Compat Patcher Core Documentation

Release 1.2

Pascal Chambon & others

Jan 19, 2021

Contents

Long term API compatibility for fast-moving projects

Welcome to **Compat Patcher Core**, a mini-framework to build **Compatibility Patchers**. These companion applications allow your favorite software (a framework, a library. . .) to keep long term API stability towards its ecosystem (plugins, bridges to other applications. . .), while still keeping the main codebase clean, and getting new features at a good pace.

Compatibility Patchers inject backward/forward compatibility shims (like class/function/attribute aliases, restore features which were dropped/externalized because “not used enough” or “outside the scope of the library”, and tweak the signatures and behaviour of callables (eg. for arguments which disappeared, or which became mandatory). It can even setup lazy “import aliases”, so that code can import a moved module both from its old and new location.

These shims allows you to upgrade your dependencies one at a time, when their maintainer finally had some time for a code update, or when missing features and bugfixes justify a fork. Most importantly, they allow you to not get stuck, when deadlines are tight, and crucial dependencies have conflicting expectations regarding the software version.

Note that compatibility Patchers are not supposed to undo changes related to security (default access permissions, markup escaping, cookie parameters. . .), because of the risks involved. Also, changes that only impact project-level code (eg. new mandatory *settings*) should not get patched, since it’s easier and cleaner for to simply update project code.

Technically, Compatibility Patchers are packages which manage a set of **fixers**, tiny utilities (often less than 10 lines of code) which advertise the change that they make, the software versions that they support, and which patch the target code on demand. By applying these fixers in a proper order (sometimes before, sometimes after the initialization of the patched software itself), Compatibility Patchers can easily “time travel”, and work around multiple breaking changes which target the same part of the code (e.g. a content handler being added and then removed).

Compat Patcher Core holds the core logice of that system, via easily extendable classes: a registry system, monkey-patching utilities, and a generic runner to be called at the very start of your application. It also contains a cookiecutter recipe to setup your own Compatibility Patcher in a few minutes, with test scaffolding and packaging metadata ready for launch.

Documentation is available on Read The Docs

Sources are available on Github

Which applications currently benefit from this system?

- The Django Web Framework via Django-Compat-Patcher

CHAPTER 1

Topics

1.1 Setup your own Compatibility Patcher

To make a new framework/library (here called the **software**) benefit from such a companion application, we recommend starting from the provided project skeleton.

It provides a standard structure for utilities and sets of fixers (preventing import loops), and activates several helpers: checking that every fixer has its test, checking that no stdlib "warnings.warn()" is directly used by compatibility shims, checking that Tox properly tests the setuptools-deployed package and not the one of the "src/" directory, generating a readme file with the list of available fixers…

It also provides basic metadata files for packaging, as well as Pytest/Tox integration, Travis integration for Continuous Integration of your Git repository..

1.1.1 Prerequisites

As usual, it is advised that you create and activate a separate python virtual environment before working an a new project.

Recent python versions will just need `python -m venv <env_name>`, while older ones will use `virtualenv <env_name>` after installing *virtualenv* package, or `mkvirtualenv <env_name>` if they prefer the commodities of *virtualenvwrapper*.

Be sure that your virtualenv is activated before proceeding (`pip freeze` should show an empty list of installed packages).

Then install cookiecutter as explained in their docs.

1.1.2 Deploying the project skeleton

The recipe for compatibility patchers is stored in the compat-patcher-core repository, so just do:

```
cookiecutter https://github.com/pakal/compat-patcher-core
```

Enter the requested information, in particuler the **project_prefix** which should be the short name of the software which you intend to provides comaptibility fixers for. By convention, your compatibility patcher should have a name in the form "<software>-compat-patcher".

Then *cd* into the newly created project, and explore the few files that were templated for you.

1.1.3 Launching tests

You can run `python setup.py test`, which will install test requirements in a temporary folder, and launch pytest.

But it is advised to install all dev dependencies in your virtualenv, and directly run `pytest`:

```
pip install -r requirements-dev.txt

pytest
```

In case of failed tests, you might want to tweak the *pytest.ini* file, or directly ask for more verbosity:

```
pytest -vl --tb=long -x
```

To test cross-version compatibility of your changes, run the `tox` tool from inside the repository; and ensure that eventually all combinations of Python and your to-be-patched framework are present in your *tox.ini* file.

1.1.4 Implementing software-specific patchers

Now begins the real value-added work:

- Adding your target software as a dependency, in *setup.py* and *requirements.txt*
- Fixing the implementation of *get_current_software_version()* (in *registry.py*) to fetch the real current version of said software
- Writing your first compatibility fixers, as well as their tests (look at the dummy *fix_behaviour_submodule_element()* and then remove it).
- Tweaking *README.in*, *CHANGELOG* and *CONTRIBUTING* according to your needs, and (re)generating the real *README.rst* with *generate_readme.py*
- Optionally modifying the root *patch()* function to fetch its default settings from the configuration files of your target framework

Note that if you don't plan to provide a standalone patcher, but an additional fixers registry for an existing Compatibility Patcher, you can remove all of the *patch()* functionality.

1.2 Guidelines for writing fixers

compat_patcher_core.PatchingRegistry.

- Follow PEP8 as much as you can, or use Black to solve the problem once and for all.
- Fixers have to be registered with the *`register_compatibility_fixer()`* method of a patching registry. Hint: you can be DRY by using `functools.partial()` to preconfigure a set of similar fixers.

- Fixers must have a docstring explaining what they do (what breaking changes they deal with), and tests checking that both old and new behaviour keep working. These tests must have a name matching the form "test_<fixer-name>()" to be recognized.
- Fixers should have a signature following the convention `fix_<kind>_<path>_<element>(utils)`, where:
  - `<kind>` is one of the following:
    - "deletion" (when an element was removed from sources and must be put back)
    - "behavior" (when a signature changed, when an optional arguments became mandatory…)
    - "outsourcing" (when a whole submodule was turned into an external python package)
  - `<path>` is the trail of submodules leading to the patched element (with dots replaced by underscores), **not** including the root module of the library
  - `<element>` is the element to be patched ; for clarity you should use the same character casing as the real element name
  - `utils` is the injected parameter, which exposes patching utilities
- Fixers should be coded quite *defensively*, since they are highly expected to repair breakages, not introduce new ones.
- Fixers should follow these rules of good behaviour:
  - Use monkey-patching utilities from the injected `utils` object, not direct assignments
  - Not do logging or warnings by themselves either, but use the injected `utils` object once again (or a dedicated WarningsProxy instance)
  - Not do global imports of software submodules or external libraries, but import them *locally* inside the fixer function
- Fixers should not reintroduce code which raises security issues (like XSS attacks…), unless these fixers are OFF by default, deeply documented, and risks can be properly mitigated by project maintainers.
- Fixers should not be created to modify project-level code, like framework settings, since these are supposed to be easily updatable by project maintainers.

1.3 API

1.3.1 Patching launcher

Each package implementing a compat-patcher for a specific framework/library should expose a *patch()* function at the top level, responsible for the whole process of retrieving configuration, instantiating miscellaneous utilities, selecting relevant fixers, and applying them.

These building blocks are exposed to ease the process fo creating this launcher.

`compat_patcher_core.`**`generic_patch_software`**(*settings*, *patching_registry*, *patching_utilities_class=<class 'compat_patcher_core.utilities.PatchingUtilities'>*, *patching_runner_class=<class 'compat_patcher_core.runner.PatchingRunner'>*, *warnings_proxy=None*)

Load all dependencies, and apply relevant fixers from the *patching_registry*, according to the settings of the provided *settings*.

You can provide custom classes to be instantiated instead of default ones, and/or an existing WarningsProxy which will be updated with the new settings as soon as possible.

`compat_patcher_core.`**`make_safe_patcher`**(*f*)

This decorator makes a patching launcher thread-safe with a recursive lock.

Other checks and misc. features might be added in the future, so packages using this patching framework should always decorate their main "patch()" entrypoint with this utility.

1.3.2 Patching registry

`class` `compat_patcher_core.`**`PatchingRegistry`**(*family_prefix*, *populate_callable=None*, *current_software_version=None*)

This registry is used to store and select a set of fixers related to some specific software.

family_prefix will be used to constructe family names, along with the software reference version provided by the fixer.

populate_callable, if provided, is a callable taking the registry as first argument, and which will be called by *populate()*.

current_software_version may be a version tuple or a string. If it's None, then an override value will have to be provided when calling *get_relevant_fixers*.

`get_all_fixers`()

Return the list of all fixers (as dicts) known by this registry.

`get_fixer_by_id`(*fixer_id*)

Return the fixer having this (unqualified) ID, or raise KeyError.

`get_relevant_fixer_ids`(*qualified=False*, ***kwargs*)

"Same as *get_relevant_fixers*, but only returns IDs of selected fixers.

If *qualified* is True, returns a fixers IDs dot-prefixed with the family name.

`get_relevant_fixers`(*include_fixer_ids='*'*, *include_fixer_families=None*, *exclude_fixer_ids=None*, *exclude_fixer_families=None*, *current_software_version=None*, *log=None*)

Return the list of fixers (as dicts) to be applied for the target software version, based on the metadata of fixers, as well as inclusion/exclusion lists provided as arguments.

For inclusion/exclusion filters, a special "*" value means "all fixers", else a list of strings is expected.

An output callable *log* may be provided, expecting a string as argument, to debug the reasons why some fixers weren't selected.

This method forces a populate() on the registry.

`populate`()

Trigger the registration of potential lazy fixers, which might be in other submodules, or waiting in factory functions.

`register_compatibility_fixer`(*fixer_reference_version*, *fixer_applied_from_version=None*, *fixer_applied_upto_version=None*, *feature_supported_from_version=None*, *feature_supported_upto_version=None*, *fixer_tags=None*)

Register a compatibility fixer, which will be activated only if current software version is >= *fixer_applied_from_version* and < *fixer_applied_upto_version* (let them be None to have no limit).

The "fixer_reference_version" parameters identifies the software version where the breaking change was introduced (for backwards compatibility fixers), or where the new feature was introduced (for forwards compatibility fixers). It is not related to the appearance of corresponding DeprecationWarnings in the software. It is also used to sort fixers when applying them, and to generate the name of the family of fixers concerned.

feature_supported_from_version (included) and *feature_supported_upto_version* (excluded) may be used to limit the range of software versions for which related unit-tests are expected to work (i.e versions for which the feature is available, either as a monkey-paching or as standard code).

Version identifiers must be dotted strings, eg. "1.9.1". None means "no limit" here.

fixer_tags is a **list** of strings, which can be used to differentiate fixers which will be applied at different moments of software startup.

class `compat_patcher_core.`**`MultiPatchingRegistry`**(*registries*)

This patching registry wraps a list of other registries, each having its own fixers and current software version.

It concatenates and returns selected fixers on demand, assuming that they are compatible with each other.

`get_all_fixers`(**args*, ***kwargs*)

Return the concatenation of all fixers of underlying registries.

`get_fixer_by_id`(*fixer_id*, **args*, ***kwargs*)

In case of duplicate fixers having the same ID, just return the first one.

`get_relevant_fixer_ids`(*qualified=False*, ***kwargs*)

"Same as *get_relevant_fixers*, but only returns IDs of selected fixers.

If *qualified* is True, returns a fixers IDs dot-prefixed with the family name.

`get_relevant_fixers`(**args*, ***kwargs*)

Populate underlying registries, and return the concatenation of their selected fixers.

Forcing a *current_software_version* as parameter of this method is still possible, but beware that underlying registries all deal with the same software stack, in this case.

1.3.3 Patching utilities

class `compat_patcher_core.`**`PatchingUtilities`**(*settings*)

An instance of this class is provided as first argument to each compatibility fixer applied.

It provides handy tools to monkey-patch the software environment, and emit logs and Warnings in a controllable way.

For better forward-compatibility, please call injection utilities though keyword arguments, and not positional ones (due to python2.7 support, it's not enforced yet).

`settings_keys_used = ['logging_level', 'enable_warnings', 'patch_injected_objects']`

`apply_settings`(*settings*)

This method can be called at runtime, mainly to alter the emission of logs and Warnings by fixers. it's possible to provide only a subset of settings, the others remaining as is.

`emit_log`(*message*, *level='INFO'*)

A logger printing to stderr, since at some stages of patching, logging is not yet setup.

Log is only output if *level* is gerater or equal the current *logging_level* setting.

`emit_warning`(*message*, *category=<class 'DeprecationWarning'>*, *stacklevel=1*)

Similar to "warnings.warn()" of the stdlib, but only emits the Warning if *enable_warnings* setting is True.

`inject_attribute`(*target_object*, *target_attrname*, *attribute*)

Inject an attribute into an object of any type (module, class, instance…).

Parameters

- **`target_object`** – The object to patch
- **`target_attrname`** – The name given to the new attribute in the object to patch
- **`attribute`** – The attribute to inject, which must not be a callable

`inject_callable`(*target_object*, *target_callable_name*, *patch_callable*)
Inject a simple callable (not a class) into an object of any type (module, class, instance. . .).

Parameters

- **`target_object`** – The object to patch
- **`target_callable_name`** – The name given to the new callable in the object to patch
- **`patch_callable`** – The callable to inject, which must be a callable, but not a class

`inject_callable_alias`(*target_object*, *target_attrname*, *source_object*, *source_attrname*)
Create and inject an alias for the source callable (not a class), which also triggers a deprecation warning when called.

Returns the created alias callable.

Parameters

- **`target_object`** – The object to patch
- **`target_attrname`** – The name of the callable on the target object
- **`source_object`** – The object from which to get the callable
- **`source_attrname`** – The name of the callable on the source object

`inject_class`(*target_object*, *target_klassname*, *klass*)
Inject a class into an object of any type (module, class, instance. . .).

Parameters

- **`target_object`** – The object to patch
- **`target_klassname`** – The name given to the new class in the object to patch
- **`klass`** – The class to inject

`inject_import_alias`(*alias_name*, *real_name*)
Create an import alias for the selected module.

This doesn't directly patch sys.modules, but instead uses the imports hooks of python, so that "import <alias_name>" and "import <real_name>" both end up importing the same the (sub)module object.

When the alias is a submodule, its parent module(s) must exist too.

Parameters

- **`alias_name`** – The dotted name of the alias module
- **`real_name`** – The dotted name of the real module

`inject_module`(*target_module_name*, *module*)
Inject a module in sys.modules, under the selected dotted name.

When injecting a submodule, its parent module(s) must exist too.

Parameters

- **`target_module_name`** – The dotted name of the new module in sys.modules
- **`module`** – The module object to inject

class `compat_patcher_core.`**`WarningsProxy`**

An instance of this class acts as a replacement for the stdlib "warnings" package, but it relies on a PatchingUtilities instance as soon as this one is provided - thus making Warnings controllable by compat patcher settings.

`compat_patcher_core.`**`tuplify_software_version`**(*version*)

Coerces the version string (if not None), to a version tuple. E.g. "1.7.0" becomes (1, 7, 0). No version suffix like "alpha" is expected.

`compat_patcher_core.`**`detuplify_software_version`**(*version*)

Coerces the version tuple (if not None), to a version string. E.g. (1, 7, 0) becomes "1.7.0".

1.3.4 Patching runner

class `compat_patcher_core.`**`PatchingRunner`**(*settings*, *patching_registry*, *patching_utilities*)

This class is in charge of fetching relevant fixers from the registry, and applying them in a proper order, while skipping those who have already been applied (i.e same family name and ID) and those which raised SkipFixerException.

`patch_software`()

Patch the software according to plans.

Return a dict with, at least field "fixers_just_applied", the list of fixers that were successfully applied during this call.

1.3.5 Patching exceptions

class `compat_patcher_core.`**`SkipFixerException`**

Exception to signal a fixer which is not applicable in that project context.

1.3.6 Patching configuration

The *settings* expected by classes above must be a dict-like object (just a __getitem__() method is enough), which raises KeyError if a setting is not found.

`compat_patcher_core.`**`DEFAULT_SETTINGS = {'enable_warnings': True, 'exclude_fixer_families'`**

Example configuration to copy() and adapt

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Index