a target represents an existing attribute at the destination about to be hit by a patch.

See also:

Patch, *create_patches()*

---

gorilla.**destination**(*value*)

Modifier decorator to update a patch's destination.

This only modifies the behaviour of the *create_patches()* function and the *patches()* decorator, given that their parameter `use_decorators` is set to `True`.

Parameters **value** (*object*) – Patch destination.

Returns The decorated object.

Return type object

---

gorilla.**name**(*value*)

Modifier decorator to update a patch's name.

This only modifies the behaviour of the *create_patches()* function and the *patches()* decorator, given that their parameter `use_decorators` is set to `True`.

Parameters **value** (*object*) – Patch name.

Returns The decorated object.

Return type object

---

gorilla.**settings**(***kwargs*)

Modifier decorator to update a patch's settings.

This only modifies the behaviour of the *create_patches()* function and the *patches()* decorator, given that their parameter `use_decorators` is set to `True`.

Parameters **kwargs** – Settings to update. See *Settings* for the list.

Returns The decorated object.

Return type object

---

gorilla.**filter**(*value*)

Modifier decorator to force the inclusion or exclusion of an attribute.

This only modifies the behaviour of the *create_patches()* function and the *patches()* decorator, given that their parameter `use_decorators` is set to `True`.

Parameters **value** (*bool*) – `True` to force inclusion, `False` to force exclusion, and `None` to inherit from the behaviour defined by *create_patches()* or *patches()*.

Returns The decorated object.

Return type object

1.5.3 Utilities

default_filter	Attribute filter.
create_patches	Create a patch for each member of a module or a class.
find_patches	Find all the patches created through decorators.
get_attribute	Retrieve an attribute while bypassing the descriptor protocol.
get_original_attribute	Retrieve an overriden attribute that has been stored.
DecoratorData	Decorator data.
get_decorator_data	Retrieve any decorator data from an object.

---

gorilla.**default_filter**(*name*, *obj*)

Attribute filter.

It filters out module attributes, and also methods starting with an underscore _.

This is used as the default filter for the *create_patches()* function and the *patches()* decorator.

Parameters

- **name** (*str*) – Attribute name.
- **obj** (*object*) – Attribute value.

Returns Whether the attribute should be returned.

Return type bool

---

gorilla.**create_patches**(*destination*, *root*, *settings=None*, *traverse_bases=True*, *filter=<function default_filter>*, *recursive=True*, *use_decorators=True*)

Create a patch for each member of a module or a class.

Parameters

- **destination** (*object*) – Patch destination.
- **root** (*object*) – Root object, either a module or a class.
- **settings** (gorilla.Settings) – Settings.
- **traverse_bases** (*bool*) – If the object is a class, the base classes are also traversed.
- **filter** (*function*) – Attributes for which the function returns `False` are skipped. The function needs to define two parameters: `name`, the attribute name, and `obj`, the attribute value. If `None`, no attribute is skipped.
- **recursive** (*bool*) – If `True`, and a hit occurs due to an attribute at the destination already existing with the given name, and both the member and the target attributes are classes, then instead of creating a patch directly with the member attribute value as is, a patch for each of its own members is created with the target as new destination.
- **use_decorators** (*bool*) – `True` to take any modifier decorator into consideration to allow for more granular customizations.

Returns The patches.

Return type list of gorilla.Patch

Note: A 'target' differs from a 'destination' in that a target represents an existing attribute at the destination about to be hit by a patch.

See also:

patches()

---

gorilla.**find_patches**(*modules*, *recursive=True*)

Find all the patches created through decorators.

Parameters

- **modules** (*list of module*) – Modules and/or packages to search the patches in.
- **recursive** (*bool*) – `True` to search recursively in subpackages.

Returns Patches found.

Return type list of gorilla.Patch

Raises `TypeError` – The input is not a valid package or module.

See also:

patch(), *patches()*

---

gorilla.**get_attribute**(*obj*, *name*)
Retrieve an attribute while bypassing the descriptor protocol.

As per the built-in `getattr()` function, if the input object is a class then its base classes might also be searched until the attribute is found.

Parameters

- **obj** (*object*) – Object to search the attribute in.
- **name** (*str*) – Name of the attribute.

Returns The attribute found.

Return type object

Raises `AttributeError` – The attribute couldn't be found.

---

gorilla.**get_original_attribute**(*obj*, *name*, *id='default'*)
Retrieve an overriden attribute that has been stored.

Parameters

- **obj** (*object*) – Object to search the attribute in.
- **name** (*str*) – Name of the attribute.
- **id** (*str*) – Identifier of the original attribute to retrieve from the stack.

Returns The attribute found.

Return type object

Raises `AttributeError` – The attribute couldn't be found.

See also:

Settings.allow_hit

---

class gorilla.**DecoratorData**
Decorator data.

patches
Patches created through the decorators.

Type list of gorilla.Patch

override
Any overriding value defined by the *destination()*, *name()*, and *settings()* decorators.

Type dict

filter
Value defined by the *filter()* decorator, if any, or `None` otherwise.

Type bool or None

__init__()
Constructor.

---

gorilla.**get_decorator_data**(*obj*, *set_default=False*)
Retrieve any decorator data from an object.

Parameters

- **obj** (*object*) – Object.
- **set_default** (*bool*) – If no data is found, a default one is set on the object and returned, otherwise `None` is returned.

Returns The decorator data or `None`.

Return type *gorilla.DecoratorData*

CHAPTER 2

Developer's Guide

2.1 Running the Tests

After making any code change in Gorilla, tests need to be evaluated to ensure that the library still behaves as expected.

Note: Some of the commands below are wrapped into `make` targets for convenience, see the file `Makefile`.

2.1.1 unittest

The tests are written using Python's built-in `unittest` module. They are available in the `tests` directory and can be fired through the `tests/run.py` file:

```
$ python tests/run.py
```

It is possible to run specific tests by passing a space-separated list of partial names to match:

```
$ python tests/run.py ThisTestClass and_that_function
```

The `unittest`'s command line interface is also supported:

```
$ python -m unittest discover -s tests -v
```

Finally, each test file is a **standalone** and can be directly executed.

2.1.2 tox

Test environments have been set-up with `tox` to allow testing Gorilla against each supported version of Python:

```
$ tox
```

2.1.3 coverage

The package `coverage` is used to help localize code snippets that could benefit from having some more testing:

```
$ coverage run --source gorilla -m unittest discover -s tests
$ coverage report
$ coverage html
```

In no way should `coverage` be a race to the 100% mark since it is *not* always meaningful to cover each single line of code. Furthermore, **having some code fully covered isn't synonym to having quality tests**. This is our responsability, as developers, to write each test properly regardless of the coverage status.

CHAPTER 3

Additional Information

3.1 Changelog

Version numbers comply with the Sementic Versioning Specification (SemVer).

3.1.1 Unreleased

3.1.2 v0.4.0 (2021-04-17)

Added

- Implement a new public function to revert a patch.
- Support applying stacks of patches.
- Include the utf-8 shebang to all source files.
- Enforce Python 3 compatibility with the `__future__` module.
- Testing with Python versions 3.7, 3,8, and 3.9.
- Set the `__all__` attribute.
- Make use of styling and linting tools.

Removed

- Testing with Python version 3.3.
- Testing of the representation outputs.

Changed

- Update the setup file.
- Rework the project's metadata.
- Shorten docstrings for non-public functions.
- Make minor tweaks to the code.
- Use the 'new' string formatting method.
- Update the contact's email.

Fixed

- Fix `__weakref__` showing up in the doc.
- Fix the changelog reference.

3.1.3 v0.3.0 (2017-01-18)

Added

- Add the decorator data to the public interface.
- Add support for coverage and tox.
- Add continuous integration with Travis and coveralls.
- Add a few bling-bling badges to the readme.
- Add a Makefile to regroup common actions for developers.

Changed

- Improve the documentation.
- Improve the unit testing workflow.
- Remove the `__slots__` attribute from the `Settings` and `Patch` classes.
- Refocus the content of the readme.
- Define the 'long_description' and 'extras_require' metadata to setuptools' setup.
- Update the documentation's Makefile with a simpler template.
- Rework the '.gitignore' files.
- Rename the changelog to 'CHANGELOG'!
- Make minor tweaks to the code.

Fixed

- Fix the settings not being properly inherited.
- Fix the decorator data not supporting class inheritance.

3.1.4 v0.2.0 (2016-12-20)

Changed

- Rewrite everything from scratch. Changes are not backwards compatible.

3.1.5 v0.1.0 (2014-06-29)

Added

- Add settings to modify the behaviour of the patching process.
- Added a FAQ section to the doc.

Changed

- Refactor the class `ExtensionSet` towards using an `add()` method.
- Clean-up the `Extension.__init__()` method from the parameters not required to construct the class.
- Get the `ExtensionsRegistrar.register_extensions()` function to return a single `ExtensionSet` object.
- Make minor tweaks to the code and documentation.

3.1.6 v0.0.1 (2014-06-21)

- Initial release.

3.2 Versioning

Version numbers comply with the Sementic Versioning Specification (SemVer).

In summary, version numbers are written in the form `MAJOR.MINOR.PATCH` where:

- incompatible API changes increment the MAJOR version.
- functionalities added in a backwards-compatible manner increment the MINOR version.
- backwards-compatible bug fixes increment the PATCH version.

Major version zero (0.y.z) is considered a special case denoting an initial development phase. Anything may change at any time without the MAJOR version being incremented.

3.3 License

3.4 Out There

Projects using Gorilla include:

- bana
- mlflow

Index

Symbols

A

C

D

F

G

N

O

P

R

S