PyScaffold Documentation

Release 3.3.1.post0.dev1+g34cba4b

Blue Yonder

Jan 18, 2021
Contents

1 Features 3
1.1 Configuration, Packaging & Distribution 3
1.2 Versioning and Git Integration 4
1.3 Sphinx Documentation 5
1.4 Dependency Management in a Breeze 6
1.5 Unittest & Coverage 6
1.6 Management of Requirements & Licenses 7
1.7 Extensions 7
1.8 Easy Updating 8

2 Installation 9
2.1 Requirements 9
2.2 Installation 9
2.3 Additional Requirements 10

3 Examples 11

4 Configuration 13

5 Dependency Management 17
5.1 Test Dependencies 18
5.2 Development Environment 18

6 Migration to PyScaffold 21

7 Extending PyScaffold 23
7.1 Project Structure Representation 23
7.2 Scaffold Actions 24
7.3 What are Extensions? 25
7.4 Creating an Extension 25
7.5 Examples 29
7.6 Conventions for Community Extensions 41
7.7 Final Considerations 42

8 Embedding PyScaffold 43

9 Cookiecutter templates with PyScaffold 45
9.1 Suitable templates 45
PyScaffold helps you setup a new Python project. It is as easy as:

```
putup my_project
```

This will create a new folder called `my_project` containing a perfect project template with everything you need for some serious coding. After the usual:

```
python setup.py develop
```

you are all set and ready to go which means in a Python shell you can do the following:

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

Type `putup -h` to learn about more configuration options. PyScaffold assumes that you have Git installed and set up on your PC, meaning at least your name and email are configured. The project template in `my_project` provides you with a lot of features. PyScaffold 3 is compatible with Python 3.4 and greater. For legacy Python 2.7 support please install PyScaffold 2.5. There is also a video tutorial on how to develop a command-line application with the help of PyScaffold.

Note: This document refers to the version v3.3 of PyScaffold. If you are looking for the latest development version (v4), please check pyscaffold.org/en/latest.
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.

### 1.1 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.

In order to build a source, binary or wheel distribution, just run `python setup.py sdist`, `python setup.py bdist` or `python setup.py bdist_wheel` *(recommended)*.

#### 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 `twine` to upload it to PyPI, e.g.:

```
pip install twine
twine upload dist/`
```

For this to work, you have to first register a PyPI account. If you just want to test, please be kind and use TestPyPI before uploading to PyPI.

Please also note that PyPI does not allow uploading local versions 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:** Be aware that the usage of `python setup.py upload` for PyPI uploads also works but is nowadays strongly discouraged and even some of the new PyPI features won’t work correctly if you don’t use `twine`.
Namespace Packages

Optionally, namespace packages can be used, if you are planning to distribute a larger package as a collection of smaller ones. For example, use:

```bash
cputup my_project --package my_package --namespace com.my_domain
```

to define `my_package` inside the namespace `com.my_domain` in java-style.

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 (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:

```python
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`:

```python
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 we only specify the file `data.txt`, no path is allowed. The library `importlib.resources` provides a backport of this feature. Even another way, provided by setuptools’s `pkg_resources` is:

```python
from pkg_resources import resource_string
data = resource_string(__name__, 'path/to/my/data/relative/to/module.txt')
```

Yes, actually “there should be one– and preferably only one –obvious way to do it.”;-)

Please have in mind that the `include_package_data` option in `setup.cfg` is only guaranteed to be read when creating wheels. 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`.

**Warning:** 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.

1.2 Versioning and Git Integration

Your project is already an initialised Git repository and `setup.py` 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 PEP440-compliant version. This version will be used when building a package and is also accessible through `my_project`.
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 `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 `python setup.py install` (or `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 `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 practise for distributing my package? First of all, cloning your repository or just copying 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:

  ```bash
  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`.

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.

1.3 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. The documentation also works with Read the Docs.

The Numpy and Google style docstrings are activated by default. Just make sure Sphinx 1.3 or above is installed.
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`. Alternatively, if your project was created with the `--tox` option, simply run `tox -e docs` or `tox -e doctests`.

### 1.4 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 to avoid *dependency hell* during both development and production stages.

In particular, 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 (details in *Dependency Management*).

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

### 1.5 Unittest & Coverage

PyScaffold relies on `py.test` to run all unittests defined in the subfolder `tests`. Some sane default flags for `py.test` are already defined in the `[pytest]` section of `setup.cfg`. The `py.test` 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:

```
py.test -h
```

to show the help of `py.test` (requires `py.test` to be installed in your system or virtualenv).

**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 `--travis` to generate templates of the Travis configuration files `.travis.yml` and `tests/travis_install.sh` which even features the coverage and stats system Coveralls. In order to use the virtualenv management and test tool tox the flag `--tox` can be specified. If you are using GitLab you can get a default `.gitlab-ci.yml` also running `pytest-cov` with the flag `--gitlab`.

**Managing test environments with tox**

Run `tox` to generate test virtual environments for various python environments defined in the generated `tox.ini`. Testing and building `sdists` for python 2.7 and python 3.4 is just as simple with `tox` as:

```
tox -e py27,py34
```

Environments for tests with the the static code analyzers pyflakes and pep8 which are bundled in `flake8` are included as well. Run it explicitly with:

```
tox -e flake8
```
With tox, you can use the --recreate flag to force tox to create new environments. By default, PyScaffold’s tox configuration will execute tests for a variety of python versions. If an environment is not available on the system the tests are skipped gracefully. You can rely on the tox documentation for detailed configuration options.

# 1.6 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`, or maximum versions, e.g. `pandas<0.12`, are used sometimes.

More specific installation requirements should go into requirements.txt. This file can also be managed with the help of `pip compile` from pip-tools that basically pins packages to the current version, e.g. `numpy==1.13.1`. The packages defined in requirements.txt can be easily installed with:

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

All licenses from choosealicense.com can be easily selected with the help of the --license flag.

# 1.7 Extensions

PyScaffold comes with several extensions:

- If you want a project setup for a *Data Science* task, just use `--dsproject` after having installed pyscaffoldext-dsproject.
- Create a *Django* project with the flag `--django` which is equivalent to `django-admin startproject my_project` enhanced by PyScaffold’s features.
- Create a template for your own PyScaffold extension with `--custom-extension` after having installed pyscaffoldext-custom-extension with pip.
- Have a README.md based on MarkDown instead of README.rst by using `--markdown` after having installed pyscaffoldext-markdown with pip.
- Add a pyproject.toml file according to PEP 518 to your template by using `--pyproject` after having installed pyscaffoldext-pyproject with pip.
- With the help of Cookiecutter it is possible to further customize your project setup with a template tailored for PyScaffold. Just use the flag `--cookiecutter TEMPLATE` to use a cookiecutter template which will be refined by PyScaffold afterwards.
- ... and many more like `--gitlab` to create the necessary files for GitLab.

There is also documentation about *writing extensions*. Find more extensions within the PyScaffold organisation and consider contributing your own. All extensions can easily be installed with `pip install pyscaffoldext-NAME`.

---

**Warning:** *Deprecation Notice* - In the next major release both Cookiecutter and Django extensions will be extracted into independent packages. After PyScaffold v4.0, you will need to explicitly install pyscaffoldext-cookiecutter and pyscaffoldext-django in your system/virtualenv in order to be able to use them.
1.8 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! Also check out if configuration options in `setup.cfg` have changed.

1.8.1 Updates from PyScaffold 2

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.

1.8.2 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 `.travis.yml` was created with `putup my_project` it can later be added by issuing `putup --update my_project --travis`. 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.
2.1 Requirements

The installation of PyScaffold only requires a recent version of setuptools, i.e. at least version 38.3, as well as a working installation of Git. 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`.

Additionally, if you want to create a Django project or want to use cookiecutter:

- Django
- Cookiecutter

**Note:** It is recommended to use an isolated environment as provided by `virtualenv` or even better `Anaconda` for your work with Python in general.

2.2 Installation

Make sure you have `pip` installed, 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 extensions, run:

```
pip install --upgrade pyscaffold[all]
```
PyScaffold is also available at conda-forge and thus can be installed with conda:

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

## 2.3 Additional Requirements

If you run commands like `py.test` and `make -C docs` within your project, some additional requirements like `py.test` and Sphinx may be required. 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 inside your virtual environment. In order to avoid this just install following packages inside your virtual environment:

- Sphinx
- py.test
- pytest-cov

Alternatively, you can setup build automation with `tox`. An easy way to do that is to generate your project passing the `--tox` option. The commands `tox` and `tox -e docs` should be able to run your tests or build your docs out of the box.
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 skynet -l gpl3 -d "Finally, the ultimate AI!" -u http://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 `http://sky.net`.

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

**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`. 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 --tox --travis**  This will create a project and package `thoroughly_tested` with files `tox.ini` and `.travis.yml` for `Tox` and `Travis`.

**putup my_zope_subpackage --namespace zope -l gpl3**  This will create a project and subpackage named `my_zope_subpackage` in the namespace `zope`. 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.
Projects set up with PyScaffold feature an easy package configuration with `setup.cfg`. Check out the example below as well as the documentation of `setuptools`.

```
[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
    Programming Language :: Python :: 3.4
    Programming Language :: Python :: 3.5
    Programming Language :: Python :: 3.6
    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:
    include_package_data = True
package_dir =
    =src
```

(continues on next page)
setup_requires = pyscaffold>=3.1a0,<3.2a0
# Add here dependencies of your project (semicolon/line-separated)
install_requires =
    pandas
    scikit-learn

[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
testing-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

[test]
# py.test options when running `python setup.py test`
addopts = --verbose
extras = True

[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 pyscaffold --cov-report term-missing
    --verbose
    norecursedirs =
    dist
    build
    .tox
testpaths = tests

[aliases]
release = sdist bdist_wheel upload

[bdist_wheel]
universal = 1
[build_sphinx]
# Options for Sphinx build
source_dir = docs
build_dir = docs/_build

[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
    bdist_wheel

[flake8]
# Some sane defaults for the code style checker flake8
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 = 3.0
package = my_package
extensions =
    namespace
namespace = ns1.ns2
CHAPTER 5

Dependency Management

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

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](https://www.m吟iss.com/abstract-dependencies/).

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.
5.1 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 run `py.test` runner, you will have to install those dependencies manually, or do a editable install of your package with `pip install -e .[testing]`.

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

**Note:** 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 :)

**Warning:** PyScaffold strongly advocates the use of test runners to guarantee your project is correctly packaged/works in isolated environments. A good start is to create a new project passing the `--tox` option to `putup` and try running `tox` in your project root.

5.2 Development Environment

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, someone could do:

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

However, someone could argue that this process is pretty manual and laborious to maintain specially when the developer changes the abstract dependencies. Moreover, it is 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 two popular command line tools, supported by PyScaffold, to tackle these issues.
5.2.1 How to integrate 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.

**Note:** 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:

```bash
$ pip install pyscaffold pipenv
$ putup myproj --tox
$ 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.
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:

   ```shell
   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` with:

   ```shell
   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 `python setup.py install` and `python setup.py sdist`. If those two commands don’t work, check `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? After these
basic commands, try also to run `make -C docs html` and `py.test` (or preferably their `tox` equivalents) to check that Sphinx and PyTest run correctly.
Extending PyScaffold

PyScaffold is carefully designed to cover the essentials of authoring and distributing Python packages. Most of the time, tweaking `putup` options is enough to ensure proper configuration of a project. However, for advanced use cases a deeper level of programmability may be required and PyScaffold’s extension systems provides this.

PyScaffold can be extended at runtime by other Python packages. Therefore it is possible to change the behaviour of the `putup` command line tool without changing the PyScaffold code itself. In order to explain how this mechanism works, the following sections define a few important concepts and present a comprehensive guide about how to create custom extensions.

Additionally, Cookiecutter templates can also be used but writing a native PyScaffold extension is the preferred way.

Note: 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`.

### 7.1 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:

```python
{
    'project': {
        'folder': {
            'file.txt': 'Hello World!',
            'another-folder': {
                'empty-file.txt': '
            }
        }
    }
}
```
represents a project/folder directory in the file system containing 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`. In turn, `another-folder` contains an empty file named `empty-file.txt`.

Additionally, tuple values are also allowed in order to specify some useful metadata. In this case, the first element of the tuple is the file content. For example, the dict:

```python
{
    'project': {
        'namespace': {
            'module.py': ('print("Hello World!")', helpers.NO_OVERWRITE)
        }
    }
}
```

represents a project/namespace/module.py file with content `print("Hello World!")`, that will not be overwritten if already exists.

**Note:** The `NO_OVERWRITE` flag is defined in the module `pyscaffold.api.helpers`.

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

### 7.2 Scaffold Actions

PyScaffold organizes the generation of a project into a series of steps with well defined purposes. 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:

```python
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, and 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.api:get_default_options`
2. `pyscaffold.api:verify_options_consistency`
3. `pyscaffold.structure:define_structure`
4. `pyscaffold.structure:apply_update_rules`
5. `pyscaffold.structure:create_structure`
6. `pyscaffold.api:init_git`
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`.

### 7.3 What are Extensions?

From the standpoint of PyScaffold, an extension is just an class inheriting from `Extension` overriding and implementing certain methods. This methods allow inject actions at arbitrary positions in the aforementioned list. Furthermore, extensions can also remove actions.

### 7.4 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, registers a custom action that will be called later and returns a modified version of the list of actions:

```python
from ..api import Extension
from ..api import helpers

class MyExtension(Extension):
    """Help text on commandline when running putup -h""

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

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

        Returns:
            list: updated list of actions
        ""
        actions = helpers.register(actions, self.action, after="create_structure")
        actions = helpers.unregister(actions, "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
```
Note: The register and unregister methods implemented in the module pyscaffold.api.helpers basically create modified copies of the action list by inserting/removing the specified functions, with some awareness about their execution order.

7.4.1 Action List Helper Methods

As implied by the previous example, the helpers 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:

```
helpers.register(actions, hook1, before='define_structure')
helpers.register(actions, 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:

```
helpers.unregister(actions, 'init_git')
helpers.unregister(actions, 'pyscaffold.api:init_git')
```

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

Note: 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 actions = self.register(actions, self.my_action).

7.4.2 Structure Helper Methods

PyScaffold also provides extra facilities to manipulate the project structure. The following functions are accessible through the helpers 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 metadata 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 its metadata (for example adding NO_OVERWRITE or NO_CREATE flags).

**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:

```python
from pathlib import PurePath
from ..api import Extension
from ..api import helpers

MY_AWESOME_FILE = """\n# -*- coding: utf-8 -*-\n__author__ = "{author}"
__copyright__ = "{author}"
__license__ = "{license}"

def awesome():
    return "Awesome!"
""

MY_AWESOME_TEST = """\nimport 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 helpers.register(actions, self.define_awesome_files)

    def define_awesome_files(self, struct, opts):
        struct = helpers.merge(
            struct,
            {
                "src": {
                    "{package}": {
                        "awesome.py": MY_AWESOME_FILE.format(**opts),
                    },
                },
                "tests": {
                    "awesome_test.py": {
                        MY_AWESOME_TEST.format(**opts),
                        helpers.NO_OVERWRITE,
                    },
                },
            },
        )
```

(continues on next page)
for filename in ["awesome_file1", "awesome_file2"]:
    struct = helpers.ensure(
        struct,
        PurePath(opts["project"], "src", "awesome", filename),
        content="AWESOME!",
        update_rule=helpers.NO_CREATE,
    )
    # The second argument is the file path, represented by a
    # list of file parts or a string.
    # Alternatively in this example:
    # path = '{project}/src/awesome/{filename}'.format(
    #     filename=filename, **opts)

    # The `reject` can be used to avoid default files being generated.
    struct = helpers.reject(
        struct,
        "{project}/src/{package}/skeleton.py".format(**opts)
    )
    # Alternatively in this example:
    # path = [opts['project'], 'src', opts['package'], 'skeleton.py']

    # `modify` can be used to change contents in an existing file
    struct = helpers.modify(
        struct,
        PurePath(opts["project"], "tests", "awesome_test.py"),
        lambda content: "import pdb\n" + content,
    )

    # And/or change the update behavior
    struct = helpers.modify(
        struct,
        [opts["project"], ".travis.yml"], update_rule=helpers.NO_CREATE
    )

    # It is important to remember the return values
    return struct, opts

Note: The project and package options should be used to provide the correct location of the files relative to the current working directory.

As shown by the previous example, the helpers module also presents constants that can be used as metadata. The NO_OVERWRITE flag avoids an existing file to be overwritten when putup is used in update mode. Similarly, NO_CREATE avoids creating a file from template in update mode, even if it does not exist.

For more sophisticated extensions which need to read and parse their own command line arguments it is necessary to override activate 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 as well as the pyproject extension which serves as a blueprint for new extensions.
7.4.3 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
```

7.5 Examples

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

7.5.1 Namespace Extension

```python
# -*- coding: utf-8 -*-

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

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

```python
import argparse
import os
from os.path import isdir
from os.path import join as join_path

from .. import templates, utils
from ..api import Extension, helpers
from ..log import logger

class Namespace(Extension):
    """Add a namespace (container package) to the generated package."""

    def augment_cli(self, parser):
        """Add an option to parser that enables the namespace extension.

        Args:
            parser (argparse.ArgumentParser): CLI parser object
        """
        parser.add_argument(
            self.flag,
            dest=self.name,
            default=None,
            action=create_namespace_parser(self),
```
metavar="NS1[.NS2]",
    help="put your project inside a namespace package",
)

```python
def activate(self, actions):
    """Register an action responsible for adding namespace to the package.
    Args:
        actions (list): list of actions to perform
    Returns:
        list: updated list of actions
    ""
    actions = helpers.register(actions, enforce_namespace_options, after="get_default_options")
    actions = helpers.register(actions, add_namespace, before="apply_update_rules")
    return helpers.register(actions, move_old_package, after="create_structure")
```

def create_namespace_parser(obj_ref):
    """Create a namespace parser.
    Args:
        obj_ref (Extension): object reference to the actual extension
    Returns:
        NamespaceParser: parser for namespace cli argument
    ""
    class NamespaceParser(argparse.Action):
        """Consumes the values provided, but also appends the extension function to the extensions list.
        ""
        def __call__(self, parser, namespace, values, option_string=None):
            namespace.extensions.append(obj_ref)
            namespace.extensions.append(obj_ref)
            # Now the extra parameters can be stored
            setattr(namespace, self.dest, values)
            # save the namespace cli argument for later
            obj_ref.args = values
            return NamespaceParser
```

def enforce_namespace_options(struct, opts):
    """Make sure options reflect the namespace usage.""
    opts.setdefault("namespace", None)
    if opts["namespace"]:
        opts["ns_list"] = utils.prepare_namespace(opts["namespace"])
        opts["root_pkg"] = opts["ns_list"][0]
```python
opts["qual_pkg"] = ".".join([opts["ns_list"][-1], opts["package"]])

return struct, opts

def add_namespace(struct, opts):
    """Prepend the namespace to a given file structure
    Args:
        struct (dict): directory structure as dictionary of dictionaries
        opts (dict): options of the project
    Returns:
        tuple(dict, dict):
            directory structure as dictionary of dictionaries and input options
    """
    if not opts["namespace"]:
        return struct, opts

    namespace = opts["ns_list"][-1].split(".")
    base_struct = struct
    struct = base_struct[opts["project"]]["src"]
    pkg_struct = struct[opts["package"]] [opts["package"]]
    del struct[opts["package"]]
    for sub_package in namespace:
        struct[sub_package] = {"__init__.py": templates.namespace(opts)}
        struct = struct[sub_package]
    struct[opts["package"]] = pkg_struct

    return base_struct, opts

def move_old_package(struct, opts):
    """Move old package that may be eventually created without namespace
    Args:
        struct (dict): directory structure as dictionary of dictionaries
        opts (dict): options of the project
    Returns:
        tuple(dict, dict):
            directory structure as dictionary of dictionaries and input options
    """
    old_path = join_path(opts["project"], "src", opts["package"])
    namespace_path = opts["qual_pkg"].replace(".", os.sep)
    target = join_path(opts["project"], "src", namespace_path)

    old_exists = opts["pretend"] or isdir(old_path)
    # ^ When pretending, pretend also an old folder exists
    # to show a worst case scenario log to the user...
    if old_exists and opts["qual_pkg"] != opts["package"]:
        if not opts["pretend"]: 
            logger.warning( 
                "\nA folder $r exists in the project directory, and it is " 
                "likely to have been generated by a PyScaffold extension or " 
                "manually by one of the current project authors.\n"
```

(continues on next page)
"Moving it to %r, since a namespace option was passed.\n" +
"Please make sure to edit all the files that depend on this " +
"package to ensure the correct location.\n",
opts["package"],
namespace_path,
)

    utils.move(old_path, target=target, log=True, pretend=opts["pretend"])

return struct, opts

7.5.2 No Skeleton Extension

# -*- coding: utf-8 -*-

""
Extension that omits the creation of file `skeleton.py`
""

from pathlib import PurePath as Path
from ..api import Extension, helpers

class NoSkeleton(Extension):
    """Omit creation of skeleton.py and test_skeleton.py""

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

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

        Returns:
            list: updated list of actions
        ""
        return self.register(actions, self.remove_files, after="define_structure")

    def remove_files(self, struct, opts):
        """Remove all skeleton files from structure

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

        Returns:
            struct, opts: updated project representation and options
        ""
        # Namespace is not yet applied so deleting from package is enough
        file = Path(opts["project"], "src", opts["package"], "skeleton.py")
        struct = helpers.reject(struct, file)
        file = Path(opts["project"], "tests", "test_skeleton.py")
        struct = helpers.reject(struct, file)
        return struct, opts
### 7.5.3 Cookiecutter Extension

```python
# -*- coding: utf-8 -*-

"""
Extension that integrates cookiecutter templates into PyScaffold.

Warning:
*Deprecation Notice* - In the next major release the Cookiecutter extension
will be extracted into an independent package.
After PyScaffold v4.0, you will need to explicitly install
`pyscaffoldext-cookiecutter` in your system/virtualenv in order to be
able to use it.
"""

import argparse

from ..api import Extension
from ..api.helpers import logger, register
from ..warnings import UpdateNotSupported

class Cookiecutter(Extension):
    """Additionally apply a Cookiecutter template"""
    mutually_exclusive = True

    def augment_cli(self, parser):
        """Add an option to parser that enables the Cookiecutter extension

        Args:
            parser (argparse.ArgumentParser): CLI parser object
        """
        parser.add_argument(
            self.flag,
            dest=self.name,
            action=create_cookiecutter_parser(self),
            metavar="TEMPLATE",
            help="additionally apply a Cookiecutter template. 
            "Note that not all templates are suitable for PyScaffold. 
            "Please refer to the docs for more information."
        )

    def activate(self, actions):
        """Register before_create hooks to generate project using Cookiecutter

        Args:
            actions (list): list of actions to perform
        """
        # `get_default_options` uses passed options to compute derived ones,
        # so it is better to prepend actions that modify options.
        actions = register(
            actions, enforce_cookiecutter_options, before="get_default_options"
        )
```

(continues on next page)
# `apply_update_rules` uses CWD information, so it is better to prepend actions that modify it.

```python
actions = register(actions, create_cookiecutter, before="apply_update_rules")

return actions
```

def create_cookiecutter_parser(obj_ref):
    """Create a Cookiecutter parser.

    Args:
        obj_ref (Extension): object reference to the actual extension

    Returns:
        NamespaceParser: parser for namespace cli argument
    """

class CookiecutterParser(argparse.Action):
    """Consumes the values provided, but also append the extension function to the extensions list.
    """

def __call__(self, parser, namespace, values, option_string=None):
    # First ensure the extension function is stored inside the
    # 'extensions' attribute:
    extensions = getattr(namespace, "extensions", [])
    extensions.append(obj_ref)
    namespace.extensions = extensions

    # Now the extra parameters can be stored
    setattr(namespace, self.dest, values)

    # save the cookiecutter cli argument for later
    obj_ref.args = values

    return CookiecutterParser

def enforce_cookiecutter_options(struct, opts):
    """Make sure options reflect the cookiecutter usage.

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

    Returns:
        struct, opts: updated project representation and options
    """
    opts["force"] = True

    return struct, opts

def create_cookiecutter(struct, opts):
    """Create a cookie cutter template

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

    Returns:
        struct, opts: updated project representation and options
    """
    return struct, opts
Args:

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

Returns:

- `struct, opts`: updated project representation and options

```python
if opts.get("update"):
    logger.warning(UpdateNotSupported(extension="cookiecutter"))
    return struct, opts

try:
    from cookiecutter.main import cookiecutter
except Exception as e:
    raise NotInstalled from e

extra_context = dict(
    full_name=opts["author"],
    author=opts["author"],
    email=opts["email"],
    project_name=opts["project"],
    package_name=opts["package"],
    repo_name=opts["package"],
    project_short_description=opts["description"],
    release_date=opts["release_date"],
    version="unknown",  # will be replaced later
    year=opts["year"],
)

if "cookiecutter" not in opts:
    raise MissingTemplate

logger.report("run", "cookiecutter " + opts["cookiecutter"])
if not opts.get("pretend"):
    cookiecutter(opts["cookiecutter"], no_input=True, extra_context=extra_context)

return struct, opts
```

class NotInstalled(RuntimeError):
    """This extension depends on the `cookiecutter` package."""

    DEFAULT_MESSAGE = "cookiecutter is not installed, run: pip install cookiecutter"

    def __init__(self, message=DEFAULT_MESSAGE, *args, **kwargs):
        super(NotInstalled, self).__init__(message, *args, **kwargs)

class MissingTemplate(RuntimeError):
    """A cookiecutter template (git url) is required."""

    DEFAULT_MESSAGE = "missing `cookiecutter` option"
def __init__(self, message=DEFAULT_MESSAGE, *args, **kwargs):
    super(MissingTemplate, self).__init__(message, *args, **kwargs)

7.5.4 Django Extension

# -*- coding: utf-8 -*-

""" Extension that creates a base structure for the project using django-admin. """

Warning:
*Deprecation Notice* - In the next major release the Django extension will be extracted into an independent package.
After PyScaffold v4.0, you will need to explicitly install `pyscaffoldext-django` in your system/virtualenv in order to be able to use it.

"""
import os
import shutil
from os.path import join as join_path

from .. import shell
from ..api import Extension, helpers
from ..warnings import UpdateNotSupported

class Django(Extension):
    """Generate Django project files"""
    mutually_exclusive = True

    def activate(self, actions):
        """Register hooks to generate project using django-admin.

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

        Returns:
            list: updated list of actions
        """
        # get_default_options uses passed options to compute derived ones,
        # so it is better to prepend actions that modify options.
        actions = helpers.register(
            actions, enforce_django_options, before="get_default_options"
        )
        # apply_update_rules uses CWD information,
        # so it is better to prepend actions that modify it.
        actions = helpers.register(
            actions, create_django_proj, before="apply_update_rules"
        )

        return actions

    def enforce_django_options(struct, opts):

(continues on next page)
Make sure options reflect the Django usage.

Args:

struct (dict): project representation as (possibly) nested :obj:`dict`

opts (dict): given options, see :obj:`create_project` for an extensive list.

Returns:

struct, opts: updated project representation and options

opts["package"] = opts["project"]  # required by Django
opts["force"] = True
opts.setdefault("requirements", []).append("django")

return struct, opts

def create_django_proj(struct, opts):
    
    Creates a standard Django project with django-admin

Args:

struct (dict): project representation as (possibly) nested :obj:`dict`

opts (dict): given options, see :obj:`create_project` for an extensive list.

Returns:

struct, opts: updated project representation and options

Raises:

:obj:`RuntimeError`: raised if django-admin is not installed

if opts.get("update"):
    helpers.logger.warning(UpdateNotSupported(extension="django"))
    return struct, opts

try:
    shell.django_admin("--version")
except Exception as e:
    raise DjangoAdminNotInstalled from e

pretend = opts.get("pretend")
shell.django_admin("startproject", opts["project"], log=True, pretend=pretend)
if not pretend:
    src_dir = join_path(opts["project"], "src")
    os.mkdir(src_dir)
    shutil.move(
        join_path(opts["project"], opts["project"]),
        join_path(src_dir, opts["package"]),
    )

return struct, opts

class DjangoAdminNotInstalled(RuntimeError):
    
    This extension depends on the `django-admin` cli script.""
DEFAULT_MESSAGE = "django-admin is not installed, " "run: pip install django"

```python
def __init__(self, message=DEFAULT_MESSAGE, *args, **kwargs):
    super(DjangoAdminNotInstalled, self).__init__(message, *args, **kwargs)
```

7.5.5 Pre Commit Extension

```python
# -*- coding: utf-8 -*-

"""
Extension that generates configuration files for Yelp `pre-commit`

.. _pre-commit: http://pre-commit.com
"""

from ..api import Extension, helpers
from ..log import logger
from ..templates import isort_cfg, pre_commit_config

class PreCommit(Extension):
    """Generate pre-commit configuration file"""

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

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

        Returns:
            list: updated list of actions
        """
        return self.register(actions, self.add_files, after="define_structure") + [
            self.instruct_user
        ]

    @staticmethod
def 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)

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

        Returns:
            struct, opts: updated project representation and options
        """
        files = {
            ".pre-commit-config.yaml": (pre_commit_config(opts), helpers.NO_  
            ...OVERWRITE),
            
```
```
7.5.6 Tox Extension

```python
# -*- coding: utf-8 -*-

from ..api import Extension, helpers
from ..templates import tox as tox_ini

class Tox(Extension):
    
    def activate(self, actions):
        
        def add_files(self, struct, opts):
            
            return struct, opts
```

7.5. Examples
opts (dict): given options, see :obj:`create_project` for an extensive list.

Returns:
struct, opts: updated project representation and options

files = {"tox.ini": (tox_ini(opts), helpers.NO_OVERWRITE)}

return helpers.merge(struct, {opts["project"]: files}), opts

7.5.7 Travis Extension

```python
# -*- coding: utf-8 -*-

""" Extension that generates configuration and script files for Travis CI. """

from ..api import Extension, helpers
from ..templates import travis, travis_install

class Travis(Extension):
    """Generate Travis CI configuration files"""

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

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

    Returns:
        list: updated list of actions
    """
    return self.register(actions, self.add_files, after="define_structure")

def add_files(self, struct, opts):
    """Add some Travis files to structure"

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

    Returns:
        struct, opts: updated project representation and options
    """
    files = {
        ".travis.yml": (travis(opts), helpers.NO_OVERWRITE),
        "tests": {
            "travis_install.sh": (travis_install(opts), helpers.NO_OVERWRITE)
        },
    }

    return helpers.merge(struct, {opts["project"]: files}), opts
```
### 7.5.8 GitLab-CI Extension

```python
# -*- coding: utf-8 -*-
"
Extension that generates configuration and script files for GitLab CI.
"

from ..api import Extension, helpers
from ..templates import gitlab_ci

class GitLab(Extension):
    """Generate GitLab CI configuration files""

    def activate(self, actions):
        """Activate extension"
        Args:
            actions (list): list of actions to perform
        Returns:
            list: updated list of actions
        """
        return self.register(actions, self.add_files, after="define_structure")

    def add_files(self, struct, opts):
        """Add .gitlab-ci.yml file to structure"
        Args:
            struct (dict): project representation as (possibly) nested :obj:`dict`
            opts (dict): given options, see :obj:`create_project` for an extensive list.
        Returns:
            struct, opts: updated project representation and options
        """
        files = {".gitlab-ci.yml": (gitlab_ci(opts), helpers.NO_OVERWRITE)}

        return helpers.merge(struct, {opts["project"]: files}), opts
```

### 7.6 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:
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 (\texttt{--awesome-files}).

### 7.7 Final Considerations

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 helpers module provides a special logger object useful in these situations. Please refer to Cookiecutter Extension 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.
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 is exposed by the `pyscaffold.api` module, which contains the main function `create_project`. The following example illustrates a typical embedded usage of PyScaffold:

```python
import logging

from pyscaffold.api import create_project
from pyscaffold.extensions.tox import Tox
from pyscaffold.extensions.travis import Travis
from pyscaffold.extensions.namespace import Namespace
from pyscaffold.log import DEFAULT_LOGGER as LOGGER_NAME

logging.getLogger(LOGGER_NAME).setLevel(logging.INFO)
create_project(
    project="my-proj-name",
    author="Your Name",
    namespace="some.namespace",
    license="mit",
    extensions=[Tox("tox"), Travis("travis"), Namespace("namespace")],
)
```

Note that no built-in extension (e.g. tox, travis and namespace support) 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.
Cookiecutter templates with PyScaffold

Cookiecutter is a flexible utility that allows the definition of templates for a diverse range of software projects. On the other hand, PyScaffold is focused in a good out-of-the-box experience for developing distributable Python packages (exclusively). Despite the different objectives, it is possible to combine the power of both tools to create a custom Python project setup. For instance, the following command creates a new package named mypkg, that uses a Cookiecutter template, but is enhanced by PyScaffold's features:

```
$ putup mypkg --cookiecutter gh:audreyr/cookiecutter-pypackage
```

This is roughly equivalent to first create a project using the Cookiecutter template and convert it to PyScaffold afterwards:

```
$ cookiecutter --no-input gh:audreyr/cookiecutter-pypackage project_name=mypkg
$ putup mypkg --force
```

**Note:** For complex Cookiecutter templates calling `cookiecutter` and `putup` separately may be a better option, since it is possible to answer specific template questions or at least set values for Cookiecutter variables.

**Warning:** Although using Cookiecutter templates is a viable solution to customize a project that was set up with PyScaffold, the recommended way is to help improve PyScaffold by contributing an extension.

### 9.1 Suitable templates

Note that PyScaffold will overwrite some files generated by Cookiecutter, like `setup.py`, the `__init__.py` file under the package folder and most of the `docs` folder, in order to provide `setuptools_scm` and `sphinx` integration. Therefore not all Cookiecutter templates are suitable for this approach.

Ideally, interoperable templates should focus on the file structure inside the `src` folder instead of packaging or distributing, since PyScaffold already handles it under-the-hood. This also means that your template should adhere to the
src-layout if you want to generate files within your Python package.

In addition, PyScaffold runs Cookiecutter with the `--no-input` flag activated and thus the user is not prompted for manual configuration. Instead, PyScaffold injects the following parameters:

```plaintext
author
e-mail
project_name
description
package_name
project_short_description
```

Accordingly, the template file structure should be similar to:

```plaintext
cookiecutter-something/
  └── {{cookiecutter.project_name}}/
      └── src/
          └── {{cookiecutter.package_name}}/
              ... 
```

See Cookiecutter for more information about template creation.
PyScaffold was started by Blue Yonder developers to help automating and standardizing the process of project setups. Nowadays it is a pure community project and you are very welcome to join in our effort if you would like to contribute.

### 10.1 Issue Reports

If you experience bugs or in general issues with PyScaffold, please file an issue report on our issue tracker.

### 10.2 Code Contributions

#### 10.2.1 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.

#### 10.2.2 Create an environment

Before you start coding we recommend to install Miniconda which allows to setup a dedicated development environment named `pyscaffold` with:

```
conda create -n pyscaffold python=3 six virtualenv pytest pytest-cov
```

Then activate the environment `pyscaffold` with:

```
source activate pyscaffold
```
10.2.3 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
   ```

4. Run `python setup.py egg_info --egg-base` after a fresh checkout. This will generate some critically needed files. Typically after that, you should run `python setup.py develop` to be able run `putup`.
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.
6. Create a branch to hold your changes:

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

   and start making changes. Never work on the master branch!
7. Start your work on this branch. When you’re done editing, do:

   ```
   git add modified_files
   git commit
   ```

   to record your changes in Git, then push them to GitHub with:

   ```
   git push -u origin my-feature
   ```
8. Please check that your changes don’t break any unit tests with:

   ```
   python setup.py test
   ```

   or even a more thorough test with `tox` after having installed `tox` with `pip install tox`. Don’t forget to also add unit tests in case your contribution adds an additional feature and is not just a bugfix.

   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 PROCESS>` 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.
9. Use `flake8` to check your code style.
10. Add yourself to the list of contributors in `AUTHORS.rst`.
11. 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 creating a PR.
12. If you are submitting a change related to an existing continuous integration (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.

10.3 Release

As a PyScaffold maintainer following steps are needed 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. Clean up the `dist` and `build` folders with `rm -rf dist build` to avoid confusion with old builds and Sphinx docs.

4. Run `python setup.py dists` 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.

5. Run `twine upload dist/*` and check that everything was uploaded to PyPI correctly.

10.4 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.
Frequently Asked Questions

In case you have a general question that is not answered here, consider submitting a new issue.

1. 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.

2. Does my project depend on PyScaffold when I use it to set my project up?

The short answer is no if you later distribute your project in the recommended wheel format. The longer answer is that only during development PyScaffold is needed as a setup dependency. That means if someone clones your repository and runs setup.py, setuptools checks for the setup_requires argument which includes PyScaffold and installs PyScaffold automatically as egg file into .eggs if PyScaffold is not yet installed. This mechanism is provided by setuptools and definitely beyond the scope of this answer. The same applies for the deprecated source distribution (sdist) but not for a binary distribution (bdist). Anyways, the recommend way is nowadays a binary wheel distribution (bdist_wheel) which will not depend on PyScaffold at all.

3. 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!”).

4. **Can I use PyScaffold 3 to develop a Python package that is Python 2 & 3 compatible?**

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.

5. **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. To answer the question, it’s actually really simple. Within setup.py just remove the use_pyscaffold argument from the setup() call which will deactivate all of PyScaffold’s functionality that goes beyond what is provided by setuptools. 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,
- `python setup.py test` and `python setup.py doctest`, just use `py.test` directly,
- `python setup.py docs` for building your Sphinx documentation, just enter the docs folder and type `make html` 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’s `conf.py`,
- `.gitignore` with some nice defaults and other dot files depending on the flags used when running putup,
- some sane defaults for py.test.

For further cleanups, feel free to remove pyscaffold from the setup_requires key in setup.cfg as well as the complete [pyscaffold] section.
The MIT License (MIT)

Copyright (c) 2014 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.
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
14.1 Development version

14.1.1 Version 4.0

- Support for Python 3.5 dropped, issue #226

14.2 Current versions

14.2.1 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

14.2.2 Version 3.2.3, 2019-10-12

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

14.3 Older versions

14.3.1 Version 3.2.2, 2019-09-12

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

14.3.2 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

14.3.3 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`
14.3.4 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 #160 & #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

14.3.5 Version 2.5.11, 2018-04-14

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

14.3.6 Version 3.0.3, 2018-04-14

- Set `install_requires` to `setuptools>=30.3.0`

14.3.7 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

14.3.8 Version 2.5.10, 2018-03-21

- Updated `setuptools_scm` to version 1.17.0

14.3. Older versions
14.3.9 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

14.3.10 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

14.3.11 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.
14.3.12 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

14.3.13 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

14.3.14 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

14.3.15 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

14.3.16 Version 2.5.4, 2016-02-10

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

14.3.17 Version 2.5.3, 2016-01-16

- Fix classifier metadata (`classifiers` to `classifier` in `setup.cfg`)
14.3.18 Version 2.5.2, 2016-01-02

- Fix is_git_installed

14.3.19 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

14.3.20 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 #72
- Cool logo was designed by Eva Schmücker, issue #66

14.3.21 Version 2.4.4, 2015-10-29

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

14.3.22 Version 2.4.3, 2015-10-27

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

14.3.23 Version 2.4.2, 2015-09-16

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

14.3.24 Version 2.4.1, 2015-09-09

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

14.3.25 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.

### 14.3.26 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.

### 14.3.27 Version 2.2.1, 2015-06-18

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

### 14.3.28 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/confstest.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

14.3. Older versions
14.3.29 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

14.3.30 Version 2.0.4, 2015-03-17

- Typo and wrong Sphinx usage in the RTD documentation

14.3.31 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

14.3.32 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)

14.3.33 Version 2.0.1, 2015-02-27

- FIX: Installation bug under Windows 7

14.3.34 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

14.3.35 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

14.3.36 Version 1.3.2, 2014-12-02

• Fix: path of Travis install script

14.3.37 Version 1.3.1, 2014-11-24

• Fix: –with-tox tuple bug #28

14.3.38 Version 1.3, 2014-11-17

• Support for Tox (https://tox.readthedocs.org/)
• 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


• Support pre-commit hooks (http://pre-commit.com/)

14.3.40 Version 1.1, 2014-09-29

• Changed COPYING to LICENSE
• Support for all licenses from http://choosealicense.com/
• Fix: Allow update of license again
• Update to Versioneer 0.12
14.3.41 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

14.3.42 Version 0.9, 2014-07-27

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

14.3.43 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`

14.3.44 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

14.3.45 Version 0.6.1, 2014-05-15

- Fix: Removed six dependency in `setup.py`
14.3.46 Version 0.6, 2014-05-14

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

14.3.47 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

14.3.48 Version 0.4.1, 2014-04-27

- Problem fixed with pytest-cov installation

14.3.49 Version 0.4, 2014-04-23

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

14.3.50 Version 0.3, 2014-04-18

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

14.3.51 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

14.3. Older versions
14.3.52 Version 0.1.2, 2014-04-10

- Bugfix in Manifest.in
- Python 2.6 problems fixed

14.3.53 Version 0.1.1, 2014-04-10

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

14.3.54 Version 0.1, 2014-04-03

- First release
15.1 pyscaffold package

15.1.1 Subpackages

pyscaffold.api package

Submodules

pyscaffold.api.helpers module

Useful functions for manipulating the action list and project structure.

```python
pyscaffold.api.helpers.NO_CREATE = 1
# Do not create the file during an update

pyscaffold.api.helpers.NO_OVERWRITE = 0
# Do not overwrite an existing file during update (still created if not exists)

pyscaffold.api.helpers.ensure(struct, path, content=None, update_rule=None)
# Ensure a file exists in the representation of the project tree with the provided content. All the parent directories are automatically created.
```

**Parameters**

- `struct (dict)` – project representation as (possibly) nested `dict`. See `merge`.

- `path (os.PathLike)` – path-like string or object relative to the structure root. The following examples are equivalent:

```python
from pathlib import PurePath

'docs/api/index.html'
PurePath('docs', 'api', 'index.html')
```
**Deprecated** - Alternatively, a list with the parts of the path can be provided, ordered from the structure root to the file itself.

- **content** *(str)* – file text contents, None by default. The old content is preserved if None.
- **update_rule** – see *FileOp*, None by default

**Returns** updated project tree representation

**Return type** *dict*

**Note:** Use an empty string as content to ensure a file is created empty.

---

**Warning:** *Deprecation Notice* - In the next major release, the usage of lists for the *path* argument will result in an error. Please use *pathlib.PurePath* instead.

```python
pyscaffold.api.helpers.logger = <ReportLogger pyscaffold.log (WARNING)>
```

Logger wrapper, that provides methods like *report*. See *ReportLogger*.

```python
pyscaffold.api.helpers.merge(old, new)
```

Merge two dict representations for the directory structure.

Basically a deep dictionary merge, except from the leaf update method.

**Parameters**

- **old** *(dict)* – directory descriptor that takes low precedence during the merge
- **new** *(dict)* – directory descriptor that takes high precedence during the merge

The directory tree is represented as a (possibly nested) dictionary. The keys indicate the path where a file will be generated, while the value indicates the content. Additionally, tuple values are allowed in order to specify the rule that will be followed during an *update* operation (see *FileOp*). In this case, the first element is the file content and the second element is the update rule. For example, the dictionary:

```
{'project': {
    'namespace': {
        'module.py': ('print("Hello World!")', 
                     helpers.NO_OVERWRITE)}}
```

represents a *project/namespace/module.py* file with content *print("Hello World!")*, that will be created only if not present.

**Returns** resulting merged directory representation

**Return type** *dict*

**Note:** Use an empty string as content to ensure a file is created empty. (None contents will not be created).

```python
pyscaffold.api.helpers.modify(struct, path, modifier=<function _id_func>, update_rule=None)
```

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** *(dict)* – project representation as (possibly) nested *dict*. See *merge*.  

---
**path** *(os.PathLike)* – path-like string or object relative to the structure root. The following examples are equivalent:

```python
from pathlib import PurePath

docs/api/index.html
PurePath('docs', 'api', 'index.html')
```

*Deprecated* - Alternatively, a list with the parts of the path can be provided, ordered from the structure root to the file itself.

**modifier** *(callable)* – function (or callable object) that receives the old content as argument and returns the new content. If no modifier is passed, the identity function will be used. Note that, if the file does not exist in `struct`, `None` will be passed as argument. Example:

```python
modifier = lambda old: (old or '') + 'APPENDED CONTENT'!
modifier = lambda old: 'PREPENDED CONTENT!' + (old or '')
```

**update_rule** – see `FileOp`, `None` by default. Note that, if no `update_rule` is passed, the previous one is kept.

**Returns** updated project tree representation

**Return type** dict

**Note:** Use an empty string as content to ensure a file is created empty (`None` contents will not be created).

**Warning:** *Deprecation Notice* - In the next major release, the usage of lists for the `path` argument will result in an error. Please use `pathlib.PurePath` instead.

```python
pyscaffold.api.helpers.register(actions, action, before=None, after=None)
```

Register a new action to be performed during scaffold.

**Parameters**

- **actions** *(list)* – previous action list.
- **action** *(callable)* – 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:

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

- ****kwars** *(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:

```python
helpers.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.
```

(continues on next page)
helpers.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:

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

**Returns** modified action list.

**Return type** list

pyscaffold.api.helpers.reject(struct, path)
Remove a file from the project tree representation if existent.

**Parameters**

- **struct** (*dict*) – project representation as (possibly) nested dict. See merge.
- **path** (*os.PathLike*) – path-like string or object relative to the structure root. The following examples are equivalent:

```python
from pathlib import PurePath

'docs/api/index.html'
PurePath('docs', 'api', 'index.html')
```

**Deprecated** - Alternatively, a list with the parts of the path can be provided, ordered from the structure root to the file itself.

**Returns** modified project tree representation

**Return type** dict

**Warning:** Deprecation Notice - In the next major release, the usage of lists for the path argument will result in an error. Please use pathlib.PurePath instead.

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

**Parameters**

- **actions** (*list*) – 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

Module contents

Exposed API for accessing PyScaffold via Python.

In addition to the functions and classes exposed in this module, please also consider `pyscaffold.templates.get_template` to be part of PyScaffold’s public API.

class pyscaffold.api.Extension(name)
   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.

   activate (actions)
   Activates the extension by registering its functionality

   Parameters actions (list) – list of action to perform

   Returns updated list of actions

   Return type list

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

flag
mutually_exclusive = False

static register (*args, **kwargs)
   Shortcut for helpers.register

static unregister (*args, **kwargs)
   Shortcut for helpers.unregister

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:

   Naming

   • project (str)
• package (str)

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)

Some of these options are equivalent to the command line options, others are used for creating the basic python package meta information, but the last tree can change the way PyScaffold 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.

Finally, the extensions list may contain any function that follows the extension API. Note that some PyScaffold features, such as travis, tox and pre-commit support, are implemented as built-in extensions. In order to use these features it is necessary to include the respective functions in the extension list. All built-in extensions are accessible via pyscaffold.extensions submodule.

Note that extensions may define extra options. For example, built-in cookiecutter extension define a cookiecutter option that should be the address to the git repository used as template.

pyscaffold.api.discover_actions(extensions)
Retrieve the action list.

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

Parameters
• extensions (list) – list of functions responsible for activating the

Returns scaffold actions.

Return type list

pyscaffold.api.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** (*dict*) – project representation as (possibly) nested *dict*.
• **opts** (*dict*) – given options, see *create_project* for an extensive list.

Returns

project representation and options with default values set

Return type

dict, dict

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.api.*.init_git(*struct*, *opts*)

Add revision control to the generated files.

Parameters

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

Returns

updated project representation and options

Return type

dict, dict

*pyscaffold.api.*.verify_options_consistency(*struct*, *opts*)

Perform some sanity checks about the given options.

Parameters

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

Returns

updated project representation and options

Return type

dict, dict

*pyscaffold.api.*.verify_project_dir(*struct*, *opts*)

Check if PyScaffold can materialize the project dir structure.

Parameters

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

Returns

updated project representation and options

Return type

dict, dict

*pyscaffold.contrib package*

Subpackages
pyscaffold.contrib.setuptools_scm package

Submodules

pyscaffold.contrib.setuptools_scm.config module

configuration

class pyscaffold.contrib.setuptools_scm.config.Configuration(relative_to=None, root='.', version_scheme='guess-next-dev', local_scheme='node-and-date', write_to=None, write_to_template=None, tag_regex='^(?:\[\w-]+-)?(?P<version>[vV]\d+(?:\.|\d+)?[\d\.|\d]*(?:\+)?(?P<build_num>[0-9]+)?)?(?P<version>[vV]0\d+)?(?P<build_num>[0-9]+)?)?$', parent_dir_prefix_version=None, fallback_version=None, fallback_root='.', parse=None, git_describe_command=None)

Bases: object

Global configuration model

absolute_root

fallback_root

classmethod from_file(name='pyproject.toml')

Read Configuration from pyproject.toml (or similar). Raises exceptions when file is not found or toml is not installed or the file has invalid format or does not contain the [tool.setuptools_scm] section.

relative_to

root

tag_regex

pyscaffold.contrib.setuptools_scm.discover module

pyscaffold.contrib.setuptools_scm.discover.iter_matching_entrypoints(path, entry-point)

pyscaffold.contrib.setuptools_scm.file_finder module

pyscaffold.contrib.setuptools_scm.file_finder.scm_find_files(path, scm_files, scm_dirs)

setuptools compatible file finder that follows symlinks
• path: the root directory from which to search
• scm_files: set of scm controlled files and symlinks (including symlinks to directories)
• scm_dirs: set of scm controlled directories (including directories containing no scm controlled files)

scm_files and scm_dirs must be absolute with symlinks resolved (realpath), with normalized case (normcase)

Spec here: http://setuptools.readthedocs.io/en/latest/setuptools.html# adding-support-for-revision-control-systems

```python
pyscaffold.contrib.setuptools_scm.file_finder_git module

class pyscaffold.contrib.setuptools_scm.git.GitWorkdir(path)
    Bases: object
    experimental, may change at any time
    count_all_nodes()
    do_ex(cmd)
    fetch_shallow()
    classmethod from_potential_worktree(wd)
    get_branch()
    is_dirty()
    is_shallow()
    node()

pyscaffold.contrib.setuptools_scm.git.fail_on_shallow(wd)
    experimental, may change at any time

pyscaffold.contrib.setuptools_scm.git.fetch_on_shallow(wd)
    experimental, may change at any time

pyscaffold.contrib.setuptools_scm.git.parse(root, describe_command='git describe --dirty --tags --long --match *.*', pre_parse=<function warn_on_shallow>, config=None)

    Parameters pre_parse -- experimental pre_parse action, may change at any time

pyscaffold.contrib.setuptools_scm.git.warn_on_shallow(wd)
    experimental, may change at any time
```
pyscaffold.contrib.setuptools_scm.hacks module

pyscaffold.contrib.setuptools_scm.hacks.fallback_version(root, config=None)
pyscaffold.contrib.setuptools_scm.hacks.parse_pip_egg_info(root, config=None)
pyscaffold.contrib.setuptools_scm.hacks.parse_pkginfo(root, config=None)

pyscaffold.contrib.setuptools_scm.hg module

pyscaffold.contrib.setuptools_scm.hg.archival_to_version(data, config=None)
pyscaffold.contrib.setuptools_scm.hg.get_graph_distance(root, rev1, rev2='. ')
pyscaffold.contrib.setuptools_scm.hg.get_latest_normalizable_tag(root)
pyscaffold.contrib.setuptools_scm.hg.parse(root, config=None)
pyscaffold.contrib.setuptools_scm.hg.parse_archival(root, config=None)

pyscaffold.contrib.setuptools_scm.integration module

pyscaffold.contrib.setuptools_scm.integration.find_files(path='')
pyscaffold.contrib.setuptools_scm.integration.infer_version(dist)
pyscaffold.contrib.setuptools_scm.integration.version_keyword(dist, keyword, value)

pyscaffold.contrib.setuptools_scm.utils module

utils

class pyscaffold.contrib.setuptools_scm.utils.UTC(Bases: datetime.tzinfo)

dst (dt)
datetime -> DST offset as timedelta positive east of UTC.

tzname (dt)
datetime -> string name of time zone.

utcoffset (dt)
datetime -> timedelta showing offset from UTC, negative values indicating West of UTC

pyscaffold.contrib.setuptools_scm.utils.data_from_mime(path)
pyscaffold.contrib.setuptools_scm.utils.do(cmd, cwd='')
pyscaffold.contrib.setuptools_scm.utils.do_ex(cmd, cwd='')
pyscaffold.contrib.setuptools_scm.utils.ensure_stripped_str(str_or_bytes)
pyscaffold.contrib.setuptools_scm.utils.function_has_arg(fn, argname)
pyscaffold.contrib.setuptools_scm.utils.has_command(name)
pyscaffold.contrib.setuptools_scm.utils.no_git_env(env)
pyscaffold.contrib.setuptools_scm.utils.trace(*k)
pyscaffold.contrib.setuptools_scm.utils.trace_exception()

**pyscaffold.contrib.setuptools_scm.version module**

```python
class pyscaffold.contrib.setuptools_scm.version.ScmVersion(tag_version, distance=None, node=None, dirty=False, preformatted=False, branch=None, config=None, **kw)

    Bases: object
    exact
    extra
    format_choice (clean_format, dirty_format, **kw)
    format_next_version (guess_next, fmt='{guessed}.dev{distance}', **kw)
    format_with (fmt, **kw)
```

**exception pyscaffold.contrib.setuptools_scm.version.SetuptoolsOutdatedWarning**

```python
Bases: Warning
```

```python
pyscaffold.contrib.setuptools_scm.version.callable_or_entrypoint (group, callable_or_name)
pyscaffold.contrib.setuptools_scm.version.format_version (version, **config)
pyscaffold.contrib.setuptools_scm.version.get_local_dirty_tag (version)
pyscaffold.contrib.setuptools_scm.version.get_local_node_and_date (version)
pyscaffold.contrib.setuptools_scm.version.get_local_node_and_timestamp (version, fmt='%Y%m%d%H%M%S')
pyscaffold.contrib.setuptools_scm.version.get_no_local_node ()
pyscaffold.contrib.setuptools_scm.version.guess_next_dev_version (version)
pyscaffold.contrib.setuptools_scm.version.guess_next_simple_semver (version, retain, increment=True)
pyscaffold.contrib.setuptools_scm.version.guess_next_version (tag_version)
pyscaffold.contrib.setuptools_scm.version.meta (tag, distance=None, dirty=False, node=None, preformatted=False, branch=None, config=None, **kw)
pyscaffold.contrib.setuptools_scm.version.postrelease_version (version)
pyscaffold.contrib.setuptools_scm.version.release_branch_semver (version)
pyscaffold.contrib.setuptools_scm.version.release_branch_semver_version (version)
pyscaffold.contrib.setuptools_scm.version.simplified_semver_version (version)
pyscaffold.contrib.setuptools_scm.version.tag_to_version (tag, config=None)
```

- take a tag that might be prefixed with a keyword and return only the version part
- :param config: optional configuration object

15.1. pyscaffold package
pyscaffold.contrib.setuptools_scm.version.tags_to_versions(tags, config=None)
take tags that might be prefixed with a keyword and return only the version part
:param tags: an iterable of tags
:param config: optional configuration object

pyscaffold.contrib.setuptools_scm.win_py31_compat module

Module contents

copyright 2010-2015 by Ronny Pfannschmidt
license MIT

pyscaffold.contrib.setuptools_scm.dump_version(root, version, write_to, template=None)
pyscaffold.contrib.setuptools_scm.get_version(root='.', version_scheme='guess-next-dev', local_scheme='node-and-date', write_to=None, write_to_template=None, relative_to=None, tag_regex='^(?:\[\w-]+-)?(?P<version>[vV]\d+(?:\.\d+){0,2}(?:\+\d+)*(?:\+.*))?$', parentdir_prefix_version=None, fallback_version=None, fallback_root='.', parse=None, git_describe_command=None)

If supplied, relative_to should be a file from which root may be resolved. Typically called by a script or module that is not in the root of the repository to direct setuptools_scm to the root of the repository by supplying __file__.

pyscaffold.contrib.setuptools_scm.version_from_scm(root)

Submodules

pyscaffold.contrib.configupdater module

Configuration file updater.

A configuration file consists of sections, lead by a “[section]” header, and followed by “name: value” entries, with continuations and such in the style of RFC 822.

The basic idea of ConfigUpdater is that a configuration file consists of three kinds of building blocks: sections, comments and spaces for separation. A section itself consists of three kinds of blocks: options, comments and spaces. This gives us the corresponding data structures to describe a configuration file.

A general block object contains the lines which were parsed and make up the block. If a block object was not changed then during writing the same lines that were parsed will be used to express the block. In case a block, e.g. an option, was changed, it is marked as updated and its values will be transformed into a corresponding string during an update of a configuration file.

Note: ConfigUpdater was created by starting from Python’s ConfigParser source code and changing it according to my needs. Thus this source code is subject to the PSF License in a way but I am not a lawyer.

exception pyscaffold.contrib.configupdater.NoSectionError(section)
Bases: configparser.Error
Raised when no section matches a requested option.

```python
exception pyscaffold.contrib.configupdater.DuplicateOptionError(section, option, source=None, lineno=None)
```

**Bases:** `configparser.Error`

Raised by strict parsers when an option is repeated in an input source.

Current implementation raises this exception only when an option is found more than once in a single file, string or dictionary.

```python
exception pyscaffold.contrib.configupdater.DuplicateSectionError(section, source=None, lineno=None)
```

**Bases:** `configparser.Error`

Raised when a section is repeated in an input source.

Possible repetitions that raise this exception are: multiple creation using the API or in strict parsers when a section is found more than once in a single input file, string or dictionary.

```python
exception pyscaffold.contrib.configupdater.NoOptionError(option, section)
```

**Bases:** `configparser.Error`

A requested option was not found.

```python
exception pyscaffold.contrib.configupdater.NoConfigFileReadError
```

**Bases:** `configparser.Error`

Raised when no configuration file was read but update requested.

```python
exception pyscaffold.contrib.configupdater.ParsingError(source=None, filename=None)
```

**Bases:** `configparser.Error`

Raised when a configuration file does not follow legal syntax.

```python
append(lineno, line)
```

**filename**

Deprecated, use ‘source’.

```python
exception pyscaffold.contrib.configupdater.MissingSectionHeaderError(filename, lineno, line)
```

**Bases:** `configparser.ParsingError`

Raised when a key-value pair is found before any section header.

```python
class pyscaffold.contrib.configupdater.ConfigUpdater(allow_no_value=False, *, delimiters=('=', ':'), comment_prefixes=('#', ';'), in_line_comment_prefixes=None, strict=True, space_around_delimiters=True)
```

**Bases:** `pyscaffold.contrib.configupdater.Container`, `collections.abc.MutableMapping`

Parser for updating configuration files.

**ConfigUpdater follows the API of ConfigParser with some differences:**

- inline comments are treated as part of a key’s value,
• only a single config file can be updated at a time,
• empty lines in values are not valid,
• the original case of sections and keys are kept,
• control over the position of a new section/key.

Following features are deliberately not implemented:
• interpolation of values,
• propagation of parameters from the default section,
• conversions of values,
• passing key/value-pairs with default argument,
• non-strict mode allowing duplicate sections and keys.

NONSPACECRE = re.compile('\\S')
OPTCRE = re.compile('\\n (?P<option>.*?[^=]) # very permissive!
 \s*(?P<vi>=|:)=\s* # any number of space/tab,
 # followed by any of t, re.VERBOSE)
OPTCRE_NV = re.compile('\\n (?P<option>.*?[^=]) # very permissive!
 \s*(?: # any number of space/tab,
 (?P<vi>=|:)=\s* # optionally followed , re.VERBOSE)
SECTCRE = re.compile('\\n \[ # 
 (?P<header>[^\]]*) # very permissive!
 \] # ]
 ', re.VERBOSE)

add_section(section)
Create a new section in the configuration.
Raise DuplicateSectionError if a section by the specified name already exists. Raise ValueError if name is DEFAULT.

Parameters
section (str or Section) – name or Section type

get(section, option)
Gets an option value for a given section.

Parameters
• section (str) – section name
• option (str) – option name

Returns Option object holding key/value pair

Return type Option

has_option(section, option)
Checks for the existence of a given option in a given section.

Parameters
• section (str) – name of section
• option (str) – name of option

Returns whether the option exists in the given section

Return type bool

has_section(section)
Returns whether the given section exists.

Parameters
section (str) – name of section

Returns wether the section exists

Return type bool
items (section=<object object>)
    Return a list of (name, value) tuples for options or sections.
    If section is given, return a list of tuples with (name, value) for each option in the section. Otherwise, return a list of tuples with (section_name, section_type) for each section.
    Parameters section (str) – optional section name, default UNSET
    Returns list of Section or Option objects
    Return type list

options (section)
    Returns list of configuration options for the named section.
    Parameters section (str) – name of section
    Returns list of option names
    Return type list

optionxform (optionstr)
    Converts an option key to lower case for unification
    Parameters optionstr (str) – key name
    Returns unified option name
    Return type str

read (filename, encoding=None)
    Read and parse a filename.
    Parameters
        • filename (str) – path to file
        • encoding (str) – encoding of file, default None

read_file (f, source=None)
    Like read() but the argument must be a file-like object.
    The f argument must be iterable, returning one line at a time. Optional second argument is the source specifying the name of the file being read. If not given, it is taken from f.name. If f has no name attribute, <???> is used.
    Parameters
        • f – file like object
        • source (str) – reference name for file object, default None

read_string (string, source='<string> ')
    Read configuration from a given string.
    Parameters
        • string (str) – string containing a configuration
        • source (str) – reference name for file object, default '<string>'

remove_option (section, option)
    Remove an option.
    Parameters
        • section (str) – section name
• **option** *(str)* – option name
  *Returns* whether the option was actually removed
  *Return type* bool

`remove_section(name)`
Remove a file section.

  *Parameters* name – name of the section
  *Returns* whether the section was actually removed
  *Return type* bool

`sections()`
Return a list of section names

  *Returns* list of section names
  *Return type* list

`sections_blocks()`
Returns all section blocks

  *Returns* list of Section blocks
  *Return type* list

`set(section, option, value=None)`
Set an option.

  *Parameters*
  *• section (str)* – section name
  *• option (str)* – option name
  *• value (str)* – value, default None

`to_dict()`
Transform to dictionary

  *Returns* dictionary with same content
  *Return type* dict

`update_file(validate=True)`
Update the read-in configuration file.

  *Parameters* validate (Boolean) – validate format before writing

`validate_format(**kwargs)`
Call ConfigParser to validate config

  *Parameters* kwargs – are passed to configparser.ConfigParser

`write(fp, validate=True)`
Write an .ini-format representation of the configuration state.

  *Parameters*
  *• fp (file-like object)* – open file handle
  *• validate (Boolean)* – validate format before writing
pyscaffold.contrib.ptr module

Implementation

class pyscaffold.contrib.ptr.CustomizedDist (attrs=None)
    Bases: setuptools.dist.Distribution

    allow_hosts = None

    fetch_build_egg (req)
        Specialized version of Distribution.fetch_build_egg that respects allow_hosts and index_url.

    index_url = None

class pyscaffold.contrib.ptr.PyTest (dist, **kw)
    Bases: setuptools.command.test.test

    static ensure_setuptools_version ()
        Due to the fact that pytest-runner is often required (via setup-requires directive) by toolchains that never invoke it (i.e. they're only installing the package, not testing it), instead of declaring the dependency in the package metadata, assert the requirement at run time.

    finalize_options ()
        Set final values for all the options that this command supports. This is always called as late as possible, i.e. after any option assignments from the command-line or from other commands have been done. Thus, this is the place to code option dependencies: if ‘foo’ depends on ‘bar’, then it is safe to set ‘foo’ from ‘bar’ as long as ‘foo’ still has the same value it was assigned in ‘initialize_options()’.

        This method must be implemented by all command classes.

    initialize_options ()
        Set default values for all the options that this command supports. Note that these defaults may be overridden by other commands, by the setup script, by config files, or by the command-line. Thus, this is not the place to code dependencies between options; generally, ‘initialize_options()’ implementations are just a bunch of “self.foo = None” assignments.

        This method must be implemented by all command classes.

    install_dists (dist)
        Extend install_dists to include extras support

    install_extra_dists (dist)
        Install extras that are indicated by markers or install all extras if ‘–extras’ is indicated.

    static marker_passes (marker)
        Given an environment marker, return True if the marker is valid and matches this environment.

    run ()
        Override run to ensure requirements are available in this session (but don’t install them anywhere).

    run_tests ()
        Invoke pytest, replacing argv. Return result code.

    user_options = [('extras', None, 'Install (all) setuptools extras when running tests')]
Module contents

Contribution packages used by PyScaffold

All packages inside contrib are external packages that come with their own licences and are not part of the PyScaffold source code itself. The reason for shipping these dependencies directly is to avoid problems in the resolution of setup_requires dependencies that occurred more often than not, see issues #71 and #72.

Currently the contrib packages are:

1) setuptools_scm 4.1.2
2) pytest-runner 5.2
3) configupdater 1.1.2

The packages/modules were just copied over.

pyscaffold.contrib.scm_find_files(*args, **kwargs)
pyscaffold.contrib.scm_get_local_dirty_tag(*args, **kwargs)
pyscaffold.contrib.scm_get_local_node_and_date(*args, **kwargs)
pyscaffold.contrib.scm_guess_next_dev_version(*args, **kwargs)
pyscaffold.contrib.scm_parse_archival(*args, **kwargs)
pyscaffold.contrib.scm_parse_git(*args, **kwargs)
pyscaffold.contrib.scm_parse_hg(*args, **kwargs)
pyscaffold.contrib.scm_parse_pkginfo(*args, **kwargs)
pyscaffold.contrib.scm_postrelease_version(*args, **kwargs)
pyscaffold.contrib.warn_about_deprecated_pyscaffold()
pyscaffold.contrib.write_pbr_json(*args, **kwargs)

pyscaffold.extensions package

Submodules

pyscaffold.extensions.cirrus module

Extension that generates configuration for Cirrus CI.

class pyscaffold.extensions.cirrus.Cirrus(name)
    Bases: pyscaffold.api.Extension

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

    activate(actions)
    Activate extension

        Parameters actions (list) – list of actions to perform

        Returns updated list of actions

        Return type list
**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.

:param parser: current parser object

```python
class pyscaffold.extensions.cirrus.IncludeExtensions (option_strings, dest, nargs=None, const=None, default=None, type=None, choices=None, required=False, help=None, metavar=None)
```

Bases: `argparse.Action`

Automatically activate tox and pre-commit together with cirrus.

```python
def add_files(struct, opts)
```

Add `.cirrus.yaml` to the file structure

**Parameters**

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

**Returns** updated project representation and options

**Return type** `struct, opts`

```python
def cirrus_descriptor(_opts)
```

Returns the rendered template

---

**pyscaffold.extensions.cookiecutter module**

Extension that integrates cookiecutter templates into PyScaffold.

---

**Warning:** *Deprecation Notice* - In the next major release the Cookiecutter extension will be extracted into an independent package. After PyScaffold v4.0, you will need to explicitly install `pyscaffoldext-cookiecutter` in your system/virtualenv in order to be able to use it.

```python
class pyscaffold.extensions.cookiecutter.Cookiecutter(name)
```

Bases: `pyscaffold.api.Extension`

Additionally apply a Cookiecutter template

**activate(actions)**

Register before_create hooks to generate project using Cookiecutter

**Parameters** `actions` *(list)* — list of actions to perform

**Returns** updated list of actions

**Return type** `list`

**augment_cli(parser)**

Add an option to parser that enables the Cookiecutter extension

**Parameters** `parser` *(argparse.ArgumentParser)* — CLI parser object

`mutually_exclusive = True`
exception pyscaffold.extensions.cookiecutter.MissingTemplate(message='missing `cookiecutter` option', *args, **kwargs)

Bases: RuntimeError

A cookiecutter template (git url) is required.

DEFAULT_MESSAGE = 'missing `cookiecutter` option'

exception pyscaffold.extensions.cookiecutter.NotInstalled(message='cookiecutter is not installed, run: pip install cookiecutter', *args, **kwargs)

Bases: RuntimeError

This extension depends on the cookiecutter package.

DEFAULT_MESSAGE = 'cookiecutter is not installed, run: pip install cookiecutter'

pyscaffold.extensions.cookiecutter.create_cookiecutter(struct, opts)

Create a cookiecutter template

Parameters

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

Returns updated project representation and options

Return type struct, opts

pyscaffold.extensions.cookiecutter.create_cookiecutter_parser(obj_ref)

Create a Cookiecutter parser.

Parameters **obj_ref (Extension)** – object reference to the actual extension

Returns parser for namespace cli argument

Return type NamespaceParser

pyscaffold.extensions.cookiecutter.enforce_cookiecutter_options(struct, opts)

Make sure options reflect the cookiecutter usage.

Parameters

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

Returns updated project representation and options

Return type struct, opts

pyscaffold.extensions.django module

Extension that creates a base structure for the project using django-admin.

Warning: Deprecation Notice - In the next major release the Django extension will be extracted into an independent package. After PyScaffold v4.0, you will need to explicitly install pyscaffoldext-django in your system/virtualenv in order to be able to use it.
class pyscaffold.extensions.django.Django(name)
    Bases: pyscaffold.api.Extension

Generate Django project files

activate(actions)
    Register hooks to generate project using django-admin.

    Parameters actions(list) – list of actions to perform
    Returns updated list of actions
    Return type list

    mutually_exclusive = True

exception pyscaffold.extensions.django.DjangoAdminNotInstalled(message='django-admin is not installed, run: pip install django', *args, **kwargs)
    Bases: RuntimeError

This extension depends on the django-admin cli script.

DEFAULT_MESSAGE = 'django-admin is not installed, run: pip install django'
pyscaffold.extensions.django.create_django_proj(struct, opts)
    Creates a standard Django project with django-admin

    Parameters

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

    Returns updated project representation and options
    Return type struct, opts

    Raises RuntimeError – raised if django-admin is not installed

pyscaffold.extensions.django.enforce_django_options(struct, opts)
    Make sure options reflect the Django usage.

    Parameters

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

    Returns updated project representation and options
    Return type struct, opts

pyscaffold.extensions.gitlab_ci module

Extension that generates configuration and script files for GitLab CI.

class pyscaffold.extensions.gitlab_ci.GitLab(name)
    Bases: pyscaffold.api.Extension

Generate GitLab CI configuration files

15.1. pyscaffold package
activate (actions)
   Activate extension
   Parameters actions (list) – list of actions to perform
   Returns updated list of actions
   Return type list

add_files (struct, opts)
   Add .gitlab-ci.yml file to structure
   Parameters
   • struct (dict) – project representation as (possibly) nested dict.
   • opts (dict) – given options, see create_project for an extensive list.
   Returns updated project representation and options
   Return type struct, opts

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)
   Bases: pyscaffold.api.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) – 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 (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.create_namespace_parser (obj_ref)
   Create a namespace parser.
   Parameters obj_ref (Extension) – object reference to the actual extension
Returns parser for namespace cli argument

Return type NamespaceParser

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.no_skeleton module

Extension that omits the creation of file skeleton.py

class pyscaffold.extensions.no_skeleton.NoSkeleton(name)
  Bases: pyscaffold.api.Extension

  Omit creation of skeleton.py and test_skeleton.py
  
  activate(actions)
  Activate extension

  Parameters actions (list) – list of actions to perform

  Returns updated list of actions

  Return type list

   remove_files(struct, opts)
   Remove all skeleton files from structure

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

   Returns updated project representation and options

   Return type struct, opts

pyscaffold.extensions.pre_commit module

Extension that generates configuration files for Yelp pre-commit.

class pyscaffold.extensions.pre_commit.PrecCommit(name)
  Bases: pyscaffold.api.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

static 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)

Parameters

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

Returns updated project representation and options

Return type struct, opts

static instruct_user(struct, opts)

pyscaffold.extensions.tox module

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

class pyscaffold.extensions.tox.Tox(name)
   Bases: pyscaffold.api.Extension
   Generate Tox configuration file

activate(actions)
Activate extension

Parameters actions (list) – list of actions to perform

Returns updated list of actions

Return type list

add_files(struct, opts)
Add .tox.ini file to structure

Parameters

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

Returns updated project representation and options

Return type struct, opts

pyscaffold.extensions.travis module

Extension that generates configuration and script files for Travis CI.

class pyscaffold.extensions.travis.Travis(name)
   Bases: pyscaffold.api.Extension
   Generate Travis CI configuration files

activate(actions)
Activate extension
Parameters `actions` *(list)* – list of actions to perform

Returns updated list of actions

Return type `list`

`add_files` *(struct, opts)*

Add some Travis files to structure

Parameters

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

Returns updated project representation and options

Return type `struct, opts`

### Module contents

Built-in extensions for PyScaffold.

#### pyscaffold.templates package

### Module contents

Templates for all files of a project’s scaffold

pyscaffold.templates.**authors**(opts)

Template of AUTHORS.rst

Parameters `opts` – mapping parameters as dictionary

Returns file content as string

Return type `str`

pyscaffold.templates.**changelog**(opts)

Template of CHANGELOG.rst

Parameters `opts` – mapping parameters as dictionary

Returns file content as string

Return type `str`

pyscaffold.templates.**conftest_py**(opts)

Template of conftest.py

Parameters `opts` – mapping parameters as dictionary

Returns file content as string

Return type `str`

pyscaffold.templates.**coveragerc**(opts)

Template of .coveragerc

Parameters `opts` – mapping parameters as dictionary

Returns file content as string

Return type `str`
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

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.gitignore(opts)
Template of .gitignore

Parameters opts – mapping parameters as dictionary

Returns file content as string

Return type str

pyscaffold.templates.gitignore_empty(opts)
Template of empty .gitignore
Parameters **opts** – mapping parameters as dictionary

Returns file content as string

Return type *str*

```pyscaffold.templates.gitlab_ci(opts)```
Template of `.gitlab-ci.yml`

Parameters **opts** – mapping parameters as dictionary

Returns file content as string

Return type *str*

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

Parameters **opts** – mapping parameters as dictionary

Returns file content as string

Return type *str*

```pyscaffold.templates.isort_cfg(opts)```
Template of `.isort.cfg`

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 = {'affero': 'license_affero_3.0', 'apache': 'license_apache', 'artistic': 'license_artistic_2.0', 'cc0': 'license_cc0', 'gpl': 'license_gpl', 'mit': 'license_mit', 'mozilla': 'license_mozilla', 'odc-sa': 'license_odc_sa', 'proprietary': 'license_none', 'public-domain': 'license_public_domain', 'simple-bsd': 'license_simplified_bsd'}```
All available licences

```pyscaffold.templates.namespace(opts)```
Template of `__init__.py` defining a namespace package

Parameters **opts** – mapping parameters as dictionary

Returns file content as string

Return type *str*

```pyscaffold.templates.pre_commit_config(opts)```
Template of `.pre-commit-config.yaml`

Parameters **opts** – mapping parameters as dictionary

Returns file content as string

Return type *str*

```pyscaffold.templates.readme(opts)```
Template of `README.rst`

Parameters **opts** – mapping parameters as dictionary

Returns file content as string
Return type str

pyscaffold.templates.requirements\(\texttt{opts}\)
Template of requirements.txt

Parameters \texttt{opts} – mapping parameters as dictionary

Returns file content as string

Return type str

pyscaffold.templates.setup_cfg\(\texttt{opts}\)
Template of setup.cfg

Parameters \texttt{opts} – mapping parameters as dictionary

Returns file content as string

Return type str

pyscaffold.templates.setup_py\(\texttt{opts}\)
Template of setup.py

Parameters \texttt{opts} – mapping parameters as dictionary

Returns file content as string

Return type str

pyscaffold.templates.skeleton\(\texttt{opts}\)
Template of skeleton.py defining a basic console script

Parameters \texttt{opts} – mapping parameters as dictionary

Returns file content as string

Return type str

pyscaffold.templates.sphinx_authors\(\texttt{opts}\)
Template of authors.rst

Parameters \texttt{opts} – mapping parameters as dictionary

Returns file content as string

Return type str

pyscaffold.templates.sphinx_changelog\(\texttt{opts}\)
Template of changelog.rst

Parameters \texttt{opts} – mapping parameters as dictionary

Returns file content as string

Return type str

pyscaffold.templates.sphinx_conf\(\texttt{opts}\)
Template of conf.py

Parameters \texttt{opts} – mapping parameters as dictionary

Returns file content as string

Return type str

pyscaffold.templates.sphinx_index\(\texttt{opts}\)
Template of index.rst

Parameters \texttt{opts} – mapping parameters as dictionary
Returns  file content as string
Return type  str

```python
pyscaffold.templates.sphinx_license(opts)
```
Template of license.rst

Parameters  opts – mapping parameters as dictionary
Returns  file content as string
Return type  str

```python
pyscaffold.templates.sphinx_makefile(opts)
```
Template of Sphinx’s Makefile

Parameters  opts – mapping parameters as dictionary
Returns  file content as string
Return type  str

```python
pyscaffold.templates.test_skeleton(opts)
```
Template of unittest for skeleton.py

Parameters  opts – mapping parameters as dictionary
Returns  file content as string
Return type  str

```python
pyscaffold.templates.tox(opts)
```
Template of tox.ini

Parameters  opts – mapping parameters as dictionary
Returns  file content as string
Return type  str

```python
pyscaffold.templates.travis(opts)
```
Template of .travis.yml

Parameters  opts – mapping parameters as dictionary
Returns  file content as string
Return type  str

```python
pyscaffold.templates.travis_install(opts)
```
Template of travis_install.sh

Parameters  opts – mapping parameters as dictionary
Returns  file content as string
Return type  str

### 15.1.2 Submodules

### 15.1.3 pyscaffold.cli module

Command-Line-Interface of PyScaffold

```python
pyscaffold.cli.add_default_args(parser)
```
Add the default options and arguments to the CLI parser.
Parameters **parser** (*argparse.ArgumentParser*) – CLI parser object

```python
pyscaffold.cli.list_actions(opts)
```
Do not create a project, just list actions considering extensions

**Parameters** **opts** (*dict*) – command line options as dictionary

```python
pyscaffold.cli.main(args)
```
Main entry point for external applications

**Parameters** **args** (*[str]*) – command line arguments

```python
pyscaffold.cli.parse_args(args)
```
Parse command line parameters respecting extensions

**Parameters** **args** (*[str]*) – command line parameters as list of strings

**Returns** command line parameters

**Return type** *dict*

```python
pyscaffold.cli.process_opts(opts)
```
Process and enrich command line arguments

**Parameters** **opts** (*dict*) – dictionary of parameters

**Returns** dictionary of parameters from command line arguments

**Return type** *dict*

```python
pyscaffold.cli.run()
```
Entry point for console script

```python
pyscaffold.cli.run_scaffold(opts)
```
Actually scaffold the project, calling the python API

**Parameters** **opts** (*dict*) – command line options as dictionary

### 15.1.4 pyscaffold.exceptions module

Custom exceptions used by PyScaffold to identify common deviations from the expected behavior.

```python
exception pyscaffold.exceptions.ActionNotFound(name, *args, **kwargs)
```
Base: *KeyError*

Impossible to find the required action.

```python
exception pyscaffold.exceptions.DirectoryAlreadyExists
```
Base: *RuntimeError*

The project directory already exists, but no update or force option was used.

```python
exception pyscaffold.exceptions.DirectoryDoesNotExist
```
Base: *RuntimeError*

No directory was found to be updated.

```python
exception pyscaffold.exceptions.GitDirtyWorkspace(message="Your working tree is dirty. Commit your changes first or use ‘--force’.", *args, **kwargs)
```
Base: *RuntimeError*

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:
    git config --global user.email "you@example.com"
    git config --global username "Your Name"
to set your account’s default identity.', *args, **kwargs)
Bases: RuntimeError
PyScaffold tries to read user.name and user.email from git config.

DEFAULT_MESSAGE = 'Make sure git is configured. Run:
    git config --global user.email "you@example.com"
    git config --global username "Your Name"
to set your account’s default identity.'

exception pyscaffold.exceptions.GitNotInstalled(message='Make sure git is installed and working.', *args, **kwargs)
Bases: RuntimeError
PyScaffold requires git to run.

DEFAULT_MESSAGE = 'Make sure git is installed and working.'

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: RuntimeError
PyScaffold cannot update a project that it hasn’t generated

DEFAULT_MESSAGE = 'Could not update project. Was it generated with PyScaffold?'

exception pyscaffold.exceptions.OldSetuptools(message='Your setuptools version is too old (<38.3). Use `pip install -U setuptools` to upgrade. If you have the deprecated 'distribute' package installed remove it or update to version 0.7.3.', *args, **kwargs)
Bases: RuntimeError
PyScaffold requires a recent version of setuptools.

DEFAULT_MESSAGE = 'Your setuptools version is too old (<38.3). Use `pip install -U setuptools` to upgrade. If you have the deprecated 'distribute' package installed remove it or update to version 0.7.3.'

exception pyscaffold.exceptions.PyScaffoldTooOld(message='setup.cfg has no section [pyscaffold]! Are you trying to update a pre 3.0 version?', *args, **kwargs)
Bases: RuntimeError
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(message, *args, **kwargs)
Bases: RuntimeError
Outputs proper logging when a ShellCommand fails

15.1. pyscaffold package
15.1.5 pyscaffold.info module

Provide general information about the system, user etc.

class pyscaffold.info.GitEnv
    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.best_fit_license(txt)
    Finds proper license name for the license defined in txt
    Parameters txt (str) – license name
    Returns license name
    Return type str

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.email()
    Retrieve the user’s email
    Returns user’s email
    Return type str

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
    Return type bool

pyscaffold.info.is_git_installed()
    Check if git is installed
    Returns True if git is installed, False otherwise
    Return type bool

pyscaffold.info.is_git_workspace_clean(path)
    Checks if git workspace is clean
    Parameters path (str) – path to git repository
    Returns condition if workspace is clean or not
Return type

bool

Raises: 
  GitNotInstalled: when git command is not available
  GitNotConfigured: when git does not know user information

pyscaffold.info.project(opts)

Update user options with the options of an existing PyScaffold project

Params: 
  opts (dict): options of the project

Returns 
  options with updated values

Return type 
  dict

Raises

  • PyScaffoldTooOld – when PyScaffold is to old to update from
  • NoPyScaffoldProject – when project was not generated with PyScaffold

pyscaffold.info.username()

Retrieve the user’s name

Returns 
  user’s name

Return type 
  str

15.1.6 pyscaffold.integration module

Integration part for hooking into distutils/setuptools

Rationale: The use_pyscaffold keyword is unknown to setuptools’ setup(...) command, therefore the entry_points are checked for a function to handle this keyword which is pyscaffold_keyword below. This is where we hook into setuptools and apply the magic of setuptools_scm as well as other commands.

class pyscaffold.integration.DeprecatedPyTest(dist, **kw)

Bases: pyscaffold.contrib.ptr.PyTest

initialize_options()

Set default values for all the options that this command supports. Note that these defaults may be overridden by other commands, by the setup script, by config files, or by the command-line. Thus, this is not the place to code dependencies between options; generally, ‘initialize_options()’ implementations are just a bunch of “self.foo = None” assignments.

This method must be implemented by all command classes.

pyscaffold.integration.build_cmd_docs()

Return Sphinx’s BuildDoc if available otherwise a dummy command

Returns 
  command object

Return type 
  Command

pyscaffold.integration.local_version2str(version)

Create the local part of a PEP440 version string

Parameters

  version (setuptools_scm.version.ScmVersion) – version object

Returns 
  local version

Return type 
  str
pyscaffold.integration.pyscaffold_keyword (dist, keyword, value)
Handles the use_pyscaffold keyword of the setup(...) command

Parameters

- dist (setuptools.dist) – distribution object as
- keyword (str) – keyword argument = 'use_pyscaffold'
- value – value of the keyword argument

pyscaffold.integration.setuptools_scm_config(value)
Generate the configuration for setuptools_scm

Parameters

- value – value from entry_point

Returns dictionary of options

Return type dict

pyscaffold.integration.version2str(version)
Creates a PEP440 version string

Parameters

- version (setuptools_scm.version.ScmVersion) – version object

Returns version string

Return type str

15.1.7 pyscaffold.log module

Custom logging infrastructure to provide execution information for the user.

class pyscaffold.log.ColoredReportFormatter (fmt=None, datefmt=None, style='%')
Bases: pyscaffold.log.ReportFormatter

Format logs with ANSI colors.

ACTIVITY_STYLES = {'create': ('green', 'bold'), 'delete': ('red', 'bold'), 'invoke': (...}

CONTEXT_PREFIX = '\x1b[35m\x1b[1mfrom\x1b[0m'

LOG_STYLES = {'critical': ('red', 'bold'), 'debug': ('green',), 'error': ('red',), ...

SUBJECT_STYLES = {'invoke': ('blue',)}

TARGET_PREFIX = '\x1b[35m\x1b[1mtoto\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.

class pyscaffold.log.ReportFormatter (fmt=None, datefmt=None, style='%')
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)
    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 de-
      fault a ReportFormatter is created and used.
    • extra (dict) – extra attributes to be merged into the log record. Options, empty by
      default.

    wrapped
        underlying logger object.
        Type logging.Logger

    handler
        stream handler configured for providing user feedback in PyScaffold CLI.
        Type logging.Handler

    formatter
        formatter configured in the default handler.
        Type logging.Formatter

    nesting
        current nesting level of the report.
copy ()
    Produce a copy of the wrapped logger.

    Sometimes, it is better to make a copy of the report logger to keep indentation consistent.

indent (count=1)
    Temporarily adjust padding while executing a context.

Example

```python
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.

level
    Effective level of the logger

process(msg, kwargs)
    Method overridden to augment LogRecord with the nesting attribute.

reconfigure (opts=None, **kwargs)
    Reconfigure some aspects of the logger object.

    Parameters
    opts (dict) – dict with the same elements as the keyword arguments below

    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) – usually a path in the file system or an action identifier.
    • context (str) – path where the activity take place.
    • target (str) – 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:

```python
logger.report('copy', 'my/file', target='my/awesome/path')
logger.report('run', 'command', context='current/working/dir')
```

**pyscaffold.log.configure_logger**(opts)
Configure the default logger

**Parameters**
- **opts**(dict) – command line parameters

**Warning:** *Deprecation Notice* - In the next major release, this function will be removed. Please call *ReportLogger.reconfigure* instead:

```python
from pyscaffold.log import logger
logger.reconfigure(...)```

**pyscaffold.log.logger** = `<ReportLogger pyscaffold.log (WARNING)>`
Default logger configured for PyScaffold.

### 15.1.8 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**(str) – path to the project
- **tag_name**(str) – name of the tag
- **message**(str) – 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**(str) – 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 (dict)** – directory structure as dictionary of dictionaries
• **prefix (str)** – 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 (str)** – path to the project
• **struct (dict)** – 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

Parameters

• **folder (str)** – path

15.1.9 pyscaffold.shell module

Shell commands like git, django-admin etc.

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 (str)** – command to handle
• **shell (bool)** – run the command in the shell
• **cwd (str)** – 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.

pyscaffold.shell.command_exists(cmd)

Check if command exists

Parameters

• **cmd** – executable name

pyscaffold.shell.django_admin = <pyscaffold.shell.ShellCommand object>

Command for django-admin

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.*shell_command_error2exit_decorator* *(func)*

Decorator to convert given ShellCommandException to an exit message
This avoids displaying nasty stack traces to end-users

### 15.1.10 *pyscaffold.structure* module

Functionality to generate and work with the directory structure of a project

```python
class pyscaffold.structure.FileOp
    Bases: object

    Namespace for file operations during an update

    NO_CREATE = 1
    -> Do not create the file during an update

    NO_OVERWRITE = 0
    -> Do not overwrite an existing file during update (still created if not exists)
```

```python
pyscaffold.structure.*create_structure*(struct, opts, prefix=None)

Manifests a directory structure in the filesystem

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

Return type
tuple(dict, dict)

Raises
.RuntimeError – raised if content type in struct is unknown
```

```python
pyscaffold.structure.*define_structure*(_, opts)

Creates the project structure as dictionary of dictionaries

Parameters
- **_** *(dict)* – previous directory structure (ignored)
- **opts** *(dict)* – options of the project

Returns
structure as dictionary of dictionaries and input options

Return type
tuple(dict, dict)
```

### 15.1.11 *pyscaffold.termui* module

Basic support for ANSI code formatting.

```python
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

15.1.12 pyscaffold.update module

Functionality to update one PyScaffold version to another

def add_entrypoints(struct, opts)
Add [options.entry_points] to setup.cfg

Parameters

- **struct** (*dict*) – previous directory structure (ignored)
- **opts** (*dict*) – options of the project

Returns structure as dictionary of dictionaries and input options

Return type tuple(dict, dict)

def add_setup_requires(struct, opts)
Add setup_requires in setup.cfg

Parameters

- **struct** (*dict*) – previous directory structure (ignored)
- **opts** (*dict*) – options of the project

Returns structure as dictionary of dictionaries and input options

Return type tuple(dict, dict)
pyscaffold.update.apply_update_rule_to_file(path, value, opts)
Applies the update rule to a given file path

Parameters

•  **path** *(str)* – file path

•  **value** *(tuple or str)* – content (and update rule)

•  **opts** *(dict)* – options of the project, containing the following flags:
  –  **update**: if the project already exists and should be updated
  –  **force**: overwrite all the files that already exist

Returns  content of the file if it should be generated or None otherwise.

pyscaffold.update.apply_update_rules(struct, opts, prefix=None)
Apply update rules using **FileOp** to a directory structure.
As a result the filtered structure keeps only the files that actually will be written.

Parameters

•  **opts** *(dict)* – options of the project, containing the following flags:
  –  **update**: when the project already exists and should be updated
  –  **force**: overwrite all the files that already exist

•  **struct** *(dict)* – directory structure as dictionary of dictionaries (in this tree representation, each leaf can be just a string or a tuple also containing an update rule)

•  **prefix** *(str)* – prefix path for the structure

Returns  directory structure with keys removed according to the rules (in this tree representation, all the leaves are strings) and input options

Return type  tuple(dict, dict)

pyscaffold.update.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.update.invoke_action(action, struct, opts)
Invoke action with proper logging.

Parameters

•  **struct** *(dict)* – project representation as (possibly) nested dict.

•  **opts** *(dict)* – given options, see **create_project** for an extensive list.

Returns  updated project representation and options

Return type  tuple(dict, dict)

pyscaffold.update.read_setupcfg(project_path)
Reads-in setup.cfg for updating

Parameters  **project_path** *(str)* – path to project

Returns:
pyscaffold.update.update_pyscaffold_version(project_path, pretend)

Update setup_requires in setup.cfg

Parameters

• **project_path** (str) – path to project
• **pretend** (bool) – only pretend to do something

pyscaffold.update.version_migration(struct, opts)

Migrations from one version to another

Parameters

• **struct** (dict) – previous directory structure (ignored)
• **opts** (dict) – options of the project

Returns structure as dictionary of dictionaries and input options

Return type tuple(dict, dict)

15.1.13 pyscaffold.utils module

Miscellaneous utilities and tools

pyscaffold.utils.ERROR_INVALID_NAME = 123

Windows-specific error code indicating an invalid pathname.

pyscaffold.utils.chdir(path, **kwargs)

Contextmanager to change into a directory

Parameters **path** (str) – 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.utils.check_setuptools_version()

Check minimum required version of setuptools

Check that setuptools has all necessary capabilities for setuptools_scm as well as support for configuration with the help of setup.cfg.

Raises OldSetuptools – raised if necessary capabilities are not met

pyscaffold.utils.create_directory(path, update=False, pretend=False)

Create a directory in the given path.

This function reports the operation in the logs.

Parameters

• **path** (str) – path in the file system where contents will be written.
• **update** (bool) – false by default. A OSErr or is 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.utils.create_file(path, content, pretend=False)
Create a file in the given path.
This function reports the operation in the logs.

Parameters
- **path** *(str)* – path in the file system where contents will be written.
- **content** *(str)* – what will be written.
- **pretend** *(bool)* – false by default. File is not written when pretending, but operation is logged.

pyscaffold.utils.dasherize(word)
Replace underscores with dashes in the string.

Example:

```python
>>> dasherize("foo_bar")
"foo-bar"
```

Parameters **word** *(str)* – input word

Returns input word with underscores replaced by dashes

pyscaffold.utils.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

pyscaffold.utils.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

Returns identifier

Return type **str**

pyscaffold.utils.get_setup_requires_version()
Determines the proper setup_requires string for PyScaffold

E.g. setup_requires = pyscaffold>=3.0a0,<3.1a0

Returns requirement string for setup_requires

Return type **str**

pyscaffold.utils.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.

```python
pyscaffold.utils.is_valid_identifier(string)
```
Check if string is a valid package name

Parameters `string` *(str)* – package name

Returns `True` if string is valid package name else `False`

Return type `bool`

```python
pyscaffold.utils.levenshtein(s1, s2)
```
Calculate the Levenshtein distance between two strings

Parameters

- `s1` *(str)* – first string
- `s2` *(str)* – second string

Returns distance between `s1` and `s2`

Return type `int`

```python
pyscaffold.utils.localize_path(path_string)
```
Localize path for Windows, Unix, i.e. / or :param path_string: path using / :type path_string: str

Returns path depending on OS

Return type `str`

```python
pyscaffold.utils.make_valid_identifier(string)
```
Try to make a valid package name identifier from a string

Parameters `string` *(str)* – invalid package name

Returns valid package name as string or `RuntimeError`

Return type `str`

```
Raises InvalidIdentifier – raised if identifier can not be converted
```

```python
pyscaffold.utils.move(*src, **kwargs)
```
Move files or directories to (into) a new location

Parameters `*src` *(str[])* – one or more files/directories to be moved

Keyword Arguments

- `target` *(str)* – if target is a directory, `src` will be moved inside it. Otherwise, it will be the new path (note that it may be overwritten)
- `pretend` *(bool)* – skip execution (but log) when pretending. Default False.

```python
pyscaffold.utils.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.utils.prepare_namespace(namespace_str)
Check the validity of namespace_str and split it up into a list

Parameters namespace_str (str) – namespace, e.g. “com.blue_yonder”

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

Return type [str]

Raises InvalidIdentifier – raised if namespace is not valid

pyscaffold.utils.rm_rf(path)
Remove a path by all means like rm -rf in Linux.

Args (str): Path to remove:

15.1.14 pyscaffold.warnings module

Warnings used by PyScaffold to identify issues that can be safely ignored but that should be displayed to the user.

exception pyscaffold.warnings.UpdateNotSupported(*args, extension=None, **kwargs)
Bases: RuntimeWarning

Extensions that make use of external generators are not able to do updates by default.

DEFAULT_MESSAGE = ‘Updating code generated using external tools is not supported. The extension `{}` will be ignored, only changes in PyScaffold core features will take place.’

15.1.15 Module contents
CHAPTER 16

Indices and tables

• genindex
• modindex
• search
Python Module Index

p
pyscaffold, 113
pyscaffold.api, 73
pyscaffold.api.helpers, 69
pyscaffold.cli, 97
pyscaffold.contrib, 86
pyscaffold.contrib.configupdater, 80
pyscaffold.contrib.ptr, 85
pyscaffold.contrib.setuptools_scm, 80
pyscaffold.contrib.setuptools_scm.config, 76
pyscaffold.contrib.setuptools_scm.discover, 76
pyscaffold.contrib.setuptools_scm.file_finder, 76
pyscaffold.contrib.setuptools_scm.file_finder_git, 77
pyscaffold.contrib.setuptools_scm.file_finder_hg, 77
pyscaffold.contrib.setuptools_scm.git, 77
pyscaffold.contrib.setuptools_scm.hacks, 78
pyscaffold.contrib.setuptools_scm.hg, 78
pyscaffold.contrib.setuptools_scm.integration, 78
pyscaffold.contrib.setuptools_scm.utils, 78
pyscaffold.contrib.setuptools_scm.version, 79
pyscaffold.exceptions, 98
pyscaffold.extensions, 93
pyscaffold.extensions.cirrus, 86
pyscaffold.extensions.cookiecutter, 87
pyscaffold.extensions.django, 88
pyscaffold.extensions.gitlab_ci, 89
pyscaffold.extensions.namespace, 90
pyscaffold.extensions.no_skeleton, 91
pyscaffold.extensions.pre_commit, 91
pyscaffold.extensions.tox, 92
pyscaffold.extensions.travis, 92
pyscaffold.info, 100
pyscaffold.integration, 101
pyscaffold.log, 102
pyscaffold.repo, 105
pyscaffold.shell, 106
pyscaffold.structure, 107
pyscaffold.templates, 93
pyscaffold.termui, 107
pyscaffold.update, 108
pyscaffold.utils, 110
pyscaffold.warnings, 113
absolute_root (pyscaffold.contrib.setuptools_scm.config.Configuration attribute), 76
ActionNotFound, 98
activate() (pyscaffold.api.Extension method), 73
activate() (pyscaffold.extensions.cirrus.Cirrus method), 86
activate() (pyscaffold.extensions.cookiecutter.Cookiecutter method), 87
activate() (pyscaffold.extensions.django.Django method), 89
activate() (pyscaffold.extensions.gitlab_ci.GitLab method), 89
activate() (pyscaffold.extensions.namespace.Namespace method), 90
activate() (pyscaffold.extensions.no_skeleton.NoSkeleton method), 91
activate() (pyscaffold.extensions.pre_commit.PreCommit method), 91
activate() (pyscaffold.extensions.tox.Tox method), 92
activate() (pyscaffold.extensions.travis.Travis method), 92
add_default_args() (in module pyscaffold.cli), 97
add_entrypoints() (in module pyscaffold.update), 108
add_files() (in module pyscaffold.extensions.cirrus), 87
add_files() (pyscaffold.extensions.pre_commit.PreCommit static method), 90
add_files() (pyscaffold.extensions.tox.Tox method), 92
add_files() (pyscaffold.extensions.travis.Travis method), 93
add_files() (pyscaffold.extensions.gitlab_ci.GitLab method), 90
add_namespace() (in module pyscaffold.extensions.namespace), 90
add_setup_requires() (in module pyscaffold.update), 108
add_tag() (in module pyscaffold.repo), 105
allow_hosts (pyscaffold.contrib.ptr.CustomizedDist attribute), 85
append() (pyscaffold.contrib.configupdater.ParsingError method), 81
apply_update_rule_to_file() (in module pyscaffold.update), 109
apply_update_rules() (in module pyscaffold.update), 109
archival_to_version() (in module pyscaffold.contrib.setuptools_scm.hg), 78
augment_cl() (pyscaffold.api.Extension method), 73
augment_cl() (pyscaffold.extensions.cirrus.Cirrus method), 86
augment_cl() (pyscaffold.extensions.cookiecutter.Cookiecutter method), 87
augment_cl() (pyscaffold.extensions.namespace.Namespace method), 90
author_date (pyscaffold.info.GitEnv attribute), 100
author_email (pyscaffold.info.GitEnv attribute), 100
author_name (pyscaffold.info.GitEnv attribute), 100
authors() (in module pyscaffold.templates), 93
B

best_fit_license() (in module pyscaffold.info), 100
build_cmd_docs() (in module pyscaffold.integration), 101

callable_or_entrypoint() (in module pyscaffold.contrib.setuptools_scm.version), 79
changelog() (in module pyscaffold.scm.templates), 79
chdir() (in module pyscaffold.utils), 110
check_git() (in module pyscaffold.info), 100
check_setuptools_version() (in module pyscaffold.utils), 110
Cirrus (class in pyscaffold.extensions.cirrus), 86
cirrus_descriptor() (in module pyscaffold.extensions.cirrus), 87
ColoredReportFormatter (class in pyscaffold.log), 102
commiter_email (pyscaffold.info.GitEnv attribute), 100
commiter_name (pyscaffold.info.GitEnv attribute), 100
ConfigUpdater (class in pyscaffold.contrib.configupdater), 81
Configuration (class in pyscaffold.contrib.setuptools_scm.config), 76
configure_logger() (in module pyscaffold.log), 105
conftest_py() (in module pyscaffold.templates), 93
CONTEXT_PREFIX (pyscaffold.log.ColoredReportFormatter attribute), 102
CONTEXt_PREFIX (pyscaffold.log.ReportFormatter attribute), 102
Cookiecutter (class in pyscaffold.extensions.cookiecutter), 87
copy() (pyscaffold.log.ReportLogger method), 104
count_all_nodes() (pyscaffold.contrib.setuptools_scm.git.GitWorkdir method), 77
coveragerc() (in module pyscaffold.templates), 93
create_cookiecutter() (in module pyscaffold.extensions.cookiecutter), 88
create_cookiecutter_parser() (in module pyscaffold.extensions.cookiecutter), 88
create_directory() (in module pyscaffold.utils), 110
create_django_proj() (in module pyscaffold.extensions.django), 89
create_file() (in module pyscaffold.utils), 110
create_namespace_parser() (in module pyscaffold.extensions.namespace), 90
create_padding() (pyscaffold.log.ReportFormatter method), 103
create_project() (in module pyscaffold.api), 73
create_structure() (in module pyscaffold.structure), 107
curses_available() (in module pyscaffold.termui), 107
CustoMedDist (class in pyscaffold.contrib.ptr), 85

dasherize() (in module pyscaffold.utils), 111
data_from_mime() (in module pyscaffold.contrib.setuptools_scm.utils), 78
decorate() (in module pyscaffold.termui), 107
DEFAULT_MESSAGE (pyscaffold.exceptions.GitDirtyWorkspace attribute), 98
DEFAULT.Message (pyscaffold.exceptions.GitNotConfigured attribute), 99
DEFAULT_MESSAGE (pyscaffold.exceptions.GitNotInstalled attribute), 99
DEFAULT_MESSAGE (pyscaffold.exceptions.NoPyScaffoldProject attribute), 99
DEFAULT_MESSAGE (pyscaffold.exceptions.OldSetuptools attribute), 99
DEFAULT_MESSAGE (pyscaffold.exceptions.PyScaffoldTooOld attribute), 99
DEFAULT_MESSAGE (pyscaffold.exceptions.Cookiecutter.MissingTemplate attribute), 88
DEFAULT_MESSAGE (pyscaffold.extensions.cookiecutter.NotInstalled attribute), 88
DEFAULT_MESSAGE (pyscaffold.extensions.django.DjangoAdminNotInstalled attribute), 89
DEFAULT_MESSAGE (pyscaffold.warnings.UpdateNotSupported attribute), 113
define_structure() (in module pyscaffold.structure), 107
DeprecationPyTest (class in pyscaffold.integration), 101
DirectoryAlreadyExists, 98
DirectoryDoesNotExist, 98
discover_actions() (in module pyscaffold.api), 74
Django (class in pyscaffold.extensions.django), 88
django_admin (in module pyscaffold.shell), 106
DjangoAdminNotInstalled, 89
do() (in module pyscaffold.contrib.setuptools_scmutils), 78
do_ex() (in module pyscaffold.contrib.setuptools_scmutils), 78
do_ex() (pyscaffold.contrib.setuptools_scm.git.GitWorkdir method), 77
dst() (pyscaffold.contrib.setuptools_scm.utils.UTC method), 78
dump_version() (in module pyscaffold.contrib.setuptools_scm), 80
DuplicateOptionError, 81
DuplicateSectionError, 81

E

email() (in module pyscaffold.info), 100
enforce_cookiecutter_options() (in module pyscaffold.extensions.cookiecutter), 88
enforce_django_options() (in module pyscaffold.extensions.django), 89
enforce_namespace_options() (in module pyscaffold.extensions.namespaces), 91
ensure() (in module pyscaffold.api.helpers), 69
ensure_setuptools_version() (pyscaffold.contrib.ptr.PyTest static method), 85
ensure_stripped_str() (in module pyscaffold.contrib.setuptools_scmutils), 78
ERROR_INVALID_NAME (in module pyscaffold.utils), 110
exact (pyscaffold.contrib.setuptools_scm.version.ScmVersion attribute), 79
exceptions2exit() (in module pyscaffold.utils), 111
Extension (class in pyscaffold.utils), 73
extra (pyscaffold.contrib.setuptools_scm.version.ScmVersion attribute), 79

F

fail_on_shallow() (in module pyscaffold.contrib.setuptools_scm.git), 77
fallback_root (pyscaffold.contrib.setuptools_scm.config.Configuration attribute), 76
fallback_version() (in module pyscaffold.contrib.setuptools_scm), 77
fetch_build_egg() (pyscaffold.contrib.ptr.CustomizedDist method), 85
fetch_on_shallow() (in module pyscaffold.contrib.setuptools_scm.git), 77
fetch_shallow() (pyscaffold.contrib.setuptools_scm.git.GitWorkdir method), 77
filename (pyscaffold.contrib.configupdater.ParsingError attribute), 81
FileOp (class in pyscaffold.structure), 107
finalize_options() (pyscaffold.contrib.ptr.PyTest method), 85
find_files() (in module pyscaffold.contrib.setuptools_scm.integration), 78
flag (pyscaffold.api.Extension attribute), 73
format() (pyscaffold.log.ReportFormatter method), 103
format_activity() (pyscaffold.log.ColoredReportFormatter method), 102
format_activity() (pyscaffold.log.ReportFormatter method), 103
format_choice() (pyscaffold.contrib.setuptools_scm.version.ScmVersion method), 79
format_context() (pyscaffold.log.ReportFormatter method), 103
format_default() (pyscaffold.log.ColoredReportFormatter method), 102
format_default() (pyscaffold.log.ReportFormatter method), 103
format_next_version() (pyscaffold.contrib.setuptools_scm.version.ScmVersion method), 79
format_path() (pyscaffold.log.ReportFormatter method), 103
format_report() (pyscaffold.log.ReportFormatter method), 103
format_subject() (pyscaffold.log.ColoredReportFormatter method), 102
format_subject() (pyscaffold.log.ReportFormatter method), 103
format_target() (pyscaffold.log.ReportFormatter method), 103
format_version() (in module pyscaffold.contrib.setuptools_scm.version), 79
format_with() (pyscaffold.contrib.setuptools_scm.version.ScmVersion method), 79
formatter (pyscaffold.log.ReportLogger attribute), 103
from_file() (pyscaffold.contrib.setuptools_scm.config.Configuration class method), 76
from_potential_worktree() (pyscaffold.contrib.setuptools_scm.git.GitWorkdir class method), 77
function_has_arg() (in module pyscaffold--
Index

H

handler (pyscaffold.log.ReportLogger attribute), 103
has_command () (in module pyscaffold.contrib.setuptools_scm.utils), 78
has_option (pyscaffold.contrib.configupdater.ConfigUpdater method), 82
has_section (pyscaffold.contrib.configupdater.ConfigUpdater method), 82
hg_find_files (in module pyscaffold.contrib.setuptools_scm.file_finder_hg), 77

I

IncludeExtensions (class in pyscaffold.extensions.cirrus), 87
indent (pyscaffold.log.ReportLogger method), 104
index_url (pyscaffold.contrib.ptr.CustomizedDist attribute), 85
infer_version (in module pyscaffold.contrib.setuptools_scm.integration), 78
init (in module pyscaffold.templates), 95
init_color() (in module pyscaffold.termui), 108
init_commit_repo (in module pyscaffold.repo), 106
init_git (in module pyscaffold.api), 75
initialize_options (pyscaffold.contrib.ptr.PyTest method), 85
initialize_options (pyscaffold.integration.DeprecatedPyTest method), 101
install_dists (pyscaffold.contrib.ptr.PyTest method), 85
install_extra_dists (pyscaffold.contrib.ptr.PyTest method), 85
instruct_user (pyscaffold.extensions.pre_commit.PreCommit static method), 92
InvalidIdentifier, 99
invoke_action (in module pyscaffold.update), 109
is_dirty (pyscaffold.contrib.setuptools_scm.git.GitWorkdir method), 77
is_git_configured (in module pyscaffold.info), 100
is_git_installed (in module pyscaffold.info), 100
is_git_repo (in module pyscaffold.repo), 106
is_git_worktree_clean (in module pyscaffold.info), 100
is_pathname_valid (in module pyscaffold.utils), 111
Index

is_shallow() (pyscaffold.contrib.setupools_scm.git.GitWorkdir method), 77
is_valid_identifier() (in module pyscaffold.utils), 112
isatty() (in module pyscaffold.termui), 108
isortCfg() (in module pyscaffold.templates), 95
iter_matching_entrypoints() (in module pyscaffold.contrib.setuptools_scm.discover), 76

L
level (pyscaffold.log.ReportLogger attribute), 104
levenshtein() (in module pyscaffold.utils), 112
license() (in module pyscaffold.templates), 95
licenses (in module pyscaffold.templates), 95
list_actions() (in module pyscaffold.cli), 98
localize_path() (in module pyscaffold.utils), 112
LOG_STYLES (pyscaffold.log.ColoredReportFormatter attribute), 102
logger (in module pyscaffold.api.helpers), 70
logger (in module pyscaffold.log), 105

M
main() (in module pyscaffold.cli), 98
make_valid_identifier() (in module pyscaffold.utils), 112
marker_passes() (pyscaffold.contrib.ptr.PyTest static method), 85
merge() (in module pyscaffold.api.helpers), 70
meta() (in module pyscaffold.contrib.setupools_scm.version), 79
MissingSectionHeaderError, 81
MissingTemplate, 87
modify() (in module pyscaffold.api.helpers), 70
move() (in module pyscaffold.utils), 112
move_old_package() (in module pyscaffold.extensions.namespace), 91
mutually_exclusive (pyscaffold.api.Extension attribute), 73
mutually_exclusive (pyscaffold.extensions.cookiecutter.Cookiecutter attribute), 87
mutually_exclusive (pyscaffold.extensions.django.Django attribute), 89

N
Namespace (class in pyscaffold.extensions.namespace), 90

Index 123
pyscaffold.integrate (module), 101
pyscaffold (module), 102
pyscaffold.repo (module), 105
pyscaffold.shell (module), 106
pyscaffold.structure (module), 107
pyscaffold.templates (module), 93
pyscaffold.termui (module), 107
pyscaffold.update (module), 108
pyscaffold.utils (module), 110
pyscaffold.warnings (module), 113
pyscaffold_keyword (module), 101
PyScaffoldTooOld, 99
PyTest (class in pyscaffold.contrib.ptr), 85
read (pyscaffold.contrib.configupdater.ConfigUpdater method), 83
read_file (pyscaffold.contrib.configupdater.ConfigUpdater method), 83
read_setupcfg (in module pyscaffold.update), 109
read_string (pyscaffold.contrib.configupdater.ConfigUpdater method), 83
readme (in module pyscaffold.templates), 95
reconfigure (pyscaffold.log.ReportLogger method), 104
register (in module pyscaffold.api.helpers), 71
register (pyscaffold.api.Extension static method), 73
relative_to (pyscaffold.contrib.setuptools_scm.config.Configuration attribute), 76
release_branch_semver (in module pyscaffold.contrib.setuptools_scm.version), 79
release_branch_semver_version (in module pyscaffold.contrib.setuptools_scm.version), 79
remove_files (pyscaffold.extensions.no_skeleton.NoSkeleton method), 91
remove_option (pyscaffold.contrib.configupdater.ConfigUpdater method), 83
remove_section (pyscaffold.contrib.configupdater.ConfigUpdater method), 84
report (pyscaffold.log.ReportLogger method), 104
ReportFormatter (class in pyscaffold.log), 102
ReportLogger (class in pyscaffold.log), 103
requirements (in module pyscaffold.templates), 96
rm_rf (in module pyscaffold.utils), 113
Index 125

root (pyscaffold.contrib.setuptools_scm.config.Configuration, 76
run () (in module pyscaffold.cli), 98
run () (pyscaffold.contrib.ptr.PyTest method), 85
run_scaffold () (in module pyscaffold.cli), 98
run_tests () (pyscaffold.contrib.ptr.PyTest method), 85

S
scm_find_files () (in module pyscaffold.contrib), 86
scm_find_files () (in module pyscaffold.contrib.setuptools_scm.file_finder), 76
scm_get_local_dirty_tag () (in module pyscaffold.contrib), 86
scm_get_local_node_and_date () (in module pyscaffold.contrib), 86
scm_guess_next_dev_version () (in module pyscaffold.contrib), 86
scm_parse_archival () (in module pyscaffold.contrib), 86
scm_parse_git () (in module pyscaffold.contrib), 86
scm_parse_pkginfo () (in module pyscaffold.contrib), 86
scm_postrelease_version () (in module pyscaffold.contrib), 86
ScmVersion (class in pyscaffold.contrib.setuptools_scm.version), 79
SECTCRE (pyscaffold.contrib.configupdater.ConfigUpdater attribute), 82
sections () (pyscaffold.contrib.configupdater.ConfigUpdater method), 84
sections_blocks () (pyscaffold.contrib.configupdater.ConfigUpdater method), 84
set () (pyscaffold.contrib.configupdater.ConfigUpdater method), 84
setup_cfg () (in module pyscaffold.templates), 96
setup_py () (in module pyscaffold.templates), 96
setupools_scm_config () (in module pyscaffold.contrib.setuptools_scm_integration), 102
SetupoolsOutdatedWarning, 79
shell_command_error2exit_decorator () (in module pyscaffold.shell), 107
ShellCommand (class in pyscaffold.shell), 106
ShellCommandException, 99
simplified_semver_version () (in module pyscaffold.contrib.setuptools_scm.version), 79
skeleton () (in module pyscaffold.templates), 96
SPACING (pyscaffold.log.ReportFormatter attribute), 102

T
tag_regex (pyscaffold.contrib.setuptools_scm.config.Configuration attribute), 76
tag_to_version () (in module pyscaffold.contrib.setuptools_scm.version), 79
tags_to_versions () (in module pyscaffold.contrib.setuptools_scm.version), 79
TARGET_PREFIX (pyscaffold.log.ColoredReportFormatter attribute), 102
TARGET_PREFIX (pyscaffold.log.ReportFormatter attribute), 103
test_skeleton () (in module pyscaffold.templates), 97
to_dict () (pyscaffold.contrib.configupdater.ConfigUpdater method), 84
Tox (class in pyscaffold.extensions.tox), 92
tox () (in module pyscaffold.templates), 97
trace () (in module pyscaffold.contrib.setuptools_scm.utils), 78
trace_exception () (in module pyscaffold.contrib.setuptools_scm.utils), 78
Travis (class in pyscaffold.extensions.travis), 92
travis () (in module pyscaffold.templates), 97
travis_install () (in module pyscaffold.templates), 97
tzname () (pyscaffold.contrib.setuptools_scm.utils.UTC method), 78

U
unregister () (in module pyscaffold.api.helpers), 72
unregister () (pyscaffold.api.Extension static method), 73
update_file () (pyscaffold.contrib.configupdater.ConfigUpdater method), 84
update_pyscaffold_version () (in module pyscaffold.update), 109
UpdateNotSupported, 113
user_options (pyscaffold.contrib.ptr.PyTest attribute), 85
username() (in module pyscaffold.info), 101
UTC (class in pyscaffold.contrib.setuptools_scm.utils), 78
utcoffset() (pyscaffold.contrib.setuptools_scm.utils.UTC method), 78

\V
validate_format() (pyscaffold.contrib.configupdater.ConfigUpdater method), 84
verify_options_consistency() (in module pyscaffold.api), 75
verify_project_dir() (in module pyscaffold.api), 75
version2str() (in module pyscaffold.integration), 102
version_from_scm() (in module pyscaffold.contrib.setuptools_scm), 80
version_keyword() (in module pyscaffold.contrib.setuptools_scm.integration), 78
version_migration() (in module pyscaffold.update), 110

\W
warn_about_deprecated_pyscaffold() (in module pyscaffold.contrib), 86
warn_on_shallow() (in module pyscaffold.contrib.setuptools_scm.git), 77
wrapped (pyscaffold.log.ReportLogger attribute), 103
write() (pyscaffold.contrib.configupdater.ConfigUpdater method), 84
write_pbr_json() (in module pyscaffold.contrib), 86