Projy Documentation

Release 0.3

Stéphane Péchard

December 24, 2015

Contents

Projy is a template-based skeleton generator. In one command line, you can generate project skeletons like Python packages, LaTeX document or any file structure composed of directories and files.

Each file is generated by a different template. It uses the simple core templating system from Python, nothing fancy on that part. You can easily add new templates and new ways to collect data to insert in the created files. As much as possible, Projy tries to be simple to use and extend.

CHAPTER 1

Content

1.1 Installation

If you are familiar with Python, it is strongly suggested that you install Projy in virtualenv.

1.1.1 Pip and Distribute

To install Projy system-wide, just type:

```
$ sudo pip install projy
```

If no pip available, try `easy_install`:

```
$ sudo easy_install projy
```

1.1.2 Play the game

If you want to code, hack, enhance or just understand Projy, you can get the latest code at Github:

```
$ git clone http://github.com/stephanepechard/projy
```

Then create the local virtualenv and install Projy:

```
$ cd projy && source bootstrap && fab install
```

1.2 Usage

As an example, let's create a Python package. The Projy template mostly follows recommendations from The Hitchhiker's Guide to Packaging.

1.2.1 A Python package example

Use simply:

```
$ projy PythonPackage TowelStuff
```

In the same directory as you typed this command, you now have a *TowelStuff* directory, with the following structure:

```
TowelStuff/
    bin/
    bootstrap
    CHANGES.txt
    docs/
        index.rst
    LICENSE.txt
    MANIFEST.in
    README.txt
    setup.py
    towelstuff/
        __init__.py
```

Each file has been created with a specific template, so the package is fully functional, yet empty. Now, let's give a little explanation on each file. You can find further information here.

bin/, *docs/* and *towelstuff/* directories

Three directories are created by default:

- bin/ contains your package's scripts ;
- docs/, contains the documentation you write for the package. A primary *index.rst* file waits for you to write into it. Yes, it uses reStructuredText format.
- towelstuff/, is where you put the files of your package. It is the lower case version of the project name. By default, it already contains an empty *__init__.py* file.

See the links for more information.

bootstrap

This file is a little treat, not present in The Hitchhiker's Guide to Packaging. Using the `BootstrapScriptFileTemplate` template, it is a simple bash file creating a virtual environment easily. Use it with a simple:

```
$ source bootstrap
```

By default, it installs three packages from pypi:

- nose is "nicer testing for Python" ;
- pylint, a Python code static checker ;
- sphinx, the Python documentation generator.

Everything you need to write quality code :-) Of course, you can add any other package you may need, it's up to you. You can even externalize this list of package to a requirement file.

CHANGES.txt

The template of the CHANGES.txt file simply contains:

```
v<version>, <date> -- Initial release.
```

LICENSE.txt

By default, the Python package template contains the GPL v3 as *LICENSE.txt*. Change it as your convenience.

MANIFEST.in

The manifest is an important file that contains this:

```
include CHANGES.txt
include LICENSE.txt
include MANIFEST.in
include README.txt
recursive-include bin *
recursive-include docs *
recursive-include towelstuff *
```

README.txt

The usual README file, written in reStructuredText format.

setup.py

The setup.py file created from the template contains:

```python
# -*- coding: utf-8 -*-
""" $project setup.py script """

# system
from distutils.core import setup
from os.path import join, dirname


setup(
    name='TowelStuff',
    version='0.1.0',
    author='Stéphane Péchard',
    author_email='stephanepechard@provider.com',
    packages=['towelstuff','towelstuff.test'],
    url='http://',
    license='LICENSE.txt',
    long_description=open(join(dirname(__file__), 'README.txt')).read(),
    install_requires=[''],
    test_suite='towelstuff.test',
)
```

1.2.2 A more elaborate example: customizing the substitutions

You can modify the substitutions used by the template through the command line:

```
$ projy PythonPackage TowelStuff "author,Monty" "date,2012-06-18"
```

Then the substitutes `author` (normally get from git) and `date` (defaulted to the current day) are defined by the given values, not those computed by Projy. The format of such substitutions should be `"key,value"`. **Neither the key or the value should therefore include a comma.** Leading and trailing spaces are removed from both key and value.

To know which substitutions can be overwritten this way, use the `-i` option as described in the dedicated section. You can add substitutions that are not listed with the `-i` option but **they won't have any effect if the template file does not consider them.**

1.2.3 Options

Projy comes also with some useful command line option.

Listing templates

Type:

```
$ projy -l
```

and you'll see the list of available templates in your installation. That's an easy way to copy/paste the name of the template you want to use on the next command.

What's inside a template

Type:

```
$ projy -i PythonPackage
```

and you'll see the detailed structure of the `PythonPackage` template. It shows the created directories and files, with the substitutions included in the template.

1.3 Available templates

Here is a list of all the templates, but also collectors, integrated into Projy at the moment. Of course, you can propose new templates, they'll be integrated into Projy.

1.3.1 Project templates

Project templates are used to create a files/directories structure. That's the second argument of the command line. For this list, the projects we create are all called `TowelStuff`. They are somewhat ordered by the programming language they use.

LaTeX

These are LaTeX templates.

LaTeX book

The command:

```
$ projy LaTeXBook TowelStuff
```

produces:

```
TowelStuff/
    TowelStuff.tex          - LaTeXBookFileTemplate
    references.bib          - BibTeXFileTemplate
    Makefile                - LaTeXMakefileFileTemplate
```

Note: the `Makefile` uses Latexmk.

Python

These are Python templates.

Python package

The command:

```
$ projy PythonPackage TowelStuff
```

produces:

```
TowelStuff/
    bootstrap               - BootstrapScriptFileTemplate
    CHANGES.txt             - PythonPackageCHANGESFileTemplate
    LICENSE.txt             - GPL3FileTemplate
    MANIFEST.in             - PythonPackageMANIFESTFileTemplate
    README.txt              - READMEReSTFileTemplate
    setup.py                - PythonPackageSetupFileTemplate

TowelStuff/docs/
    index.rst

TowelStuff/towelstuff/
    __init__.py
```

Python script

The command:

```
$ projy PythonScript TowelStuff
```

produces:

```
TowelStuff/
    TowelStuff.py           - PythonScriptFileTemplate
```

Fabric file

The command:

```
$ projy Fabfile TowelStuff
```

produces:

```
/
    fabfile.py              - FabfileFileTemplate
```

This one is probably not generic enough, I added some stuff I use. Feel free to customize it.

Bootstrap

The command:

```
$ projy Bootstrap TowelStuff
```

produces:

```
/
    bootstrap                 - BootstrapScriptFileTemplate
```

Yes, the name has no impact on the produced file. Don't hesitate to make it short!

Projy itself!

Finally, a bit of a special template, which lets you create a Projy template and an empty file template from Projy itself. Call it meta if you want :-) See *Extending Projy* to know how such templates are meant to be written.

The command:

```
$ projy ProjyTemplate TowelStuff
```

produces:

```
/
    TowelStuffTemplate.py       - ProjyTemplateFileTemplate
    TowelStuffFileTemplate.txt
```

1.3.2 File templates

LaTeX files

- `BibTeXFileTemplate`
- `LaTeXBookFileTemplate`
- `LaTeXMakefileFileTemplate`

Python files

- `ProjyTemplateFileTemplate`
- `PythonPackageCHANGESFileTemplate`
- `PythonPackageMANIFESTFileTemplate`
- `PythonPackageSetupFileTemplate`
- `PythonScriptFileTemplate`
- `FabfileFileTemplate`

Bash files

- `BootstrapScriptFileTemplate`

Text files

- `READMEReSTFileTemplate`

Licenses

- `AGPL3FileTemplate`
- `ApacheLicenseFileTemplate`
- `BSDLicenseFileTemplate`
- `DWTFYWTPLFileTemplate`
- `GPL2FileTemplate`
- `GPL3FileTemplate`
- `LaTeX3LicenseFileTemplate`
- `LGPL3FileTemplate`
- `MITLicenseFileTemplate`
- `MPL2FileTemplate`
- `PythonLicense2FileTemplate`

1.3.3 Collectors

Here is the list of currently available collectors:

- `AuthorCollector`
- `AuthorMailCollector`

1.4 Extending Projy

Writing new templates and data collectors is easy. Let's continue reviewing our example.

1.4.1 Project templates

Here is the project template used to create a Python package:

```python
# -*- coding: utf-8 -*-
""" Projy template for PythonPackage. """

# system
from datetime import date
# parent class
from projy.templates.ProjyTemplate import ProjyTemplate
# collectors
```

```python
from projy.collectors.AuthorCollector import AuthorCollector
from projy.collectors.AuthorMailCollector import AuthorMailCollector


class PythonPackageTemplate(ProjyTemplate):
    """ Projy template class for PythonPackage. """

    def __init__(self):
        ProjyTemplate.__init__(self)


    def directories(self):
        """ Return the names of directories to be created. """
        directories_description = [
            self.project_name,
            self.project_name + '/' + self.project_name.lower(),
            self.project_name + '/docs',
        ]
        return directories_description


    def files(self):
        """ Return the names of files to be created. """
        files_description = [
            [ self.project_name,
              'bootstrap',
              'BootstrapScriptFileTemplate' ],
            [ self.project_name,
              'CHANGES.txt',
              'PythonPackageCHANGESFileTemplate' ],
            [ self.project_name,
              'LICENSE.txt',
              'GPL3FileTemplate' ],
            [ self.project_name,
              'MANIFEST.in',
              'PythonPackageMANIFESTFileTemplate' ],
            [ self.project_name,
              'README.txt',
              'READMEReSTFileTemplate' ],
            [ self.project_name,
              'setup.py',
              'PythonPackageSetupFileTemplate' ],
            [ self.project_name + '/' + self.project_name.lower(),
              '__init__.py',
              None ],
            [ self.project_name + '/docs',
              'index.rst',
              None ],
        ]
        return files_description


    def substitutes(self):
        """ Return the substitutions for the templating replacements. """
        author_collector = AuthorCollector()
        mail_collector = AuthorMailCollector()
        substitute_dict = dict(
            project = self.project_name,
            project_lower = self.project_name.lower(),
            date = date.today().isoformat(),
            author = author_collector.collect(),
            author_email = mail_collector.collect(),
        )
        return substitute_dict
```

To write a new template, you have to specify four parts:

- the name of the template, which is the name of the class ;
- the `directories`, `files` and `substitutes` functions.

When writing a new template, you can use the `self.project_name` variable which contains the name of the project as you typed it. In our example, it is `TowelStuff`.

Name of the template

Here it is simply `PythonPackageTemplate`. This is the name you type in the command line plus `Template` at the end. The created template inherits from the father of all templates, the `ProjyTemplate` class.

The *directories* function

`directories``()`

Returns a tuple containing all the names of the directories to be created.

Return type list of directory names

In our example, the created directories are `TowelStuff`, `TowelStuff/towelstuff` and `TowelStuff/docs`.

The *files* function

`files``()`

This function should return a tuple containing three informations for each file:

- the directory the file is in. It is defined as in *the directories function* ;
- the name of the file ;
- the template of the file, which is not the same as the project template. See *File templates*.

Return type list of file names

In our example, eight files are created:

- `bootstrap` created from `BootstrapScriptFileTemplate` ;
- `CHANGES.txt` created from `PythonPackageCHANGESFileTemplate` ;
- `LICENSE.txt` created from `GPL3FileTemplate` ;
- `MANIFEST.in` created from `PythonPackageMANIFESTFileTemplate` ;
- `README.txt` created from `READMEReSTFileTemplate` ;
- `setup.py` created from `PythonPackageSetupFileTemplate` ;

- `__init__.py` into the `TowelStuff/towelstuff` directory, created from `PythonPackageSetupFileTemplate` ;
- `index.rst` into the `TowelStuff/docs` directory, created empty.

Details on the content of each file is given on *Usage*.

The *substitutes* function

substitutes()

This function should return a dictionary containing the string substitutions used in the template.

Return type list of file names

In our example, the substitutions made in all the created files are:

- `$project` is replaced by `TowelStuff` ;
- `$project_lower` is replaced by `towelstuff` ;
- `$date` is replaced by the current date, in the format 2012-11-23 ;
- `$author` is replaced by what returns the `AuthorCollector` ;
- `$author_email` is replaced by what returns the `AuthorMailCollector` ;

1.4.2 File templates

From all the templated files we created, let's see how the `PythonPackageSetupFileTemplate` is made. Here is its content:

```
# -*- coding: utf-8 -*-
""" $project setup.py script """

# $project
from $project_lower import __version__

# system
try:
    from setuptools import setup
except ImportError:
    from distutils.core import setup
from os.path import join, dirname


setup(
    name=__version__,
    version='0.1.0',
    description='My $project project',
    author='$author',
    author_email='$author_email',
    packages=['$project_lower','$project_lower.test'],
    url='http://stephanepechard.github.com/projy',
    long_description=open('README.txt').read(),
    install_requires=[''],
    test_suite='$project_lower.test',
    classifiers=[
        'Development Status :: 3 - Alpha',
        'License :: OSI Approved :: GNU General Public License v3 (GPLv3)',
        'Programming Language :: Python',
    ],
)
```

It is simply the file you want to create with the variables that will be substitute in the creation process. Each variable should begin by `$` as described in the Template mechanism. Nothing fancy on this side, as you can see.

1.4.3 Data collectors

A data collector, as its name suggest, collects data. It is used by Projy to complete the *File templates*. Here is the data collector for the author data:

```python
# -*- coding: utf-8 -*-
""" AuthorCollector class
    Tries to find the program user name, as accuratly as possible.

    Put the functions alphabetical order in the same order as their importance.
    For example here, author_from_git should be taken before author_from_system
    as it is probably better.
"""

# system
import getpass
import os
from subprocess import Popen, PIPE, CalledProcessError
# parent class
from projy.collectors.Collector import Collector


class AuthorCollector(Collector):
    """ The AuthorCollector class. """

    def __init__(self):
        self.author = None


    def author_from_git(self):
        """ Get the author name from git information. """
        self.author = None
        try:
            # launch git command and get answer
            cmd = Popen(["git", "config", "--get", "user.name"], stdout=PIPE)
            stdoutdata = cmd.communicate()
            if (stdoutdata[0]):
                self.author = stdoutdata[0].rstrip(os.linesep)
        except ImportError:
            pass
        except CalledProcessError:
            pass
        except OSError:
            pass

        return self.author


    def author_from_system(self):
        """ Get the author name from system information.
            This is just the user name, not the real name.
        """
        self.author = getpass.getuser()
        return self.author
```

A data collector defines as many functions as necessary. In the case of the author, two ways of finding it are written. The first uses git. As many users of Projy would probably use it, chances are that its configuration will reflect the author's information. As a fallback in case git does not return the wanted data, the user name is taken as the system current user name. There are probably other methods to find it, so feel free to propose some more.

Functions are treated in the alphabetical order, which means that the most accurate functions should come before the least accurate ones. Of course, one may not always know what the most accurate way of finding a particular data is. Be smart then!

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Index