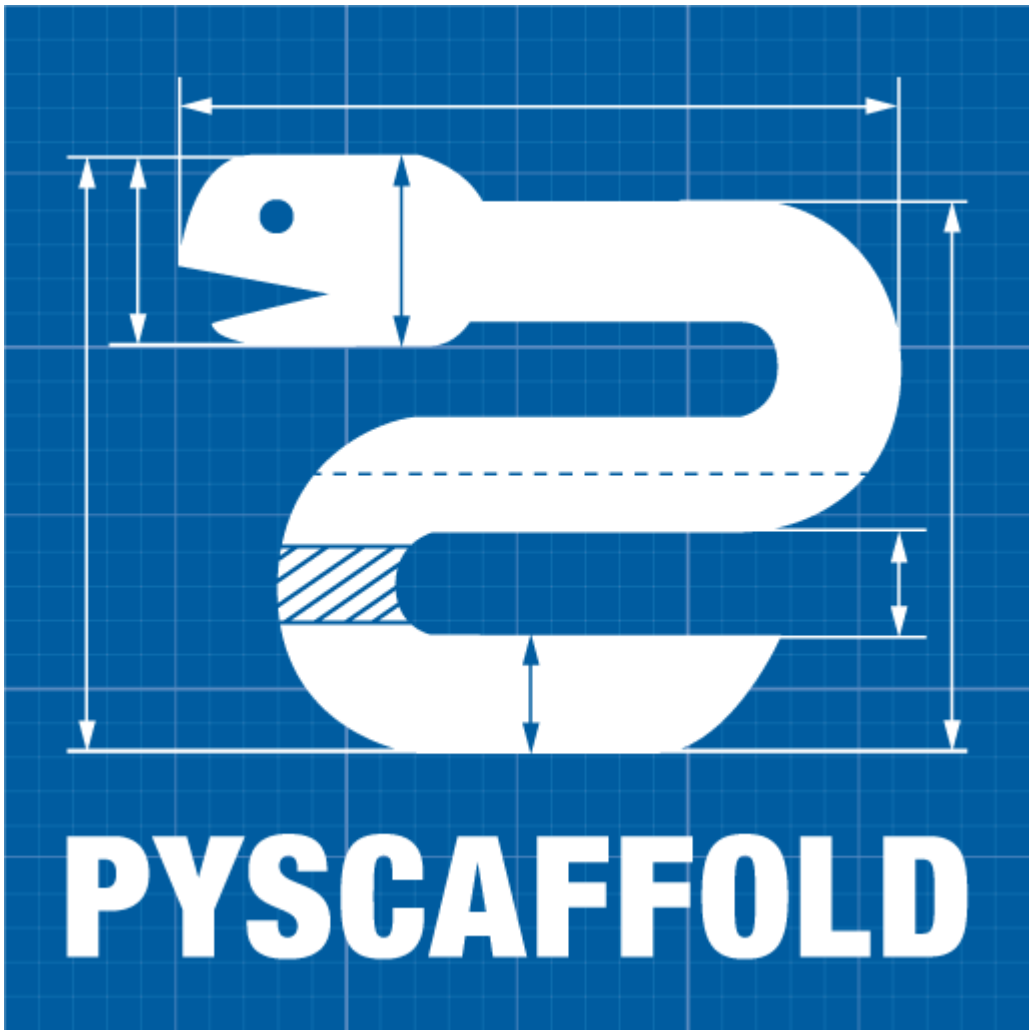


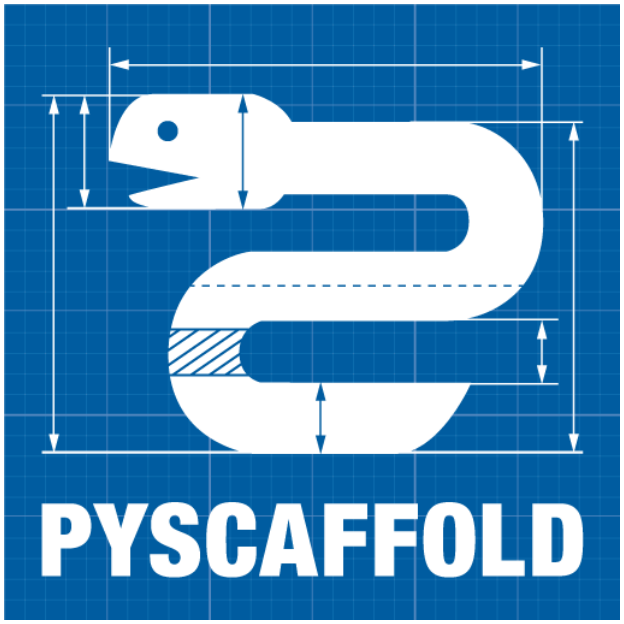
PyScaffold Documentation

Release 4.1.1

Blue Yonder

Oct 18, 2021

CONTENTS



PyScaffold is a project generator for bootstrapping high-quality Python packages, ready to be shared on PyPI and installable via pip. It is easy to use and encourages the adoption of the best tools and practices of the Python ecosystem, helping you and your team to stay sane, happy and productive. The best part? It is stable and has been used by thousands of developers for over half a decade!

Note: This document refers to the latest version of PyScaffold (v4). Please refer to v3.3 for the previous stable version. Both versions are compatible with Python 3.6 and greater.

For legacy Python 2.7 please install PyScaffold 2.5 *(not officially supported)*.

CHAPTER

ONE

INSTALLATION

1.1 Requirements

The installation of PyScaffold only requires a recent version of of setuptools, (at least version 46.1), pip, as well as a working installation of Git (meaning at least your *name and email were configured* in your first-time git setup). Especially Windows users should make sure that the command `git` is available on the command line. Otherwise, check and update your `PATH` environment variable or run PyScaffold from the *Git Bash*.

Tip: It is recommended to use an isolated development environment as provided by virtualenv or conda for your work with Python in general. You might want to install PyScaffold globally in your system, but consider using virtual environments when developing your packages.

1.2 Installation

PyScaffold relies on a Python package manager for its installation. The easiest way of getting started is via our loved pip. Make sure you have `pip` installed[1], then simply type:

```
pip install --upgrade pyscaffold
```

to get the latest stable version. The most recent development version can be installed with:

```
pip install --pre --upgrade pyscaffold
```

Using `pip` also has the advantage that all requirements are automatically installed.

If you want to install PyScaffold with all official extensions, run:

```
pip install --upgrade pyscaffold[all]
```

[1] In some operating systems, e.g. Ubuntu, this means installing a `python3-pip` package or similar via the OS's global package manager.

1.3 Alternative Methods

It is very easy to get PyScaffold installed with pip, but some people do prefer other package managers such as conda while doing their work.

If you do lots of number crunching or data science in general[2] and you already rely on conda-forge packages, you might also use the following method:

```
conda install -c conda-forge pyscaffold
```

It is also very common for developers to have more then one Python version installed on their machines, and a plethora of virtual environments spread all over the place… Instead of constantly re-installing PyScaffold in each one of these installations and virtual environments, you can use pipx to do a "minimally-invasive" system-wide installation and have the `putup` command always available independently of which Python you are using:

```
pipx install pyscaffold
```

Please check the documentation of each tool to understand how they work with extra requirements (e.g. `[all]`) or how to add extensions (e.g. `pipx inject pyscaffold pyscaffoldext-dsproject`).

1.4 Additional Requirements

We strongly recommend installing tox together with PyScaffold (both can be installed with pip, conda or pipx), so you can take advantage of its automation capabilities and avoid having to install dependencies/requirements manually. If you do that, just by running the commands `tox` and `tox -e docs`, you should able to run your tests or build your docs out of the box (a list with all the available tasks is obtained via the `tox -av` command).

If you dislike tox, or are having problems with it, you can run commands (like `pytest` and `make -C docs`) manually within your project, but then you will have to deal with additional requirements and dependencies yourself. It might be the case you are already have them installed but this can be confusing because these packages won't be available to other packages when you use a virtual environment. If that is the case, just install following packages inside the environment you are using for development:

- Sphinx
- pytest
- pytest-cov

Note: If you have problems using PyScaffold, please make sure you are using Python 3.6 or greater.

---

[2] conda is a very competent package manager for Python, not only when you have to deal with numbers. In general, when you rely on native extensions, hardware acceleration or lower level programming languages integration (such as C or C++), conda might just be the tool you are looking for.

CHAPTER

TWO

USAGE & EXAMPLES

2.1 Quickstart

A single command is all you need to quickly start coding like a Python rockstar, skipping all the difficult and tedious bits:

```
putup my_project
```

This will create a new folder called `my_project` containing a *perfect project template* with everything you need for getting things done. Checkout out this demo project, which was set up using Pyscaffold.

Tip: New in version 4.0: We are trying out a brand new *interactive mode* that makes it even easier to use PyScaffold in its full potential. If you want to give it a shot, use the `--interactive`, or simply `-i` option.

The interactive command equivalent to the previous example is: `putup -i my_project`.

You can `cd` into your new project and interact with it from the command line after creating (or activating) an isolated development environment (with virtualenv, conda or your preferred tool), and performing the usual editable install:

```
pip install -e .
```

... all set and ready to go! Try the following in a Python shell:

```
>>> from my_project.skeleton import fib
>>> fib(10)
55
```

Or if you are concerned about performing package maintainer tasks, make sure to have tox installed and see what we have prepared for you out of the box:

```
tox -e docs  # to build your documentation
tox -e build  # to build your package distribution
tox -e publish  # to test your project uploads correctly in test.pypi.org
tox -e publish -- --repository pypi  # to release your package to PyPI
tox -av  # to list all the tasks available
```

The following figure demonstrates the usage of `putup` with the new experimental interactive mode for setting up a simple project. It uses the *–cirrus* flag to add CI support (via Cirrus CI), and tox to run automated project tasks like building a package file for distribution (or publishing).

Type `putup -h` to learn about *other things PyScaffold can do* for your project, and if you are not convinced yet, have a look on these *reasons to use PyScaffold*.

There is also a video tutorial on how to develop a command-line application with the help of PyScaffold.

2.1.1 Notes

1. PyScaffold's project template makes use of a dedicated `src` folder to store all the package files meant for distribution (additional files like tests and documentation are kept in their own separated folders). You can find some comments and useful links about this design decision in our *FAQ*.
2. The `pip install -e .` command installs your project in editable mode, making it available in import statements as any other Python module. It might fail if your have an old version of Python's package manager and tooling in your current environment. Please make sure you are using the intended environment (either a virtual environment [*recommended*] or the default installation of Python in the operating system) and try to update them with `python -m pip install -U pip setuptools`.
3. If you are using a virtual environment, please remember to re-activate it everytime you close your shell, otherwise you will not be able to import your project in the REPL. To check if you have already activated it you can run `which python` on Linux and OSX, `where python` on the classical Windows command prompt, or `Get-Command python` on PowerShell.

2.2 Examples

Just a few examples to get you an idea of how easy PyScaffold is to use:

`putup my_little_project` The simplest way of using PyScaffold. A directory `my_little_project` is created with a Python package named exactly the same. The MIT license will be used.

`putup -i my_little_project` If you are unsure on how to use PyScaffold, or keep typing `putup --help` all the time, the **experimental** `--interactive` (or simply `-i`), is your best friend. It will open your default text editor with a file containing examples and explanations on how to use `putup` (think of it as an "editable" `--help` text, once the file is saved and closed all the values you leave there are processed by PyScaffold). You might find some familiarities in the way this option works with `git rebase -i`, including the capacity of choosing a different text editor by setting the `EDITOR` (or `VISUAL`) environment variable in your terminal.

`putup skynet -l GPL-3.0-only -d "Finally, the ultimate AI!" -u https://sky.net` This will create a project and package named *skynet* licensed under the GPL3. The *description* inside `setup.cfg` is directly set to "Finally, the ultimate AI!" and the homepage to `https://sky.net`.

`putup Scikit-Gravity -p skgravity -l BSD-3-Clause` This will create a project named *Scikit-Gravity* but the package will be named *skgravity* with license new-BSD[1].

`putup youtub --django --pre-commit -d "Ultimate video site for hot tub fans"` This will create a web project and package named *youtub* that also includes the files created by Django's `django-admin`[2]. The *description* in `setup.cfg` will be set and a file `.pre-commit-config.yaml` is created with a default setup for pre-commit.

`putup thoroughly_tested --cirrus` This will create a project and package *thoroughly_tested* with files `tox.ini` and `.cirrus.yml` for tox and Cirrus CI.

`putup my_zope_subpackage --name my-zope-subpackage --namespace zope --package subpackage` This will create a project under the `my_zope_subpackage` directory with the *installation name* of

[1] Notice the usage of SPDX identifiers for specifying the license in the CLI
[2] Requires the installation of pyscaffoldext-django.

`my-zope-subpackage` (this is the name used by pip and PyPI), but with the following corresponding import statement:

```
from zope import subpackage
# zope is the namespace and subpackage is the package name
```

To be honest, there is really only the Zope project that comes to my mind which is using this exotic feature of Python's packaging system. Chances are high, that you will never ever need a namespace package in your life. To learn more about namespaces in the Python ecosystem, check PEP 420.

2.3 Package Configuration

Projects set up with PyScaffold rely on setuptools, and therefore can be easily configured/customised via `setup.cfg`. Check out the example below:

```
# Docs on setup.cfg:
# http://setuptools.readthedocs.io/en/latest/setuptools.html#configuring-setup-using-
↪setup-cfg-files

[metadata]
name = my_project
description = A test project that was set up with PyScaffold
author = Florian Wilhelm
author_email = Florian.Wilhelm@blue-yonder.com
license = MIT
url = https://...
long_description = file: README.rst
platforms = any
classifiers =
    Development Status :: 5 - Production/Stable
    Topic :: Utilities
    Programming Language :: Python
    Programming Language :: Python :: 3
    Environment :: Console
    Intended Audience :: Developers
    License :: OSI Approved :: MIT License
    Operating System :: POSIX :: Linux
    Operating System :: Unix
    Operating System :: MacOS
    Operating System :: Microsoft :: Windows

[options]
zip_safe = False
packages = find_namespace:
python_requires = >=3.6
include_package_data = True
package_dir =
    =src
# Add here dependencies of your project (semicolon/line-separated)
install_requires =
    pandas
    scikit-learn
```

(continues on next page)

(continued from previous page)

```
[options.packages.find]
where = src
exclude =
    tests

[options.extras_require]
# Add here additional requirements for extra features, like:
# pdf = ReportLab>=1.2; RXP
# rest = docutils>=0.3; pack ==1.1, ==1.3
all = django; cookiecutter
# Add here test requirements (semicolon/line-separated)
testing =
    pytest
    pytest-cov

[options.entry_points]
# Add here console scripts like:
# console_scripts =
#     script_name = ${package}.module:function
# For example:
# console_scripts =
#     fibonacci = ${package}.skeleton:run
# And any other entry points, for example:
# pyscaffold.cli =
#     awesome = pyscaffoldext.awesome.extension:AwesomeExtension

[tool:pytest]
# Options for py.test:
# Specify command line options as you would do when invoking py.test directly.
# e.g. --cov-report html (or xml) for html/xml output or --junitxml junit.xml
# in order to write a coverage file that can be read by Jenkins.
addopts =
    --cov my_project --cov-report term-missing
    --verbose
norecursedirs =
    dist
    build
    .tox
testpaths = tests
markers =
    slow: mark tests as slow (deselect with '-m "not slow"')

[bdist_wheel]
universal = 1

[devpi:upload]
# Options for the devpi: PyPI server and packaging tool
# VCS export must be deactivated since we are using setuptools-scm
no_vcs = 1
formats =
    sdist
```

(continues on next page)

(continued from previous page)

```
    bdist_wheel

[flake8]
# Some sane defaults for the code style checker flake8
max_line_length = 88
extend_ignore = E203, W503
# ^  Black-compatible
#    E203 and W503 have edge cases handled by black
exclude =
    .tox
    build
    dist
    .eggs
    docs/conf.py

[pyscaffold]
# PyScaffold's parameters when the project was created.
# This will be used when updating. Do not change!
version = 4.0
package = my_package
extensions =
    namespace
namespace = ns1.ns2
```

You might also want to have a look on pyproject.toml for specifying dependencies required during the build:

```
[build-system]
# AVOID CHANGING REQUIRES: IT WILL BE UPDATED BY PYSCAFFOLD!
requires = ["setuptools>=46.1.0", "setuptools_scm[toml]>=5", "wheel"]
build-backend = "setuptools.build_meta"

[tool.setuptools_scm]
# See configuration details in https://github.com/pypa/setuptools_scm
version_scheme = "no-guess-dev"
```

Please note PyScaffold will add some internal information to `setup.cfg`, we do that to make updates a little smarter.

Note: To avoid splitting the configuration and build parameters among several files, PyScaffold uses the same file as setuptools (`setup.cfg`). Storing configuration in pyproject.toml is not supported. In the future, if the default build metadata location changes (as proposed by PEP 621), PyScaffold will follow the same pattern.

2.4 PyScaffold's Own Configuration

PyScaffold also allows you to save your favourite configuration to a file that will be automatically read every time you run `putup`, this way you can avoid always retyping the same command line options.

The locations of the configuration files vary slightly across platforms, but in general the following rule applies:

- Linux: `$XDG_CONFIG_HOME/pyscaffold/default.cfg` with fallback to `~/.config/pyscaffold/default.cfg`
- OSX: `~/Library/Preferences/pyscaffold/default.cfg`
- Windows(7): `%APPDATA%\pyscaffold\pyscaffold\default.cfg`

The file format resembles the `setup.cfg` generated automatically by PyScaffold, but with only the `metadata` and `pyscaffold` sections, for example:

```
[metadata]
author = John Doe
author-email = john.joe@gmail.com
license = MPL-2.0

[pyscaffold]
extensions =
    cirrus
    pre-commit
```

With this file in place, typing only:

```
$ putup myproj
```

will have the same effect as if you had typed:

```
$ putup --license MPL-2.0 --cirrus --pre-commit myproj
```

Note: For the time being, only the following options are allowed in the config file:

- **metadata** section: `author`, `author-email` and `license`
- **pyscaffold** section: `extensions` (and associated opts)

Options associated with extensions are the ones prefixed by an extension name.

To prevent PyScaffold from reading an existing config file, you can pass the `--no-config` option in the CLI. You can also save the given options when creating a new project with the `--save-config` option. Finally, to read the configurations from a location other then the default, use the `--config PATH` option. See `putup --help` for more details.

Warning: *Experimental Feature* - We are still evaluating how this new and exciting feature will work, so its API (including file format and name) is not considered stable and might change between minor versions. As previously stated, if the configuration file for setuptools changes (e.g. with PEP 621), PyScaffold will follow that and change its own configuration.

This means that in future versions, PyScaffold will likely adopt a more *pyproject.toml*-style configuration (and as a consequence the file name and extension might change).

CHAPTER

THREE

ADVANCED USAGE & FEATURES

3.1 Dependency Management

Warning: *Experimental Feature* - PyScaffold support for virtual environment management is experimental and might change in the future.

3.1.1 Foundations

The greatest advantage in packaging Python code (when compared to other forms of distributing programs and libraries) is that packages allow us to stand on the shoulders of giants: you don't need to implement everything by yourself, you can just declare dependencies on third-party packages and `setuptools`, `pip`, PyPI and their friends will do the heavy lifting for you.

Of course, with great power comes great responsibility. Package authors must be careful when declaring the versions of the packages they depend on, so the people consuming the final work can do reliable installations, without facing dependency hell. In the opensource community, two main strategies have emerged in the last few years:

- the first one is called **abstract** and consists of having permissive, minimal and generic dependencies, with versions specified by ranges, so anyone can install the package without many conflicts, sharing and reusing as much as possible dependencies that are already installed or are also required by other packages
- the second, called **concrete**, consists of having strict dependencies, with pinned versions, so all the users will have repeatable installations

Both approaches have advantages and disadvantages, and usually are used together in different phases of a project. As a rule of thumb, libraries tend to emphasize abstract dependencies (but can still have concrete dependencies for the development environment), while applications tend to rely on concrete dependencies (but can still have abstract dependencies specially if they are intended to be distributed via PyPI, e.g. command line tools and auxiliary WSGI apps/middleware to be mounted inside other domain-centric apps). For more information about this topic check Donald Stufft post.

Since PyScaffold aims the development of Python projects that can be easily packaged and distributed using the standard PyPI and `pip` flow, we adopt the specification of **abstract dependencies** using `setuptools' install_requires`. This basically means that if PyScaffold generated projects specify dependencies inside the `setup.cfg` file (using general version ranges), everything will work as expected.

3.1.2 Test Dependencies

While specifying the final dependencies for packages is pretty much straightforward (you just have to use `install_requires` inside `setup.cfg`), dependencies for running the tests can be a little bit trick.

Historically, `setuptools` provides a `tests_require` field that follows the same convention as `install_requires`, however this field is not strictly enforced, and `setuptools` doesn't really do much to enforce the packages listed will be installed before the test suite runs.

PyScaffold's recommendation is to create a `testing` field (actually you can name it whatever you want, but let's be explicit!) inside the `[options.extras_require]` section of `setup.cfg`. This way multiple test runners can have a centralised configuration and authors can avoid double bookkeeping.

If you use `tox` (recommended), you can list `testing` under the the `extras` configuration field option (PyScaffold template for `tox.ini` already takes care of this configuration for you).

If running `pytest` directly, you will have to install those dependencies manually, or do a editable install of your package with `pip install -e .[testing]`.

Tip: If you prefer to use just `tox` and keep everything inside `tox.ini`, please go ahead and move your test dependencies. Every should work just fine :)

Note: PyScaffold strongly advocates the use of test runners to guarantee your project is correctly packaged/works in isolated environments. New projects will ship with a default `tox.ini` file that is a good starting point, with a few useful tasks. Run `tox -av` to list all the available tasks.

3.1.3 Basic Virtualenv

As previously mentioned, PyScaffold will get you covered when specifying the **abstract** or test dependencies of your package. We provide sensible configurations for `setuptools` and `tox` out-of-the-box. In most of the cases this is enough, since developers in the Python community are used to rely on tools like `virtualenv` and have a workflow that take advantage of such configurations. As an example, you could do:

```
$ pip install pyscaffold
$ putup myproj
$ cd myproj
$ virtualenv .venv
# OR python -m venv .venv
$ source .venv/bin/activate
$ pip install -U pip setuptools setuptools_scm tox
# ... edit setup.cfg to add dependencies ...
$ pip install -e .
$ tox
```

However, someone could argue that this process is pretty manual and laborious to maintain specially when the developer changes the **abstract** dependencies.

PyScaffold can alleviate this pain a little bit with the *venv* extension:

```
$ putup myproj --venv --venv-install PACKAGE
# Is equivalent of running:
#
```

(continues on next page)

(continued from previous page)

```
#     putup myproj
#     cd myproj
#     virtualenv .venv OR python -m venv .venv
#     pip install PACKAGE
```

But it is still desirable to keep track of the version of each item in the dependency graph, so the developer can have environment reproducibility when trying to use another machine or discuss bugs with colleagues.

In the following sections, we describe how to use a few popular command line tools, supported by PyScaffold, to tackle these issues.

Tip: When called with the `--venv` option, PyScaffold will try first to use `virtualenv` (there are some advantages on using it, such as being faster), and if it is not installed, will fallback to Python stdlib's `venv`. Plese notice however that even `venv` might not be available by default in your system: some OS/distributions split Python's stdlib in several packages and require the user to explicitly install them (e.g. Ubuntu will require you to do `apt install python3-venv`). If you run into problems, try installing `virtualenv` and run the command again.

3.1.4 Integration with Pipenv

We can think in Pipenv as a virtual environment manager. It creates per-project virtualenvs and generates a `Pipfile.lock` file that contains a precise description of the dependency tree and enables re-creating the exact same environment elsewhere.

Pipenv supports two different sets of dependencies: the default one, and the *dev* set. The default set is meant to store runtime dependencies while the dev set is meant to store dependencies that are used only during development.

This separation can be directly mapped to PyScaffold strategy: basically the default set should mimic the `install_requires` option in `setup.cfg`, while the dev set should contain things like `tox`, `sphinx`, `pre-commit`, `ptpython` or any other tool the developer uses while developing.

Tip: Test dependencies are internally managed by the test runner, so we don't have to tell Pipenv about them.

The easiest way of doing so is to add a `-e .` dependency (in resemblance with the non-automated workflow) in the default set, and all the other ones in the dev set. After using Pipenv, you should add both `Pipfile` and `Pipfile.lock` to your git repository to achieve reproducibility (maintaining a single `Pipfile.lock` shared by all the developers in the same project can save you some hours of sleep).

In a nutshell, PyScaffold+Pipenv workflow looks like:

```
$ pip install pyscaffold pipenv
$ putup myproj
$ cd myproj
# ... edit setup.cfg to add dependencies ...
$ pipenv install
$ pipenv install -e .  # proxy setup.cfg install_requires
$ pipenv install --dev tox sphinx  # etc
$ pipenv run tox       # use `pipenv run` to access tools inside env
$ pipenv lock          # to generate Pipfile.lock
$ git add Pipfile Pipfile.lock
```

After adding dependencies in `setup.cfg`, you can run `pipenv update` to add them to your virtual environment.

Warning: *Experimental Feature* - Pipenv is still a young project that is moving very fast. Changes in the way developers can use it are expected in the near future, and therefore PyScaffold support might change as well.

3.1.5 Integration with pip-tools

Contrary to Pipenv, `pip-tools` does not replace entirely the aforementioned "manual" workflow. Instead, it provides lower level command line tools that can be integrated to it, in order to achieve better reproducibility.

The idea here is that you have two types files describing your dependencies: `*requirements.in` and `*requirements.txt`. The `.in` files are the ones used to list **abstract** dependencies, while the `.txt` files are generated by running `pip-compile`.

Again the easiest way of having the `requirements.in` file to mimic `setup.cfg`' `install_requires` is to add *something like* `-e .` to it.

Warning: For the time being adding `-e file:.` is a working solution that is tested by `pip-tools` team (`-e .` will generate absolute file paths in the compiled file, which will make it impossible to share). However this situation might change in the near future. You can find more details about this topic and monitor any changes in https://github.com/jazzband/pip-tools/issues/204.

When using `-e file:.` in your `requirements.in` file, the compiled `requirements.txt` needs to be installed via `pip-sync` instead of `pip install -r requirements.txt`

You can also create multiple environments and have multiple *"profiles"*, by using different files, e.g. `dev-requirements.in` or `ci-requirements.in`, but keeping it simple and using `requirements.in` to represent all the tools you need to run common tasks in a development environment is a good practice, since you can omit the arguments when calling `pip-compile` and `pip-sync`. After all, if you need to have a separated test environment you can use tox, and the minimal dependencies of your packages are already listed in `setup.cfg`.

Note: The existence of a `requirements.txt` file in the root of your repository does not imply all the packages listed there will be considered direct dependencies of your package. This was valid for older versions of PyScaffold (3), but is no longer the case. If the file exists, it is completely ignored by PyScaffold and setuptools.

A simple a PyScaffold + `pip-tools` workflow looks like:

```
$ putup myproj --venv --venv-install pip-tools setuptools_scm && cd myproj
$ source .venv/bin/activate
# ... edit setup.cfg to add dependencies ...
$ echo '-e file:.' > requirements.in
$ echo -e 'tox\nsphinx\nptpython' >> requirements.in  # etc
$ pip-compile
$ pip-sync
$ tox
# ... do some debugging/live experimentation running Python in the terminal
$ ptpython
$ git add *requirements.{in,txt}
```

After adding dependencies in `setup.cfg` (or to `requirements.in`), you can run `pip-compile && pip-sync` to add them to your virtual environment. If you want to add a dependency to the dev environment only, you can also:

```
$ echo "mydep>=1.2,<=2" >> requirements.in && pip-compile && pip-sync
```

Warning: *Experimental Feature* - the methods described here for integrating `pip-tools` and PyScaffold in a single workflow are tested to a certain degree and not considered stable. The usage of relative paths in the compiled `requirements.txt` file is a feature that have being several years in the making and still is under discussion. As everything in Python's packaging ecosystem right now, the implementation, APIs and specs might change in the future so it is up to the user to keep an eye on the official docs and use the logic explained here to achieve the expected results with the most up-to-date API `pip-tools` have to offer.

The issue https://github.com/jazzband/pip-tools/issues/204 is worth following.

If you find that the procedure here no longer works, please open an issue on https://github.com/pyscaffold/pyscaffold/issues.

3.1.6 Integration with conda

Conda is an open-source package manager very popular in the Python ecosystem that can be used as an alternative to pip. It is especially helpful when distributing packages that rely on compiled libraries (e.g. when you need to use some C code to achieve performance improvements) and uses Anaconda as its standard repository (the PyPI equivalent in the conda world).

The main advantage of conda compared to `virtualenv`/`venv` based tools is that it unifies several different tools and has a deeper isolation than the pip package manager. For instance conda allows you to create isolated environments by specifying also the Python version and even system libraries like `glibc`. In the pip ecosystem, one needs a tool like pyenv to choose the Python version and the installation of system libraries besides the current ones is not possible at all.

Note: Unfortunately, since conda environments are more complex and feature-rich than the ones produced by `virtualenv`/`venv` based tools, package installations usually take longer. If all your dependencies are pure Python packages and you don't need to use any compiled libraries, `virtualenv`/`venv` might provide a faster dev experience.

To use conda with a project setup generated by PyScaffold just:

1. Create a file `environment.yml`, e.g. like this example for data science projects. Note that `name: my_conda_env` defines the name of the environment. Also note that besides the conda dependencies you can still add pip-installable packages by adding `- pip` as dependency and a section defining additional packages as well as the project setup itself:

   ```
   - pip:
     - -e .
     - other-pip-based-package
   ```

   This will install your project as well as `other-pip-based-package` within the conda environment. Be careful though that some pip-based packages might not work perfectly within a conda environment but this concerns only certain packages that tamper with the environment itself like tox for instance. As a rule of thumb, always define a requirement as conda package if available and only resort to pip packages if not available as conda package.

2. Create an environment based on this file with:

   ```
   conda env create -f environment.yml
   ```

Tip: Mamba is a new and much faster drop-in replacemenet for conda. For large environments, conda often requires several minutes or hours to solve dependencies while mamba normally completes within seconds.

To create an environment with mamba, you can run the following command:

```
mamba env create -f environment.yml
```

3. Activate the environment with:

```
conda activate my_conda_env
```

You can read more about conda in the excellent guide writen by WhiteBox. Also checkout the PyScaffold's dsproject extension that already comes with a proper `environment.yml`.

Creating a conda package

The process of creating conda packages consists basically in creating some extra files that describe general recipe to build your project in different operating systems. These recipe files can in theory coexist within the same repository as generated by PyScaffold.

While this approach is completely fine and works well, a package uploaded by a regular user to Anaconda will not be available if someone simply try to install it via `conda install <pkg name>`. This happens because Anaconda and conda are organised in terms of channels and regular users cannot upload packages to the default channel. Instead, separated personal channels need to be used for the upload and explicitly selected with the `-c <channel name>` option in `conda install`.

It is important however to consider that mixing many channels together might create clashes in dependencies (although conda tries very hard to avoid clashes by using channel preference ordering and a clever resolution algorithm).

A general practice that emerged in the conda ecosystem is to organise packages in large communities that share a single and open repository in Anaconda, that rely on specific procedures and heavy continuous integration for publishing cohesive packages. These procedures, however, might involve creating a second repository (separated from the main code base) to just host the recipe files. For that reason, PyScaffold does not currently generate conda recipe files when creating new projects.

Instead, if you are an open-source developer and are interested in distributing packages via conda, our recommendation is to try publishing your package on conda-forge (unless you want to target a specific community such as bioconda). conda-forge is one of the largest channels in Anaconda and works as the central hub for the Python developers in the conda ecosystem.

Once you have your package published to PyPI using the project generated by PyScaffold, you can create a *conda-forge feedstock*[1] using a special tool called grayskull and following the documented instructions. Please make sure to check PyScaffold community tips in discussion #422.

If you still need to use a personal custom channel in Anaconda, please checkout conda-build tutorials for further information.

Tip: It is not strictly necessary to publish your package to Anaconda for your users to be able to install it if they are using conda – `pip install` can still be used from a conda environment. However, if you have dependencies that are also published in Anaconda and are not pure Python projects (e.g. `numpy` or `matplotlib`), or that rely on virtual environments, it is generally advisable to do so.

[1] **feedstock** is the term used by conda-forge for the companion repository with recipe files

3.2 Migration to PyScaffold

Migrating your existing project to PyScaffold is in most cases quite easy and requires only a few steps. We assume your project resides in the Git repository `my_project` and includes a package directory `my_package` with your Python modules.

Since you surely don't want to lose your Git history, we will just deploy a new scaffold in the same repository and move as well as change some files. But before you start, please make sure that your working tree is not dirty, i.e. all changes are committed and all important files are under version control.

Let's start:

1. Change into the parent folder of `my_project` and type:

   ```
   putup my_project --force --no-skeleton -p my_package
   ```

   in order to deploy the new project structure in your repository.

2. Now change into `my_project` and move your old package folder into `src` (if your existing project does not follow a src layout yet):

   ```
   git mv my_package/* src/my_package/
   ```

   Use the same technique if your project has a test folder other than `tests` or a documentation folder other than `docs`.

3. Use `git status` to check for untracked files and add them with `git add`.

4. Eventually, use `git difftool` to check all overwritten files for changes that need to be transferred. Most important is that all configuration that you may have done in `setup.py` by passing parameters to `setup(...)` need to be moved to `setup.cfg`. You will figure that out quite easily by putting your old `setup.py` and the new `setup.cfg` template side by side. Checkout the documentation of setuptools for more information about this conversion. In most cases you will not need to make changes to the new `setup.py` file provided by PyScaffold. The only exceptions are if your project uses compiled resources, e.g. Cython.

5. In order to check that everything works, run `pip install .` and `tox -e build` (or `python setup.py sdist`). If those two commands don't work, check `pyproject.toml`, `setup.cfg`, `setup.py` as well as your package under `src` again. Were all modules moved correctly? Is there maybe some `__init__.py` file missing? Be aware that projects containing a `pyproject.toml` file will build in a different, and sometimes non backwards compatible, way. If that is your case, you can try to keep the legacy behaviour by deleting `pyproject.toml` and building the distributions exclusively with `setup.py`. Please see our *updating guide* for some *extra steps* you might want to execute manually. Finally, try also to run `make -C docs html` and `pytest` (or preferably their `tox` equivalents) to check that Sphinx and PyTest run correctly.

3.3 Updating from Previous Versions

When updating a project generated with the same major version of PyScaffold[1], running `putup --update` should be enough to get you going. However updating from previous major versions of PyScaffold will probably require some manual adjustments. The following sections describe how to update from one major version into the following one.

Tip: Before updating make sure to commit all the pending changes in your repository. If something does not work

[1] PyScaffold uses 3 numbers for its version: `MAJOR.MINOR.PATCH` (when the numbers on the right are missing, just assume them as being 0), so PyScaffold 3.1.2 has the same major version as PyScaffold 3.3.1, but not PyScaffold 4.

exactly how you expected after the update, please revise the changes using a `diff` and perform the necessary corrections.

3.3.1 Updates from PyScaffold 2 to PyScaffold 3

Since the overall structure of a project set up with PyScaffold 2 differs quite much from a project generated with PyScaffold 3 it is not possible to just use the `--update` parameter. Still with some manual efforts an update from a scaffold generated with PyScaffold 2 to PyScaffold 3's scaffold is quite easy. Assume the name of our project is `old_project` with a package called `old_package` and no namespaces then just:

1) make sure your worktree is not dirty, i.e. commit all your changes,
2) run `putup old_project --force --no-skeleton -p old_package` to generate the new structure inplace and `cd` into your project,
3) move with `git mv old_package/* src/old_package/ --force` your old package over to the new `src` directory,
4) check `git status` and add untracked files from the new structure,
5) use `git difftool` to check all overwritten files, especially `setup.cfg`, and transfer custom configurations from the old structure to the new,
6) check if `python setup.py test sdist` works and commit your changes.

3.3.2 Updates from PyScaffold 3 to PyScaffold 4

Most of the time, updating from PyScaffold 3 should be completely automatic. However, since in version 4 we have adopted Python's new standards for packaging (PEP 517/PEP 518), you might find the new build process incompatible.

If that is the case, you might want to try reverting to the legacy behaviour and preventing the build tools from using isolated builds (PEP 517). That can be easily done by deleting the *pyproject.toml* file from your package root.

You will need, though, to manually follow a few extra steps to make sure everything works:

1) Remove PyScaffold from your build dependencies (`setup_requires` in `setup.cfg`) and add setuptools-scm.

   Note: The use of `setup_requires` is discouraged. When updating to v4 PyScaffold will remove this field automatically and transfer the dependencies to the `pyproject.toml :: build-system.requires` field, which means you may need to manually place them back when deleting `pyproject.toml`. Alternatively you can ditch `setup_requires` completely and rely on other tools like tox or make to build your project with the correct dependencies in place inside a virtual environment. This have the advantage of increasing reproducibility. With tox you can specify a `build` testenv with the skip_install option and the required build time dependencies listed in `deps`.

2) Migrate any configuration options for tools that might be using `pyproject.toml` to alternative files. For example if you have `isort` and `coverage` configurations in your `pyproject.toml`, you might want to rewrite them in the *.isort.cfg* and *.coveragerc* files respectively.
3) Please open an issue with PyScaffold so we understand with kind of backward incompatibilities PEP 517 and PEP 518 might be causing and try to help. Similarly you might also consider opening an issue with setuptools.

Warning: For the time being you can use the **transitional** `--no-pyproject` option, when running `putup`, but have in mind that this option will be removed in future versions of PyScaffold.

PyScaffold 4 also adopts the PEP 420 scheme for implicit namespaces and will automatically migrate existing packages. This is incompatible with the previously adopted pkg_resources methodology. **Fortunately, this will not affect you if you are not using namespaces**, but in the case you are, installing a new PEP 420-compliant package in an environment that already contains other packages with the same namespace but that use the pkg_resources methodology, will likely result in errors (please check the official packaging namespace packages guides for more information).

To solve this problem you will need to either migrate the existing packages to PEP 420 or revert some specific configurations in `setup.cfg` after the update. In particular `packages = find_namespace:` should be converted back to `packages = find:` in the `[options]` section (use a `git difftool` to help you with that). If using Sphinx for the documentation, you can also remove the `--implicit-namespaces` option in the `cmd_line_template` variable in the `docs/conf.py` file.

Tip: Existing regular Python files (or other directories containing Python files) that do not belong to the package distribution but are placed inside the `src` folder (such as example files not meant to be packaged), can cause problems when building your package.

Please move these files if necessary to their own separated folders (e.g. the `docs` folder or a new `examples` folder in the root of the repository), or revert back to the pkg_resources implementation. Just have in mind that PyScaffold, considers the `src` directory to be exclusively dedicated to store files meant to be distributed, and will rely on that assumption on its future versions and updates.

3.4 Extending PyScaffold

PyScaffold is carefully designed to cover the essentials of authoring and distributing Python packages. Most of time, tweaking `putup` options is enough to ensure proper configuration of a project. However, for advanced use cases PyScaffold can be extended at runtime by other Python packages, providing a deeper level of programmability and customization.

From the standpoint of PyScaffold, an extension is just an class inheriting from *`Extension`* overriding and implementing certain methods that allow the manipulation of a *in-memory* **project structure representation** via PyScaffold's internal **action pipeline** mechanism. The following sections describe these two key concepts in detail and present a comprehensive guide about how to create custom extensions.

Tip: A perfect start for your own custom extension is the extension custom_extension for PyScaffold. Just install it with `pip install pyscaffoldext-custom-extension` and then create your own extension template with `putup --custom-extension pyscaffoldext-my-own-extension`.

3.4.1 Project Structure Representation

Each Python package project is internally represented by PyScaffold as a tree data structure, that directly relates to a directory entry in the file system. This tree is implemented as a simple (and possibly nested) `dict` in which keys indicate the path where files will be generated, while values indicate their content. For instance, the following dict:

```
{
    "folder": {
        "file.txt": "Hello World!",
        "another-folder": {
            "empty-file.txt": ""
        }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

represents a project directory in the file system that contains a single directory named `folder`. In turn, `folder` contains two entries. The first entry is a file named `file.txt` with content `Hello World!` while the second entry is a sub-directory named `another-folder`. Finally, `another-folder` contains an empty file named `empty-file.txt`.

Note: Changed in version 4.0: Prior to version 4.0, the project structure included the top level directory of the project. Now it considers everything **under** the project folder.

Additionally, tuple values are also allowed in order to specify a **file operation** (or simply **file op**) that will be used to produce the file. In this case, the first element of the tuple is the file content, while the second element will be a function (or more generally a `callable` object) responsible for writing that content to the disk. For example, the dict:

```
from pyscaffold.operations import create

{
    "src": {
        "namespace": {
            "module.py": ('print("Hello World!")', create)
        }
    }
}
```

represents a `src/namespace/module.py` file, under the project directory, with content `print("Hello World!")`, that will written to the disk. When no operation is specified (i.e. when using a simple string instead of a tuple), PyScaffold will assume *`create`* by default.

Note: The *`create`* function simply creates a text file to the disk using UTF-8 encoding and the default file permissions. This behaviour can be modified by wrapping *`create`* within other fuctions/callables, for example:

```
from pyscaffold.operations import create, no_overwrite

{"file": ("content", no_overwrite(create))}
```

will prevent the `file` to be written if it already exists. See *`pyscaffold.operations`* for more information on how to write your own file operation and other options.

Finally, while it is simple to represent file contents as a string directly, most of the times we want to *customize* them according to the project parameters being created (e.g. package or author's name). So PyScaffold also accepts `string.Template` objects and functions (with a single `dict` argument and a `str` return value) to be used as contents. These templates and functions will be called with *`PyScaffold's options`* when its time to create the file to the disk.

Note: `string.Template` objects will have `safe_substitute` called (not simply `substitute`).

This tree representation is often referred in this document as **project structure** or simply **structure**.

3.4.2 Action Pipeline

PyScaffold organizes the generation of a project into a series of steps with well defined purposes. As shown in the figure bellow, each step is called **action** and is implemented as a simple function that receives two arguments: a project structure and a `dict` with options (some of them parsed from command line arguments, other from default values).

An action **MUST** return a tuple also composed by a project structure and a `dict` with options. The return values, thus, are usually modified versions of the input arguments. Additionally an action can also have side effects, like creating directories or adding files to version control. The following pseudo-code illustrates a basic action:

```
def action(project_structure, options):
    new_struct, new_opts = modify(project_structure, options)
    some_side_effect()
    return new_struct, new_opts
```

The output of each action is used as the input of the subsequent action, forming a pipeline. Initially the structure argument is just an empty `dict`. Each action is uniquely identified by a string in the format `<module name>:<function name>`, similarly to the convention used for a setuptools entry point. For example, if an action is defined in the `action` function of the `extras.py` file that is part of the `pyscaffoldext.contrib` project, the **action identifier** is `pyscaffoldext.contrib.extras:action`.

By default, the sequence of actions taken by PyScaffold is:

1. *pyscaffold.actions:get_default_options*
2. *pyscaffold.actions:verify_options_consistency*
3. *pyscaffold.structure:define_structure*
4. *pyscaffold.actions:verify_project_dir*
5. *pyscaffold.update:version_migration*
6. *pyscaffold.structure:create_structure*
7. *pyscaffold.actions:init_git*
8. *pyscaffold.actions:report_done*

(as given by *pyscaffold.actions.DEFAULT*)

The project structure is usually empty until *define_structure* This action just loads the in-memory `dict` representation, that is only written to disk by the *create_structure* action.

Note that, this sequence varies according to the command line options. To retrieve an updated list, please use `putup --list-actions` or `putup --dry-run`.

3.4.3 Creating an Extension

In order to create an extension it is necessary to write a class that inherits from *Extension* and implements the method *activate* that receives a list of actions (interpret this argument as a sequence of actions to be executed, or pipeline), registers a custom action that will be called later and returns a modified version of the list of actions:

```
from pyscaffold import actions
from pyscaffold.extensions import Extension


class MyExtension(Extension):
```

(continues on next page)

(continued from previous page)

```
    """Help text on commandline when running putup -h"""

    def activate(self, pipeline):
        """Activate extension

        Args:
            pipeline (list): list of actions to perform

        Returns:
            list: updated list of actions
        """
        pipeline = actions.register(pipeline, self.action, after="create_structure")
        pipeline = actions.unregister(pipeline, "init_git")
        return actions

    def action(self, struct, opts):
        """Perform some actions that modifies the structure and options

        Args:
            struct (dict): project representation as (possibly) nested
                :obj:`dict`.
            opts (dict): given options, see :obj:`create_project` for
                an extensive list.

        Returns:
            new_struct, new_opts: updated project representation and options
        """
        ...
        return new_struct, new_opts
```

Tip: The `register` and `unregister` methods implemented in the module *pyscaffold.actions* basically create modified copies of the action list by inserting/removing the specified functions, with some awareness about their execution order.

Action List Helper Methods

As implied by the previous example, the *pyscaffold.actions* module provides a series of useful functions and makes it easier to manipulate the action list, by using *register* and *unregister*.

Since the action order is relevant, the first function accepts special keyword arguments (`before` and `after`) that should be used to place the extension actions precisely among the default actions. The value of these arguments can be presented in 2 different forms:

```
actions.register(action_sequence, hook1, before="define_structure")
actions.register(action_sequence, hook2, after="pyscaffold.structure:create_structure")
```

The first form uses as a position reference the first action with a matching name, regardless of the module. Accordingly, the second form tries to find an action that matches both the given name and module. When no reference is given, *register* assumes as default position `after="pyscaffold.structure:define_structure"`. This position is special since most extensions are expected to create additional files inside the project. Therefore, it is possible to easily amend the project structure before it is materialized by `create_structure`.

The *unregister* function accepts as second argument a position reference which can similarly present the module name:

```
actions.unregister(action_sequence, "init_git")
actions.unregister(action_sequence, "pyscaffold.api:init_git")
```

Note: These functions **DO NOT** modify the actions list, instead they return a new list with the changes applied.

Tip: For convenience, the functions *register* and *unregister* are aliased as instance methods of the *Extension* class.

Therefore, inside the *activate* method, one could simply call `action_sequence = self.register(action_sequence, self.my_action)`.

Structure Helper Methods

PyScaffold also provides extra facilities to manipulate the project structure. The following functions are accessible through the *structure* module:

- *merge*
- *ensure*
- *reject*
- *modify*

The first function can be used to deep merge a dictionary argument with the current representation of the to-be-generated directory tree, automatically considering any file op present in tuple values. On the other hand, the second and third functions can be used to ensure a single file is present or absent in the current representation of the project structure, automatically handling parent directories. Finally, *modify* can be used to change the contents of an existing file in the project structure and/or the assigned file operation (for example wrapping it with *no_overwrite*, *skip_on_update* or *add_permissions*).

Note: Similarly to the actions list helpers, these functions also **DO NOT** modify the project structure. Instead they return a new structure with the changes applied.

The following example illustrates the implementation of a `AwesomeFiles` extension which defines the `define_awesome_files` action:

```
from pathlib import Path
from string import Template
from textwrap import dedent

from pyscaffold import structure
from pyscaffold.extensions import Extension
from pyscaffold.operations import create, no_overwrite, skip_on_update


def my_awesome_file(opts):
    return dedent(
```

(continues on next page)

(continued from previous page)

```
        """\
        __author__ = "{author}"
        __copyright__ = "{author}"
        __license__ = "{license}"

        def awesome():
            return "Awesome!"
        """.format(
            **opts
        )
    )


MY_AWESOME_TEST = Template(
    """\
import pytest
from ${qual_pkg}.awesome import awesome

def test_awesome():
    assert awesome() == "Awesome!"
"""
)


class AwesomeFiles(Extension):
    """Adding some additional awesome files"""

    def activate(self, actions):
        return self.register(actions, self.define_awesome_files)

    def define_awesome_files(self, struct, opts):
        struct = structure.merge(
            struct,
            {
                "src": {
                    opts["package"]: {"awesome.py": my_awesome_file},
                },
                "tests": {
                    "awesome_test.py": (MY_AWESOME_TEST, no_overwrite(create)),
                    "other_test.py": ("# not so awesome", no_overwrite(create)),
                },
            },
        )

        struct[".python-version"] = ("3.6.1", no_overwrite(create))

        for filename in ["awesome_file1", "awesome_file2"]:
            struct = structure.ensure(
                struct,
                f"src/{opts['package']}/{filename}",
                content="AWESOME!",
                file_op=skip_on_update(create),
```

(continues on next page)

(continued from previous page)

```
                # The second argument is the file path, represented by a
                # os.PathLike object or string.
                # Alternatively in this example:
                # Path("src", opts["package"], filename),
            )

        # The `reject` can be used to avoid default files being generated.
        struct = structure.reject(struct, Path("src", opts["package"], "skeleton.py"))

        # `modify` can be used to change contents in an existing file
        # and/or change the assigned file operation
        def append_pdb(prev_content, prev_op):
            retrun(prev_content + "\nimport pdb", skip_on_update(prev_op)),

        struct = structure.modify(struct, "tests/other_test.py", append_pdb)

        # It is import to remember the return values
        return struct, opts
```

As shown by the previous example, the *operations* module also contains file operation **modifiers** that can be used to change the assigned file op. These modifiers work like standard Python decorators: instead of being a file op themselves, they receive a file operation as argument and return a file operation, and therefore can be used to *wrap* the original file operation and modify its behaviour.

Tip: By default, all the file op modifiers in the *pyscaffold.operations* package don't even need an explicit argument, when called with zero arguments *create* is assumed.

---

no_overwrite avoids an existing file to be overwritten when `putup` is used in update mode. Similarly, *skip_on_update* avoids creating a file from template in update mode, even if it does not exist. On the other hand, *add_permissions* will change the file access permissions if it is created or already exists in the disk.

Note: See *pyscaffold.operations* for more information on how to write your own file operation or modifiers.

Activating Extensions

PyScaffold extensions are not activated by default. Instead, it is necessary to add a CLI option to do it. This is possible by setting up a setuptools entry point under the `pyscaffold.cli` group. This entry point should point to our extension class, e.g. `AwesomeFiles` like defined above. If you for instance use a scaffold generated by PyScaffold to write a PyScaffold extension (we hope you do ;-), you would add the following to the `options.entry_points` section in `setup.cfg`:

```
[options.entry_points]
pyscaffold.cli =
    awesome_files = your_package.your_module:AwesomeFiles
```

Tip: In order to guarantee consistency and allow PyScaffold to unequivocally find your extension, the name of the entry point should be a "underscore" version of the name of the extension class (e.g. an entry point `awesome_files` for the `AwesomeFiles` class). If you really need to customize that behaviour, please overwrite the `name` property of

your class to match the entry point.

By inheriting from *pyscaffold.extensions.Extension*, a default CLI option that already activates the extension will be created, based on the dasherized version of the name in the setuptools entry point. In the example above, the automatically generated option will be `--awesome-files`.

For more sophisticated extensions which need to read and parse their own command line arguments it is necessary to override *augment_cli* that receives an `argparse.ArgumentParser` argument. This object can then be modified in order to add custom command line arguments that will later be stored in the `opts` dictionary. Just remember the convention that after the command line arguments parsing, the extension function should be stored under the `extensions` attribute (a list) of the `argparse` generated object. For reference check out the implementation of the namespace extension. Another convention is to avoid storing state/parameters inside the extension class, instead store them as you would do regularly with `argparse` (inside the `argparse.Namespace` object).

Persisting Extensions for Future Updates

PyScaffold will save the name of your extension in a **pyscaffold** section inside the `setup.cfg` files and automatically activate it again every time the user runs `putup --update`. To prevent it from happening you can set `persist = False` in your extension instances or class.

PyScaffold can also save extension-specific options if the names of those options start with an "underscore" version of your extension's name (and setuptools entry point). For example, the namespace extension stores the `namespace` option in `setup.cfg`.

If the name of your extension class is `AwesomeFiles`, then anything like `opts["awesome_files"]`, `opts["awesome_files1"]`, `opts["awesome_files_SOMETHING"]` would be stored. Please ensure you have in mind the limitations of the `configparser` serialisation mechanism and supported data types to avoid errors (it should be safe to use string values without line breaks).

Extra Configurations

Similarly to `persist = False`, existing extensions might accept some sort of metadata to be defined by new extensions.

This is the case of the *pyscaffold.extensions.interactive*, that allows users to interactively choose PyScaffold's parameters by editing a file containing available options alongside a short description (similarly to `git rebase -i`). The *interactive* extension accepts a `interactive` attribute defined by extension instances or classes. This attribute might define a dictionary with keys: `"ignore"` and `"comment"`. The value associated with the key `"ignore"` should be a list of CLI options to be simply ignored when creating examples (e.g. `["--help"]`). The value associated with the key `"comment"` should be a list of CLI options to be commented in the created examples, even if they appear in the original `sys.argv`.

> **Warning:** The *interactive* extension is still **experimental** and might not work exactly as expected. More importatly, due to limitations on the way `argparse` is implemented, there are several limitations and complexities on how to manipulate command line options when not using them directly. This means that the interactive extension might render your extension's options in a sub-optimal way. If you ever encounter this challenge we strongly encourage you to open a pull request (or at least an issue or discussion).

If your extension accepts metadata and interact with other extensions, you can also rely in informative attributes, but please be sure to make these optional with good fallback values and a comprehensive documentation.

3.4.4 Examples

Some options for the `putup` command are already implemented as extensions and can be used as reference implementation, such as:

- no-skeleton
- no-tox
- cirrus
- gitlab

For more advanced extensions, please check:

- namespace
- pre-commit

3.4.5 Public API

The following methods, functions and constants are considered to be part of the public API of PyScaffold for creating extensions and will not change signature and described overall behaviour (although implementation details might change) in a backwards incompatible way between major releases (semantic versioning):

- *pyscaffold.actions.register*
- *pyscaffold.actions.unregister*
- pyscaffold.extensions.Extension.__init__
- *pyscaffold.extensions.Extension.persist*
- *pyscaffold.extensions.Extension.name*
- *pyscaffold.extensions.Extension.augment_cli*
- *pyscaffold.extensions.Extension.activate*
- *pyscaffold.extensions.Extension.register*
- *pyscaffold.extensions.Extension.unregister*
- *pyscaffold.extensions.include*
- *pyscaffold.extensions.store_with*
- *pyscaffold.operations.create*
- *pyscaffold.operations.no_overwrite*
- *pyscaffold.operations.skip_on_update*
- *pyscaffold.structure.ensure*
- *pyscaffold.structure.merge*
- *pyscaffold.structure.modify*
- *pyscaffold.structure.reject*
- *pyscaffold.templates.get_template*

In addition to these, the definition of action (given by *pyscaffold.actions.Action*), project structure (given by *pyscaffold.structure.Structure*), and operation (given by `pyscaffold.operation.FileOp`) are also part of the public API. The remaining functions and methods are no guaranteed to be stable and are subject to incompatible changes even in minor/patch releases.

3.4.6 Conventions for Community Extensions

In order to make it easy to find PyScaffold extensions, community packages should be namespaced as in `pyscaffoldext.${EXT_NAME}` (where `${EXT_NAME}` is the name of the extension being developed). Although this naming convention slightly differs from PEP423, it is close enough and shorter.

Similarly to `sphinxcontrib-*` packages, names registered in PyPI should contain a dash `-`, instead of a dot `.`. This way, third-party extension development can be easily bootstrapped with the command:

```
putup pyscaffoldext-${EXT_NAME} -p ${EXT_NAME} --namespace pyscaffoldext --no-skeleton
```

If you put your extension code in the module `extension.py` then the `options.entry_points` section in `setup.cfg` looks like:

```
[options.entry_points]
pyscaffold.cli =
    awesome_files = pyscaffoldext.${EXT_NAME}.extension:AwesomeFiles
```

In this example, `AwesomeFiles` represents the name of the class that implements the extension and `awesome_files` is the string used to create the flag for the `putup` command (`--awesome-files`).

Tip: If you want to write a PyScaffold extension, check out our custom_extension generator. It can get you pretty far in just a few minutes.

3.4.7 Final Considerations

1. When writing extensions, it is important to be consistent with the default PyScaffold behavior. In particular, PyScaffold uses a `pretend` option to indicate when the actions should not run but instead just indicate the expected results to the user, that **MUST** be respected.

   The `pretend` option is automatically observed for files registered in the project structure representation, but complex actions may require specialized coding. The *`log`* module provides a special *`logger`* object useful in these situations. Please refer to pyscaffoldext-cookiecutter for a practical example.

   Other options that should be considered are the `update` and `force` flags. See *`pyscaffold.api.create_project`* for a list of available options.

2. Don't forget that packages can be created inside namespaces. To be on the safe side when writing templates prefer explicit relative import statements (e.g. `from . import module`) or use the template variable `${qual_pkg}` provided by PyScaffold. This variable contains the fully qualified package name, including possible namespaces.

   ```
   # Yes:
   import ${qual_pkg}
   from . import module
   from .module import function
   from ${qual_pkg} import module
   from ${qual_pkg}.module import function

   # No:
   import ${package}
   from ${package} import module
   from ${package}.module import function
   ```

CHAPTER

FOUR

WHY PYSCAFFOLD?

Stable and battle-tested PyScaffold was created in 2014 to make the lives of developers easier. Since then it has been used to create many awesome Python packages for data science, industrial automation, academic research, telecom, web development and many other sectors.

Constantly evolving The stability of PyScaffold does not come at the price of stagnation. Throughout its existence, PyScaffold has adapted itself to better solve the needs of its users and evolved to accommodate the best practices and standards of the Python ecosystem. In every single major release, we offered a clear update path for our users, automating things as much as possible, so everyone can benefit from PyScaffold's improvements without being afraid of breaking things.

Thoroughly tested PyScaffold has an extensive automated test suite that runs for all major operating systems and versions of Python for every commit or pull request. Moreover, PyScaffold is used by its maintainers in their day-to-day programming and for all PyScaffold's extensions, so we also have people constantly keeping an eye on it.

Do one thing and do it well It might sound old-fashioned, but we like this piece of good old UNIX wisdom. Pyscaffold creates a perfect project structure that compiles the best practices for Python packaging and has tons of useful defaults… *that is it!*

It does not attempt to do dependency management or build a distribution, because there are dedicated tools exactly for that purpose which have survived the test of time and are well-established within the Python community.

Instead of reinventing the wheel, we spent all these years curating a incredibly smart project template that ships with ready-to-use configuration for all the tools needed by the vast majority of Python developers.

Standing on the shoulder of giants PyScaffold incentives its users to use the best tools and practices available in the Python ecosystem.

A generated project will contain sane default configurations for setuptools (the de facto standard for building Python packages), Sphinx (the one & only Python documentation tool), pytest and tox (most commonly used Python testing framework & task runner), so the users can run these common tasks using e.g. `tox -e build`, `tox -e docs`, or `tox -e publish`.

For those who want to go the extra mile, PyScaffold can also bring pre-commit into the mix to run a set of prolific linters and automatic formatters in each commit in order to adhere to common coding standards like pep8 and black.

Composable PyScaffold shows its strengths when combined with other tools, and indeed we bring configurations for lots of them by default. In the end of the day, a project generated by PyScaffold is just a plain, standard Python package, and will interoperate well with the majority of the tools you might want to use in your development environment.

Extensible *Don't like something about PyScaffold? Wish the templates were a little different? Particular workflow? Different tools? Have you got a nice set of templates that you would like to re-use?*

Well, go ahead and make PyScaffold yours... We have developed a *powerful extension system* that allows users to make the most out of PyScaffold. In fact, PyScaffold's core is very minimal and several of the options are implemented themselves as extensions and shipped by default.

Easy and yet powerful PyScaffold provides you one simple, yet powerful, command with intuitive options. Newcomers can achieve a lot by just running `putup your_project_name`, while power users can dig into our docs/help and discover all our *Features*.

It works with existing projects too PyScaffold is useful not only for starting new projects from scratch. If you have existing code that you have been playing around, or that was generated by other tools, you can convert it with PyScaffold's `putup --force your_project_folder` command. Check our *Migration to PyScaffold* guides.

No *lock-in* Once you have generated your project with PyScaffold you can later come back to use the update features. Other than that, there are no ties to PyScaffold at all! Meaning that PyScaffold will be no install dependency of your project and starting from version 4.0 on not even a development dependency. If you would want to erase all traces of the fact that you used PyScaffold to set up your project, we got you, and *have documented even that...*

Batteries included PyScaffold offers a lot out of the box: we have everything a Python developer needs to start coding right away and be 100% productive from the start.

Have a look in our extensive list of *Features*. Using PyScaffold is like having a Python Packaging Guru, who has spent a lot of time researching how to create the best project setups, as a friend that is helping you with your project.

Curious? Checkout out our demo project, or *install* PyScaffold and type `putup -h` to *get started*.

CHAPTER

FIVE

FEATURES

PyScaffold comes with a lot of elaborated features and configuration defaults to make the most common tasks in developing, maintaining and distributing your own Python package as easy as possible.

5.1 Configuration, Packaging & Distribution

All configuration can be done in `setup.cfg` like changing the description, URL, classifiers, installation requirements and so on as defined by setuptools. That means in most cases it is not necessary to tamper with `setup.py`. The syntax of `setup.cfg` is pretty much self-explanatory and well commented, check out this *example* or setuptools' documentation.

If you use tox, PyScaffold will already configure everything out of the box[1] so you can easily build your distribution, in a PEP 517/PEP 518 compliant way, by just running:

```
tox -e build
```

Alternatively, if you are not a huge fan of isolated builds, or prefer running the commands yourself, you can execute `python setup.py bdist_wheel`. Source distributions, i.e. `sdist`, are obsolete and no longer recommended as they ignore several options in `setup.cfg`. If `universal=1` in the `[bdist_wheel]` section of `setup.cfg`, a generated wheel distribution will be as flexible as a source distribution and can be installed on every architecture.

Uploading to PyPI

Of course uploading your package to the official Python package index PyPI for distribution also works out of the box. Just create a distribution as mentioned above and use tox to publish with:

```
tox -e publish
```

This will first upload your package using TestPyPI, so you can be a good citizen of the Python world, check/test everything is fine, and then, when you are absolutely sure the moment has come for your package to shine, you can go ahead and run `tox -e --publish -- --repository pypi`[2]. Just remember that for this to work, you have to first register a PyPI account (and also a TestPyPI one).

Under the hood, tox uses twine for uploads to PyPI (as configured by PyScaffold in the `tox.ini` file), so if you prefer running things yourself, you can also do:

[1] Tox is a virtual environment management and test tool that allows you to define and run custom tasks that call executables from Python packages. In general, PyScaffold will already pre-configure tox to do the most common tasks for you. You can have a look on what is available out of the box by running `tox -av`, or go ahead and check tox docs to automatise your own tasks.

[2] The verbose command is intentional here to prevent later regrets. Once a package version is published to PyPI, it cannot be replaced. Therefore, be always sure your are done and all set before publishing.

```
pip install twine
twine upload --repository testpypi dist/*
```

Please notice that PyPI does not allow uploading local versions, e.g. `0.0.dev5+gc5da6ad`, for practical reasons. Thus, you have to create a git tag before uploading a version of your distribution. Read more about it in the *versioning* section below.

Warning: Old guides might mention `python setup.py upload`, but its use is strongly discouraged nowadays and even some of the new PyPI features won't work correctly if you don't use twine.

Namespace Packages

If you want to work with namespace packages, you will be glad to hear that PyScaffold supports the PEP 420 specification for implicit namespaces, which is very useful to distribute a larger package as a collection of smaller ones. `putup` can automatically setup everything you need with the `--namespace` option. For example, use:

```
putup my_project --package my_package --namespace com.my_domain
```

to define `my_package` inside the namespace `com.my_domain`, Java-style.

Note: Prior to PyScaffold 4.0, namespaces were generated explicitly with pkg_resources, instead of PEP 420. Moreover, if you are developing "subpackages" for already existing namespaces, please check which convention the namespaces are currently following. Different styles of namespace packages might be incompatible. If you don't want to update existing namespace packages to PEP 420, you will probably need to manually copy the `__init__.py` file for the umbrella namespace folder from an existing project. Additionally have a look in our *FAQ* about how to disable implicit namespaces.

Package and Files Data

Additional data, e.g. images and text files, that **must reside within** your package, e.g. under `my_project/src/my_package`, and are tracked by Git will automatically be included if `include_package_data = True` in `setup.cfg`. It is not necessary to have a `MANIFEST.in` file for this to work. Just make sure that all files are added to your repository. To read this data in your code, use:

```
from pkgutil import get_data
data = get_data('my_package', 'path/to/my/data.txt')
```

Starting from Python 3.7 an even better approach is using importlib.resources:

```
from importlib.resources import read_text, read_binary
data = read_text('my_package.sub_package', 'data.txt')
```

Note that we need a proper package structure in this case, i.e. directories need to contain `__init__.py` and be named as a valid Python package (which follow the same rules as variable names). We only specify the file `data.txt`, no path is allowed. The library importlib_resources provides a backport of this feature.

Please have in mind that the `include_package_data` option in `setup.cfg` is only guaranteed to be read when creating a wheels distribution. Other distribution methods might behave unexpectedly (e.g. always including data files even when `include_package_data = False`). Therefore, the best option if you want to have data files in your repository **but not as part of the pip installable package** is to add them somewhere **outside** the `src` directory (e.g. a `files`

directory in the root of the project, or inside `tests` if you use them for checks). Additionally you can exclude them explicitly via the `[options.packages.find] exclude` option in `setup.cfg`. More information about data files support is available on the `setuptools` website.

Tip: Using package files to store runtime configuration or mutable data is not considered good practice. Package files should be read-only. If you need configuration files, or files that should be written at runtime, please consider doing so inside standard locations in the user's home folder (appdirs is a good library for that). If needed you can even create them at the first usage from a read-only template, which in turn can be a package file.

5.2 Versioning and Git Integration

Your project is already an initialised Git repository and setuptools uses the information of tags to infer the version of your project with the help of setuptools_scm. To use this feature you need to tag with the format `MAJOR.MINOR[.PATCH]` , e.g. `0.0.1` or `0.1`. Run `python setup.py --version` to retrieve the current PEP 440-compliant version. This version will be used when building a package and is also accessible through `my_project.__version__`. If you want to upload to PyPI you have to tag the current commit before uploading since PyPI does not allow local versions, e.g. `0.0.dev5+gc5da6ad`, for practical reasons.

Please check our docs for the *best practices and common errors with version numbers*.

Pre-commit Hooks

Unleash the power of Git by using its pre-commit hooks. This feature is available through the `--pre-commit` flag. After your project's scaffold was generated, make sure pre-commit is installed, e.g. `pip install pre-commit`, then just run `pre-commit install`.

It goes unsaid that also a default `.gitignore` file is provided that is well adjusted for Python projects and the most common tools.

5.3 Sphinx Documentation

PyScaffold will prepare a `docs` directory with all you need to start writing your documentation. Start editing the file `docs/index.rst` to extend the documentation and note that even the Numpy and Google style docstrings are activated by default.

If you have tox in your system, simply run `tox -e docs` or `tox -e doctests` to compile the docs or run the doctests.

Alternatively, if you have make and Sphinx installed in your computer, build the documentation with `make -C docs html` and run doctests with `make -C docs doctest`. Just make sure Sphinx 1.3 or above is installed.

The documentation also works with Read the Docs. Please check the RTD guides to learn how to import your documents into the website.

Note: In order to generate the docs locally, you will need to install any dependency used to build your doc files (and probably all your project dependencies) in the same Python environment where Sphinx is installed (either the global Python installation or a conda/virtualenv/venv environment). For example, if you want to use the Read the Docs classic theme, the `sphinx_rtd_theme` package should be installed.

If you are using `tox -e docs`, tox will take care of generating a virtual environment and installing all these dependencies automatically. You will only need to list your doc dependencies (like `sphinx_rtd_theme`) under the `deps` property

of the `[testenv:{docs,doctests}]` section in the `tox.ini` file. Your can also use the `docs/requirements.txt` file to store them. This file can be used by both Read the Docs and tox when generating the docs.

5.4 Dependency Management in a Breeze

PyScaffold out of the box allows developers to express abstract dependencies and take advantage of `pip` to manage installation. It also can be used together with a virtual environment (also called *virtual env*) to avoid dependency hell during both development and production stages.

If you like the traditional style of dependency management using a virtual env co-located with your package, PyScaffold can help to reduce the boilerplate. With the `--venv` option, a virtualenv will be bootstrapped and waiting to be activated. And if you are the kind of person that always install the same packages when creating a virtual env, PyScaffold's option `--venv-install PACKAGE` will be the right one for you. You can even integrate pip-tools in this workflow, by putting a `-e file:.` in your *requirements.in*.

Alternatively, PyPA's Pipenv can be integrated in any PyScaffold-generated project by following standard setuptools conventions. Keeping abstract requirements in `setup.cfg` and running `pipenv install -e .` is basically what you have to do.

You can check the details on how all of that works in *Dependency Management*.

Warning: *Experimental Feature* - Pipenv and pip-tools support is experimental and might change in the future.

5.5 Automation, Tests & Coverage

PyScaffold relies on pytest to run all automated tests defined in the subfolder `tests`. Some sane default flags for pytest are already defined in the `[tool:pytest]` section of `setup.cfg`. The pytest plugin pytest-cov is used to automatically generate a coverage report. It is also possible to provide additional parameters and flags on the commandline, e.g., type:

```
pytest -h
```

to show the help of pytest (requires pytest to be installed in your system or virtual environment).

JUnit and Coverage HTML/XML

For usage with a continuous integration software JUnit and Coverage XML output can be activated in `setup.cfg`. Use the flag `--cirrus` to generate templates of the Cirrus CI configuration file `.cirrus.yml` which even features the coverage and stats system Coveralls. If you are using GitLab you can get a default *.gitlab-ci.yml* also running *pytest-cov* with the flag `--gitlab`.

Managing test environments and tasks with tox

Projects generated with PyScaffold are configured by default to use tox to run some common tasks. Tox is a virtual environment management and test tool that allows you to define and run custom tasks that call executables from Python packages.

If you simply install tox and run from the root folder of your project:

```
tox
```

tox will download the dependencies you have specified, build the package, install it in a virtual environment and run the tests using pytest, so you are sure everything is properly tested. You can rely on the tox documentation for detailed configuration options (which include the possibility of running the tests for different versions of Python).

You are not limited to running your tests, with tox you can define all sorts of automation tasks. Here are a few examples for you:

```
tox -e build  # will bundle your package and create a distribution inside the `dist`␣
↪folder
tox -e publish  # will upload your distribution to a package index server
tox -e docs  # will build your docs
```

but you can go ahead and check tox examples, or this tox tutorial from Sean Hammond for more ideas, e.g. running static code analyzers (pyflakes and pep8) with flake8. Run `tox -av` to list all the available tasks.

5.6 Management of Requirements & Licenses

Installation requirements of your project can be defined inside `setup.cfg`, e.g. `install_requires = numpy; scipy`. To avoid package dependency problems it is common to not pin installation requirements to any specific version, although minimum versions, e.g. `sphinx>=1.3`, and/or maximum versions, e.g. `pandas<0.12`, are used frequently in accordance with semantic versioning.

For test/dev purposes, you can additionally create a `requirements.txt` pinning packages to specific version, e.g. `numpy==1.13.1`. This helps to ensure reproducibility, but be sure to read our *Dependency Management Guide* to understand the role of a `requirements.txt` file for library and application projects (`pip-compile` from pip-tools can help you to manage that file). Packages defined in `requirements.txt` can be easily installed with:

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

The most popular open source licenses can be easily added to your project with the help of the `--license` flag. You only need to specify the license identifier according to the SPDX index so PyScaffold can generate the appropriate `LICENSE.txt` and configure your package. For example:

```
putup --license MPL-2.0 my_project
```

will create the `my_project` package under the Mozilla Public License 2.0 The available licenses can be listed with `putup --help`, and you can find more information about each license in the SPDX index and choosealicense.com.

5.7 Extensions

PyScaffold offers several extensions:

- If you want a project setup for a *Data Science* task, just use `--dsproject` after having installed pyscaffoldext-dsproject.
- Have a `README.md` based on Markdown instead of `README.rst` by using `--markdown` after having installed pyscaffoldext-markdown.
- Create a Django project with the flag `--django` which is equivalent to `django-admin startproject my_project` enhanced by PyScaffold's features (requires pyscaffoldext-django).
- ... and many more like `--gitlab` to create the necessary files for GitLab, `--travis` for Travis CI (see pyscaffoldext-travis), or `--cookiecutter` for Cookiecutter integration (see pyscaffoldext-cookiecutter).

Find more extensions within the PyScaffold organisation and consider contributing your own, it is very easy! You can quickly generate a template for your extension with the `--custom-extension` option after having installed pyscaffoldext-custom-extension. Have a look in our guide on *writing extensions* to get started.

All extensions can easily be installed with `pip install pyscaffoldext-NAME`.

5.8 Easy Updating

Keep your project's scaffold up-to-date by applying `putup --update my_project` when a new version of PyScaffold was released. An update will only overwrite files that are not often altered by users like `setup.py`. To update all files use `--update --force`. An existing project that was not setup with PyScaffold can be converted with `putup --force existing_project`. The force option is completely safe to use since the git repository of the existing project is not touched! Please check out the *Updating from Previous Versions* docs for more information on how to migrate from old versions and *configuration options* in `setup.cfg`.

5.8.1 Adding features

With the help of an **experimental** updating functionality it is also possible to add additional features to your existing project scaffold. If a scaffold lacking `.cirrus.yml` was created with `putup my_project` it can later be added by issuing `putup my_project --update --cirrus`. For this to work, PyScaffold stores all options that were initially used to put up the scaffold under the `[pyscaffold]` section in `setup.cfg`. Be aware that right now PyScaffold provides no way to remove a feature which was once added.

5.9 PyScaffold Configuration

After having used PyScaffold for some time, you probably will notice yourself repeating the same options most of the time for every new project. Don't worry, PyScaffold now allows you to set default flags using the **experimental** `default.cfg` file[3]. Check out our *Configuration* section to get started.

[3] Experimental features can change the way they work (or be removed) between any releases. If you are scripting with PyScaffold, please avoid using them.

CHAPTER

SIX

FREQUENTLY ASKED QUESTIONS

In case you have a general question that is not answered here, please have a look at our discussions and consider submitting a new one for the Q&A.

6.1 Pyscaffold Usage

Does my project depend on PyScaffold when I use it to set my project up? Starting from version 4, your package is completely independent from PyScaffold, we just kick-start your project and take care of the boilerplate. However, we do include some build-time dependencies that make your life easier, such as setuptools_scm. But don't worry, if you distribute your project in the recommended wheel format those dependencies will not affect the final users, they are just required during development to assembling the package file.

That means if someone clones your repository and tries to build it, the dependencies in `pyproject.toml` will be automatically pulled. This mechanism is described by PEP 517/PEP 518 and definitely beyond the scope of this answer.

Can I use PyScaffold 3 to develop a Python package that is Python 2 & 3 compatible? Python 2 reached *end-of-life* in 2020, which means that no security updates will be available, and therefore any software running on Python 2 is potentially vulnerable. PyScaffold strongly recommends all packages to be ported to the latest supported version of Python.

That being said, Python 3 is actually only needed for the `putup` command and whenever you use `setup.py`. This means that with PyScaffold 3 you have to use Python 3 during the development of your package for practical reasons. If you develop the package using six you can still make it Python 2 & 3 compatible by creating a *universal* `bdist_wheel` package. This package can then be installed and run from Python 2 and 3. Just have in mind that no support for Python 2 will be provided.

How can I get rid of PyScaffold when my project was set up using it? First of all, I would really love to understand **why** you want to remove it and **what** you don't like about it. You can create an issue for that or just text me on Twitter. But the good news is that your project is completely independent of PyScaffold, even if you uninstall it, everything will be fine.

If you still want to remove setuptools_scm (a build-time dependency we add by default), it's actually really simple:

- Within `setup.py` remove the `use_scm_version` argument from the `setup()`
- Remove the `[tool.setuptools_scm]` section of `pyproject.toml`.

This will deactivate the automatic version discovery. In practice, following things will **no** longer work:

- `python setup.py --version` and the dynamic versioning according to the git tags when creating distributions, just put e.g. `version = 0.1` in the `metadata` section of `setup.cfg` instead,

That's already everything you gonna lose. Not that much. You will still benefit from:

- the smart project layout,
- the declarative configuration with `setup.cfg` which comes from setuptools,
- some sane defaults in Sphinx' `conf.py`,
- `.gitignore` with some nice defaults and other dot files depending on the flags used when running `putup`,
- some sane defaults for pytest.

For further cleanups, feel free to remove the dependencies from the `requires` key in `pyproject.toml` as well as the complete `[pyscaffold]` section in `setup.cfg`.

Why would I use PyScaffold instead of Cookiecutter? PyScaffold is focused on a good out-of-the-box experience for developing distributable Python packages (exclusively). The idea is to standardize the structure of Python packages. Thus, PyScaffold sticks to

> "There should be one– and preferably only one –obvious way to do it."

from the Zen of Python. The long-term goal is that PyScaffold becomes for Python what Cargo is for Rust. Still, with the help of PyScaffold's *extension system* customizing a project scaffold is possible.

Cookiecutter on the other hand is a really flexible templating tool that allows you to define own templates according to your needs. Although some standard templates are provided that will give you quite similar results as PyScaffold, the overall goal of the project is quite different.

Still, if you so desire, PyScaffold allows users to augment PyScaffold projects with certain types of cookiecutter templates, through its pyscaffoldext-cookiecutter extension.

How can I embed PyScaffold into another application? PyScaffold is expected to be used from terminal, via `putup` command line application. It is, however, possible to write an external script or program that embeds PyScaffold and use it to perform some custom actions.

The public Python API for embedding PyScaffold is composed by the main function *pyscaffold.api.create_project* in addition to *pyscaffold.api.NO_CONFIG*, *pyscaffold.log.DEFAULT_LOGGER*, *pyscaffold.log.logger* (partially, see details bellow), and the constructors for the extension classes belonging to the `pyscaffold.extenstions` module (the other methods and functions are not considered part of the API). This API, as explicitly listed, follows Semantic Versioning and will not change in a backwards incompatible way between releases. The remaining methods and functions are not guaranteed to be stable.

The following example illustrates a typical embedded usage of PyScaffold:

```
import logging

from pyscaffold.api import create_project
from pyscaffold.extenstions.cirrus import Cirrus
from pyscaffold.extenstions.namespace import Namespace
from pyscaffold.log import DEFAULT_LOGGER as LOGGER_NAME

logging.getLogger(LOGGER_NAME).setLevel(logging.INFO)

create_project(
    project_path="my-proj-name",
    author="Your Name",
    namespace="some.namespace",
    license="MIT",
    extensions=[Cirrus(), Namespace()],
)
```

Note that no built-in extension (e.g. **cirrus** and **namespace**) is activated by default. The `extensions` option should be manually populated when convenient.

PyScaffold uses the logging infrastructure from Python standard library, and emits notifications during its execution. Therefore, it is possible to control which messages are logged by properly setting the log level (internally, most of the messages are produced under the `INFO` level). By default, a `StreamHandler` is attached to the logger, however it is possible to replace it with a custom handler using `logging.Logger.removeHandler` and `logging.Logger.addHandler`. The logger object is available under the *`logger`* variable of the *`pyscaffold.log`* module. The default handler is available under the *`handler`* property of the *`logger`* object.

How can I use PyScaffold if my project is nested within a larger repository, e.g. in a monorepo? If you use PyScaffold to create a Python project within another larger repository, you will see the following error when building your package:

```
LookupError: setuptools-scm was unable to detect version for '/path/to/your/project
↪'::
```

This is due to the fact that setuptools_scm assumes that the root of your repository is where `pyproject.toml` resides. In order to tell setuptools_scm where the actual root is some changes have to be made. In the example below we assume that the root of the repository is the parent directory of your project, i.e. `..` as relative path. In any case you need to specify the root of the repository relative to the root of your project.

1. `pyproject.toml`:

```
[tool.setuptools_scm]
# See configuration details in https://github.com/pypa/setuptools_scm
version_scheme = "no-guess-dev"
# ADD THE TWO LINES BELOW
root = ".."
relative_to = "setup.py"
```

2. `setup.py`:

```
setup(use_scm_version={"root": "..",  # ADD THIS...
                       "relative_to": __file__,  # ... AND THAT!
                       "version_scheme": "no-guess-dev"})
```

In future versions of PyScaffold this will be much simpler as `pyproject.toml` will completely replace `setup.py`.

What is the license of the generated project scaffold? Is there anything I need to consider? The source code of PyScaffold itself is MIT-licensed with the exception of the **.template* files under the `pyscaffold.templates` subpackage, which are licensed under the BSD 0-Clause license (0BSD). Thus, also the generated boilerplate code for your project is 0BSD-licensed and consequently you have no obligations at all and can do whatever you want except of suing us ;-)

6.2 File Organisation and Directory Structure

Why does PyScaffold 3 have a `src` folder which holds the actual Python package? This avoids quite many problems compared to the case when the actual Python package resides in the same folder as `setup.py`. A nice blog post by Ionel gives a thorough explanation why this is so. In a nutshell, the most severe problem comes from the fact that Python imports a package by first looking at the current working directory and then into the `PYTHONPATH` environment variable. If your current working directory is the root of your project directory you are thus not testing the installation of your package but the local package directly. Eventually, this always leads to huge confusion (*"But the unit tests ran perfectly on my machine!"*).

Moreover, having a dedicated `src` directory to store the package files, makes it easy to comply with recent standards in the Python community (for example PEP 420).

Please notice that PyScaffold assumes all the files inside `src` are meant to be part of the package.

Can I have other files inside the `src` folder that are not meant for distribution? PyScaffold considers the `src` directory to be exclusively dedicated to store files meant to be distributed, and relies on this assumption to generate configuration for the several aspects of your project. Therefore it is not recommended to include any file not meant to distribution inside the `src` folder. (Temporary files and directories automatically generated by `setuptools` might appear from times to times though).

Where should I put extra files not meant for distribution? You can use the `docs` folder (if applicable) or create another dedicated folder in the root of your repository (e.g. `examples`). The additional project structure created by the pyscaffoldext-dsproject is a good example on how to use extra folders to achieve good project organisation.

6.3 Namespaces

How can I get rid of the implicit namespaces (PEP 420)? PyScaffold uses `setup.cfg` to ensure setuptools will follow PEP 420. If this configuration particularly messes up with your package, or you simply want to follow the old behavior, please replace `packages = find_namespace:` with `packages = find:` in the `[options]` section of that file.

You should also remove the `--implicit-namespaces` option in the `cmd_line_template` variable in the `docs/conf.py` file.

Finally, if want to keep a namespace but use an explicit implementation (old behavior), make sure to have a look on the packaging namespace packages official guide. If there are already other projects with packages registered in the same namespace, chances are you just need to copy from them a sample of the `__init__.py` file for the umbrella folder working as namespace.

My namespaced package and/or other packages that use the same namespace broke after updating to PyScaffold 4. How can I fi
That is likely to be happening because PyScaffold 4 removed support for pkg_resources namespaces in favour of PEP 420. Unfortunately these two methodologies for creating namespaces are not compatible, as documented in the packaging namespace packages official guide. To fix this problem you (or other maintainers) will need to either **(a)** update all the existing "subpackages" in the same namespace to be implicit (PEP 420-style), or **(b)** get rid of the implicit namespace configuration PyScaffold automatically sets up during project creation/update. Please check the answers for these other questions about *removing* or *adding* implicit namespaces and the *updating* guides for some tips on how to achieve that.

How can I convert an existing package to use implicit namespaces (PEP 420)? The easiest answer for that question is to **(a)** convert the existing package to a PyScaffold-enabled project (*if it isn't yet*; please check *our guides* for instructions) and **(b)** *update* your existing project to the latest version of PyScaffold passing the correct `--namespace` option.

The slightly more difficult answer for that question is to **(a)** make sure your project uses a src layout, **(b)** remove the `__init__.py` file from the umbrella folder that is serving as namespace for your project, **(c)** configure `setup.cfg` to include your namespace – have a look on setuptools, for packages that use the `src-layout` that basically means that you want to have something similar to:

```
[options]
# ...
packages = find_namespace:
package_dir =
    =src
# ...

[options.packages.find]
where = src
```

in your `setup.cfg` – and finally, **(d)** configure your documentation to include the implicit namespace (for Sphinx users, in general that will mean that you want to run `sphinx-apidoc` with the `--implicit-namespaces` flag after extending the `PYTHONPATH` with the `src` folder).

The previous steps assume your existing package uses setuptools and you are willing to have a src layout, if that is not the case refer to the documentation of your package creator (or the software you use to package up your Python projects) and the PEP 420 for more information.

6.4 pyproject.toml

Can I modify `requires` despite the warning in `pyproject.toml` to avoid doing that? You can definitely modify `pyproject.toml`, but it is good to understand how PyScaffold uses it. If you are just adding a new build dependency (e.g. Cython), there is nothing to worry. However, if you are trying to remove or change the version of a dependency PyScaffold included there, PyScaffold will overwrite that change if you ever run `putup --update` in the same project (in those cases `git diff` is your friend, and you should be able to manually reconcile the dependencies).

What should I do if I am not using `pyproject.toml` or if it is causing me problems? If you prefer to have legacy builds and get the old behavior, you can remove the `pyproject.toml` file and run `python setup.py bdist_wheel`, but we advise to install the build requirements (as the ones specified in the `requires` field of `pyproject.toml`) in an isolated environment and use it to run the `setup.py` commands (tox can be really useful for that). Alternatively you can use the `setup_requires` field in setup.cfg, however, this method is discouraged and might be invalid in the future.

Note: For the time being you can use the **transitional** `--no-pyproject` option, when running `putup`, but have in mind that this option will be removed in future versions of PyScaffold.

Please check our *updating guide* for *extra steps* you might need to execute manually.

6.5 Best Practices and Common Errors with Version Numbers

How do I get a clean version like 3.2.4 when I have 3.2.3.post0.dev9+g6817bd7? Just commit all your changes and create a new tag using `git tag v3.2.4`. In order to build an old version checkout an old tag, e.g. `git checkout -b v3.2.3 v3.2.3` and run `tox -e build` or `python setup.py bdist_wheel`.

Why do I see *unknown* as version? In most cases this happens if your source code is no longer a proper Git repository, maybe because you moved or copied it or Git is not even installed. In general using `pip install -e .`, `python setup.py install` or `python setup.py develop` to install your package is only recommended for developers of your Python project, which have Git installed and use a proper Git repository anyway. Users of your project should always install it using the distribution you built for them e.g. `pip install my_project-3.2.3-py3-none-any.whl`. You build such a distribution by running `tox -e build` (or `python setup.py bdist_wheel`) and then find it under `./dist`.

Is there a good versioning scheme I should follow? The most common practice is to use Semantic Versioning. Following this practice avoids the so called dependency hell for the users of your package. Also be sure to set attributes like `python_requires` and `install_requires` appropriately in `setup.cfg`.

Is there a best practice for distributing my package? First of all, cloning your repository or just coping your code around is a really bad practice which comes with tons of pitfalls. The *clean* way is to first build a distribution and then give this distribution to your users. This can be done by just copying the distribution file or uploading it to some artifact store like PyPI for public packages or devpi, Nexus, etc. for private packages. Also check out this article about packaging, versioning and continuous integration.

Using some CI service, why is the version *unknown* or *my_project-0.0.post0.dev50*? Some CI services use shallow git clones, i.e. `--depth N`, or don't download git tags to save bandwidth. To verify that your repo works as expected, run:

```
git describe --dirty --tags --long --first-parent
```

which is basically what setuptools_scm does to retrieve the correct version number. If this command fails, tweak how your repo is cloned depending on your CI service and make sure to also download the tags, i.e. `git fetch origin --tags`.

How can I build a distribution if I have only the source code without a proper git repo? If you see an error message like:

```
setuptools-scm was unable to detect version for 'your/project'.
```

This means that `setuptools-scm` could not find an intact git repository. If you still want to build a distribution from the source code there is a workaround. In `setup.cfg` in the section `[metadata]` define a version manually with e.g. `version = 1.0`. Now remove from `pyproject.toml` the requirement `use_scm_version={"version_scheme": "no-guess-dev"}` if you use isolated builds with `tox` and/or `"setuptools_scm[toml]>=5"` from `setup.cfg` if you use `python setup.py bdist_wheel` to build.

CHAPTER

SEVEN

CONTRIBUTING

PyScaffold was started by Blue Yonder developers to help automating and standardizing the process of project setups. Nowadays it is a pure community project driven by volunteer work. Every little gesture is really appreciated (including issue reports!), and if you are interested in joining our continuous effort for making PyScaffold better, welcome aboard! We are pleased to help you in this journey .

Note: This document is an attempt to get any potential contributor familiarized with PyScaffold's community processes, but by no means is intended to be a complete reference.

Please feel free to contact us for help and guidance in our GitHub discussions page.

Please notice, all the members of the PyScaffold community (and in special contributors) are expected to be **open, considerate, reasonable, and respectful**. and follow the Python Software Foundation's Code of Conduct when interacting with PyScaffold's codebases, issue trackers, chat rooms and mailing lists (or equivalent).

Tip: If you are new to open source or have never contributed before to a software project, please have a look at contribution-guide.org and the How to Contribute to Open Source guide. Other resources are also listed in the excellent guide created by FreeCodeCamp.

7.1 How to contribute to PyScaffold?

This guide focus on issue reports, documentation improvements, and code contributions, but there are many other ways to contribute to PyScaffold, even if you are not an experienced programmer or don't have the time to code. Skills like graphical design, event planning, teaching, mentoring, public outreach, tech evangelism, code review, between many others are greatly appreciated. Please reach us out, we would love to have you on board and discuss what can be done!

7.1.1 Issue Reports

If you experience bugs or general issues with PyScaffold, please have a look on our issue tracker.

Note: Please don't forget to include the closed issues in your search. Sometimes another person has already experienced your problem and reported a solution. If you don't see anything useful there, feel free to fire a new issue report

---

New issue reports should include information about your programming environment (e.g., operating system, Python version) and steps to reproduce the problem. Please try also to simplify the reproduction steps to a very minimal

example that still illustrates the problem you are facing. By removing other factors, you help us to identify the root cause of the issue.

7.1.2 Documentation Improvements

You can help us improve our docs by making them more readable and coherent, or by adding missing information and correcting mistakes (including spelling and grammar errors).

Already known and discussed documentation issues that would benefit from contributions are marked in our issue tracker with the **documentation** label (we also do the same for all existing extensions under the PyScaffold organization on GitHub). But you are also welcomed to propose completely new changes (e.g., if you find new problems or would like to see a complicated topic better explained).

PyScaffold's documentation is written in reStructuredText and uses Sphinx as its main documentation compiler[1]. This means that the docs are kept in the same repository as the project code, and that any documentation update is done via GitHub pull requests, as if it was a code contribution.

While that might be scary for new programmers, it is actually a very nice way of getting started in the open source community, since doc contributions are not as difficult to make as other code contributions (for example, they don't require any automated testing).

Please have a look in the steps described below and in case of doubts, contact us at the GitHub discussions page for help.

When working on changes to PyScaffold's docs in your local machine, you can compile them using `tox`:

```
tox -e docs
```

and use Python's built-in web server for a preview in your web browser (`http://localhost:8000`):

```
python3 -m http.server --directory 'docs/_build/html'
```

Tip: Please notice that the GitHub web interface provides a quick way of propose changes in PyScaffold's files, that do not require you to have a lot of experience with git or programing in general. While this mechanism can be tricky for normal code contributions, it works perfectly fine for contributing to the docs, and can be quite handy.

If you are interested in trying this method out, please navigate to PyScaffold's `docs` folder in the main repository, find which file you would like to propose changes and click in the little pencil icon at the top, to open GitHub's code editor. Once you finish editing the file, please write a nice message in the form at the bottom of the page describing which changes have you made and what are the motivations behind them and submit your proposal.

7.1.3 Code Contributions

PyScaffold uses GitHub's fork and pull request workflow for code contributions, which means that anyone can propose changes in the code base.

Once proposed changes are submitted, our continuous integration (CI) service, Cirrus-CI, will run a series of automated checks to make sure everything is OK and the pull request (PR) itself will be reviewed by one of PyScaffold maintainers, before being merged in the code base. In some cases, changes might be required to fix problems pointed out by the CI, or the maintainers might want to discuss a bit about the PR and suggest adjustments. Please don't worry if that happens, this kind of iterative development is very common in the open source community and usually makes the software better. Besides, we will do our best to provide feedback (and support for eventual doubts) as soon as we can.

[1] The same is valid for the extensions under the PyScaffold organization on GitHub, although some extension, like pyscaffoldext-markdown and pyscaffoldext-dsproject use CommonMark with MyST extensions as an alternative to reStructuredText.

If you are unsure about what to contribute, please have a look in our issue tracker (or the issue tracker of any extension under the PyScaffold organization on GitHub). Contributions on issues marked with the **help wanted** label are particularly appreciated. Moreover, the **good first issue** label marks issues that do not require a huge understanding on how the project works and therefore can be tackled by new members of the community. Please also notice that some issues are not ready yet for a follow up implementation or bug fix, these are usually signed with other labels, such as **needs discussion** and **waiting response**. When in doubt, please engage in the conversation by posting a message to the open issue.

Understanding how PyScaffold works

If you have a change in mind, but don't know how to implement it, please have a look in our *Developer Guide*. It explains the main aspects of PyScaffold internals and provide a brief overview of how the project is organized.

Submit an issue

Before you work on any non-trivial code contribution it's best to first create an issue report to start a discussion on the subject. This often provides additional considerations and avoids unnecessary work.

Create an environment

Before you start coding, we recommend creating an isolated virtual environment to avoid any problems with your installed Python packages. This can easily be done via either `virtualenv`:

```
virtualenv <PATH TO VENV>
source <PATH TO VENV>/bin/activate
```

or Miniconda:

```
conda env create -d environment.yml
conda activate pyscaffold
```

Clone the repository

1. Create a GitHub account if you do not already have one.
2. Fork the project repository: click on the *Fork* button near the top of the page. This creates a copy of the code under your account on the GitHub server.
3. Clone this copy to your local disk:

   ```
   git clone git@github.com:YourLogin/pyscaffold.git
   cd pyscaffold
   ```

4. You should run:

   ```
   pip install -U pip setuptools -e .
   ```

   to be able run `putup --help`.

5. Install `pre-commit`:

   ```
   pip install pre-commit
   pre-commit install
   ```

PyScaffold project comes with a lot of hooks configured to automatically help the developer to check the code being written.

Implement your changes

1. Create a branch to hold your changes:

```
git checkout -b my-feature
```

   and start making changes. Never work on the master branch!

2. Start your work on this branch. Don't forget to add docstrings to new functions, modules and classes, especially if they are part of public APIs.

3. Add yourself to the list of contributors in `AUTHORS.rst`.

4. When you're done editing, do:

```
git add <MODIFIED FILES>
git commit
```

   to record your changes in git. Please make sure to see the validation messages from `pre-commit` and fix any eventual issues. This should automatically use flake8/black to check/fix the code style in a way that is compatible with PyScaffold.

   Important: Don't forget to add unit tests and documentation in case your contribution adds an additional feature and is not just a bugfix.

   Moreover, writing a descriptive commit message is highly recommended. In case of doubt, you can check the commit history with:

```
git log --graph --decorate --pretty=oneline --abbrev-commit --all
```

   to look for recurring communication patterns.

   ---

5. Please check that your changes don't break any unit tests with:

```
tox
```

   (after having installed `tox` with `pip install tox` or `pipx`).

   To speed up running the tests, you can try to run them in parallel, using `pytest-xdist`. This plugin is already added to the test dependencies, so everything you need to do is adding `-n auto` or `-n <NUMBER OF PROCESSES>` in the CLI. For example:

```
tox -- -n 15
```

   Please have in mind that PyScaffold test suite is IO intensive, so using a number of processes slightly bigger than the available number of CPUs is a good idea. For quicker feedback you can also try:

```
tox -e fast
```

   or select individual tests using the `-k` flag from pytest:

```
tox -- -k <NAME OF THE TEST FUNCTION>
```

You can also use `tox` to run several other pre-configured tasks in the repository. Try `tox -av` to see a list of the available checks.

Submit your contribution

1. If everything works fine, push your local branch to GitHub with:

```
git push -u origin my-feature
```

2. Go to the web page of your PyScaffold fork and click "Create pull request" to send your changes to the maintainers for review. Find more detailed information in creating a PR. You might also want to open the PR as a draft first and mark it as ready for review after the feedbacks from the continuous integration (CI) system or any required fixes.
3. If you are submitting a change related to an existing CI system template (e.g., travis, cirrus, or even tox and pre-commit), please consider first submitting a companion PR to PyScaffold's ci-tester, with the equivalent files changes, so we are sure it works.

   If you are proposing a new CI system template, please send us a link of a simple repository generated with your templates (a simple `putup --<YOUR EXTENSION> ci-tester` will do) and the CI logs for that repository.

   This helps us a lot to control breaking changes that might appear in the future.

Troubleshooting

I've got a strange error related to versions in `test_update.py` when executing the test suite or about an `entry_point` that cannot be found.

Make sure to fetch all the tags from the upstream repository, the command `git describe --abbrev=0 --tags` should return the version you are expecting. If you are trying to run the CI scripts in a fork repository, make sure to push all the tags. You can also try to remove all the egg files or the complete egg folder, i.e., `.eggs`, as well as the `*.egg-info` folders in the `src` folder or potentially in the root of your project. Afterwards run `python setup.py egg_info --egg-base .` again.

I've got a strange syntax error when running the test suite. It looks like the tests are trying to run with Python 2.7 ...

Try to create a dedicated virtual environment using Python 3.6+ (or the most recent version supported by PyScaffold) and use a `tox` binary freshly installed. For example:

```
virtualenv .venv
source .venv/bin/activate
.venv/bin/pip install tox
.venv/bin/tox -e all
```

I have found a weird error when running `tox`. It seems like some dependency is not being installed.

Sometimes `tox` misses out when new dependencies are added, especially to `setup.cfg` and `docs/requirements.txt`. If you find any problems with missing dependencies when running a command with `tox`, try to recreate the `tox` environment using the `-r` flag. For example, instead of:

```
tox -e docs
```

Try running:

```
tox -r -e docs
```

I am trying to debug the automatic test suite, but it is very hard to understand what is happening.

Pytest can drop you in an interactive session in the case an error occurs. In order to do that you need to pass a `--pdb` option (for example by running `tox -- -k <NAME OF THE FALLING TEST> --pdb`). While `pdb` does not have the best user interface in the world, if you feel courageous, it is possible to use an alternate implementation like ptpdb and bpdb (please notice some of them might require additional options, such as `--pdbcls ptpdb:PtPdb`/`--pdbcls bpdb:BPdb`). You will need to temporarily add the respective package as a dependency in your `tox.ini` file. You can also setup breakpoints manually instead of using the `--pdb` option.

7.1.4 Code Reviews and Issue Triage

If you are an experienced developer and wants to help, but do not have the time to create complete pull requests, you can still help by reviewing existing open pull requests, or going through the open issues and evaluating them according to our labels and even suggesting possible solutions or workarounds.

7.1.5 Maintainer tasks

PyScaffold maintainers not only carry out most of the source code development, but also are responsible for planning new releases, reviewing pull requests, and managing CI tools between many other tasks. If you are interested in becoming a maintainer, the best is to keep "hanging out" in the community, helping with the issues, proposing PRs and doing some code review (either in the main repository or the extensions under the PyScaffold organization on GitHub). Eventually, one of the existing maintainers will approach you and ask you to join .

This section describes some technical aspects of recurring tasks and is meant as a guide for new maintainers (or old ones that need a memory refresher).

Releases

New PyScaffold releases should be automatically uploaded to PyPI by one of our GitHub actions every time a new tag is pushed to the repository. Therefore, as a PyScaffold maintainer, the following steps are all you need to release a new version:

1. Make sure all unit tests on Cirrus-CI are green.
2. Tag the current commit on the master branch with a release tag, e.g., `v1.2.3`.
3. Push the new tag to the upstream repository, e.g., `git push upstream v1.2.3`
4. After a few minutes check if the new version was uploaded to PyPI

If, for some reason, you need to manually create a new distribution file and upload to PyPI, the following extra steps can be used:

1. Clean up the `dist` and `build` folders with `tox -e clean` (or `rm -rf dist build`) to avoid confusion with old builds and Sphinx docs.
2. Run `tox -e build` and check that the files in `dist` have the correct version (no `.dirty` or git hash) according to the git tag. Also sizes of the distributions should be less than 500KB, otherwise unwanted clutter may have been included.
3. Run `tox -e publish -- --repository pypi` and check that everything was uploaded to PyPI correctly.

Important: When working in a new **external extension**, it is important that the first distribution is manually uploaded to PyPI, to make sure it will have the correct ownership.

After successful releases (especially of new major versions), it is a good practice to re-generate our example repository. To manually do that, please visit our GitHub actions page and run the **Make Demo Repo** workflow (please check if it was not automatically triggered already).

Working on multiple branches and splitting complex changes

PyScaffold follows semantic versioning. As a consequence, most of the times the `master` (or `main`) branch for either the main repository or the extensions under the PyScaffold organization on GitHub, should be pointing out to the latest published minor version, or the next minor version still under development. We also tend (but are not committed to) keep some level of support for the previous major version, which means that once a major version is superseded, the maintainers should create a new branch pointing to this previous version.

For this reason, Read the Docs should always be configured to show the **stable** version by default instead of **latest**. The **stable** version corresponds to the latest commit that received a git tag, while the **latest** version points to the **master/main** branch.

During the transition period between major versions, it is common practice to create a new *development* version that is kept apart from the master branch and will only be merged when everything is ready for release. For example, a `v4.0.x` branch was used for all the development related to PyScaffold v4, while the `master` branch was still being used for bug fixes to v3.

When working in complex features or refactoring, it might also be interesting to create a new long-living branch that will receive multiple PRs from other short-lived auxiliary branch splitting the changes into smaller steps. Please be aware that splitting complex changes into smaller PRs can be very tricky. Whenever possible, try to create independent PRs, i.e., PRs that can be merged independently into a long-living branch, without causing conflicts between themselves. When that is not possible, please coordinate a review and merge strategy with the other maintainers reviewing your code.

One possible strategy is to create a single PR, but ask your reviewers to consider each commit (that should be small) as if it was an independent PR. A different strategy is to use **stacked PRs**, as described by the following references:

- Timothy Andrew's Blog
- Doctor McKayla's Blog
- Div's Blog
- LogRocket's Blog

Please also notice that independently of the strategy you and the reviewers agree on, it might be worthy to ask them to just review the PRs without merging (so you are responsible for closing the PRs and bringing their code to the long-lived branch via git `merge`, `pull` or `cherry-pick`). This might avoid confusion since GitHub does not provide any special mechanism for dealing with dependencies between PRs. Moreover, the merging might be just easier via git CLI.

Note: PyScaffold's repositories also contain `archives/*` branches. These branches correspond to old experiments and alternative feature implementations that, although not merged, are kept for reference as interesting (or very complex) pieces of code that might be useful in the future.

7.1.6 Spread the Word

Finally, another way to contribute to PyScaffold is to recommend it. You can speak about it with your work colleagues, in a conference, or even writing a blog post about the project.

If you need to pitch PyScaffold to your boss or co-workers, please check out our docs. We have enumerated a few *reasons for using PyScaffold* in our website, that can be handy to have around .

CHAPTER

EIGHT

DEVELOPER GUIDE

This document describes the internal architecture and the main concepts behind PyScaffold. It assumes the reader has some experience in *using* PyScaffold (specially its command line interface, `putup`) and some familiarity with Python's package ecosystem.

Please notice this document does not target PyScaffold's users, instead it provides **internal** documentation for those who are involved in PyScaffold's development.

8.1 Architecture

As indicated in the figure bellow, PyScaffold can be divided in two main execution blocks: a pure Python API and the command line interface wrapping it as an executable program that runs on the shell.

The CLI is responsible for defining all arguments `putup` accepts and parsing the user input accordingly. The result is a `dict` that contains options expressing the user preference and can be fed into PyScaffold's main API, *`create_project`*.

This function is responsible for combining the provided options `dict` with pre-existing project configurations that might be available in the project directory (the `setup.cfg` file, if present) and globally defined default values (via *PyScaffold's own configuration file*). It will then create an (initially empty) *in-memory* representation of the project structure and run PyScaffold's action pipeline, which in turn will (between other tasks) write customized versions of PyScaffold's templates to the disk as project files, according to the combined scaffold options.

The project representation and the action pipeline are two key concepts in PyScaffold's architecture and are described in detail in the following sections.

8.2 Project Structure Representation

Each Python package project is internally represented by PyScaffold as a tree data structure, that directly relates to a directory entry in the file system. This tree is implemented as a simple (and possibly nested) `dict` in which keys indicate the path where files will be generated, while values indicate their content. For instance, the following dict:

```
{
    "folder": {
        "file.txt": "Hello World!",
        "another-folder": {
            "empty-file.txt": ""
        }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

represents a project directory in the file system that contains a single directory named `folder`. In turn, `folder` contains two entries. The first entry is a file named `file.txt` with content `Hello World!` while the second entry is a sub-directory named `another-folder`. Finally, `another-folder` contains an empty file named `empty-file.txt`.

Note: Changed in version 4.0: Prior to version 4.0, the project structure included the top level directory of the project. Now it considers everything **under** the project folder.

Additionally, tuple values are also allowed in order to specify a **file operation** (or simply **file op**) that will be used to produce the file. In this case, the first element of the tuple is the file content, while the second element will be a function (or more generally a `callable` object) responsible for writing that content to the disk. For example, the dict:

```
from pyscaffold.operations import create

{
    "src": {
        "namespace": {
            "module.py": ('print("Hello World!")', create)
        }
    }
}
```

represents a `src/namespace/module.py` file, under the project directory, with content `print("Hello World!")`, that will written to the disk. When no operation is specified (i.e. when using a simple string instead of a tuple), PyScaffold will assume *`create`* by default.

Note: The *`create`* function simply creates a text file to the disk using UTF-8 encoding and the default file permissions. This behaviour can be modified by wrapping *`create`* within other fuctions/callables, for example:

```
from pyscaffold.operations import create, no_overwrite

{"file": ("content", no_overwrite(create))}
```

will prevent the `file` to be written if it already exists. See *`pyscaffold.operations`* for more information on how to write your own file operation and other options.

Finally, while it is simple to represent file contents as a string directly, most of the times we want to *customize* them according to the project parameters being created (e.g. package or author's name). So PyScaffold also accepts `string.Template` objects and functions (with a single `dict` argument and a `str` return value) to be used as contents. These templates and functions will be called with *`PyScaffold's options`* when its time to create the file to the disk.

Note: `string.Template` objects will have `safe_substitute` called (not simply `substitute`).

This tree representation is often referred in this document as **project structure** or simply **structure**.

8.3 Action Pipeline

PyScaffold organizes the generation of a project into a series of steps with well defined purposes. As shown in the figure bellow, each step is called **action** and is implemented as a simple function that receives two arguments: a project structure and a `dict` with options (some of them parsed from command line arguments, other from default values).

An action **MUST** return a tuple also composed by a project structure and a `dict` with options. The return values, thus, are usually modified versions of the input arguments. Additionally an action can also have side effects, like creating directories or adding files to version control. The following pseudo-code illustrates a basic action:

```
def action(project_structure, options):
    new_struct, new_opts = modify(project_structure, options)
    some_side_effect()
    return new_struct, new_opts
```

The output of each action is used as the input of the subsequent action, forming a pipeline. Initially the structure argument is just an empty `dict`. Each action is uniquely identified by a string in the format `<module name>:<function name>`, similarly to the convention used for a setuptools entry point. For example, if an action is defined in the `action` function of the `extras.py` file that is part of the `pyscaffoldext.contrib` project, the **action identifier** is `pyscaffoldext.contrib.extras:action`.

By default, the sequence of actions taken by PyScaffold is:

1. *pyscaffold.actions:get_default_options*
2. *pyscaffold.actions:verify_options_consistency*
3. *pyscaffold.structure:define_structure*
4. *pyscaffold.actions:verify_project_dir*
5. *pyscaffold.update:version_migration*
6. *pyscaffold.structure:create_structure*
7. *pyscaffold.actions:init_git*
8. *pyscaffold.actions:report_done*

(as given by *pyscaffold.actions.DEFAULT*)

The project structure is usually empty until *define_structure* This action just loads the in-memory `dict` representation, that is only written to disk by the *create_structure* action.

Note that, this sequence varies according to the command line options. To retrieve an updated list, please use `putup --list-actions` or `putup --dry-run`.

8.4 Extensions

Extensions are a mechanism provided by PyScaffold to modify its action pipeline at runtime and the preferred way of adding new functionality. There are **built-in extensions** (e.g. *pyscaffold.extensions.cirrus*) and **external extensions** (e.g. pyscaffoldext-dsproject), but both types of extensions work exactly in the same way. This division is purely based on the fact that some of PyScaffold features are implemented as extensions that ship by default with the `pyscaffold` package, while other require the user to install additional Python packages.

Extensions are required to add at least one CLI argument that allow the users to opt-in for their behaviour. When `putup` runs, PyScaffold's will dynamically discover installed extensions via setuptools entry points and add their defined

arguments to the main CLI parser. Once activated, a extension can use the helper functions defined in *pyscaffold.actions* to manipulate PyScaffold's action pipeline and therefore the project structure.

For more details on extensions, please consult our *Extending PyScaffold* guide.

8.5 Code base Organization

PyScaffold is organized in a series of internal Python modules, the main ones being:

- *api*: top level functions for accessing PyScaffold functionality, by combining together the other modules
- *cli*: wrapper around the API to create a command line executable program
- *actions*: default action pipeline and helper functions for manipulating it
- *structure*: functions specialized in defining the in-memory project structure representation and in taking this representation and creating it as part of the file system.
- *update*: steps required for updating projects generated with old versions of PyScaffold
- *extensions*: main extension mechanism and subpackages corresponding to the built-in extensions

Additionally, a series of internal auxiliary libraries is defined in:

- *dependencies*: processing and manipulating of package dependencies and requirements
- *exceptions*: custom PyScaffold exceptions and exception handlers
- *file_system*: wrappers around file system functions that make them easy to be used from PyScaffold.
- *identification*: creating and processing of project/package/function names and other general identifiers
- *info*: general information about the system, user and package being generated
- *log*: custom logging infrastructure for PyScaffold, specialized in its verbose execution
- *operations*: file operations that can be embedded in the in-memory project structure representation
- *repo*: wrapper around the `git` command
- *shell*: helper functions for working with external programs
- *termui*: basic support for ANSI code formatting
- *toml*: thin adapter layer around third-party TOML parsing libraries, focused in API stability

For more details about each module and its functions and classes, please consult our *module reference*.

When contributing to PyScaffold, please try to maintain this overall project organization by respecting each module's own purpose. Moreover, when introducing new files or renaming existing ones, please try to use meaningful naming and avoid terms that are too generic, e.g. `utils.py` (when in doubt, Peter Hilton has a great article about naming smells and a nice presentation aboug how to name things).

CHAPTER

NINE

CONTRIBUTORS

- Florian Wilhelm
- Felix Wick
- Holger Peters
- Uwe Korn
- Patrick Mühlbauer
- Florian Rathgeber
- Eva Schmücker
- Tim Werner
- Julian Gethmann
- Will Usher
- Anderson Bravalheri
- David Hilton
- Pablo Aguiar
- Vicky C Lau
- Reuven Podmazo
- Juan Leni
- Anthony Sottile
- Henning Häcker
- Noah Pendleton
- John Vandenberg
- Ben Mares

CHAPTER

TEN

CHANGELOG

10.1 Current versions

10.1.1 Version 4.1.1, 2021-10-18

- Ensure required extensions are installed on `--update`, PR #512
- Prevent extension from crashing when persisting `None` in `setup.cfg`, PR #510
- Prevent multi-line descriptions to crash `putup`, PR #509
- Warn users about empty namespaces, PR #508
- Prevent parsing errors during dependency deduplication, PR #518
- Add `license_files` to `setup.cfg` template, issue #524

10.1.2 Version 3.3, 2020-12-24

- Code base changed to Black's standards
- New docs about version numbers and git integration
- Updated pre-commit hooks
- Updated `docs/Makefile` to use Sphinx "make mode"
- *deprecated* setuptools extensions/commands `python setup.py test/docs/doctests`, issue #245
- New tox test environments for generating docs and running doctests
- New built-in extension for Cirrus CI, issue #251
- *experimental* `get_template` is now part of the public API and can be used by extensions, issue #252
- Updated `setuptools_scm` to version 4.1.2 in contrib
- Updated `configupdater` to version 1.1.2 in contrib
- precommit automatically fixes line endings by default
- *deprecated* `log.configure_logger`, use `log.logger.reconfigure` instead

Note: PyScaffold 3.3 is the last release to support Python 3.5

10.2 Older versions

10.2.1 Version 4.1, 2021-09-22

- Added *linkcheck* task to `tox.ini`, PR #456
- Updated configuration for Sphinx and ReadTheDocs, PR #455
- Note that templates and the generated boilerplate code is 0BSD-licensed, PR #461
- Added 0BSD license template
- Added `CONTRIBUTING.rst` template, issue #376
- Added PyScaffold badge to `README` template, issue #473
- Updated Cirrus CI config and templates, including better `coveralls` integration, issue #449
- Adopted global `isolated_build` for `tox` configuration, issue #483, PR #491
- Loop counter explicitly marked as unused in `skeleton.py` (`flake8-bugbear B007`), PR #495
- Ensure update include added extensions in `setup.cfg`, PR #496

10.2.2 Version 4.0.2, 2021-05-26

- Restructured docs
- Fix WSL2 installation problem, issue #440
- Fix for interactive mode under Windows, issue #430

10.2.3 Version 4.0.1, 2021-03-17

- Fix `tox -e build` issue when running on Conda, PR #417
- Ensure `snake_case` for keys in `setup.cfg`, issue #418
- Update dependencies on `configupdater` and `pyscaffoldext-django`
- Remove broken checks for old `setuptools`, issue #428

10.2.4 Version 4.0, 2021-03-04

- Cookiecutter, Django and Travis extensions extracted to their own repositories, issue #175 and issue #355
- Support for Python 3.4 and 3.5 dropped, issue #226
- Dropped deprecated `requirements.txt` file, issue #182
- Added support for global configuration (avoid retyping common `putup`'s options), issue #236
- PyScaffold is no longer a build-time dependency, it just generates the project structure
- Removed `contrib` subpackage, vendorized packages are now runtime dependencies, PR #290
- `setuptools_scm` is included by default in `setup.cfg`, `setup.py` and `pyproject.toml`
- API changed to use `pyscaffold.operations` instead of integer flags, PR #271
- Allow `string.Template` and `callable` as file contents in project structure, PR #295

- Extract file system functions from `utils.py` into `file_system.py`
- Extract identification/naming functions from `utils.py` into `identification.py`
- Extract action related functions from `api/__init__.py` to `actions.py`
- `helpers.{modify,ensure,reject}` moved to `structure.py`
- `helpers.{register,unregister}` moved to `actions.py`
- New extension for automatically creating virtual environments (`--venv`)
- Added instructions to use `pip-tools` to docs
- `pre-commit` extension now attempts to install hooks automatically
- A nice message is now displayed when PyScaffold finishes running (`actions.report_done`)
- Removed mutually exclusive argparse groups for extensions, PR #315
- Progressive type annotations adopted in the code base together with mypy linting
- Simplified isort config
- `pyproject.toml` and isolated builds adopted by default, issue #256
- Added comment to `setup.cfg` template instructing about extra links, issue #268
- Generate `tox.ini` by default, issue #296
- Replace `pkg_resources` with `importlib.{metadata,resources}` and `packaging`, issue #309
- Adopt PEP 420 for namespaces, issue #218
- Adopt SPDX identifiers for the license field in `setup.cfg`, issue #319
- Removed deprecated `log.configure_logger`
- Add links to issues and pull requests to changelog, PR #363
- Add an experimental `--interactive` *mode* (inspired by `git rebase -i`), issue #191 (additional discussion: PR #333, PR #325, PR #362)
- Reorganise the **FAQ** (including version questions previously in **Features**)
- Updated `setuptools` and `setuptools_scm` dependencies to minimal versions 46.1 and 5, respectively
- Adopted `no-guess-dev` version scheme from `setuptools_scm` (semantically all stays the same, but non-tag commits are now versioned `LAST_TAG.post1.devN` instead of `LAST_TAG.post0.devN`)
- Fix problem of not showing detailed log with `--verbose` if error happens when loading extensions issue #378

10.2.5 Version 3.2.3, 2019-10-12

- Updated `configupdater` to version 1.0.1
- Changed Travis to Cirrus CI
- Fix some problems with Windows

10.2.6 Version 3.2.2, 2019-09-12

- Write files as UTF-8, fixes `codec can't encode characters` error

10.2.7 Version 3.2.1, 2019-07-11

- Updated pre-commit configuration and set max-line-length to 88 (Black's default)
- Change build folder of Sphinx's Makefile
- Fix creation of empty files which were just ignored before

10.2.8 Version 3.2, 2019-06-30

- *deprecated* use of lists with `helpers.{modify,ensure,reject}`, issue #211
- Add support for `os.PathLike` objects in `helpers.{modify,ensure,reject}`, issue #211
- Remove `release` alias in `setup.cfg`, use `twine` instead
- Set `project-urls` and `long-description-content-type` in `setup.cfg`, issue #216
- Added additional command line argument `very-verbose`
- Assure clean workspace when updating existing project, issue #190
- Show stacktrace on errors if `--very-verbose` is used
- Updated `configupdater` to version 1.0
- Use `pkg_resources.resource_string` instead of `pkgutil.get_data` for templates
- Update `setuptools_scm` to version 3.3.3
- Updated pytest-runner to version 5.1
- Some fixes regarding the order of executing extensions
- Consider `GIT_AUTHOR_NAME` and `GIT_AUTHOR_EMAIL` environment variables
- Updated `tox.ini`
- Switch to using tox in `.travis.yml` template
- Reworked all official extensions `--pyproject`, `--custom-extension` and `--markdown`

10.2.9 Version 3.1, 2018-09-05

- Officially dropped Python 2 support, issue #177
- Moved `entry_points` and `setup_requires` to `setup.cfg`, issue #176
- Updated `travis.yml` template, issue #181
- Set `install_requires` to setuptools>=31
- Better isolation of unit tests, issue #119
- Updated tox template, issues issue #160 & issue #161
- Use `pkg_resources.parse_version` instead of old `LooseVersion` for parsing
- Use `ConfigUpdater` instead of `ConfigParser`

- Lots of internal cleanups and improvements
- Updated pytest-runner to version 4.2
- Updated setuptools_scm to version 3.1
- Fix Django extension problem with src-layout, issue #196
- *experimental* extension for MarkDown usage in README, issue #163
- *experimental* support for Pipenv, issue #140
- *deprecated* built-in Cookiecutter and Django extensions (to be moved to separated packages), issue #175

10.2.10 Version 2.5.11, 2018-04-14

- Updated pbr to version 4.0.2
- Fixes Sphinx version 1.6 regression, issue #152

10.2.11 Version 3.0.3, 2018-04-14

- Set install_requires to setuptools>=30.3.0

10.2.12 Version 3.0.2, 2018-03-21

- Updated setuptools_scm to version 1.17.0
- Fix wrong docstring in skeleton.py about entry_points, issue #147
- Fix error with setuptools version 39.0 and above, issue #148
- Fixes in documentation, thanks Vicky

10.2.13 Version 2.5.10, 2018-03-21

- Updated setuptools_scm to version 1.17.0

10.2.14 Version 2.5.9, 2018-03-20

- Updated setuptools_scm to version 1.16.1
- Fix error with setuptools version 39.0 and above, issue #148

10.2.15 Version 3.0.1, 2018-02-13

- Fix confusing error message when `python setup.py docs` and Sphinx is not installed, issue #142
- Fix 'unknown' version in case project name differs from the package name, issue #141
- Fix missing `file:` attribute in long-description of setup.cfg
- Fix `sphinx-apidoc` invocation problem with Sphinx 1.7

10.2.16 Version 3.0, 2018-01-07

- Improved Python API thanks to an extension system
- Dropped pbr in favor of setuptools >= 30.3.0
- Updated setuptools_scm to v1.15.6
- Changed `my_project/my_package` to recommended `my_project/src/my_package` structure
- Renamed `CHANGES.rst` to more standard `CHANGELOG.rst`
- Added platforms parameter in `setup.cfg`
- Call Sphinx api-doc from `conf.py`, issue #98
- Included six 1.11.0 as contrib sub-package
- Added `CONTRIBUTING.rst`
- Removed `test-requirements.txt` from template
- Added support for GitLab
- License change from New BSD to MIT
- FIX: Support of git submodules, issue #98
- Support of Cython extensions, issue #48
- Removed redundant `--with-` from most command line flags
- Prefix `n` was removed from the local_version string of dirty versions
- Added a `--pretend` flag for easier development of extensions
- Added a `--verbose` flag for more output what PyScaffold is doing
- Use pytest-runner 4.4 as contrib package
- Added a `--no-skeleton` flag to omit the creation of `skeleton.py`
- Save parameters used to create project scaffold in `setup.cfg` for later updating

A special thanks goes to Anderson Bravalheri for his awesome support and inovex for sponsoring this release.

10.2.17 Version 2.5.8, 2017-09-10

- Use `sphinx.ext.imgmath` instead of `sphinx.ext.mathjax`
- Added `--with-gitlab-ci` flag for GitLab CI support
- Fix Travis install template dirties git repo, issue #107
- Updated setuptools_scm to version 1.15.6
- Updated pbr to version 3.1.1

10.2.18 Version 2.5.7, 2016-10-11

- Added encoding to __init__.py
- Few doc corrections in setup.cfg
- [tool:pytest] instead of [pytest] in setup.cfg
- Updated skeleton
- Switch to Google Sphinx style
- Updated setuptools_scm to version 1.13.1
- Updated pbr to version 1.10.0

10.2.19 Version 2.5.6, 2016-05-01

- Prefix error message with ERROR:
- Suffix of untagged commits changed from {version}-{hash} to {version}-n{hash}
- Check if package identifier is valid
- Added log level command line flags to the skeleton
- Updated pbr to version 1.9.1
- Updated setuptools_scm to version 1.11.0

10.2.20 Version 2.5.5, 2016-02-26

- Updated pbr to master at 2016-01-20
- Fix sdist installation bug when no git is installed, issue #90

10.2.21 Version 2.5.4, 2016-02-10

- Fix problem with `fibonacci` terminal example
- Update setuptools_scm to v1.10.1

10.2.22 Version 2.5.3, 2016-01-16

- Fix classifier metadata (`classifiers` to `classifier` in `setup.cfg`)

10.2.23 Version 2.5.2, 2016-01-02

- Fix `is_git_installed`

10.2.24 Version 2.5.1, 2016-01-01

- Fix: Do some sanity checks first before gathering default options
- Updated setuptools_scm to version 1.10.0

10.2.25 Version 2.5, 2015-12-09

- Usage of `test-requirements.txt` instead of `tests_require` in `setup.py`, issue #71
- Removed `--with-numpydoc` flag since this is now included by default with `sphinx.ext.napoleon` in Sphinx 1.3 and above
- Added small template for unittest
- Fix for the example skeleton file when using namespace packages
- Fix typo in devpi:upload section, issue #82
- Include `pbr` and `setuptools_scm` in PyScaffold to avoid dependency problems, issue #71 and issue #72
- Cool logo was designed by Eva Schmücker, issue #66

10.2.26 Version 2.4.4, 2015-10-29

- Fix problem with bad upload of version 2.4.3 to PyPI, issue #80

10.2.27 Version 2.4.3, 2015-10-27

- Fix problem with version numbering if setup.py is not in the root directory, issue #76

10.2.28 Version 2.4.2, 2015-09-16

- Fix version conflicts due to too tight pinning, issue #69

10.2.29 Version 2.4.1, 2015-09-09

- Fix installation with additional requirements `pyscaffold[ALL]`
- Updated pbr version to 1.7

10.2.30 Version 2.4, 2015-09-02

- Allow different py.test options when invoking with `py.test` or `python setup.py test`
- Check if Sphinx is needed and add it to *setup_requires*
- Updated pre-commit plugins
- Replaced pytest-runner by an improved version
- Let pbr do `sphinx-apidoc`, removed from `conf.py`, issue #65

Note: Due to the switch to a modified pytest-runner version it is necessary to update `setup.cfg`. Please check the *example*.

10.2.31 Version 2.3, 2015-08-26

- Format of setup.cfg changed due to usage of pbr, issue #59
- Much cleaner setup.py due to usage of pbr, issue #59
- PyScaffold can be easily called from another script, issue #58
- Internally dictionaries instead of namespace objects are used for options, issue #57
- Added a section for devpi in setup.cfg, issue #62

Note: Due to the switch to pbr, it is necessary to update `setup.cfg` according to the new syntax.

10.2.32 Version 2.2.1, 2015-06-18

- FIX: Removed putup console script in setup.cfg template

10.2.33 Version 2.2, 2015-06-01

- Allow recursive inclusion of data files in setup.cfg, issue #49
- Replaced hand-written PyTest runner by pytest-runner, issue #47
- Improved default README.rst, issue #51
- Use tests/conftest.py instead of tests/__init__.py, issue #52
- Use setuptools_scm for versioning, issue #43
- Require setuptools>=9.0, issue #56
- Do not create skeleton.py during an update, issue #55

Note: Due to the switch to *setuptools_scm* the following changes apply:

- use `python setup.py --version` instead of `python setup.py version`
- `git archive` can no longer be used for packaging (and was never meant for it anyway)
- initial tag `v0.0` is no longer necessary and thus not created in new projects
- tags do no longer need to start with *v*

10.2.34 Version 2.1, 2015-04-16

- Use alabaster as default Sphinx theme
- Parameter data_files is now a section in setup.cfg
- Allow definition of extras_require in setup.cfg
- Added a CHANGES.rst file for logging changes
- Added support for cookiecutter
- FIX: Handle an empty Git repository if necessary

10.2.35 Version 2.0.4, 2015-03-17

- Typo and wrong Sphinx usage in the RTD documentation

10.2.36 Version 2.0.3, 2015-03-17

- FIX: Removed misleading *include_package_data* option in setup.cfg
- Allow selection of a proprietary license
- Updated some documentations
- Added -U as short parameter for –update

10.2.37 Version 2.0.2, 2015-03-04

- FIX: Version retrieval with setup.py install
- argparse example for version retrieval in skeleton.py
- FIX: import my_package should be quiet (verbose=False)

10.2.38 Version 2.0.1, 2015-02-27

- FIX: Installation bug under Windows 7

10.2.39 Version 2.0, 2015-02-25

- Split configuration and logic into setup.cfg and setup.py
- Removed .pre from version string (newer PEP 440)
- FIX: Sphinx now works if package name does not equal project name
- Allow namespace packages with –with-namespace
- Added a skeleton.py as a console_script template
- Set *v0.0* as initial tag to support PEP440 version inference
- Integration of the Versioneer functionality into setup.py
- Usage of *data_files* configuration instead of *MANIFEST.in*
- Allow configuration of *package_data* in *setup.cfg*

- Link from Sphinx docs to AUTHORS.rst

10.2.40 Version 1.4, 2014-12-16

- Added numpydoc flag –with-numpydoc
- Fix: Add django to requirements if –with-django
- Fix: Don't overwrite index.rst during update

10.2.41 Version 1.3.2, 2014-12-02

- Fix: path of Travis install script

10.2.42 Version 1.3.1, 2014-11-24

- Fix: –with-tox tuple bug, PR #28

10.2.43 Version 1.3, 2014-11-17

- Support for Tox (https://tox.readthedocs.io/en/stable/)
- flake8: exclude some files
- Usage of UTF8 as file encoding
- Fix: create non-existent files during update
- Fix: unit tests on MacOS
- Fix: unit tests on Windows
- Fix: Correct version when doing setup.py install

10.2.44 Version 1.2, 2014-10-13

- Support pre-commit hooks (https://pre-commit.com/)

10.2.45 Version 1.1, 2014-09-29

- Changed COPYING to LICENSE
- Support for all licenses from https://choosealicense.com/
- Fix: Allow update of license again
- Update to Versioneer 0.12

10.2.46 Version 1.0, 2014-09-05

- Fix when overwritten project has a git repository
- Documentation updates
- License section in Sphinx
- Django project support with –with-django flag
- Travis project support with –with-travis flag
- Replaced sh with own implementation
- Fix: new *git describe* version to PEP440 conversion
- conf.py improvements
- Added source code documentation
- Fix: Some Python 2/3 compatibility issues
- Support for Windows
- Dropped Python 2.6 support
- Some classifier updates

10.2.47 Version 0.9, 2014-07-27

- Documentation updates due to RTD
- Added a –force flag
- Some cleanups in setup.py

10.2.48 Version 0.8, 2014-07-25

- Update to Versioneer 0.10
- Moved sphinx-apidoc from setup.py to conf.py
- Better support for *make html*

10.2.49 Version 0.7, 2014-06-05

- Added Python 3.4 tests and support
- Flag –update updates only some files now
- Usage of setup_requires instead of six code

10.2.50 Version 0.6.1, 2014-05-15

- Fix: Removed six dependency in setup.py

10.2.51 Version 0.6, 2014-05-14

- Better usage of six
- Return non-zero exit status when doctests fail
- Updated README
- Fixes in Sphinx Makefile

10.2.52 Version 0.5, 2014-05-02

- Simplified some Travis tests
- Nicer output in case of errors
- Updated PyScaffold's own setup.py
- Added –junit_xml and –coverage_xml/html option
- Updated .gitignore file

10.2.53 Version 0.4.1, 2014-04-27

- Problem fixed with pytest-cov installation

10.2.54 Version 0.4, 2014-04-23

- PEP8 and PyFlakes fixes
- Added –version flag
- Small fixes and cleanups

10.2.55 Version 0.3, 2014-04-18

- PEP8 fixes
- More documentation
- Added update feature
- Fixes in setup.py

10.2.56 Version 0.2, 2014-04-15

- Checks when creating the project
- Fixes in COPYING
- Usage of sh instead of GitPython
- PEP8 fixes
- Python 3 compatibility
- Coverage with Coverall.io
- Some more unittests

10.2.57 Version 0.1.2, 2014-04-10

- Bugfix in Manifest.in
- Python 2.6 problems fixed

10.2.58 Version 0.1.1, 2014-04-10

- Unittesting with Travis
- Switch to string.Template
- Minor bugfixes

10.2.59 Version 0.1, 2014-04-03

- First release

CHAPTER

ELEVEN

LICENSE

```
PyScaffold is licensed under the MIT license; see below for details.

The template files under `pyscaffold.templates`, used for the the generated
projects, are licensed under BSD 0-Clause license; see below for details.

-------------------------------------------------------------------------

The MIT License (MIT)

Copyright (c) 2018-present, PyScaffold contributors
Copyright (c) 2014-2018 Blue Yonder GmbH

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.

-------------------------------------------------------------------------

BSD 0-Clause License

Copyright (c) 2018-present, PyScaffold contributors
Copyright (c) 2014-2018 Blue Yonder GmbH

Permission to use, copy, modify, and/or distribute this software for
any purpose with or without fee is hereby granted.
```

(continues on next page)

(continued from previous page)

```
THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT,
INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM
LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

CHAPTER

TWELVE

PYSCAFFOLD

12.1 pyscaffold package

12.1.1 Subpackages

pyscaffold.extensions package

Submodules

pyscaffold.extensions.cirrus module

Extension that generates configuration for Cirrus CI.

class pyscaffold.extensions.cirrus.**Cirrus**(*name=None*)

Bases: *pyscaffold.extensions.Extension*

Add configuration file for Cirrus CI (includes *–pre-commit*)

activate(*actions*)

Activate extension, see `activate`.

augment_cli(*parser*)

Augments the command-line interface parser. See `augment_cli`.

pyscaffold.extensions.cirrus.**add_files**(*struct*, *opts*)

Add .cirrus.yml to the file structure

Parameters

- **struct** – project representation as (possibly) nested `dict`.
- **opts** – given options, see `create_project` for an extensive list.

Returns updated project representation and options

Return type struct, opts

pyscaffold.extensions.cirrus.**cirrus_descriptor**(*_opts*)

Returns the rendered template

pyscaffold.extensions.config module

CLI options for using/saving preferences as PyScaffold config files.

class pyscaffold.extensions.config.**Config**(*name=None*)
Bases: *pyscaffold.extensions.Extension*

Add a few useful options for creating/using PyScaffold config files.

activate(*actions*)
Activates the extension by registering its functionality

Parameters **actions** (*List[Action]*) – list of action to perform

Returns updated list of actions

Return type List[Action]

augment_cli(*parser*)
Augments the command-line interface parser.

A command line argument `--FLAG` where FLAG=``self.name`` is added which appends `self.activate` to the list of extensions. As help text the docstring of the extension class is used. In most cases this method does not need to be overwritten.

Parameters **parser** – current parser object

persist = False
When `True` PyScaffold will store the extension in the PyScaffold's section of `setup.cfg`. Useful for updates. Set to `False` if the extension should not be re-invoked on updates.

pyscaffold.extensions.config.**save**(*struct*, *opts*)
Save the given opts as preferences in a PyScaffold's config file.

pyscaffold.extensions.gitlab_ci module

Extension that generates configuration and script files for GitLab CI.

class pyscaffold.extensions.gitlab_ci.**GitLab**(*name=None*)
Bases: *pyscaffold.extensions.Extension*

Generate GitLab CI configuration files

activate(*actions*)
Activate extension, see `activate`.

pyscaffold.extensions.gitlab_ci.**add_files**(*struct*, *opts*)
Add .gitlab-ci.yml file to structure

Parameters

- **struct** – project representation as (possibly) nested `dict`.
- **opts** – given options, see `create_project` for an extensive list.

Returns updated project representation and options

Return type struct, opts

pyscaffold.extensions.interactive module

Similarly to `git rebase -i` this extension allows users to interactively choose which options apply to `putup`, by editing a file filled with examples.

See *CONFIG* for more details on how to tweak the text generated in the interactive mode.

New in version 4.0: *"interactive mode"* introduced as an **experimental** extension.

Warning: **NOTE FOR CONTRIBUTORS**: Due to the way `argparse` is written, it is not very easy to obtain information about which options and arguments a given parser is currently configured with. There are no public methods that allow inspection/reflection, and therefore in order to do so, one has to rely on a few non-public methods (according to Python's convention, the ones starting with a _ symbol). Since `argparse` implementation is very stable and mature, these non-public method are very unlikely to change and, therefore, it is relatively safe to use these methods, however developers and maintainers have to be aware and pay attention to eventual breaking changes. The non-public functions are encapsulated in the functions *get_actions* and *format_args* in this file, in order to centralise the usage of non-public API.

pyscaffold.extensions.interactive.**CONFIG = {'comment': ['--verbose', '--very-verbose'], 'ignore': ['--help', '--version']}**

Configuration for the options that are not associated with an extension class. This dict value consist of a set of metadata organised as follows:

- Each value must be a list of "long" `argparse` option strings (e.g. *"–help"* instead of *"-h"*).
- Each key implies on a different interpretation for the metadata:
  - `"ignore"`: Options that should be simply ignored when creating examples
  - `"comment"`: Options that should be commented when creating examples, even if they appear in the original `sys.argv`.

Extension classes (or instances) can also provide configurations by defining a `interactive` attribute assigned to a similar `dict` object.

class pyscaffold.extensions.interactive.**Interactive**(*name=None*)

Bases: *pyscaffold.extensions.Extension*

Interactively choose and configure PyScaffold's parameters

augment_cli(*parser*)

See *augment_cli*.

command(*opts*)

This method replace the regular call to *cli.run_scaffold* with an intermediate file to confirm the user's choices in terms of arguments/options.

parser

pyscaffold.extensions.interactive.**all_examples**(*parser*, *actions*, *opts*)

Generate a example of usage of the CLI options corresponding to the given `actions` including the help text.

This function will skip options that are marked in the `"ignore"` *configuration*.

See *example_with_help*.

pyscaffold.extensions.interactive.**alternative_flags**(*action*)

Get the alternative flags (i.e. not the long one) of a `argparse.Action`

pyscaffold.extensions.interactive.**comment**(*text*, *comment_mark='#'*, *indent_level=0*)

Comment each line of the given text (optionally indenting it)

pyscaffold.extensions.interactive.**example**(*parser*, *action*, *opts*)
Generate a CLI example of option usage for the given argparse.Action. The opts argument corresponds to options already processed by PyScaffold, and interferes on the generated text (e.g., when the corresponding option is already processed, the example will be adjusted accordingly; when the corresponding option is not present, the example might be commented out; ...).

This function will comment options that are marked in the "comment" *configuration*.

pyscaffold.extensions.interactive.**example_no_value**(*parser*, *action*, *opts*)
Generate a CLI example of option usage for a argparse.Action that do not expect arguments (nargs = 0).

See *example*.

pyscaffold.extensions.interactive.**example_with_help**(*parser*, *action*, *opts*)
Generate a CLI example of option usage for the given argparse.Action that includes a comment text block explaining its meaning (basically the same text displayed when using --help).

See *example*.

pyscaffold.extensions.interactive.**example_with_value**(*parser*, *action*, *opts*)
Generate a CLI example of option usage for a argparse.Action that expects one or more arguments (nargs is "?", "*", "+" or "N" > 0).

See *example*.

pyscaffold.extensions.interactive.**expand_computed_opts**(*opts*)
Pre-process the given PyScaffold options and add default/computed values (including the ones derived from setup.cfg in case of --update or PyScaffold's own configuration file in the user's home directory.

pyscaffold.extensions.interactive.**format_args**(*parser*, *action*)
Produce an example to be used together with the flag of the given action.

> **Warning:** This function uses non-public API from Python's stdlib argparse.

pyscaffold.extensions.interactive.**get_actions**(*parser*)
List actions related to options that were configured to the given ArgumentParser.

> **Warning:** This function uses non-public API from Python's stdlib argparse.

pyscaffold.extensions.interactive.**get_config**(*kind*)
Get configurations that will be used for generating examples (from both *CONFIG* and the interactive attribute of each extension).

The kind argument can assume the same values as the *CONFIG* keys.

This function is cached to improve performance. Call get_config.__wrapped__ to bypass the cache (or get_config.cache_clear, see functools.lru_cache).

pyscaffold.extensions.interactive.**has_active_extension**(*action*, *opts*)
Returns True if the given argparse.Action corresponds to an extension that was previously activated via CLI.

pyscaffold.extensions.interactive.**join_block**(**parts*, *sep='\n'*)
Join blocks of text using sep, but ignoring the empty ones

pyscaffold.extensions.interactive.**long_option**(*action*)
Get the long option corresponding to the given argparse.Action

pyscaffold.extensions.interactive.**split_args**(*text*)
Split the text from the interactive example into arguments that can be passed directly to `argparse`, as if they were invoked directly from the CLI (this includes removing comments).

pyscaffold.extensions.interactive.**wrap**(*text*, *width=70*, ***kwargs*)
Wrap text to fit lines with a maximum number of caracters

pyscaffold.extensions.namespace module

Extension that adjust project file tree to include a namespace package.

This extension adds a **namespace** option to *create_project* and provides correct values for the options **root_pkg** and **namespace_pkg** to the following functions in the action list.

class pyscaffold.extensions.namespace.**Namespace**(*name=None*)
Bases: *pyscaffold.extensions.Extension*

Add a namespace (container package) to the generated package.

activate(*actions*)
Register an action responsible for adding namespace to the package.

Parameters **actions** – list of actions to perform

Returns updated list of actions

Return type list

augment_cli(*parser*)
Add an option to parser that enables the namespace extension.

Parameters **parser** (*argparse.ArgumentParser*) – CLI parser object

pyscaffold.extensions.namespace.**add_namespace**(*struct*, *opts*)
Prepend the namespace to a given file structure

Parameters

- **struct** – directory structure as dictionary of dictionaries
- **opts** – options of the project

Returns Directory structure as dictionary of dictionaries and input options

pyscaffold.extensions.namespace.**enforce_namespace_options**(*struct*, *opts*)
Make sure options reflect the namespace usage.

pyscaffold.extensions.namespace.**move_old_package**(*struct*, *opts*)
Move old package that may be eventually created without namespace

Parameters

- **struct** (*dict*) – directory structure as dictionary of dictionaries
- **opts** (*dict*) – options of the project

Returns directory structure as dictionary of dictionaries and input options

Return type tuple(dict, dict)

pyscaffold.extensions.namespace.**prepare_namespace**(*namespace_str*)
Check the validity of namespace_str and split it up into a list

Parameters **namespace_str** – namespace, e.g. “com.blue_yonder”

Returns list of namespaces, e.g. ["com", "com.blue_yonder"]

Raises *InvalidIdentifier* – raised if namespace is not valid

pyscaffold.extensions.no_pyproject module

Extension that omits the creation of file *pyproject.toml*.

Since the isolated builds with PEP517/PEP518 are not completely backward compatible, this extension provides an escape hatch for people that want to maintain the legacy behaviour.

> **Warning:** This extension is **transitional** and will be removed in future versions of PyScaffold. Once support for isolated builds stabilises, the Python community will likely move towards using them more exclusively.

class pyscaffold.extensions.no_pyproject.**NoPyProject**(*name=None*)

Bases: *pyscaffold.extensions.Extension*

Do not include a pyproject.toml file in the project root, and thus avoid isolated builds as defined in PEP517/518 [not recommended]

activate(*actions*)

Activates the extension by registering its functionality

Parameters **actions** (*List[Action]*) – list of action to perform

Returns updated list of actions

Return type List[Action]

name = 'no_pyproject'

pyscaffold.extensions.no_pyproject.**ensure_option**(*struct*, *opts*)

Make option available in non-CLI calls (used by other parts of PyScaffold)

pyscaffold.extensions.no_pyproject.**remove_files**(*struct*, *opts*)

pyscaffold.extensions.no_skeleton module

Extension that omits the creation of file *skeleton.py*

class pyscaffold.extensions.no_skeleton.**NoSkeleton**(*name=None*)

Bases: *pyscaffold.extensions.Extension*

Omit creation of skeleton.py and test_skeleton.py

activate(*actions*)

Activate extension, see `activate`.

pyscaffold.extensions.no_skeleton.**remove_files**(*struct*, *opts*)

Remove all skeleton files from structure

Parameters

- **struct** – project representation as (possibly) nested `dict`.
- **opts** – given options, see `create_project` for an extensive list.

Returns Updated project representation and options

pyscaffold.extensions.no_tox module

Extension that removes configuration files for the Tox test automation tool.

class pyscaffold.extensions.no_tox.**NoTox**(*name=None*)
Bases: *pyscaffold.extensions.Extension*

Prevent a tox configuration file from being created

activate(*actions*)
Activate extension, see `activate`.

pyscaffold.extensions.no_tox.**remove_files**(*struct*, *opts*)
Remove .tox.ini file to structure

pyscaffold.extensions.pre_commit module

Extension that generates configuration files for Yelp pre-commit.

class pyscaffold.extensions.pre_commit.**PreCommit**(*name=None*)
Bases: *pyscaffold.extensions.Extension*

Generate pre-commit configuration file

activate(*actions*)
Activate extension

Parameters **actions** (*list*) – list of actions to perform

Returns updated list of actions

Return type list

pyscaffold.extensions.pre_commit.**add_files**(*struct*, *opts*)
Add .pre-commit-config.yaml file to structure

Since the default template uses isort, this function also provides an initial version of .isort.cfg that can be extended by the user (it contains some useful skips, e.g. tox and venv)

pyscaffold.extensions.pre_commit.**add_instructions**(*opts*, *content*, *file_op*)
Add pre-commit instructions to README

pyscaffold.extensions.pre_commit.**find_executable**(*struct*, *opts*)
Find the pre-commit executable that should run later in the next action… Or take advantage of the venv to install it…

pyscaffold.extensions.pre_commit.**install**(*struct*, *opts*)
Attempts to install pre-commit in the project

pyscaffold.extensions.venv module

Create a virtual environment for the project

pyscaffold.extensions.venv.**DEFAULT = '.venv'**
Default directory name for collocated virtual environment that will be created

exception pyscaffold.extensions.venv.**NotInstalled**(*msg=None*)
Bases: `ImportError`

Neither virtualenv or venv are installed in the computer. Please check the following alternatives:

- `virtualenv` can be installed via pip
- `venv` is supposed to installed by default with Python3, however some OS or distributions (such as Ubuntu) break the standard library in a series of packages that need to be manually installed via OS package manager. You can try to search for a `python-venv`, `python3-venv` or similar in the official repositories.

class pyscaffold.extensions.venv.**Venv**(*name=None*)
Bases: *pyscaffold.extensions.Extension*

Create a virtual environment for the project (using virtualenv or stdlib's venv). Default location: "{DEFAULT}".

If `virtualenv` is available, it will be used, since it has some advantages over stdlib's `venv` (such as being faster, see https://virtualenv.pypa.io/en/stable/).

Notice that even if part of Python's stdlib, `venv` is not guaranteed to be installed, some OS/distributions (such as Ubuntu) require an explicit installation. If you have problems, try installing virtualenv with pip and run the command again.

activate(*actions*)
Activate extension, see `activate`.

augment_cli(*parser*)
Augments the command-line interface parser. See `augment_cli`.

persist = False
When `True` PyScaffold will store the extension in the PyScaffold's section of `setup.cfg`. Useful for updates. Set to `False` if the extension should not be re-invoked on updates.

pyscaffold.extensions.venv.**create_with_stdlib**(*path*, *pretend=False*)

pyscaffold.extensions.venv.**create_with_virtualenv**(*path*, *pretend=False*)

pyscaffold.extensions.venv.**get_path**(*opts*, *default='.venv'*)
Get the path to the venv that will be created.

pyscaffold.extensions.venv.**install_packages**(*struct*, *opts*)
Install the specified packages inside the created venv.

pyscaffold.extensions.venv.**instruct_user**(*struct*, *opts*)
Simply display a message reminding the user to activate the venv.

pyscaffold.extensions.venv.**run**(*struct*, *opts*)
Action that will create a virtualenv for the project

Module contents

Built-in extensions for PyScaffold.

class pyscaffold.extensions.**Extension**(*name=None*)
Bases: `object`

Base class for PyScaffold's extensions

Parameters **name** (*str*) – How the extension should be named. Default: name of class By default, this value is used to create the activation flag in PyScaffold cli.

See our docs on how to create extensions in: https://pyscaffold.org/en/latest/extensions.html

Also check *actions*, *Structure* and *ScaffoldOpts* for more details.

Note: Please name your class using a CamelCase version of the name you use in the setuptools entrypoint (alternatively you will need to overwrite the `name` property to match the entrypoint name).

`activate`(*actions*)
Activates the extension by registering its functionality

Parameters **actions** (*List[Action]*) – list of action to perform

Returns updated list of actions

Return type List[Action]

`augment_cli`(*parser*)
Augments the command-line interface parser.

A command line argument `--FLAG` where FLAG=``self.name`` is added which appends `self.activate` to the list of extensions. As help text the docstring of the extension class is used. In most cases this method does not need to be overwritten.

Parameters **parser** – current parser object

`property flag`

`property help_text`

`property name`

`persist = True`
When `True` PyScaffold will store the extension in the PyScaffold's section of `setup.cfg`. Useful for updates. Set to `False` if the extension should not be re-invoked on updates.

`static register`(*actions*, *action*, *before=None*, *after=None*)
Shortcut for *pyscaffold.actions.register*

`static unregister`(*actions*, *reference*)
Shortcut for *pyscaffold.actions.unregister*

pyscaffold.extensions.**NO_LONGER_NEEDED = {'pyproject', 'tox'}**
Extensions that are no longer needed and are now part of PyScaffold itself

pyscaffold.extensions.**include**(**extensions*)
Create a custom `argparse.Action` that saves multiple extensions for activation.

Parameters ***extensions** – extension objects to be saved

pyscaffold.extensions.**iterate_entry_points**(*group='pyscaffold.cli'*)
Produces a generator yielding an EntryPoint object for each extension registered via setuptools entry point mechanism.

This method can be used in conjunction with *load_from_entry_point* to filter the extensions before actually loading them.

pyscaffold.extensions.**list_from_entry_points**(*group='pyscaffold.cli'*, *filtering=<function <lambda>>*)
Produces a list of extension objects for each extension registered via setuptools entry point mechanism.

Parameters

- **group** – name of the setuptools' entry_point group where extensions is being registered
- **filtering** – function returning a boolean deciding if the entry point should be loaded and included (or not) in the final list. A `True` return means the extension should be included.

pyscaffold.extensions.**load_from_entry_point**(*entry_point*)
Carefully load the extension, raising a meaningful message in case of errors

pyscaffold.extensions.**store_with**(**extensions*)
Create a custom `argparse.Action` that stores the value of the given option in addition to saving the extension for activation.

Parameters ***extensions** – extension objects to be saved for activation

pyscaffold.templates package

Module contents

Templates for all files of a project's scaffold

All template files (**.template*) within this subpackage are licensed under the BSD 0-Clause license.

pyscaffold.templates.**add_pyscaffold**(*config*, *opts*)
Add PyScaffold section to a `setup.cfg`-like file + PyScaffold's version + extensions and their associated options.

pyscaffold.templates.**get_template**(*name*, *relative_to='pyscaffold.templates'*)
Retrieve the template by name

Parameters

- **name** – name of template (the `.template` extension will be automatically added to this name)
- **relative_to** – module/package object or name to which the resource file is relative (in the standard module format, e.g. `foo.bar.baz`). Notice that `relative_to` should not represent directly a shared namespace package, since this kind of package is spread in different folders in the file system.

  Default value: `pyscaffold.templates` (**please assign accordingly when using in custom extensions**).

Examples

Consider the following package organization:

```
.
├── src
│   └── my_package
│       ├── __init__.py
│       ├── templates
│       │   ├── __init__.py
│       │   ├── file1.template
│       │   └── file2.template
│       ...
└── tests
```

Inside the file `src/my_package/__init__.py`, one can easily obtain the contents of `file1.template` by doing:

```
from pyscaffold.templates import get_template
from . import templates as my_templates
```

(continues on next page)

(continued from previous page)

```
tpl1 = get_template("file1", relative_to=my_templates)
# OR
# tpl1 = get_template('file1', relative_to=my_templates.__name__)
```

Please notice you can also use *relative_to=__name__* or a combination of *from .. import __name__ as parent* and *relative_to=parent* to deal with relative imports.

Returns template

Return type string.Template

Changed in version 3.3: New parameter **relative_to**.

pyscaffold.templates.**init**(*opts*)
Template of __init__.py

Parameters **opts** – mapping parameters as dictionary

Returns file content as string

Return type str

pyscaffold.templates.**license**(*opts*)
Template of LICENSE.txt

Parameters **opts** – mapping parameters as dictionary

Returns file content as string

Return type str

pyscaffold.templates.**licenses = {'0BSD': 'license_bsd0', 'AGPL-3.0-only': 'license_affero_3.0', 'AGPL-3.0-or-later': 'license_affero_3.0', 'Apache-2.0': 'license_apache', 'Artistic-2.0': 'license_artistic_2.0', 'BSD-2-Clause': 'license_simplified_bsd', 'BSD-3-Clause': 'license_new_bsd', 'CC0-1.0': 'license_cc0_1.0', 'EPL-1.0': 'license_eclipse_1.0', 'GPL-2.0-only': 'license_gpl_2.0', 'GPL-2.0-or-later': 'license_gpl_2.0', 'GPL-3.0-only': 'license_gpl_3.0', 'GPL-3.0-or-later': 'license_gpl_3.0', 'ISC': 'license_isc', 'LGPL-2.0-only': 'license_lgpl_2.1', 'LGPL-2.0-or-later': 'license_lgpl_2.1', 'LGPL-3.0-only': 'license_lgpl_3.0', 'LGPL-3.0-or-later': 'license_lgpl_3.0', 'MIT': 'license_mit', 'MPL-2.0': 'license_mozilla', 'Proprietary': 'license_none', 'Unlicense': 'license_public_domain'}**
All available licences (identifiers based on SPDX https://spdx.org/licenses/)

pyscaffold.templates.**parse_extensions**(*extensions*)
Given a string value for the field pyscaffold.extensions in a setup.cfg-like file, return a set of extension names.

pyscaffold.templates.**pyproject_toml**(*opts*)

pyscaffold.templates.**setup_cfg**(*opts*)
Template of setup.cfg

Parameters **opts** – mapping parameters as dictionary

Returns file content as string

Return type str

12.1.2 Submodules

12.1.3 pyscaffold.actions module

Default PyScaffold's actions and functions to manipulate them.

When generating a project, PyScaffold uses a pipeline of functions (each function will receive as arguments the values returned by the previous function). These functions have an specific purpose and are called **actions**. Please follow the *Action* signature when developing your own action.

Note: Some actions are more complex and are placed in dedicated modules together with other auxiliary functions, see *pyscaffold.structure*, *pyscaffold.update*.

---

pyscaffold.actions.**Action**
Signature of a PyScaffold action, both arguments should be treated as immutable, but a copy of the arguments, modified by the extension might be returned:

```
Callable[[Structure, ScaffoldOpts], Tuple[Structure, ScaffoldOpts]]
```

alias of Callable[[Dict[str, Union[str, None, Callable[[Dict[str, Any]], Optional[str]], string.Template, Tuple[Union[str, None, Callable[[Dict[str, Any]], Optional[str]], string.Template], Callable[[pathlib.Path, Optional[str], Dict[str, Any]], Optional[pathlib.Path]]], dict]], Dict[str, Any]], Tuple[Dict[str, Union[str, None, Callable[[Dict[str, Any]], Optional[str]], string.Template, Tuple[Union[str, None, Callable[[Dict[str, Any]], Optional[str]], string.Template], Callable[[pathlib.Path, Optional[str], Dict[str, Any]], Optional[pathlib.Path]]], dict]], Dict[str, Any]]]

pyscaffold.actions.**ActionParams**
Both argument and return type of an action `(struct, opts)`, so a sequence of actions work in pipeline:

```
Tuple[Structure, ScaffoldOpts]
```

When actions run, they can return an updated copy of `Structure` and *ScaffoldOpts*.

alias of Tuple[Dict[str, Union[str, None, Callable[[Dict[str, Any]], Optional[str]], string.Template, Tuple[Union[str, None, Callable[[Dict[str, Any]], Optional[str]], string.Template], Callable[[pathlib.Path, Optional[str], Dict[str, Any]], Optional[pathlib.Path]]], dict]], Dict[str, Any]]

pyscaffold.actions.**DEFAULT = [<function get_default_options>, <function verify_options_consistency>, <function define_structure>, <function verify_project_dir>, <function version_migration>, <function create_structure>, <function init_git>, <function report_done>]**
Default list of actions forming the main pipeline executed by PyScaffold

pyscaffold.actions.**ScaffoldOpts**
Dictionary with PyScaffold's options, see *pyscaffold.api.create_project*.

alias of Dict[str, Any]

pyscaffold.actions.**discover**(*extensions*)
Retrieve the action list.

This is done by concatenating the default list with the one generated after activating the extensions.

Parameters **extensions** – list of functions responsible for activating the extensions.

pyscaffold.actions.**get_default_options**(*struct*, *opts*)
Compute all the options that can be automatically derived.

This function uses all the available information to generate sensible defaults. Several options that can be derived are computed when possible.

Parameters

- **struct** – project representation as (possibly) nested `dict`.
- **opts** – given options, see `create_project` for an extensive list.

Returns project representation and options with default values set

Return type ActionParams

Raises

- **DirectoryDoesNotExist** – when PyScaffold is told to update an nonexistent directory
- **GitNotInstalled** – when git command is not available
- **GitNotConfigured** – when git does not know user information

---

Note: This function uses git to determine some options, such as author name and email.

---

pyscaffold.actions.**init_git**(*struct*, *opts*)
Add revision control to the generated files.

Parameters

- **struct** – project representation as (possibly) nested `dict`.
- **opts** – given options, see `create_project` for an extensive list.

Returns Updated project representation and options

pyscaffold.actions.**invoke**(*struct_and_opts*, *action*)
Invoke action with proper logging.

Parameters

- **struct_and_opts** – PyScaffold's arguments for actions
- **action** – to be invoked

Returns updated project representation and options

Return type ActionParams

pyscaffold.actions.**register**(*actions*, *action*, *before=None*, *after=None*)
Register a new action to be performed during scaffold.

Parameters

- **actions** (*List[Action]*) – previous action list.
- **action** (*Action*) – function with two arguments: the first one is a (nested) dict representing the file structure of the project and the second is a dict with scaffold options. This function **MUST** return a tuple with two elements similar to its arguments. Example:

```
def do_nothing(struct, opts):
    return (struct, opts)
```

- ****kwargs** (*dict*) – keyword arguments make it possible to choose a specific order when executing actions: when `before` or `after` keywords are provided, the argument value is used as a reference position for the new action. Example:

```
register(actions, do_nothing,
         after='create_structure')
    # Look for the first action with a name
    # `create_structure` and inserts `do_nothing` after it.
    # If more than one registered action is named
    # `create_structure`, the first one is selected.

register(
    actions, do_nothing,
    before='pyscaffold.structure:create_structure')
    # Similar to the previous example, but the probability
    # of name conflict is decreased by including the module
    # name.
```

When no keyword argument is provided, the default execution order specifies that the action will be performed after the project structure is defined, but before it is written to the disk. Example:

```
register(actions, do_nothing)
    # The action will take place after
    # `pyscaffold.structure:define_structure`
```

Returns modified action list.

Return type List[Action]

pyscaffold.actions.**report_done**(*struct*, *opts*)
Just inform the user PyScaffold is done

pyscaffold.actions.**unregister**(*actions*, *reference*)
Prevent a specific action to be executed during scaffold.

Parameters

- **actions** (*List[Action]*) – previous action list.
- **reference** (*str*) – action identifier. Similarly to the keyword arguments of *register* it can assume two formats:
  - the name of the function alone,
  - the name of the module followed by `:` and the name of the function

Returns modified action list.

Return type List[Action]

pyscaffold.actions.**verify_options_consistency**(*struct*, *opts*)
Perform some sanity checks about the given options.

Parameters

- **struct** – project representation as (possibly) nested `dict`.
- **opts** – given options, see `create_project` for an extensive list.

Returns Updated project representation and options

pyscaffold.actions.**verify_project_dir**(*struct*, *opts*)
Check if PyScaffold can materialize the project dir structure.

Parameters

- **struct** – project representation as (possibly) nested `dict`.
- **opts** – given options, see `create_project` for an extensive list.

Returns Updated project representation and options

12.1.4 pyscaffold.api module

External API for accessing PyScaffold programmatically via Python.

pyscaffold.api.**DEFAULT_OPTIONS = {'config_files': [], 'description': 'Add a short description here!', 'extensions': [], 'force': False, 'license': 'MIT', 'update': False, 'url': 'https://github.com/pyscaffold/pyscaffold/', 'version': '4.1.1'}**
Default values for PyScaffold's options.

Options that can be derived from the values of other options (e.g. `package` can be derived from `project_path` when not explicitly passed) are computed in *pyscaffold.actions.get_default_options*.

When `config_files` is empty, a default value is computed dynamically by *pyscaffold.info.config_file* before the start of PyScaffold's action pipeline.

> **Warning:** Default values might be dynamically overwritten by `config_files` or, during updates, existing `setup.cfg`.

pyscaffold.api.**NO_CONFIG = ConfigFiles.NO_CONFIG**
This constant is used to tell PyScaffold to not load any extra configuration file, not even the default ones Usage:

```
create_project(opts, config_files=NO_CONFIG)
```

Please notice that the `setup.cfg` file inside an project being updated will still be considered.

pyscaffold.api.**bootstrap_options**(*opts=None*, ***kwargs*)
Internal API: augment the given options with minimal defaults and existing configurations saved in files (e.g. `setup.cfg`)

See list of arguments in *create_project*. Returns a dictionary of options.

> **Warning:** This function is not part of the public Python API of PyScaffold, and therefore might change even in minor/patch releases (not bounded to semantic versioning).

> **Note:** This function does not replace the *pyscaffold.actions.get_default_options* action. Instead it is needed to ensure that action works correctly.

pyscaffold.api.**create_project**(*opts=None*, ***kwargs*)
Create the project's directory structure

Parameters

- **opts** (*dict*) – options of the project

- ****kwargs** – extra options, passed as keyword arguments

Returns a tuple of *struct* and *opts* dictionary

Return type tuple

Valid options include:

Project Information

- **project_path** (`os.PathLike` or `str`)

Naming

- **name** (*str*): as in `pip install` or in PyPI
- **package** (*str*): Python identifier as in `import` (without namespace)

Package Information

- **author** (*str*)
- **email** (*str*)
- **release_date** (*str*)
- **year** (*str*)
- **title** (*str*)
- **description** (*str*)
- **url** (*str*)
- **classifiers** (*str*)
- **requirements** (*list*)

PyScaffold Control

- **update** (*bool*)
- **force** (*bool*)
- **pretend** (*bool*)
- **extensions** (*list*)
- **config_files** (*list* or `NO_CONFIG`)

Some of these options are equivalent to the command line options, others are used for creating the basic python package meta information, but the ones in the "PyScaffold Control" group affects how the "scaffolding" behaves.

When the **force** flag is `True`, existing files will be overwritten. When the **update** flag is `True`, PyScaffold will consider that some files can be updated (usually the packaging boilerplate), but will keep others intact. When the **pretend** flag is `True`, the project will not be created/updated, but the expected outcome will be logged.

The **extensions** list may contain any object that follows the *extension API*. Note that some PyScaffold features, such as cirrus, tox and pre-commit support, are implemented as built-in extensions. In order to use these features it is necessary to include the respective objects in the extension list. All built-in extensions are accessible via `pyscaffold.extensions` submodule.

Finally, when `setup.cfg`-like files are added to the **config_files** list, PyScaffold will read it's options from there in addition to the ones already passed. If the list is empty, the default configuration file is used. To avoid reading any existing configuration, please pass `config_file=NO_CONFIG`. See *usage* for more details.

Note that extensions may define extra options. For example, the cookiecutter extension define a `cookiecutter` option that should be the address to the git repository used as template and the `namespace` extension define a `namespace` option with the name of a PEP 420 compatible (and possibly nested) namespace.

12.1.5 pyscaffold.cli module

Command-Line-Interface of PyScaffold

`pyscaffold.cli.`**`add_default_args`**(*parser*)
Add the default options and arguments to the CLI parser.

`pyscaffold.cli.`**`add_extension_args`**(*parser*)
Add options and arguments defined by extensions to the CLI parser.

`pyscaffold.cli.`**`add_log_related_args`**(*parser*)
Add options that control verbosity/logger level

`pyscaffold.cli.`**`get_log_level`**(*args=None*)
Get the configured log level directly by parsing CLI options from `args` or obj:*sys.argv*.

Useful when the CLI crashes before applying the changes to the logger.

`pyscaffold.cli.`**`list_actions`**(*opts*)
Do not create a project, just list actions considering extensions

Parameters **`opts`** (*dict*) – command line options as dictionary

`pyscaffold.cli.`**`main`**(*args*)
Main entry point for external applications

Parameters **`args`** – command line arguments

`pyscaffold.cli.`**`parse_args`**(*args*)
Parse command line parameters respecting extensions

Parameters **`args`** – command line parameters as list of strings

Returns command line parameters

Return type dict

`pyscaffold.cli.`**`run`**(*args=None*)
Entry point for console script

`pyscaffold.cli.`**`run_scaffold`**(*opts*)
Actually scaffold the project, calling the python API

Parameters **`opts`** (*dict*) – command line options as dictionary

12.1.6 pyscaffold.dependencies module

Internal library for manipulating package dependencies and requirements.

`pyscaffold.dependencies.`**`BUILD = ('setuptools_scm>=5', 'wheel')`**
Dependencies that will be required to build the created project

`pyscaffold.dependencies.`**`ISOLATED = ('setuptools>=46.1.0', 'setuptools_scm[toml]>=5', 'wheel')`**
Dependencies for isolated builds (PEP517/518). - setuptools min version might be slightly higher then the version required at runtime. - setuptools_scm requires an optional dependency to work with pyproject.toml

pyscaffold.dependencies.**REQ_SPLITTER = re.compile(';(?!\\s*(python|platform|implementation|os|sys)_)', re.MULTILINE)**
Regex to split requirements that considers both setup.cfg specs and PEP 508 (in a *good enough* simplified fashion).

pyscaffold.dependencies.**RUNTIME = ('importlib-metadata; python_version<"3.8"',)**
Dependencies that will be required at runtime by the created project

pyscaffold.dependencies.**add**(*requirements*, *to_add=('setuptools_scm>=5', 'wheel')*)
Given a sequence of individual requirement strings, add `to_add` to it. By default adds *BUILD* if `to_add` is not given.

pyscaffold.dependencies.**attempt_pkg_name**(*requirement*)
In the case the given string is a dependency specification (PEP 508/440), it returns the "package name" part of dependency (without versions). Otherwise, it returns the same string (removed the comment marks).

pyscaffold.dependencies.**deduplicate**(*requirements*)
Given a sequence of individual requirement strings, e.g. `["appdirs>=1.4.4", "packaging>20.0"]`, remove the duplicated packages. If a package is duplicated, the last occurrence stays.

pyscaffold.dependencies.**remove**(*requirements*, *to_remove*)
Given a list of individual requirement strings, e.g. `["appdirs>=1.4.4", "packaging>20.0"]`, remove the requirements in `to_remove`.

pyscaffold.dependencies.**split**(*requirements*)
Split a combined requirement string (such as the values for `setup_requires` and `install_requires` in `setup.cfg`) into a list of individual requirement strings, that can be used in `is_included`, `get_requirements_str`, *remove*, etc...

12.1.7 pyscaffold.exceptions module

Functions for exception manipulation + custom exceptions used by PyScaffold to identify common deviations from the expected behavior.

exception pyscaffold.exceptions.**ActionNotFound**(*name*, **args*, ***kwargs*)
Bases: KeyError

Impossible to find the required action.

exception pyscaffold.exceptions.**DirectErrorForUser**(*message=None*, **args*, ***kwargs*)
Bases: RuntimeError

Parent class for exceptions that can be shown directly as error messages to the final users.

exception pyscaffold.exceptions.**DirectoryAlreadyExists**(*message=None*, **args*, ***kwargs*)
Bases: *pyscaffold.exceptions.DirectErrorForUser*

The project directory already exists, but no `update` or `force` option was used.

exception pyscaffold.exceptions.**DirectoryDoesNotExist**(*message=None*, **args*, ***kwargs*)
Bases: *pyscaffold.exceptions.DirectErrorForUser*

No directory was found to be updated.

exception pyscaffold.exceptions.**ErrorLoadingExtension**(*extension=''*, *entry_point=None*)
Bases: *pyscaffold.exceptions.DirectErrorForUser*

There was an error loading '{extension}'. Please make sure you have installed a version of the extension that is compatible with PyScaffold {version}. You can also try unininstalling it.

exception pyscaffold.exceptions.**ExtensionNotFound**(*extensions*)
Bases: *pyscaffold.exceptions.DirectErrorForUser*

The following extensions were not found: {extensions}. Please make sure you have the required versions installed. You can look for official extensions at https://github.com/pyscaffold.

exception pyscaffold.exceptions.**GitDirtyWorkspace**(*message="Your working tree is dirty. Commit your changes first or use '--force'.", *args, **kwargs*)
Bases: *pyscaffold.exceptions.DirectErrorForUser*

Workspace of git is empty.

DEFAULT_MESSAGE = "Your working tree is dirty. Commit your changes first or use '--force'."

exception pyscaffold.exceptions.**GitNotConfigured**(*message='Make sure git is configured. Run:\n git config --global user.email "you@example.com"\n git config --global user.name "Your Name"\nto set your account\'s default identity.', *args, **kwargs*)
Bases: *pyscaffold.exceptions.DirectErrorForUser*

PyScaffold tries to read user.name and user.email from git config.

DEFAULT_MESSAGE = 'Make sure git is configured. Run:\n git config --global user.email "you@example.com"\n git config --global user.name "Your Name"\nto set your account\'s default identity.'

exception pyscaffold.exceptions.**GitNotInstalled**(*message='Make sure git is installed and working.', *args, **kwargs*)
Bases: *pyscaffold.exceptions.DirectErrorForUser*

PyScaffold requires git to run.

DEFAULT_MESSAGE = 'Make sure git is installed and working.'

exception pyscaffold.exceptions.**ImpossibleToFindConfigDir**(*message=None, *args, **kwargs*)
Bases: *pyscaffold.exceptions.DirectErrorForUser*

An expected error occurred when trying to find the config dir.

This might be related to not being able to read the $HOME env var in Unix systems, or %USERPROFILE% in Windows, or even the username.

exception pyscaffold.exceptions.**InvalidIdentifier**
Bases: RuntimeError

Python requires a specific format for its identifiers.

https://docs.python.org/3.6/reference/lexical_analysis.html#identifiers

exception pyscaffold.exceptions.**NoPyScaffoldProject**(*message='Could not update project. Was it generated with PyScaffold?', *args, **kwargs*)
Bases: *pyscaffold.exceptions.DirectErrorForUser*

PyScaffold cannot update a project that it hasn't generated

DEFAULT_MESSAGE = 'Could not update project. Was it generated with PyScaffold?'

exception pyscaffold.exceptions.**PyScaffoldTooOld**(*message='setup.cfg has no section [pyscaffold]! Are you trying to update a pre 3.0 version?', *args, **kwargs*)
Bases: *pyscaffold.exceptions.DirectErrorForUser*

PyScaffold cannot update a pre 3.0 version

DEFAULT_MESSAGE = 'setup.cfg has no section [pyscaffold]! Are you trying to update a pre 3.0 version?'

exception pyscaffold.exceptions.**ShellCommandException**
Bases: RuntimeError

Outputs proper logging when a ShellCommand fails

pyscaffold.exceptions.**exceptions2exit**(*exception_list*)
Decorator to convert given exceptions to exit messages

This avoids displaying nasty stack traces to end-users

Parameters **[Exception]** (*exception_list*) – list of exceptions to convert

12.1.8 pyscaffold.file_system module

Internal library that encapsulate file system manipulation. Examples include: creating/removing files and directories, changing permissions, etc.

Functions in this library usually extend the behaviour of Python's standard lib by providing proper error handling or adequate logging/control flow in the context of PyScaffold (an example of adequate control flow logic is dealing with the `pretend` flag).

pyscaffold.file_system.**ERROR_INVALID_NAME = 123**
Windows-specific error code indicating an invalid pathname.

pyscaffold.file_system.**chdir**(*path*, ***kwargs*)
Contextmanager to change into a directory

Parameters **path** – path to change current working directory to

Keyword Arguments

- **log** (*bool*) – log activity when true. Default: `False`.
- **pretend** (*bool*) – skip execution (but log) when pretending. Default `False`.

pyscaffold.file_system.**chmod**(*path*, *mode*, *pretend=False*)
Change the permissions of file in the given path.

This function reports the operation in the logs.

Parameters

- **path** – path in the file system whose permissions will be changed
- **mode** – new permissions, should be a combination of :obj`stat.S_* <stat.S_IXUSR>` (see `os.chmod`).
- **pretend** (*bool*) – false by default. File is not changed when pretending, but operation is logged.

pyscaffold.file_system.**create_directory**(*path*, *update=False*, *pretend=False*)
Create a directory in the given path.

This function reports the operation in the logs.

Parameters

- **path** – path in the file system where contents will be written.
- **update** (*bool*) – false by default. A `OSError` can be raised when update is false and the directory already exists.

- **pretend** (*bool*) – false by default. Directory is not created when pretending, but operation is logged.

pyscaffold.file_system.**create_file**(*path*, *content*, *pretend=False*, *encoding='utf-8'*)
Create a file in the given path.

This function reports the operation in the logs.

Parameters

- **path** – path in the file system where contents will be written.
- **content** – what will be written.
- **pretend** (*bool*) – false by default. File is not written when pretending, but operation is logged.

Returns given path

Return type Path

pyscaffold.file_system.**is_pathname_valid**(*pathname*)
Check if a pathname is valid

Code by Cecil Curry from StackOverflow

Parameters **pathname** (*str*) – string to validate

Returns *True* if the passed pathname is a valid pathname for the current OS; *False* otherwise.

pyscaffold.file_system.**localize_path**(*path_string*)
Localize path for Windows, Unix, i.e. / or

Parameters **path_string** (*str*) – path using /

Returns path depending on OS

Return type str

pyscaffold.file_system.**move**(**src*, *target*, ***kwargs*)
Move files or directories to (into) a new location

Parameters ***src** (*PathLike*) – one or more files/directories to be moved

Keyword Arguments

- **target** (*PathLike*) – if target is a directory, `src` will be moved inside it. Otherwise, it will be the new path (note that it may be overwritten)
- **log** (*bool*) – log activity when true. Default: `False`.
- **pretend** (*bool*) – skip execution (but log) when pretending. Default `False`.

pyscaffold.file_system.**on_ro_error**(*func*, *path*, *exc_info*)
Error handler for `shutil.rmtree`.

If the error is due to an access error (read only file) it attempts to add write permission and then retries.

If the error is for another reason it re-raises the error.

Usage : `shutil.rmtree(path, onerror=onerror)`

Parameters

- **func** (*callable*) – function which raised the exception
- **path** (*str*) – path passed to *func*
- **exc_info** (*tuple of str*) – exception info returned by sys.exc_info()

pyscaffold.file_system.**rm_rf**(*path*, *pretend=False*)
Remove `path` by all means like `rm -rf` in Linux

pyscaffold.file_system.**tmpfile**(***kwargs*)
Context manager that yields a temporary `Path`

12.1.9 pyscaffold.identification module

Internal library for manipulating, creating and dealing with names, or more generally identifiers.

pyscaffold.identification.**dasherize**(*word*)
Replace underscores with dashes in the string.

Example:

```
>>> dasherize("foo_bar")
"foo-bar"
```

Parameters **word** (*str*) – input word

Returns input word with underscores replaced by dashes

pyscaffold.identification.**deterministic_name**(*obj*)
Private API that returns an string that can be used to deterministically deduplicate and sort sequences of objects.

pyscaffold.identification.**deterministic_sort**(*sequence*)
Private API that order a sequence of objects lexicographically (by *deterministic_name*), removing duplicates, which is needed for determinism.

The main purpose of this function is to deterministically sort a sequence of PyScaffold extensions (it will also sort internal extensions before external: "pyscaffold.*" < "pyscaffoldext.*").

pyscaffold.identification.**get_id**(*function*)
Given a function, calculate its identifier.

A identifier is a string in the format `<module name>:<function name>`, similarly to the convention used for setuptools entry points.

Note: This function does not return a Python 3 `__qualname__` equivalent. If the function is nested inside another function or class, the parent name is ignored.

Parameters **function** (*callable*) – function object

pyscaffold.identification.**is_valid_identifier**(*string*)
Check if string is a valid package name

Parameters **string** – package name

Returns True if string is valid package name else False

pyscaffold.identification.**levenshtein**(*s1*, *s2*)
Calculate the Levenshtein distance between two strings

Parameters

- **s1** – first string
- **s2** – second string

Returns Distance between s1 and s2

pyscaffold.identification.**make_valid_identifier**(*string*)
Try to make a valid package name identifier from a string

Parameters **string** – invalid package name

Returns Valid package name as string or RuntimeError

Raises ***InvalidIdentifier*** – raised if identifier can not be converted

pyscaffold.identification.**underscore**(*word*)
Convert CamelCasedStrings or dasherized-strings into underscore_strings.

Example:

```
>>> underscore("FooBar-foo")
"foo_bar_foo"
```

12.1.10 pyscaffold.info module

Provide general information about the system, user and the package itself.

pyscaffold.info.**CONFIG_FILE = 'default.cfg'**
PyScaffold's own config file name

class pyscaffold.info.**GitEnv**(*value*)
Bases: enum.Enum

An enumeration.

author_date = 'GIT_AUTHOR_DATE'

author_email = 'GIT_AUTHOR_EMAIL'

author_name = 'GIT_AUTHOR_NAME'

committer_date = 'GIT_COMMITTER_DATE'

committer_email = 'GIT_COMMITTER_EMAIL'

committer_name = 'GIT_COMMITTER_NAME'

pyscaffold.info.**RAISE_EXCEPTION = default.RAISE_EXCEPTION**
When no default value is passed, an exception should be raised

pyscaffold.info.**best_fit_license**(*txt*)
Finds proper license name for the license defined in txt

pyscaffold.info.**check_git**()
Checks for git and raises appropriate exception if not

Raises

- **GitNotInstalled** – when git command is not available
- **GitNotConfigured** – when git does not know user information

pyscaffold.info.**config_dir**(*prog='pyscaffold'*, *org=None*, *default=default.RAISE_EXCEPTION*)
Finds the correct place where to read/write configurations for the given app.

Parameters

- **prog** – program name (defaults to pyscaffold)

- **org** – organisation/author name (defaults to the same as `prog`)
- **default** – default value to return if an exception was raise while trying to find the config dir. If no default value is passed, an *ImpossibleToFindConfigDir* execution is raised.

Please notice even if the directory doesn't exist, if its path is possible to calculate, this function will return a Path object (that can be used to create the directory)

Returns Location somewhere in the user's home directory where to put the configs.

pyscaffold.info.**config_file**(*name='default.cfg'*, *prog='pyscaffold'*, *org=None*, *default=default.RAISE_EXCEPTION*)

Finds a file inside *config_dir*.

Parameters **name** – file you are looking for

The other args are the same as in *config_dir* and have the same meaning.

Returns Location of the config file or default if an error happened.

pyscaffold.info.**email**()

Retrieve the user's email

pyscaffold.info.**get_curr_version**(*project_path*)

Retrieves the PyScaffold version that put up the scaffold

Parameters **project_path** – path to project

Returns version specifier

Return type Version

pyscaffold.info.**is_git_configured**()

Check if user.name and user.email is set globally in git

Check first git environment variables, then config settings. This will also return false if git is not available at all.

Returns True if it is set globally, False otherwise

pyscaffold.info.**is_git_installed**()

Check if git is installed

pyscaffold.info.**is_git_workspace_clean**(*path*)

Checks if git workspace is clean

Parameters **path** – path to git repository

Raises: *GitNotInstalled*: when git command is not available *GitNotConfigured*: when git does not know user information

pyscaffold.info.**project**(*opts*, *config_path=None*, *config_file=None*)

Update user options with the options of an existing config file

Parameters

- **opts** – options of the project
- **config_path** – path where config file can be found (default: `opts["project_path"]`)
- **config_file** – if `config_path` is a directory, name of the config file, relative to it (default: `setup.cfg`)

Returns Options with updated values

Raises

- **PyScaffoldTooOld** – when PyScaffold is to old to update from

- **NoPyScaffoldProject** – when project was not generated with PyScaffold

pyscaffold.info.**read_pyproject**(*path*, *filename='pyproject.toml'*)
Reads-in a configuration file that follows a pyproject.toml format.

Parameters

- **path** – path where to find the config file
- **filename** – if `path` is a directory, `name` will be considered a file relative to `path` to read (default: setup.cfg)

Returns Object that can be used to read/edit configuration parameters.

pyscaffold.info.**read_setupcfg**(*path*, *filename='setup.cfg'*)
Reads-in a configuration file that follows a setup.cfg format. Useful for retrieving stored information (e.g. during updates)

Parameters

- **path** – path where to find the config file
- **filename** – if `path` is a directory, `name` will be considered a file relative to `path` to read (default: setup.cfg)

Returns Object that can be used to read/edit configuration parameters.

pyscaffold.info.**username**()
Retrieve the user's name

12.1.11 pyscaffold.log module

Custom logging infrastructure to provide execution information for the user.

class pyscaffold.log.**ColoredReportFormatter**(*fmt=None*, *datefmt=None*, *style='%'*, *validate=True*)
Bases: *pyscaffold.log.ReportFormatter*

Format logs with ANSI colors.

ACTIVITY_STYLES = {'create': ('green', 'bold'), 'delete': ('red', 'bold'), 'invoke': ('bold',), 'move': ('green', 'bold'), 'remove': ('red', 'bold'), 'run': ('magenta', 'bold'), 'skip': ('yellow', 'bold')}

CONTEXT_PREFIX = '\x1b[35m\x1b[1mfrom\x1b[0m'

LOG_STYLES = {'critical': ('red', 'bold'), 'debug': ('green',), 'error': ('red',), 'info': ('blue',), 'warning': ('yellow',)}

SUBJECT_STYLES = {'invoke': ('blue',)}

TARGET_PREFIX = '\x1b[35m\x1b[1mto\x1b[0m'

format_activity(*activity*)
Format the activity keyword.

format_default(*record*)
Format default log messages.

format_subject(*subject*, *activity=None*)
Format the subject of the activity.

pyscaffold.log.**DEFAULT_LOGGER = 'pyscaffold.log'**
Name of PyScaffold's default logger (it can be used with `logging.getLogger`)

class pyscaffold.log.**ReportFormatter**(*fmt=None*, *datefmt=None*, *style='%'*, *validate=True*)

Bases: logging.Formatter

Formatter that understands custom fields in the log record.

ACTIVITY_MAXLEN = 12

CONTEXT_PREFIX = 'from'

SPACING = ' '

TARGET_PREFIX = 'to'

create_padding(*activity*)
Create the appropriate padding in order to align activities.

format(*record*)
Compose message when a record with report information is given.

format_activity(*activity*)
Format the activity keyword.

format_context(*context*, *_activity=None*)
Format extra information about the activity context.

format_default(*record*)
Format default log messages.

format_path(*path*)
Simplify paths to avoid wasting space in terminal.

format_report(*record*)
Compose message when a custom record is given.

format_subject(*subject*, *_activity=None*)
Format the subject of the activity.

format_target(*target*, *_activity=None*)
Format extra information about the activity target.

class pyscaffold.log.**ReportLogger**(*logger=None*, *handler=None*, *formatter=None*, *extra=None*, *propagate=False*)

Bases: logging.LoggerAdapter

Suitable wrapper for PyScaffold CLI interactive execution reports.

Parameters

- **logger** (*logging.Logger*) – custom logger to be used. Optional: the default logger will be used.
- **handlers** (*logging.Handler*) – custom logging handler to be used. Optional: a logging.StreamHandler is used by default.
- **formatter** (*logging.Formatter*) – custom formatter to be used. Optional: by default a *ReportFormatter* is created and used.
- **extra** (*dict*) – extra attributes to be merged into the log record. Options, empty by default.
- **propagate** (*bool*) – whether or not to propagate messages in the logging hierarchy, False by default. See logging.Logger.propagate.

nesting
current nesting level of the report.

Type int

copy()
Produce a copy of the wrapped logger.

Sometimes, it is better to make a copy of th report logger to keep indentation consistent.

property formatter
Formatter configured in the default handler

property handler
Stream handler configured for providing user feedback in PyScaffold CLI

indent(*count=1*)
Temporarily adjust padding while executing a context.

Example

```
from pyscaffold.log import logger

logger.report("invoke", "custom_action")
with logger.indent():
    logger.report("create", "some/file/path")

# Expected logs:
# --------------------------------------
#       invoke  custom_action
#       create    some/file/path
# --------------------------------------
# Note how the spacing between activity and subject in the
# second entry is greater than the equivalent in the first one.
```

Note: This method is not thread-safe and should be used with care.

property level
Effective level of the logger

process(*msg*, *kwargs*)
Method overridden to augment LogRecord with the *nesting* attribute

property propagate
Whether or not to propagate messages in the logging hierarchy, See `logging.Logger.propagate`.

reconfigure(*opts=None*, ***kwargs*)
Reconfigure some aspects of the logger object.

Parameters **opts** (*dict*) – dict with the same elements as the keyword arguments bellow

Keyword Arguments

- **log_level** – One of the log levels specified in the `logging` module.
- **use_colors** – automatically set a colored formatter to the logger if ANSI codes support is detected. (Defaults to *True*).

Additional keyword arguments will be ignored.

report(*activity*, *subject*, *context=None*, *target=None*, *nesting=None*, *level=20*)
Log that an activity has occurred during scaffold.

Parameters

- **activity** (*str*) – usually a verb or command, e.g. `create`, `invoke`, `run`, `chdir`...
- **subject** (*str or os.PathLike*) – usually a path in the file system or an action identifier.
- **context** (*str or os.PathLike*) – path where the activity take place.
- **target** (*str or os.PathLike*) – path affected by the activity
- **nesting** (*int*) – optional nesting level. By default it is calculated from the activity name.
- **level** (*int*) – log level. Defaults to `logging.INFO`. See `logging` for more information.

Notes

This method creates a custom log record, with additional fields: **activity**, **subject**, **context**, **target** and **nesting**, but an empty **msg** field. The *ReportFormatter* creates the log message from the other fields.

Often **target** and **context** complement the logs when **subject** does not hold all the necessary information. For example:

```
logger.report('copy', 'my/file', target='my/awesome/path')
logger.report('run', 'command', context='current/working/dir')
```

property wrapped
Underlying logger object

pyscaffold.log.**logger = <ReportLogger pyscaffold.log (WARNING)>**
Default logger configured for PyScaffold.

12.1.12 pyscaffold.operations module

Collection of functions responsible for the "reification" of the file representation in the project structure into a file written to the disk.

A function that "reifies" a file (manifests the programmatic representation of the file in the project structure dictionary to the disk) is called here a **file operation** or simply a **file op**. Actually file ops don't even need to be functions strictly speaking, they can be any `callable` object. The only restriction is that file ops **MUST** respect the *FileOp* signature. The *create* function is a good example of how to write a new file op.

A function (or callable) that modifies the behaviour of an existing file op, by wrapping it with another function/callable is called here **modifier**. Modifiers work similarly to Python decorators and allow extending/refining the existing file ops. A modifier should receive a file op as argument and return another file op. *no_overwrite* and *skip_on_update* are good examples on how to write new file op modifiers.

While modifiers don't have a specific signature (the number of parameters might vary, but they always return a single file op value), the following conventions **SHOULD** be observed when creating new modifiers:

- Modifiers should accept at least one argument: the **file op** they modify (you don't have to be very creative, go ahead and name this parameter `file_op`, it is a good convention). This parameter should be made **optional** (the default value of *create* usually works, unless there is a better default value for the main use case of the modifier).
- Modifiers can accept arguments other then the file op they modify. These arguments should **precede** the file op in the list of arguments (the file op should be the last argument, which interoperates well with `partial`).
- When writing a modifier that happens to be a function (instead of a callable class), please name the inner function with the same name of the modifier but preceded by an _ (underscore) char. This allows better inspection/debugging.

Changed in version 4.0: Previously, file operations where simply indicated as a numeric flag (the members of `pyscaffold.structure.FileOp`) in the project structure. Starting from PyScaffold 4, file operation are functions with signature given by *FileOp*.

pyscaffold.operations.**FileContents**
When the file content is `None`, the file should not be written to disk (empty files are represented by an empty string "" as content).

alias of `Optional[str]`

pyscaffold.operations.**FileOp**
Signature of functions considered file operations:

```
Callable[[Path, FileContents, ScaffoldOpts], Union[Path, None]]
```

Parameters

- **path** (*pathlib.Path*) – file path potentially to be written to/changed in the disk.
- **contents** (*FileContents*) – usually a string that represents a text content of the file. `None` indicates the file should not be written.
- **opts** (*ScaffoldOpts*) – a dict with PyScaffold's options.

Returns If the file is written (or more generally changed, such as new access permissions), by convention they should return the `file path`. If no file was touched, `None` should be returned. Please notice a **FileOp** might return `None` if a pre-existing file in the disk is not modified.

Note: A **FileOp** usually has side effects (e.g. write a file to the disk), see `FileFileContents` and *ScaffoldOpts* for other conventions.

---

alias of `Callable[[pathlib.Path, Optional[str], Dict[str, Any]], Optional[pathlib.Path]]`

pyscaffold.operations.**ScaffoldOpts**
Dictionary with PyScaffold's options, see *pyscaffold.api.create_project*. Should be treated as immutable (if required, copy before changing).

Please notice some behaviours given by the options **SHOULD** be observed. For example, files should be overwritten when the **force** option is `True`. Similarly when **pretend** is `True`, no operation should be really performed, but any action should be logged as if realized.

alias of `Dict[str, Any]`

pyscaffold.operations.**add_permissions**(*permissions*, *file_op=<function create>*)
File op modifier. Returns a *FileOp* that will **add** access permissions to the file (on top of the ones given by default by the OS).

Parameters

- **permissions** (*int*) – permissions to be added to file:

  ```
  updated file mode = old mode | permissions  (bitwise OR)
  ```

  Preferably the values should be a combination of `stat.S_*` values (see `os.chmod`).
- **file_op** – a *FileOp* that will be "decorated". If the file exists in disk after `file_op` is called (either created or pre-existing), `permissions` will be added to it. Default: *create*.

> **Warning:** This is an **experimental** file op and might be subject to incompatible changes (or complete removal) even in minor/patch releases.

Note: File access permissions work in a completely different way depending on the operating system. This file op is straightforward on POSIX systems, but a bit tricky on Windows. It should be safe for desirable but not required effects (e.g. making a file with a shebang executable, but that can also run via `python FILE`), or ensuring files are readable/writable.

`pyscaffold.operations.`**`create`**(*path*, *contents*, *opts*)
Default *FileOp*: always create/write the file even during (forced) updates.

`pyscaffold.operations.`**`no_overwrite`**(*file_op=<function create>*)
File op modifier. Returns a *FileOp* that does not overwrite an existing file during update (still created if not exists).

Parameters **`file_op`** – a *FileOp* that will be "decorated", i.e. will be called if the `no_overwrite` conditions are met. Default: *create*.

`pyscaffold.operations.`**`remove`**(*path*, *_content*, *opts*)
Remove the file if it exists in the disk

`pyscaffold.operations.`**`skip_on_update`**(*file_op=<function create>*)
File op modifier. Returns a *FileOp* that will skip the file during a project update (the file will just be created for brand new projects).

Parameters **`file_op`** – a *FileOp* that will be "decorated", i.e. will be called if the `skip_on_update` conditions are met. Default: *create*.

12.1.13 pyscaffold.repo module

Functionality for working with a git repository

`pyscaffold.repo.`**`add_tag`**(*project*, *tag_name*, *message=None*, ***kwargs*)
Add an (annotated) tag to the git repository.

Parameters

- **`project`** – path to the project
- **`tag_name`** – name of the tag
- **`message`** – optional tag message

Additional keyword arguments are passed to the *git* callable object.

`pyscaffold.repo.`**`get_git_root`**(*default=None*)
Return the path to the top-level of the git repository or *default*.

Parameters **`default`** – if no git root is found, default is returned

Returns top-level path or *default*

Return type str

`pyscaffold.repo.`**`git_tree_add`**(*struct*, *prefix=''*, ***kwargs*)
Adds recursively a directory structure to git

Parameters

- **struct** – directory structure as dictionary of dictionaries
- **prefix** – prefix for the given directory structure

Additional keyword arguments are passed to the *git* callable object.

pyscaffold.repo.**init_commit_repo**(*project*, *struct*, ***kwargs*)
Initialize a git repository

Parameters

- **project** – path to the project
- **struct** – directory structure as dictionary of dictionaries

Additional keyword arguments are passed to the *git* callable object.

pyscaffold.repo.**is_git_repo**(*folder*)
Check if a folder is a git repository

12.1.14 pyscaffold.shell module

Shell commands like git, django-admin etc.

pyscaffold.shell.**EDITORS = {'atom': ['--wait'], 'code': ['--wait'], 'mate': ['-w'], 'nano': [], 'notepad': [], 'nvim': [], 'sensible-editor': [], 'subl': ['-w'], 'vi': [], 'vim': []}**
Programs to be tried (in sequence) when calling *edit* and *get_editor* in the case the environment variables EDITOR and VISUAL are not set.

class pyscaffold.shell.**ShellCommand**(*command*, *shell=True*, *cwd=None*)
Bases: `object`

Shell command that can be called with flags like git('add', 'file')

Parameters

- **command** – command to handle
- **shell** – run the command in the shell (`True` by default).
- **cwd** – current working dir to run the command

The produced command can be called with the following keyword arguments:

- **log** (*bool*): log activity when true. `False` by default.
- **pretend** (*bool*): skip execution (but log) when pretending. `False` by default.

The positional arguments are passed to the underlying shell command. In the case the path to the executable contains spaces of any other special shell character, `command` needs to be properly quoted.

run(**args*, ***kwargs*)
Execute command with the given arguments via `subprocess.run`.

pyscaffold.shell.**command_exists**(*cmd*)
Check if command exists

Parameters **cmd** – executable name

pyscaffold.shell.**edit**(*file*, **args*, ***kwargs*)
Open a text editor and returns back a `Path` to file, after user editing.

pyscaffold.shell.**get_command**(*name*, *prefix='/home/docs/checkouts/readthedocs.org/user_builds/pyscaffold/envs/v4.1.1'*, *include_path=True*, *shell=True*, ***kwargs*)

Similar to *get_executable* but return an instance of *ShellCommand* if it is there to be found. Additional kwargs will be passed to the *ShellCommand* constructor.

pyscaffold.shell.**get_editor**(***kwargs*)

Get an available text editor program

pyscaffold.shell.**get_executable**(*name*, *prefix='/home/docs/checkouts/readthedocs.org/user_builds/pyscaffold/envs/v4.1.1'*, *include_path=True*)

Find an executable in the system, if available.

Parameters

- **name** – name of the executable
- **prefix** – look on this directory, exclusively or in additon to $PATH depending on the value of `include_path`. Defaults to `sys.prefix`.
- **include_path** – when True the functions tries to look in the entire $PATH.

Note: The return value might contain whitespaces. If this value is used in a shell environment, it needs to be quote properly to avoid the underlying shell interpreter splitting the executable path.

---

pyscaffold.shell.**get_git_cmd**(***args*)

Retrieve the git shell command depending on the current platform

Parameters ****args** – additional keyword arguments to *ShellCommand*

pyscaffold.shell.**git = <pyscaffold.shell.ShellCommand object>**

Command for git

pyscaffold.shell.**join**(*parts*)

Join different parts of a shell command into a string, quoting whitespaces.

pyscaffold.shell.**shell_command_error2exit_decorator**(*func*)

Decorator to convert given ShellCommandException to an exit message

This avoids displaying nasty stack traces to end-users

12.1.15 pyscaffold.structure module

Functionality to generate and work with the directory structure of a project.

Changed in version 4.0: `Callable[[dict], str]` and `string.Template` objects can also be used as file contents. They will be called with PyScaffold's `opts` (`string.Template` via `safe_substitute`)

pyscaffold.structure.**AbstractContent**

Recipe for obtaining file contents

```
Union[FileContents, Callable[[ScaffoldOpts], FileContents], Template]
```

alias of `Union[str, None, Callable[[Dict[str, Any]], Optional[str]], string.Template]`

pyscaffold.structure.**ActionParams**

See *pyscaffold.actions.ActionParams*

alias of Tuple[Dict[str, Union[str, None, Callable[[Dict[str, Any]], Optional[str]], string.Template, Tuple[Union[str, None, Callable[[Dict[str, Any]], Optional[str]], string.Template], Callable[[pathlib.Path, Optional[str], Dict[str, Any]], Optional[pathlib.Path]]], dict]], Dict[str, Any]]

pyscaffold.structure.**Leaf**

Just the content of the file OR a tuple of content + file operation

```
Union[AbstractContent, ResolvedLeaf]
```

alias of Union[str, None, Callable[[Dict[str, Any]], Optional[str]], string.Template, Tuple[Union[str, None, Callable[[Dict[str, Any]], Optional[str]], string.Template], Callable[[pathlib.Path, Optional[str], Dict[str, Any]], Optional[pathlib.Path]]]]

pyscaffold.structure.**Node**

Representation of a *file system node* in the project structure (e.g. files, directories:

```
Union[Leaf, Structure]
```

alias of Union[str, None, Callable[[Dict[str, Any]], Optional[str]], string.Template, Tuple[Union[str, None, Callable[[Dict[str, Any]], Optional[str]], string.Template], Callable[[pathlib.Path, Optional[str], Dict[str, Any]], Optional[pathlib.Path]]], dict]

pyscaffold.structure.**ReifiedLeaf**

Similar to *ResolvedLeaf* but with file contents "reified", i.e. an actual string instead of a "lazy object" (such as a function or template).

alias of Tuple[Optional[str], Callable[[pathlib.Path, Optional[str], Dict[str, Any]], Optional[pathlib.Path]]]

pyscaffold.structure.**ResolvedLeaf**

Complete *recipe* for manipulating a file in disk (not only its contents but also the file operation:

```
Tuple[AbstractContent, FileOp]
```

alias of Tuple[Union[str, None, Callable[[Dict[str, Any]], Optional[str]], string.Template], Callable[[pathlib.Path, Optional[str], Dict[str, Any]], Optional[pathlib.Path]]]

pyscaffold.structure.**Structure**

The directory tree represented as a (possibly nested) dictionary:

```
Structure = Dict[str, Node]
Node = Union[Leaf, Structure]
```

The keys indicate the path where a file will be written, while the value indicates the content.

A nested dictionary represent a nested directory, while str, string.Template and callable values represent a file to be created. tuple values are also allowed, and in that case, the first element of the tuple represents the file content while the second element is a pyscaffold.operations (which can be seen as a recipe on how to create a file with the given content). Callable file contents are transformed into strings by calling them with PyScaffold's option dict as argument. Similarly, string.Template.safe_substitute are called with PyScaffold's opts.

The top level keys in the dict are file/dir names relative to the project root, while keys in nested dicts are relative to the parent's key/location.

For example:

```
from pyscaffold.operations import no_overwrite
struct: Structure = {
    'namespace': {
        'module.py': ('print("Hello World!")', no_overwrite())
    }
}
```

represents a `namespace/module.py` file inside the project folder with content `print("Hello World!")`, that will be created only if not present.

Note: `None` file contents are ignored and not created in disk.

---

alias of `Dict[str, Union[str, None, Callable[[Dict[str, Any]], Optional[str]], string.Template, Tuple[Union[str, None, Callable[[Dict[str, Any]], Optional[str]], string.Template], Callable[[pathlib.Path, Optional[str], Dict[str, Any]], Optional[pathlib.Path]]], dict]]`

pyscaffold.structure.**create_structure**(*struct*, *opts*, *prefix=None*)

Manifests/reifies a directory structure in the filesystem

Parameters

- **struct** – directory structure as dictionary of dictionaries
- **opts** – options of the project
- **prefix** – prefix path for the structure

Returns Directory structure as dictionary of dictionaries (similar to input, but only containing the files that actually changed) and input options

Raises `TypeError` – raised if content type in struct is unknown

Changed in version 4.0: Also accepts `string.Template` and `callable` objects as file contents.

pyscaffold.structure.**define_structure**(*struct*, *opts*)

Creates the project structure as dictionary of dictionaries

Parameters

- **struct** – previous directory structure (usually and empty dict)
- **opts** – options of the project

Returns Project structure and PyScaffold's options

Changed in version 4.0: `string.Template` and functions added directly to the file structure.

pyscaffold.structure.**ensure**(*struct*, *path*, *content=None*, *file_op=<function create>*)

Ensure a file exists in the representation of the project tree with the provided content. All the parent directories are automatically created.

Parameters

- **struct** – project representation as (possibly) nested.
- **path** – path-like string or object relative to the structure root. See *modify*.

  Changed in version 4.0: The function no longer accepts a list of strings of path parts.
- **content** – file text contents, `None` by default. The old content is preserved if `None`.
- **file_op** – see *pyscaffold.operations*, `create` by default.

Changed in version 4.0: Instead of a `update_rule` flag, the function now accepts a `file_op`.

Returns Updated project tree representation

Note: Use an empty string as content to ensure a file is created empty.

`pyscaffold.structure.`**`merge`**(*old*, *new*)

Merge two dict representations for the directory structure.

Basically a deep dictionary merge, except from the leaf update method.

Parameters

- **old** – directory descriptor that takes low precedence during the merge.
- **new** – directory descriptor that takes high precedence during the merge.

Changed in version 4.0: Project structure now considers everything **under** the top level project folder.

Returns Resulting merged directory representation

Note: Use an empty string as content to ensure a file is created empty. (`None` contents will not be created).

`pyscaffold.structure.`**`modify`**(*struct*, *path*, *modifier*)

Modify the contents of a file in the representation of the project tree.

If the given path does not exist, the parent directories are automatically created.

Parameters

- **struct** – project representation as (possibly) nested dict. See *merge*.
- **path** – path-like string or object relative to the structure root. The following examples are equivalent:

```
from pathlib import Path

'docs/api/index.html'
Path('docs', 'api', 'index.html')
```

  Changed in version 4.0: The function no longer accepts a list of strings of path parts.

- **modifier** – function (or callable object) that receives the old content and the old file operation as arguments and returns a tuple with the new content and new file operation. Note that, if the file does not exist in `struct`, `None` will be passed as argument. Example:

```
modifier = lambda old, op: ((old or '') + 'APPENDED CONTENT'!, op)
modifier = lambda old, op: ('PREPENDED CONTENT!' + (old or ''), op)
```

  Changed in version 4.0: `modifier` requires 2 arguments and now is a mandatory argument.

Changed in version 4.0: `update_rule` is no longer an argument. Instead the arity `modifier` was changed to accept 2 arguments instead of only 1. This is more suitable to handling the new *pyscaffold.operations* API.

Returns Updated project tree representation

Note: Use an empty string as content to ensure a file is created empty (`None` contents will not be created).

---

pyscaffold.structure.**reify_content**(*content*, *opts*)
Make content string (via __call__ or safe_substitute with opts if necessary)

pyscaffold.structure.**reify_leaf**(*contents*, *opts*)
Similar to *resolve_leaf* but applies *reify_content* to the first element of the returned tuple.

pyscaffold.structure.**reject**(*struct*, *path*)
Remove a file from the project tree representation if existent.

Parameters

- **struct** – project representation as (possibly) nested.
- **path** – path-like string or object relative to the structure root. See *modify*.

  Changed in version 4.0: The function no longer accepts a list of strings of path parts.

Returns Modified project tree representation

pyscaffold.structure.**resolve_leaf**(*contents*)
Normalize project structure leaf to be a Tuple[AbstractContent, FileOp]

12.1.16 pyscaffold.termui module

Basic support for ANSI code formatting.

pyscaffold.termui.**STYLES = {'black': 30, 'blue': 34, 'bold': 1, 'clear': 0, 'cyan': 36, 'green': 32, 'magenta': 35, 'on_black': 40, 'on_blue': 44, 'on_cyan': 46, 'on_green': 42, 'on_magenta': 45, 'on_red': 41, 'on_white': 47, 'on_yellow': 43, 'red': 31, 'white': 37, 'yellow': 33}**
Possible `styles` for *decorate*

pyscaffold.termui.**SYSTEM_SUPPORTS_COLOR = True**
Handy indicator of the system capabilities (relies on colorama if available)

pyscaffold.termui.**curses_available**()
Check if the curses package from stdlib is available.

Usually not available for windows, but its presence indicates that the terminal is capable of displaying some UI.

Returns result of check

Return type bool

pyscaffold.termui.**decorate**(*msg*, **styles*)
Use ANSI codes to format the message.

Parameters

- **msg** (*str*) – string to be formatted
- ***styles** (*list*) – the remaining arguments should be strings that represent the 8 basic ANSI colors. `clear` and `bold` are also supported. For background colors use `on_<color>`.

Returns styled and formatted message

Return type str

pyscaffold.termui.**init_colorama**()
Initialize colorama if it is available.

Returns result of check

Return type bool

pyscaffold.termui.**isatty**(*stream=None*)
Detect if the given stream/stdout is part of an interactive terminal.

Parameters **stream** – optionally the stream to check

Returns result of check

Return type bool

pyscaffold.termui.**supports_color**(*stream=None*)
Check if the stream is supposed to handle coloring.

Returns result of check

Return type bool

12.1.17 pyscaffold.toml module

Thin wrapper around the dependency so we can maintain some stability while being able to swap implementations (e.g. replace tomlkit with toml).

Despite being used in pep517, toml offers limited support for comments, so we prefer tomlkit.

pyscaffold.toml.**TOMLMapping**
Abstraction on the value returned by *loads*.

This kind of object ideally should present a dict-like interface and be able to preserve the formatting and comments of the original TOML file.

alias of MutableMapping

pyscaffold.toml.**dumps**(*obj*)
Serialize a dict-like object into a TOML str, If the object was generated via *loads*, then the style will be preserved.

pyscaffold.toml.**loads**(*text*)
Parse a string containing TOML into a dict-like object, preserving style somehow.

pyscaffold.toml.**setdefault**(*obj*, *key*, *value*)
tomlkit seems to be tricky to use together with setdefault, this function is a workaroud for that.

When `key` is string containing `'.'`, it will perform a nested setdefault.

12.1.18 pyscaffold.update module

Functionality to update one PyScaffold version to another

pyscaffold.update.**ALWAYS = VersionUpdate.ALWAYS**
Perform the update action regardless of the version

pyscaffold.update.**add_dependencies**(*setupcfg*, *opts*)
Add dependencies

pyscaffold.update.**add_entrypoints**(*setupcfg*, *opts*)
Add [options.entry_points] to setup.cfg

pyscaffold.update.**handover_setup_requires**(*setupcfg*, *opts*)
When paired with *update_pyproject_toml*, this will transfer `setup.cfg :: options.setup_requires` to `pyproject.toml :: build-system.requires`

pyscaffold.update.**replace_find_with_find_namespace**(*setupcfg*, *opts*)

pyscaffold.update.**update_pyproject_toml**(*struct*, *opts*)
Update old pyproject.toml generated by pyscaffoldext-pyproject and import *setup_requires* from *update_setup_cfg* into *build-system.requires*.

pyscaffold.update.**update_setup_cfg**(*setupcfg*, *opts*)
Update *pyscaffold* in setupcfg and ensure some values are there as expected

pyscaffold.update.**version_migration**(*struct*, *opts*)
Update projects that were generated with old versions of PyScaffold

12.1.19 Module contents

CHAPTER

THIRTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W