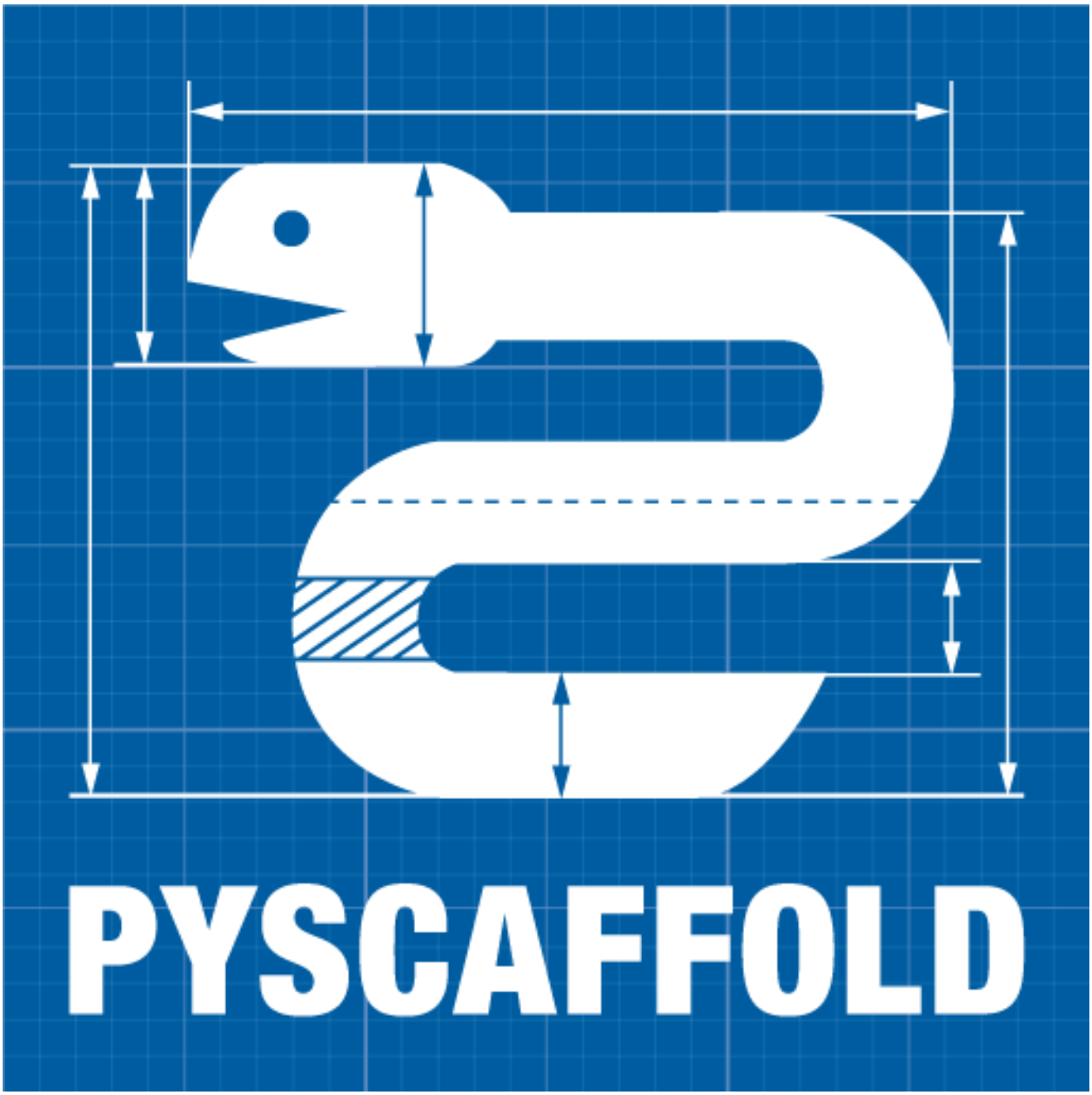


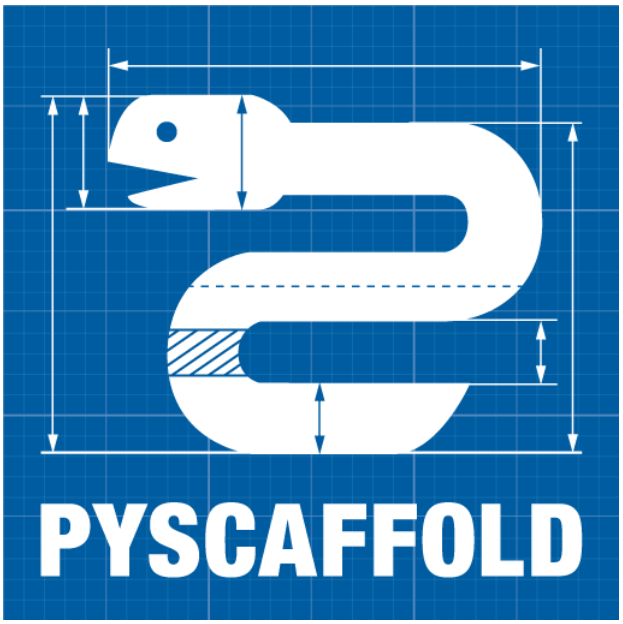
PyScaffold Documentation

Release 3.0.2.post0.dev8+g4dc74ae

Blue Yonder

Apr 16, 2018

Contents



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 is compatible with Python 2.7 and Python 3.4 and greater.

Note: Currently PyScaffold 3.0 needs at least Python 3.4 due to a bug in setuptools that only affects Python 2. For the time being use PyScaffold 2.5.* for Python 2.7 instead.

CHAPTER 1

Features

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

1.1 Configuration & Packaging

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`.

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:

```
putup 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 reside within your 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:

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

1.2 Complete 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.__version__`.

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

Build the documentation with `python setup.py docs` and run doctests with `python setup.py doctest` after you have Sphinx installed. 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.

1.4 Unittest & Coverage

Run `python setup.py test` to run all unittests defined in the subfolder `tests` with the help of py.test and pytest-runner. 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:

```
python setup.py test --addopts -h
```

to show the help of py.test.

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.

1.5 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.

1.6 Extensions

PyScaffold comes with several extensions:

- Create a Django project with the flag `--django` which is equivalent to `django-admin.py startproject my_project` enhanced by PyScaffold's features.
- 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*.

1.7 Easy Updating

Keep your project's scaffold up-to-date by applying `putput --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.

1.7.1 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.

1.7.2 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.

CHAPTER 2

Installation

2.1 Requirements

The installation of PyScaffold only requires a recent version of setuptools, i.e. at least version 30.3.0, 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 features like Django and cookiecutter support, 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 `python setup.py test` and `python setup.py docs` within your project, some additional requirements like py.test will be installed automatically as *egg*-files inside the `.eggs` folder. This is quite comfortable but 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 before you run `setup.py` commands like *doc* and *test*:

- Sphinx
- py.test
- pytest-cov

CHAPTER 3

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 *summary* 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 summary 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.

CHAPTER 4

Configuration

Projects set up with PyScaffold feature an easy package configuration with `setup.cfg`. Check out the example below as well as 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 :: 2
    Programming Language :: Python :: 2.7
    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
# Add here dependencies of your project (semicolon-separated)
install_requires = pandas; scikit-learn
# Add here test requirements (semicolon-separated)
tests_require = pytest; pytest-cov

[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

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

[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 =
    -p coverage
    --cov pyscaffold --cov-report term-missing
    --verbose
norecursedirs =
    dist
    build
    .tox

[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

[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

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 that your working tree is not dirty, i.e. all changes are committed and all important files are under version control.

Let's start:

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

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

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

2. Now change into `my_project` and move your old package folder into `src` with:

   ```
   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, or if you need to specify `entry_points`.

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. Where all modules moved correctly? Is there maybe some `__init__.py` file missing? After

these basic commands, try also to run `python setup.py docs` and `python setup.py test` to check that Sphinx and PyTest runs correctly.

CHAPTER 6

Extending PyScaffold

PyScaffold is carefully designed to cover the essentials of authoring and distributing Python packages. Most of time, tweaking `putup` options is enough to ensure proper configuration of a project. However, for advanced use cases 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 behavior 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.

6.1 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:

```
{
    '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:

```
{
    '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.

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

6.2 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:

```
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`.

6.3 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.

6.4 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:

```
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:
            struct, opts: updated project representation and options
        """
        ....
        return new_actions, new_opts
```

6.4.1 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.

6.4.2 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*

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 last two 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.

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

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

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

MY_AWESOME_FILE = """\
# -*- coding: utf-8 -*-
from __future__ import print_function

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

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

```
MY_AWESOME_TEST = """\
import pytest
from {qual_pkg}.awesome import awesome

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

class AwesomeFiles(Extension):
    """Adding some additional awesome files"""
    def activate(self, actions):
        return helpers.register(actions, self.define_awesome_files)

    def define_awesome_files(self, struct, opts):
        struct = helpers.merge(struct, {
            opts['project']: {
                opts['package']: {
                    'awesome.py': MY_AWESOME_FILE.format(**opts)
                },
                'tests': {
                    'awesome_test.py': (
                        MY_AWESOME_TEST.format(**opts),
                        helpers.NO_OVERWRITE
                    )
                }
            }
        })

        struct['.python-version'] = ('3.6.1', helpers.NO_OVERWRITE)

        for filename in ['awesome_file1', 'awesome_file2']:
            struct = helpers.ensure(
                struct, [opts['project'], '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}/awesome/{filename}'.format(
                #           filename=filename, **opts)

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

        # It is import 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,

6.4.3 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 `entry_points` variable in `setup.py`:

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

6.5 Examples

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

6.5.1 Namespace Extension

```
# -*- coding: utf-8 -*-
"""
Extension that adjust project file tree to include a namespace package.

This extension adds a **namespace** option to
:obj:`~pyscaffold.api.create_project` and provides correct values for the
options **root_pkg** and **namespace_pkg** to the following functions in the
action list.
"""
from __future__ import absolute_import

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

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


class Namespace(Extension):
    """Omit creation of skeleton.py"""
    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")

    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):
            # First ensure the extension function is stored inside the
            # 'extensions' attribute:
            extensions = getattr(namespace, 'extensions', [])
            extensions.append(obj_ref)
            setattr(namespace, 'extensions', extensions)

            # 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['namespace'] = utils.prepare_namespace(opts['namespace'])
        opts['root_pkg'] = opts['namespace'][0]
        opts['qual_pkg'] = ".".join([opts['namespace'][-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['namespace'][-1].split('.')
    base_struct = struct
    struct = base_struct[opts['project']]['src']
    pkg_struct = struct[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'
                '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
```

6.5.2 No Skeleton Extension

```
# -*- coding: utf-8 -*-
"""
Extension that omits the creation of file `skeleton.py`
"""

from ..api import Extension
from ..api import 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 = [opts['project'], 'src', opts['package'], 'skeleton.py']
        struct = helpers.reject(struct, file)
        file = [opts['project'], 'tests', 'test_skeleton.py']
        struct = helpers.reject(struct, file)
        return struct, opts
```

6.5.3 Cookiecutter Extension

```
# -*- coding: utf-8 -*-
"""
Extension that integrates cookiecutter templates into PyScaffold.
"""
from __future__ import absolute_import

import argparse

from ..api.helpers import register, logger
from ..api import Extension
from ..contrib.six import raise_from


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

        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 = register(actions, enforce_cookiecutter_options,
                           before='get_default_options')
```

```
        # `apply_update_rules` uses CWD information,
        # so it is better to prepend actions that modify it.
        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)
            setattr(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
    """
    try:
        from cookiecutter.main import cookiecutter
    except Exception as e:
        raise_from(NotInstalled, 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)
```

6.5.4 Django Extension

```
# -*- coding: utf-8 -*-
"""
Extension that creates a base structure for the project using django-admin.py.
"""
from __future__ import absolute_import

from .. import shell
from ..api import helpers
from ..api import Extension
from ..contrib.six import raise_from


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):
    """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.py

    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.py is not installed
    """
    try:
        shell.django_admin('--version')
    except Exception as e:
        raise raise_from(DjangoAdminNotInstalled, e)

    shell.django_admin('startproject', opts['project'],
                       log=True, pretend=opts.get('pretend'))

    return struct, opts


class DjangoAdminNotInstalled(RuntimeError):
    """This extension depends on the ``django-admin.py`` cli script."""

    DEFAULT_MESSAGE = ("django-admin.py is not installed, "
                       "run: pip install django")

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

6.5.5 Pre Commit Extension

```
# -*- coding: utf-8 -*-
"""
Extension that generates configuration files for Yelp `pre-commit`_.

.. _pre-commit: http://pre-commit.com
"""
from __future__ import absolute_import

from ..templates import pre_commit_config
from ..api import Extension
from ..api import helpers


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')

    def add_files(self, struct, opts):
        """Add .pre-commit-config.yaml 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 = {
            '.pre-commit-config.yaml': (
                pre_commit_config(opts), helpers.NO_OVERWRITE
            ),
        }

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

6.5.6 Tox Extension

```
# -*- coding: utf-8 -*-
"""
Extension that generates configuration files for the Tox test automation tool.
"""
from __future__ import absolute_import

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


class Tox(Extension):
    """Generate Tox 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')

    def add_files(self, struct, opts):
        """Add .tox.ini 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 = {
            'tox.ini': (tox_ini(opts), helpers.NO_OVERWRITE)
        }

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

6.5.7 Travis Extension

```
# -*- coding: utf-8 -*-
"""
Extension that generates configuration and script files for Travis CI.
"""
from __future__ import absolute_import

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


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`.
            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
```

6.5.8 GitLab-CI Extension

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

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


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
```

6.6 Conventions for Community Extensions

In order to make it easy to find PyScaffold extensions, community packages should be namespaced as in `pyscaffoldext.${EXTENSION}` (where `${EXTENSION}` 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:

```
putput pyscaffoldext-${EXTENSION} -p ${EXTENSION} --namespace pyscaffoldext
```

6.7 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 *No Skeleton 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.

CHAPTER 7

Embedding PyScaffold

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:

```
import logging

from pyscaffold.api import create_project
from pyscaffold.extenstions import tox, travis, 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.extend_project,
                           travis.extend_project,
                           namespace.extend_project])
```

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.

CHAPTER 8

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 features:

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

Note: 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*.

8.1 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 package folder instead of packaging or distributing, since PyScaffold already handles it under-the-hood. However, the following files can be safely generated by a template (will not be overwritten):

```
<project root>/docs/index.rst
<project root>/tests/conftest.py
<project root>/README.rst
<project root>/AUTHORS.rst
<project root>/LICENSE.txt
<project root>/CHANGES.rst
<project root>/setup.cfg
<project root>/requirements.txt
```

```
<project root>/test-requirements.txt
<project root>/.coveragerc
```

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:

```
author
email
project_name
package_name
project_short_description
```

Accordingly, the template file structure should be similar to:

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

See Cookiecutter for more information about template creation.

CHAPTER 9

Contributing

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

9.1 Chat

Join our chat to get in direct contact with the developers of PyScaffold.

9.2 Issue Reports

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

9.3 Code Contributions

9.3.1 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.

9.3.2 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
```

9.3.3 Clone the repository

1. Create a Gitub 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. Create a branch to hold your changes:

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

   and start making changes. Never work on the master branch!
6. 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
   ```

7. 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.
8. Use flake8 to check your code style.
9. Add yourself to the list of contributors in `AUTHORS.rst`.
10. 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.

9.4 Release

As a PyScaffold maintainer following steps are needed to release a new version:

1. Make sure all unit tests on Travis 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 test_release` 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 then 500KB, otherwise unwanted clutter may have been included.
5. Run `python setup.py release` and that everything was uploaded to PyPI correctly.

CHAPTER 10

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 in a good out-of-the-box experience for developing distributable Python packages (exclusively). The idea is to standardize the structure of Python packages. The long-term goal is that PyScaffold becomes for Python what Cargo is for Rust.

   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 3.0 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!"*).

CHAPTER 11

License

```
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.
```

CHAPTER 12

Contributors

- Florian Wilhelm <Florian.Wilhelm@gmail.com>
- Felix Wick <Felix.Wick@blue-yonder.com>
- Holger Peters <Holger.Peters@blue-yonder.com>
- Uwe Korn <Uwe.Korn@blue-yonder.com>
- Patrick Mühlbauer <Patrick.Muehlbauer@blue-yonder.com>
- Florian Rathgeber <florian.rathgeber@gmail.com>
- Eva Schmücker <email@evaschmuecker.de>
- Tim Werner <tim.werner@blue-yonder.com>
- Julian Gethmann <julian.gethmann@kit.edu>
- Will Usher <william.usher@ouce.ox.ac.uk>
- Anderson Bravalheri <andersonbravalheri@gmail.com>
- David Hilton <david.hilton.p@gmail.com>
- Pablo Aguiar <scorphus@gmail.com>
- Vicky C Lau <vcl2@sfu.ca>
- Reuven Podmazo <reuven.podmazo@gmail.com>

CHAPTER 13

Changelog

13.1 Current versions

13.1.1 Version 3.0.3, 2018-04-14

- Set install_requires to setuptools>=30.3.0

13.1.2 Version 2.5.11, 2018-04-14

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

13.2 Older versions

13.2.1 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

13.2.2 Version 2.5.10, 2018-03-21

- Updated setuptools_scm to version 1.17.0

13.2.3 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

13.2.4 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

13.2.5 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 3.0 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.

13.2.6 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

13.2.7 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

13.2.8 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

13.2.9 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

13.2.10 Version 2.5.4, 2016-02-10

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

13.2.11 Version 2.5.3, 2016-01-16

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

13.2.12 Version 2.5.2, 2016-01-02

- Fix `is_git_installed`

13.2.13 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

13.2.14 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

13.2.15 Version 2.4.4, 2015-10-29

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

13.2.16 Version 2.4.3, 2015-10-27

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

13.2.17 Version 2.4.2, 2015-09-16

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

13.2.18 Version 2.4.1, 2015-09-09

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

13.2.19 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*.

13.2.20 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.

13.2.21 Version 2.2.1, 2015-06-18

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

13.2.22 Version 2.2, 2015-06-01

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

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

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

13.2.23 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

13.2.24 Version 2.0.4, 2015-03-17

- Typo and wrong Sphinx usage in the RTD documentation

13.2.25 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

13.2.26 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)

13.2.27 Version 2.0.1, 2015-02-27

- FIX: Installation bug under Windows 7

13.2.28 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

13.2.29 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

13.2.30 Version 1.3.2, 2014-12-02

- Fix: path of Travis install script

13.2.31 Version 1.3.1, 2014-11-24

- Fix: –with-tox tuple bug #28

13.2.32 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

13.2.33 Version 1.2, 2014-10-13

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

13.2.34 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

13.2.35 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

13.2.36 Version 0.9, 2014-07-27

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

13.2.37 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*

13.2.38 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

13.2.39 Version 0.6.1, 2014-05-15

- Fix: Removed six dependency in setup.py

13.2.40 Version 0.6, 2014-05-14

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

13.2.41 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

13.2.42 Version 0.4.1, 2014-04-27

- Problem fixed with pytest-cov installation

13.2.43 Version 0.4, 2014-04-23

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

13.2.44 Version 0.3, 2014-04-18

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

13.2.45 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

13.2.46 Version 0.1.2, 2014-04-10

- Bugfix in Manifest.in
- Python 2.6 problems fixed

13.2.47 Version 0.1.1, 2014-04-10

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

13.2.48 Version 0.1, 2014-04-03

- First release

CHAPTER 14

pyscaffold

14.1 pyscaffold package

14.1.1 Subpackages

pyscaffold.api package

Submodules

pyscaffold.api.helpers module

Useful functions for manipulating the action list and project structure.

`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`** (*`str or list`*) – file path relative to the structure root. The directory separator should be `/` (forward slash) if present. Alternatively, a list with the parts of the path can be provided, ordered from the structure root to the file itself. The following examples are equivalent:

```
'docs/api/index.html'
['docs', 'api', 'index.html']
```

- **content** (*str*) – file text contents
- **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.

pyscaffold.api.helpers.**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.api.helpers.**logger = <pyscaffold.log.ReportLogger object>**

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

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.

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:

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

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

```
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.

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** (*str or list*) – file path relative to the structure root. The directory separator should be `/` (forward slash) if present. Alternatively, a list with the parts of the path can be provided, ordered from the structure root to the file itself. The following examples are equivalent:

```
'docs/api/index.html'
['docs', 'api', 'index.html']
```

Returns modified project tree representation

Return type dict

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.

class pyscaffold.api.**Extension**(*name*)
Bases: object

Base class for PyScaffold's extensions

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 list of actions

Return type list

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
- **`extensions.`** –

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.contrib package

Subpackages

pyscaffold.contrib.setuptools_scm package

Submodules

pyscaffold.contrib.setuptools_scm.discover module

pyscaffold.contrib.setuptools_scm.discover.**iter_matching_entrypoints**(*path*, *entrypoint*)

pyscaffold.contrib.setuptools_scm.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*)

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.**list_files_in_archive**(*path*)

List the files that 'git archive' generates.

pyscaffold.contrib.setuptools_scm.git.**parse**(*root*, *describe_command='git describe –dirty –tags –long –match *.*'*, *pre_parse=<function warn_on_shallow>*)

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.**parse_pip_egg_info**(*root*)

pyscaffold.contrib.setuptools_scm.hacks.**parse_pkginfo**(*root*)

pyscaffold.contrib.setuptools_scm.hg module

pyscaffold.contrib.setuptools_scm.hg.**archival_to_version**(*data*)

pyscaffold.contrib.setuptools_scm.hg.**parse**(*root*)

pyscaffold.contrib.setuptools_scm.hg.**parse_archival**(*root*)

pyscaffold.contrib.setuptools_scm.integration module

pyscaffold.contrib.setuptools_scm.integration.**find_files**(*path='.'*)

pyscaffold.contrib.setuptools_scm.integration.**version_keyword**(*dist*, *keyword*, *value*)

pyscaffold.contrib.setuptools_scm.utils module

utils

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.**has_command**(*name*)

pyscaffold.contrib.setuptools_scm.utils.**trace**(**k*)

pyscaffold.contrib.setuptools_scm.version module

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

Bases: object

exact

format_choice(*clean_format*, *dirty_format*)

format_with(*fmt*)

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.**guess_next_dev_version**(*version*)

pyscaffold.contrib.setuptools_scm.version.**guess_next_version**(*tag_version*, *distance*)

pyscaffold.contrib.setuptools_scm.version.**meta**(*tag*, *distance=None*, *dirty=False*, *node=None*, *preformatted=False*, ***kw*)

pyscaffold.contrib.setuptools_scm.version.**postrelease_version**(*version*)

pyscaffold.contrib.setuptools_scm.version.**tag_to_version**(*tag*)

pyscaffold.contrib.setuptools_scm.version.**tags_to_versions**(*tags*)

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*, *parse=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.ptr module

Copyright Jason R. Coombs

MIT-licenced

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 respects allow_hosts and index_url.

index_url = None

class pyscaffold.contrib.ptr.**PyTest**(*dist*, ***kw*)

Bases: setuptools.command.test.test

```
>>> import setuptools
>>> dist = setuptools.Distribution()
>>> cmd = PyTest(dist)
```

finalize_options()

initialize_options()

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.

static paths_on_pythonpath(*paths*)
Backward compatibility for paths_on_pythonpath; Returns a null context if paths_on_pythonpath is not implemented in orig.test. Note that this also means that the paths iterable is never consumed, which incidentally means that the None values from dist.fetch_build_eggs in older Setuptools will be disregarded.

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

run_tests()
Invoke pytest, replacing argv.

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

pyscaffold.contrib.ptr.**null**()

pyscaffold.contrib.six module

Utilities for writing code that runs on Python 2 and 3

class pyscaffold.contrib.six.**Module_six_moves_urllib**
Bases: module

Create a six.moves.urllib namespace that resembles the Python 3 namespace

error = <module 'pyscaffold.contrib.six.moves.urllib.error'>

parse = <module 'pyscaffold.contrib.six.moves.urllib_parse'>

request = <module 'pyscaffold.contrib.six.moves.urllib.request'>

response = <module 'pyscaffold.contrib.six.moves.urllib.response'>

robotparser = <module 'pyscaffold.contrib.six.moves.urllib.robotparser'>

class pyscaffold.contrib.six.**Module_six_moves_urllib_error**(*name*)
Bases: pyscaffold.contrib.six._LazyModule

Lazy loading of moved objects in six.moves.urllib_error

ContentTooShortError

HTTPError

URLError

class pyscaffold.contrib.six.**Module_six_moves_urllib_parse**(*name*)
Bases: pyscaffold.contrib.six._LazyModule

Lazy loading of moved objects in six.moves.urllib_parse

ParseResult

SplitResult

parse_qs

parse_qsl

quote

quote_plus

splitquery

splittag

splituser

splitvalue

unquote

unquote_plus

unquote_to_bytes

urldefrag

urlencode

urljoin

urlparse

urlsplit

urlunparse

urlunsplit

uses_fragment

uses_netloc

uses_params

uses_query

uses_relative

class pyscaffold.contrib.six.**Module_six_moves_urllib_request**(*name*)

Bases: pyscaffold.contrib.six._LazyModule

Lazy loading of moved objects in six.moves.urllib_request

AbstractBasicAuthHandler

AbstractDigestAuthHandler

BaseHandler

CacheFTPHandler

FTPHandler

FancyURLopener

FileHandler

HTTPBasicAuthHandler

HTTPCookieProcessor

HTTPDefaultErrorHandler

HTTPDigestAuthHandler

HTTPErrorProcessor

HTTPHandler

HTTPPasswordMgr

HTTPPasswordMgrWithDefaultRealm

HTTPRedirectHandler

HTTPSHandler

OpenerDirector

ProxyBasicAuthHandler

ProxyDigestAuthHandler

ProxyHandler

Request

URLopener

UnknownHandler

build_opener

getproxies

install_opener

parse_http_list

parse_keqv_list

pathname2url

proxy_bypass

url2pathname

urlcleanup

urlopen

urlretrieve

class pyscaffold.contrib.six.**Module_six_moves_urllib_response**(*name*)
Bases: pyscaffold.contrib.six._LazyModule

Lazy loading of moved objects in six.moves.urllib_response

addbase

addclosehook

addinfo

addinfourl

class pyscaffold.contrib.six.**Module_six_moves_urllib_robotparser**(*name*)
Bases: pyscaffold.contrib.six._LazyModule

Lazy loading of moved objects in six.moves.urllib_robotparser

RobotFileParser

class pyscaffold.contrib.six.**MovedAttribute**(*name*, *old_mod*, *new_mod*, *old_attr=None*, *new_attr=None*)
Bases: pyscaffold.contrib.six._LazyDescr

class pyscaffold.contrib.six.**MovedModule**(*name*, *old*, *new=None*)
Bases: pyscaffold.contrib.six._LazyDescr

pyscaffold.contrib.six.**add_metaclass**(*metaclass*)
Class decorator for creating a class with a metaclass.

pyscaffold.contrib.six.**add_move**(*move*)
Add an item to six.moves.

pyscaffold.contrib.six.**assertCountEqual**(*self*, **args*, ***kwargs*)

pyscaffold.contrib.six.**assertRaisesRegex**(*self*, **args*, ***kwargs*)

pyscaffold.contrib.six.**assertRegex**(*self*, **args*, ***kwargs*)

pyscaffold.contrib.six.**b**(*s*)
Byte literal

pyscaffold.contrib.six.**create_unbound_method**(*func*, *cls*)

pyscaffold.contrib.six.**get_unbound_function**(*unbound*)
Get the function out of a possibly unbound function

pyscaffold.contrib.six.**int2byte**()
S.pack(v1, v2, ...) -> bytes

Return a bytes object containing values v1, v2, ... packed according to the format string S.format. See help(struct) for more on format strings.

pyscaffold.contrib.six.**iteritems**(*d*, ***kw*)
Return an iterator over the (key, value) pairs of a dictionary.

pyscaffold.contrib.six.**iterkeys**(*d*, ***kw*)
Return an iterator over the keys of a dictionary.

pyscaffold.contrib.six.**iterlists**(*d*, ***kw*)
Return an iterator over the (key, [values]) pairs of a dictionary.

pyscaffold.contrib.six.**itervalues**(*d*, ***kw*)
Return an iterator over the values of a dictionary.

pyscaffold.contrib.six.**python_2_unicode_compatible**(*klass*)
A decorator that defines __unicode__ and __str__ methods under Python 2. Under Python 3 it does nothing.

To support Python 2 and 3 with a single code base, define a __str__ method returning text and apply this decorator to the class.

pyscaffold.contrib.six.**raise_from**(*value*, *from_value*)

pyscaffold.contrib.six.**remove_move**(*name*)
Remove item from six.moves.

pyscaffold.contrib.six.**reraise**(*tp*, *value*, *tb=None*)
Reraise an exception.

pyscaffold.contrib.six.**u**(*s*)
Text literal

pyscaffold.contrib.six.**with_metaclass**(*meta*, **bases*)
Create a base class with a metaclass.

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 v1.17.0
2. six 1.11.0
3. pytest-runner 3.0

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.cookiecutter module

Extension that integrates cookiecutter templates into PyScaffold.

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 cookie cutter 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.py.

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.py is not installed, run: pip install django'*, **args*, ***kwargs*)
Bases: RuntimeError

This extension depends on the `django-admin.py` cli script.

DEFAULT_MESSAGE = 'django-admin.py is not installed, run: pip install django'

pyscaffold.extensions.django.**create_django_proj**(*struct*, *opts*)
Creates a standard Django project with django-admin.py

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.py 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

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*

Omit creation of skeleton.py

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.**PreCommit**(*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

add_files(*struct*, *opts*)
Add .pre-commit-config.yaml 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.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*)
Retrieve the template by name

Parameters **name** – name of template

Returns template

Return type string.Template

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

Parameters **opts** – mapping parameters as dictionary

Returns file content as string

Return type str

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**(*opts*)
Template of requirements.txt

Parameters **opts** – mapping parameters as dictionary

Returns file content as string

Return type str

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

Parameters **opts** – mapping parameters as dictionary

Returns file content as string

Return type str

pyscaffold.templates.**setup_py**(*opts*)
Template of setup.py

Parameters **opts** – mapping parameters as dictionary

Returns file content as string

Return type str

pyscaffold.templates.**skeleton**(*opts*)
Template of skeleton.py defining a basic console script

Parameters **opts** – mapping parameters as dictionary

Returns file content as string

Return type str

pyscaffold.templates.**sphinx_authors**(*opts*)
Template of authors.rst

Parameters **opts** – mapping parameters as dictionary

Returns file content as string

Return type str

pyscaffold.templates.**sphinx_changelog**(*opts*)
Template of changelog.rst

Parameters **opts** – mapping parameters as dictionary

Returns file content as string

Return type str

pyscaffold.templates.**sphinx_conf**(*opts*)
Template of conf.py

Parameters **opts** – mapping parameters as dictionary

Returns file content as string

Return type str

pyscaffold.templates.**sphinx_index**(*opts*)
Template of index.rst

Parameters **opts** – mapping parameters as dictionary

Returns file content as string

Return type str

pyscaffold.templates.**sphinx_license**(*opts*)
Template of license.rst

Parameters **opts** – mapping parameters as dictionary

Returns file content as string

Return type str

pyscaffold.templates.**sphinx_makefile**(*opts*)
Template of Sphinx's Makefile

Parameters **opts** – mapping parameters as dictionary

Returns file content as string

Return type str

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

pyscaffold.templates.**tox**(*opts*)
Template of tox.ini

Parameters **opts** – mapping parameters as dictionary

Returns file content as string

Return type str

pyscaffold.templates.**travis**(*opts*)
Template of .travis.yml

Parameters **opts** – mapping parameters as dictionary

Returns file content as string

Return type str

pyscaffold.templates.**travis_install**(*opts*)
Template of travis_install.sh

Parameters **opts** – mapping parameters as dictionary

Returns file content as string

Return type str

14.1.2 Submodules

14.1.3 pyscaffold.cli module

Command-Line-Interface of PyScaffold

pyscaffold.cli.**add_default_args**(*parser*)
Add the default options and arguments to the CLI parser.

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

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

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

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

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

pyscaffold.cli.**parse_args**(*args*)
Parse command line parameters

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

Returns command line parameters

Return type dict

pyscaffold.cli.**run**()
Entry point for console script

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

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

14.1.4 pyscaffold.exceptions module

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

exception pyscaffold.exceptions.**ActionNotFound**(*name*, **args*, ***kwargs*)
Bases: KeyError

Impossible to find the required action.

exception pyscaffold.exceptions.**DirectoryAlreadyExists**
Bases: RuntimeError

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

exception pyscaffold.exceptions.**DirectoryDoesNotExist**
Bases: RuntimeError

No directory was found to be updated.

exception pyscaffold.exceptions.**GitNotConfigured**(*message='Make sure git is configured. Run:n git config –global user.email "you@example.com"n git config –global user.name "Your Name"nto set your account's default identity.'*, **args*, ***kwargs*)

Bases: RuntimeError

PyScaffold tries to read user.name and user.email from git config.

DEFAULT_MESSAGE = 'Make sure git is configured. Run:\n git config --global user.email

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 (<30.3.0). Use 'pip install -U setuptools' to upgrade.nIf 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 (>= 12).

DEFAULT_MESSAGE = 'Your setuptools version is too old (<30.3.0). Use `pip install -U s

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 p

exception pyscaffold.exceptions.**ShellCommandException**(*message*, **args*, ***kwargs*)

Bases: RuntimeError

Outputs proper logging when a ShellCommand fails

14.1.5 pyscaffold.info module

Provide general information about the system, user etc.

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

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.**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

14.1.6 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.**PyTest**(*dist*, ***kw*)
Bases: *pyscaffold.contrib.ptr.PyTest*

run_tests()

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.**version2str**(*version*)
Creates a PEP440 version string

Parameters **version** (setuptools_scm.version.ScmVersion) – version object

Returns version string

Return type str

14.1.7 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 = defaultdict(<function ColoredReportFormatter.<lambda>>, {'remove':

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

LOG_STYLES = defaultdict(<class 'tuple'>, {'info': ('blue',), 'error': ('red',), 'cr

SUBJECT_STYLES = defaultdict(<class 'tuple'>, {'invoke': ('blue',)})

TARGET_PREFIX = '\x1b[35m\x1b[1mto\x1b[0m'

format_activity(*activity*)

format_default(*record*)

format_subject(*subject*, *activity=None*)

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 default a *ReportFormatter* is created and used.
- **extra** (*dict*) – extra attributes to be merged into the log record. Options, empty by default.

wrapped
logging.Logger – underlying logger object.

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

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

nesting
int – current nesting level of the report.

copy()
Produce a copy of the wrapped logger.

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

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

Example

```
from pyscaffold.log import logger
logger.report('invoke', 'custom_action')
with logger.indent():
   logger.report('create', 'some/file/path')

# Expected logs:
# --------------------------------------
#       invoke  custom_action
#       create    some/file/path
# --------------------------------------
# Note how the spacing between activity and subject in the
# second entry is greater than the equivalent in the first one.
```

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

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:

```
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

pyscaffold.log.**logger = <pyscaffold.log.ReportLogger object>**
Default logger configured for PyScaffold.

14.1.8 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

14.1.9 pyscaffold.shell module

Shell commands like git, django-admin.py 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.py

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

14.1.10 pyscaffold.structure module

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

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)

pyscaffold.structure.**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**: when 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.structure.**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.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

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

Creates the project structure as dictionary of dictionaries

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)

14.1.11 pyscaffold.termui module

Basic support for ANSI code formatting.

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

14.1.12 pyscaffold.utils module

Miscellaneous utilities and tools

pyscaffold.utils.**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.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 `OSError` 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:

```
>>> 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.**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

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

pyscaffold.utils.**list2str**(*lst*, *indent=0*, *brackets=True*, *quotes=True*, *sep=', '*)
Generate a Python syntax list string with an indention

Parameters

- **lst** (*[str]*) – list of strings
- **indent** (*int*) – indention
- **brackets** (*bool*) – surround the list expression by brackets
- **quotes** (*bool*) – surround each item with quotes
- **sep** (*str*) – separator for each item

Returns string representation of the list

Return type str

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

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)
- **log** (*bool*) – log activity when true. Default: `False`.
- **pretend** (*bool*) – skip execution (but log) when pretending. Default `False`.

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.**utf8_decode**(*string*)

Decode a Python 2 str object to unicode for compatibility with Python 3

Parameters **string** (*str*) – Python 2 str object or Python 3 str object

Returns Python 2 unicode object or Python 3 str object

Return type str

pyscaffold.utils.**utf8_encode**(*string*)

Encode a Python 2 unicode object to str for compatibility with Python 3

Parameters **string** (*str*) – Python 2 unicode object or Python 3 str object

Returns Python 2 str object or Python 3 str object

Return type str

14.1.13 Module contents

CHAPTER 15

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

D

E

I

L

M

N

O

P

Q

R

S

T

U

V

W