py-generic-project Documentation

Release 1.2.dev0

JÃ¼rgen Hermann

2021-07-16

Contents

This is a Cookiecutter template that creates a basic Python Setuptools project, which can be later on augmented with various optional accessories. See the demo for getting a 1 impression on how this Cookiecutter template can be used, including screenshots of the terminal session.

If you have questions or need any other kind of help, please join the springerle-users Google group.

CHAPTER 1

Features

The resulting project uses rituals and invoke for task automation, and setuptools for building and distributing the project. A provided autoenv script takes care of creating a fully boot-strapped Python 3 `venv` or Python 2 `virtualenv` – it can also be called manually if you don't want to install `autoenv`.

The `setup.py` script follows the DRY principle and tries to minimize repetition of project metadata by loading it from other places (like the package's `__init__.py`). Incidently, this makes the script almost identical between different projects, and thus provides an easy update experience later on. Usually, the only specific thing in it is the docstring with the project's name and license notice. This relies on conventions, especially check out __init__.py and __main__.py in the `src` folder, for their double-underscore meta variables.

It is also importable (by using the usual `if __name__ == '__main__'` idiom), and exposes the project's setup data in a `project` dict. This allows other tools to exploit the contained data assembling code, and again supports the DRY principle. The `rituals` package uses that to provide Invoke tasks that work for any project, based on its project metadata.

Other integrated tools are `pylint` for code quality checking, `pytest` for testing support, and a Travis CI configuration.

CHAPTER 2

Documentation Contents

2.1 Using the "py-generic-project" Template

2.1.1 Preparations

In case you don't have the `cookiecutter` command line tool yet, here's how to install it.

For `py-generic-project` v1.2 and upwards, you need at least `cookiecutter` v1.1, or v1.0 with degraded functionality – for `pipsi` installs, just issue a `pipsi upgrade cookiecutter` command and you're done.

2.1.2 Project Creation

Creating a new Python project based on this template goes like this (make sure you're in the directory you want your project added to):

```
cookiecutter "https://github.com/Springerle/py-generic-project.git"
```

It's advisable to `git add` the created directory directly afterwards, before any generated files are added, that you don't want to have in your repository.

Note: To get *your* defaults for common template values `cookiecutter` will ask you for when you use a template, it makes sense to have a ~/.cookiecutterrc in your home directory. Follow the link to see an example.

Also, you should at least check these files regarding their content and adapt them according to your needs:

- `project.d/classifiers.txt` – Add the correct categories (a/k/a Trove classifiers) for your project.
- `requirements.txt` – Add any Python packages you need for your project *at runtime*.

To bootstrap the project (as mentioned, best after `git add`), use these commands from within its directory:

```
. .env --yes --develop
inv ci | less -R
python -m $(./setup.py --name | tr -- - _) --help
```

On *Windows*, please install Babun to be able to use the same procedures as on a POSIX system – the installation process is easy and painless.

2.1.3 Requirements Handling

There are *three* files that define a project's dependencies: `dev-requirements.txt`, `test-requirements.txt`, and `requirements.txt`. The first lists tools that you typically need as a developer to work on the project. It also includes the other two, so *one* call to `pip install -r dev-requirements.txt` installs *all* of the project's dependencies for developer use.

`tox` uses only the test and install requirements in the virtualenvs it creates, because the tools aren't needed there (or if they are, they belong to the test ones).

`setup.py` loads these files into the `install_requires` and `tests_require` parameters as far as possible. Special lines like `-e ...` and similar are skipped, because only `pip` supports them; the idea here is to have none of those left at the time of a release. Note that `pytest` is always added to the test requirements, since the `setup.py test` sub-command is mapped to use `pytest` as the test runner. There is also an optional file `setup-requirements.txt` loaded into `setup_requires`, in case you need to use some *setuptools* extension. If you add that file, you should also include a matching `-r setup-requirements.txt` line at the end of `dev-requirements.txt`.

2.1.4 Feature Toggles

This template has a few options that can be turned on and off even after initial creation, which the following terminal session demonstrates for Travis CI support.

At the moment of this writing, those feature are `travis`, `flake8`, and `cli`. See the `features` value in `cookiecutter.json` for a current list.

Note that since the whole template is re-created, you should make sure that you have no pending changes in your working directory, i.e. everything is either safely committed or stashed away. After changing `project.d/cookiecutter.json` and the call to `invoke moar-cookies`, you should look at the diff, and `git add` any files that can just be updated (e.g. typically `.travis.yml`, `setup.py`, and some others).

Files with considerable changes you have to merge manually, e.g. by dumping a diff, resetting the affected files, reducing the diffs to the changes you really want, and then applying the edited diff. Note that the easiest way to do such a reset to the last commit is calling `git stash && git stash drop`.

Another option is to work with two directories, i.e. clone a copy of your project for the update process, perform the update, and then selectively copy changes to your main working directory. There might be a more stream-lined way applying some `git` magic, we'll see (ideas are welcome). Still this is better than wading through commit logs to catch up with an evolving template.

2.2 Packaging Python Software

This is a how-to for developers with directions on packaging their software in ways that enable a painless installation experience for end-users. Installing Python Software is the related end-user guide.

See also these other resources on the web...

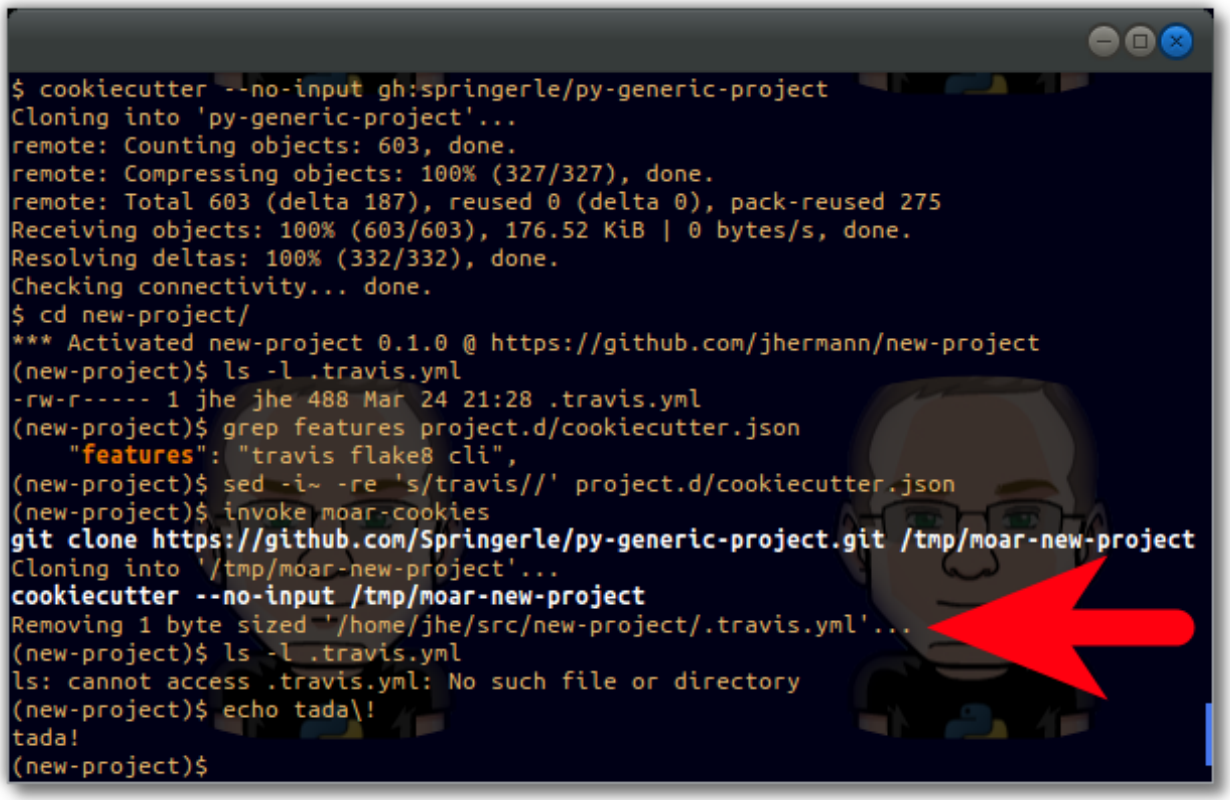

Fig. 1: Demo Terminal Session

- The Python Packaging User Guide
- The Hitchhiker's Guide to Python!
- The Sheer Joy of Packaging! – A Scipy 2018 tutorial, also covering *Conda*

The following figure gives a rough outline of what tools are involved in the development workflow. Regard services like GitLab and Artifactory as representatives, you can equally well use any git server solution and e.g. *devpi* as your artifact repository. There are also public services like GitHub, Travis, PyPI, and BinTray that can fill these roles for open-source projects.

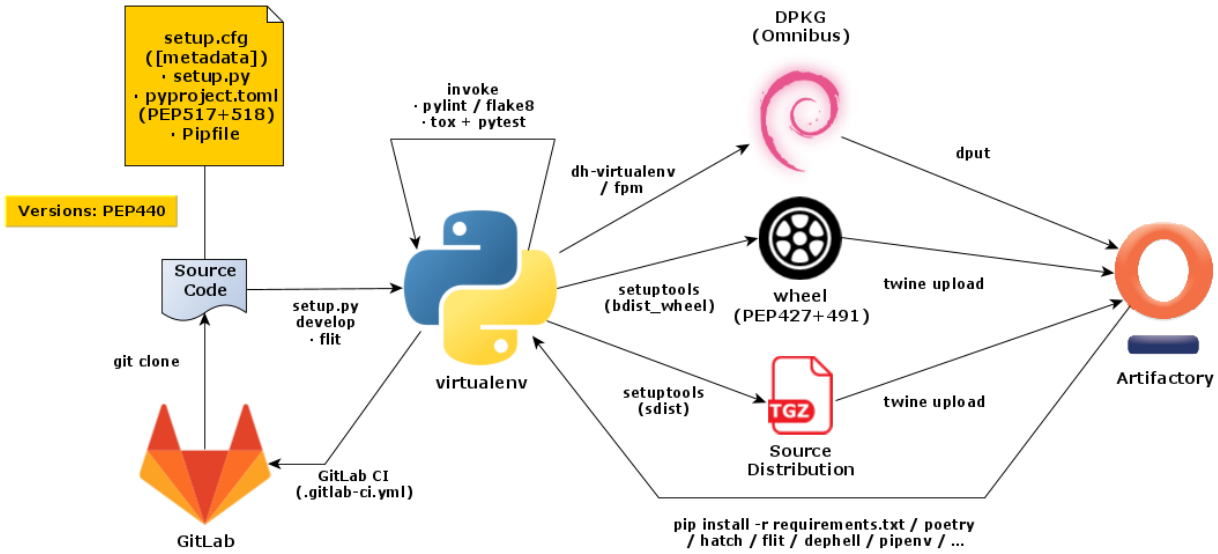

Fig. 2: Overview of Python Development Workflows

2.2.1 Packaging PyPI Releases

This is a short summary of essentals, consult the above resources for all the details. It covers the 'classic' tool-chain, there are more 'modern' tools like `poetry` and `flit` that serve similar purposes.

2.2.1.1 Building with setuptools

TODO

2.2.1.2 Packaging with wheel

TODO

2.2.1.3 Uploading with twine

Once you have your deployment artifacts ready (typically in a `dist` folder), you can upload them to `pypi.org`, or a local repository service.

There is a dedicated tool named twine for this. It supports using SSL for transfers, and also allows you to *first* build your artifacts, then test them as you see fit, and finally upload the *tested* artifacts.

Configuration is taken from `~/.pypirc`, or the environment – especially useful for CI jobs. A typical configuration might look like this:

```
[distutils]
index-servers = local pypi

[local]
repository: https://artifactory.local/artifactory/api/pypi/pypi-releases-local
username: «USER»
password: «API_TOKEN»

[pypi]
# repository: https://pypi.org/pypi
username: «PYPI_USER»
password: «PYPI_PWD»
```

You can select from the list of index servers by using `twine upload -r «repo» ...`, the default is `pypi`.

2.2.2 Building Zipapps (PEP 441)

Running Python code directly from ZIP archives is nothing new, PEP 273 made its debut in 2001, as part of Python 2.3 in the form of the `zipimport` module..

PEP 441 builds on this and describes mechanisms to bundle full applications into a single ZIP file that can be made executable. It was approved in 2015 and a first implementation appeared in Python 3.5 via the `zipapp` module.

See the PEP for details on how making a ZIP into an executable file works, but basically on POSIX systems the Python interpreter is called in a 'bang path' that is followed by the ZIP archive. The interpreter recognizes the 'script' is a whole application archive and acts accordingly. On Windows, zipapps *MUST* carry the `.pyz` extension which is bound to the `py` wrapper command, which in turn looks at the bang path and calls a matching Python interpreter from the installed set.

To display the bang path of a zipapp, use this command:

```
python3 -m zipapp --info foo.pyz
```

If you want to change the requested Python version to a newer one that is actually installed, change the bang path as part of the installation process:

```
python3 -m zipapp -p '/usr/bin/env python3.5' -o ~/bin/foo foo.pyz
```

This can also be done on an ad-hoc basis, by explicitly calling the desired interpreter:

```
python3.5 foo.pyz ...  # POSIX
py -3.5 foo.pyz ...  # Windows
```

Well-known tools to build new zipapps, outside of the Python core, are pex (Twitter) and shiv (LinkedIn). See their documentation for details on bundling your own applications, and also the next section on PEX.

2.2.3 Packaging Python EXecutables (PEX)

PEX files are **P**ython **Ex**ecutable ZIP files, a format that contains a full distribution of a Python application in a single archive (just like exectable JARs for Java). PEX files can be targeted at a specific platform and Python version, but might also support multiple runtime environments. See *Installation With PEX* for details on how to use them, and PEP 441 for a formal description of the underlying mechanics and all the details.

The Rituals task library for Invoke offers a `release.pex` task that performs all the necessary steps to create a PEX file. If you want to do it 'manually' or integrate it into another task runner, this is a concrete example:

```
pex -r requirements.txt . -c nanny \
    -o bin/nanny-0.1.0.dev5-cp27-none-linux_x86_64.pex
```

At the time of this writing, you need to install `pex 1.0.dev` directly from GitHub for the above to work.

2.3 Installing Python Software

This is a guide for end-users on how to easily install Python software on the major platforms. See *Packaging Python Software* for the related developer guide with distribution methods that enable this mostly painless installation experience.

2.3.1 TL;DR

This is a no-frills version of basic installation procedures for the three major PC platforms. Read the other sections for more details, especially if you encounter any problems with these condensed instructions. Once the basic setup is done, refer to either *Installing Releases From PyPI* or *Installing Directly From GitHub* to get an application installed – and in case the project author provides a **P**ython **Ex**ecutable archive, prefer an *Installation With PEX*.

On **Linux**, make sure you have the right version of *Python* pre-installed, and the basic developer toolset available. On Debian-like systems, the following makes sure of that:

```
sudo apt-get install python3 python3-setuptools python3-pkg-resources \
    python3-pip python3-dev libffi-dev build-essential git
```

On **Mac OS X**, install a modern *Python* tool chain and missing *GNU* utilities that are often needed by helper scripts:

```
sudo easy_install pip && sudo pip install virtualenv
brew install coreutils
```

For **Windows**, see the *Windows (python.org)* section. Developers and 'power users' with some existing Python and Linux experience might consider using Windows Subsystem for Linux (WSL), but that is outside the scope of this documentation. However, the POSIX workflows should work there.

Note: Keep in mind that the next step after the basic setup is either *Installing Releases From PyPI* or *Installing Directly From GitHub*. And that basic setup needs to be done only once.

2.3.2 Installing Python

There are different ways to get a working Python installation, depending on your computer's operating system. Note that Python 2.7 is by now increasingly unsupported, and Python 3.6 or above is the recommended version to use.

Read the documentation of any software you want to install regarding the versions of Python that particular software runs on, and act accordingly by e.g. calling `python3 -m venv` instead of just `virtualenv`.

See also these other resources on the web...

- Picking an Interpreter

2.3.2.1 POSIX (Linux, BSD, . . .)

On *POSIX systems*, use whatever package manager your distribution offers, and as the minimum install Python itself and everything to manage Python virtual environments. Usually, the Python interpreter is already installed, but some of the essential extensions and tools might be missing. For Debian-like systems, you need:

```
sudo apt-get install python3 python3-setuptools python3-pkg-resources python3-pip
```

If you need the same Python version on the stable and oldstable releases of Debian and Ubuntu, Ubuntu's Deadsnakes PPA is a means to achieve that. Python 3.6 is available from the PPA for Xenial (and Bionic comes with 3.6 by default), and you can also build the deadsnakes packages on Debian (3.6 builds on both Stretch and Buster).

Installing Extension Packages

To successfully install C extension packages like `lxml` from source into a virtual environment, you also need the necessary build tools like `gcc` or `clang`. On Debian-like systems, this means:

```
apt-get install python3-dev libffi-dev build-essential git
```

While the `wheel` format for binary distributions can make this unneccessary, there are practical limitations: wheels have to be built and uploaded to PyPI, which is seldom the case for every combination of packages and platforms. Also, wheels are not yet fully supported for POSIX at the time of this writing, so sometimes you have to install from source even if there is a pre-built wheel.

2.3.2.2 Windows (python.org)

To get the official *python.org* distribution on *Windows*, open the Python Releases for Windows page and select the appropriate version. You might want to install several Python 3 versions, to cover all possible needs of any applications – having them on one machine concurrently is no problem. Another officially supported way to get Python is the Windows Store, but at the time of this writing that is limited to Python 3.7+ and has no x86 support (for 32 bit architectures).

It's also recommended to install the Python Extensions for Windows, because many applications rely on them to access Windows-specific features.

Also note that where on a POSIX system `python3 ...` is used, that translates to `py -3 ...` on Windows.

2.3.2.3 Enabling Easy Zipapp Installs on Windows

Zipapps are a way to distribute Python applications and all of their dependencies in a single binary file, comparable to statically linked golang apps. Their main advantage is that distributing and installing them is quite simple. To learn more about zipapps, refer to *Building Zipapps (PEP 441)*.

On Windows, because there is no '+x' flag, things are a bit more complicated than on POSIX. Zipapps **MUST** have a `.pyz` extension, for which the `py` launcher is registered as the default application. The net effect is that such files become executable and are handed over to the launcher *if* you add a few environment settings to your machine.

In the user-specific environment settings, add a new `PATHEXT` variable (or extend an existing one), with the value `%PATHEXT%;.PYZ`. Also edit the `PATH` one and add a new `%LOCALAPPDATA%\bin` entry. Save everything (click "OK"), open a *new* command window, and verify the changes with

```
echo %PATHEXT% & echo %PATH%
```

Create the new bin directory by calling `md %LOCALAPPDATA%\bin`. Now you can place a zipapp file like `foo.pyz` in that directory, and it is immediately callable as `foo`.

If that makes more sense to you, you can change the system-wide variables instead of the user-specific ones, and choose paths that are global for all users (like `C:\usr\bin` or similar).

To make zipapps available network-wide, you can use `%APPDATA%` to store the zipapps, so you only have to maintain them once in case you regularly work on several machines in the same network.

2.3.2.4 Conda (Windows, Mac OS X, Linux)

Alternatively, there is also the *cross-platform*, Python-agnostic binary package manager Conda, with roots in the Scientific Python community and being part of the `Anaconda` data processing platform.

Miniconda is a minimal distribution containing only the Conda package manager and Python. Once Miniconda is installed, you can use the `conda` command to install any other packages and create environments (`conda` is the equivalent of `virtualenv` and `pip`).

2.3.2.5 RyRun (Mac OS X, Linux, FreeBSD)

Yet another contender is PyRun from *eGenix*. It is a one file Python runtime, that combines a Python interpreter with an almost complete Python standard library into a single easy-to-use executable of about 12 MiB in size. The selling point is easy installation by only handling a single file, which also results in easy relocation – ideal for using it on an USB stick for portable applications, or part of a self-contained bundle for server installations. It covers all the relevant Python versions (2.6, 2.7, and 3.4), and comes in 32bit and 64bit flavours.

From an application installation standpoint, *PyRun* allows you to efficiently create isolated runtime environments that include their own Python interpreter and standard library, i.e. are even more detached from the host setup than normal virtualenvs.

2.3.2.6 pyenv (Simple Python Version Management)

pyenv works for Mac OS X and POSIX systems and is a simple way to obtain access to Python versions that are not available from your system's software repositories, and switch between them at will.

See the pyenv installation instructions for details.

2.3.3 Installation With PEX

PEX files are **P**ython **Ex**ecutable ZIP files, a format that contains a full distribution of a Python application in a single archive (just like executable JARs for Java). PEX files can be targeted at a specific platform and Python version, but might also support multiple runtime environments. Consult the documentation of your application for further guidance.

Installing a PEX file is as easy as downloading it from the project's download page (e.g. *Bintray* or the *GitHub* releases section of a project), using your browser or `curl`, and then just start it from where you saved it to in your file system. On *Windows*, give the file a `.pyz` or `.pyzw` extension, which the *Python Launcher* is registered for. On POSIX systems, `chmod +x` the file to make it executable.

See PEP 441 and *Building Zipapps (PEP 441)* for a formal description of the underlying mechanics and all the details.

2.3.4 Installing Releases From PyPI

For releases published on PyPI, you should use `pip` to install them (i.e. do not use `easy_install` anymore). It's common procedure to not install into `/usr/local` on Linux, but instead create a so-called *virtualenv*, which is a runtime environment that is (by default) isolated against the host system and its packages, as well as against other

virtualenvs. This means that you don't have to carefully manage version numbers, you can let `pip` install exactly those versions an application works best with.

To create a virtualenv, go to the desired install location, and create the new environment, also giving it a name:

```
cd ~/.local/venvs
python3 -m venv ‹newenv›
. ‹newenv›/bin/activate
pip install -U pip setuptools  # get newest tooling
```

The third command *activates* the virtualenv, which means that when you call `python` or `pip`, they run in the context of that virtualenv.

Now all you have to do is call `pip install ‹my-new-app›` and it'll get installed into that environment. If the package provides command line tools, don't forget to add the `bin` directory to your `PATH` – or better yet symlink those commands into your `~/bin` directory or add some definitions to `~/.bash_aliases`, to make them selectively available.

To make this even simpler, dephell (via its concept of 'jails') allows installing and updating with a simple one-liner. And – at least on Linux – it also makes any exposed CLI tools immediately available in your PATH. `dephell jail` is just a convenient wrapper around `pip` and `venv`.

2.3.5 Installing Directly From GitHub

In case you *really* need the freshest source from GitHub, there are several ways to install a setuptools-enabled project from its repository. Be aware that this is nothing a casual user should really do, gain some experience using `virtualenv` and `pip` before trying this. The following shows different ways to get `pip` to download and install the source directly, with a single command.

- Via a ZIP archive download (does not need `git` installed):

  ```
  pip install "https://github.com/‹USER›/‹REPO-NAME›/archive/‹TAG-OR-SHA›.zip
  ↪#egg=‹PKG-NAME›"
  ```

  Usually, `‹TAG-OR-SHA›` will be `master` or `develop` – in the GitHub web UI, you can use the `branch` selector above the file listing to first select a branch, then the `Download ZIP` button at the bottom of the sidebar gives you the neccessary link.

- Via `git clone`:

  ```
  pip install "git+https://github.com/‹USER›/‹REPO-NAME›.git#egg=‹PKG-NAME›"
  ```

- Via `git clone` with a tag or hash:

  ```
  pip install "git+https://github.com/‹USER›/‹REPO-NAME›.git@‹TAG-OR-SHA›#egg=‹PKG-
  ↪NAME›"
  ```

- From a *working directory* you manually cloned into your file system:

  ```
  pip install "‹working-directory-path›"
  ```

- The forms that use `git+` or a `git` directory can also be done as an editable package – the difference is that the package will end up in a top-level `src` directory instead of the deeply nested `.../site-packages` one, and any changes to the source will be instantly visible to any process that imports it. When you plan to change the source or otherwise need quick access to it, that makes this easy:

```
pip install -e "git+....git#egg=‹PKG-NAME›"
```

Note that all these forms work in requirements files, which in the end are only lists of `pip install` arguments.

Tip: Use `python3 -m pip` or `python -m pip` instead of plain `pip` in case you have problems, or if you write automation scripts for unattended installations.

The advantage of this is that you always get the 'right' version of pip for the given interpreter, especially when you make that configurable and people provide 'exotic' Python executable paths.

2.4 Writing Sphinx Documentation

2.4.1 Overview

This is a directory of links to information and hints you need when you want to write (software) documentation using reStructuredText and Sphinx. Using them should improve your experience as an author as well as the end result for your readers.

Sphinx is a tool that makes it easy to create intelligent and beautiful documentation, and uses *reStructuredText* as its markup language. It was originally created for the new Python documentation, and thus has excellent facilities documenting Python projects, but is in no way limited to those.

Also visit Write the Docs, which is a place with high quality information about the art of writing documentation.

2.4.1.1 When to Use Sphinx?

Using *Sphinx* has several advantages over other options for writing documentation that has strong ties to the source code. It can be maintained and versioned together with the source, which increases the likelihood that you end up with *current* and *correct* documentation.

Sphinx was designed for that purpose – to write extensive 'prosa' documentation in addition to any in-source markup most languages offer (e.g. Javadoc), and shines when it comes to cross-linking within the documentation and into source code – for example, it's easy to refer to identifiers in your source by their name.

For a Python project, *Sphinx* is *the* obvious choice, but there are also extensions for Java and other languages (so-called *domains*). The generated output can be styled freely, and the *Sphinx* eco-system offers lots of documentation and code highlighting themes.

2.4.1.2 Feature Highlights

- **Output formats** – HTML (including Windows HTML Help), LaTeX (for printable PDF versions), Texinfo, manual pages, plain text.
- **Extensive cross-references** – Semantic markup and automatic links for functions, classes, citations, glossary terms and similar pieces of information.
- **Hierarchical structure** – Easy definition of a document tree, with automatic links to siblings, parents and children.
- **Automatic indices** – General index as well as a language-specific module indices.
- **Code handling** – Automatic highlighting using the Pygments highlighter.

- **Extensions** – Automatic testing of code snippets, inclusion of docstrings from Python modules (API docs), and more.

2.4.2 Introduction & Cheatsheets

- See the *How-Tos* section below for some quickstart advice.
- Sphinx reStructuredText primer – A brief introduction to *reStructuredText* (reST) concepts and syntax.
- Beautiful Docs – A collection of exemplary open source project documentation.
- Style guide for Sphinx-based documentations

2.4.3 Extensions & Tools

There are a lot of extensions, styles, themes, and so on available on the web. For example, see the reStructuredText tool support entry on *Stack Overflow*, or the Awesome Sphinx bookmark list on *GitHub*.

You should get a *reStructuredText* language definition enabling syntax highlighting in your favourite editor or IDE, see below for `gedit3` support.

Extensions

- PlantUML for Sphinx allows you to add PlantUML diagrams to your documentation.
- sphinxcontrib-programoutput inserts the output of arbitrary commands into documents, helping you to keep your command examples up to date.

Tools

- restview – A HTML viewer for reStructuredText documents that renders them on the fly.

gedit3

- reStructuredText preview and highlighting
  - For Python3 and gedit 3.8
  - For Python2
- gedit3 language definition for reStructuredText

2.4.4 How-Tos

2.4.4.1 Sphinx Installation and Setup

See *Installing Python Software* for the full story and all details, this is how to install *Sphinx* to your user account on a properly configured POSIX system (including *Babun* or *CygWin*):

```
venv=~/.local/venvs/sphinx
mkdir -p $(dirname $venv)
python3 -m venv $venv
$venv/bin/pip install -U pip
$venv/bin/pip install sphinx sphinx-autobuild
ln -nfs -t ~/.local/bin $venv/bin/sphinx-*
```

For a *Python* project, it makes sense to add *Sphinx* to the development requirements of the project, and install it to the project's virtualenv together with other tools. This makes you independent of the machine you build on, and also ensures that you always get the same version of *Sphinx*.

```
# Development requirements
Sphinx==2.2.2
sphinx-autobuild==0.7.1
sphinx-rtd-theme==0.4.2
```

2.4.4.2 Creating a Minimal Project

In your project directory, call `sphinx-quickstart` which will prompt you for required information. Answer the first question for a 'root path' with `docs`, and the others according to your project's needs. You will then find a working minimal Sphinx project in the `docs` folder – `git add` that immediately, before you build your documentation the first time.

To build a HTML rendering, go into `docs` and call `make html`. If all goes well, you'll find the root page of your documentation at `docs/_build/html/index.html` or `docs/_build/index.html` (with newer versions of Sphinx) – just open it with your browser.

If you use the current *Sphinx* version, the default theme is 'Alabaster'. Let's change that to the default theme used on *Read the Docs*, in `docs/conf.py`:

```
html_theme = 'sphinx_rtd_theme'
```

Call `make html` again and reload the page in your browser. You should see a difference.

2.4.4.3 Adding a New Chapter

To add a new chapter in its own file, create a file like `docs/chapter.rst` with the following content:

```
###############
My New Chapter
###############
```

Then add that file to the *toctree* of your `index.rst` file:

```
.. toctree::
   :maxdepth: 2

   chapter
```

Entries in a *toctree* are just filenames, but relative to the containing file, and without extension, so we end up with just `chapter` here.

Rebuild the docs and *"My New Chapter"* will be added to the sidebar.

See Sections in the Sphinx documentation regarding the markup for different heading levels.

2.4.4.4 Publishing Your Document

If you want to publish documentation for a project on *GitHub*, the easiest solution is Read the Docs (RTD), which is a hosting service that builds your Sphinx documentation on-the-fly based on commit triggers. That means you don't have to generate and upload anything, just commit any changes and they'll be published soon thereafter.

RTD also knows about versions (as long as you maintain them properly) and thus offers *both* the latest documentation from source as well as previously released versions. As with all these services, you log in with OAuth2 and just click on your project repository to activate building – it's *very* easy.

2.4.4.5 Automatic Preview

The best preview solution is sphinx-autobuild, which is a drop-in replacement for the `sphine-build` command. It starts a web-server bound to `localhost` that makes the documentation available, and also a watchdog that triggers a build as soon as you save any changes in your editor. Since only the part of the documentation that actually changed is rebuilt, this is usually very quick and you get a near-instant live-reload in your browser view via a Websocket connection.

If you use the rituals automation tasks library, starting `sphinx-autobuild` is as easy as...

```
invoke docs --watchdog --browse
```

This launches the daemon and waits for a complete startup, then opens a browser tab with the rendered documentation. Try to `touch docs/index.rst` and watch the activity indicator in your browser – or take a look into the `docs/watchdog.log` file.

2.4.4.6 Converting from Markdown to reST

If you have existing Markdown files you want to integrate into your documentation, the `pandoc` tool provides an easy way to convert into reST-style markup. To make it available on Debian-type system, just install the package of the same name.

Then a conversion can be done as follows:

```
pandoc --from markdown --to rst -o "‹file›.rst" "‹file›.md"
```

2.4.4.7 Adding a Custom Pygments Lexer to Sphinx

In order for Sphinx to load and recognize a custom lexer, two things are needed:

1. Add the package name of the lexer to the `extensions` list in `conf.py`. Of course, that package has to be importable, either by using a virtualenv or manipulating `sys.path`.
2. Give your lexer package a Setuptools `pygments.lexers` entry point.

Then use it in a `code-block` as if it were a built-in. That's all.

2.4.4.8 Automatic Click CLI References

If you implement CLI tools using the Click framework, you can generate a reference as part of your Sphinx documentaton using the sphinx-click extension, covering all the command line options and arguments.

The generated text is based on the information contained in the `--help` output, just formatted more prettily. Unlike manually written docs, it's always up to date by definition. All you need to do is adding `sphinx-click` to your requirements and the Sphinx configuration, and then create a new document file looking like this:

```
**********************
Complete CLI Reference
**********************

This is a full reference of the :command:`foobar` command,
with the same information as you get from using :option:`--help`.
It is generated from source code and thus always up to date.
See :doc:`usage` for a more detailed description.

```

(continues on next page)

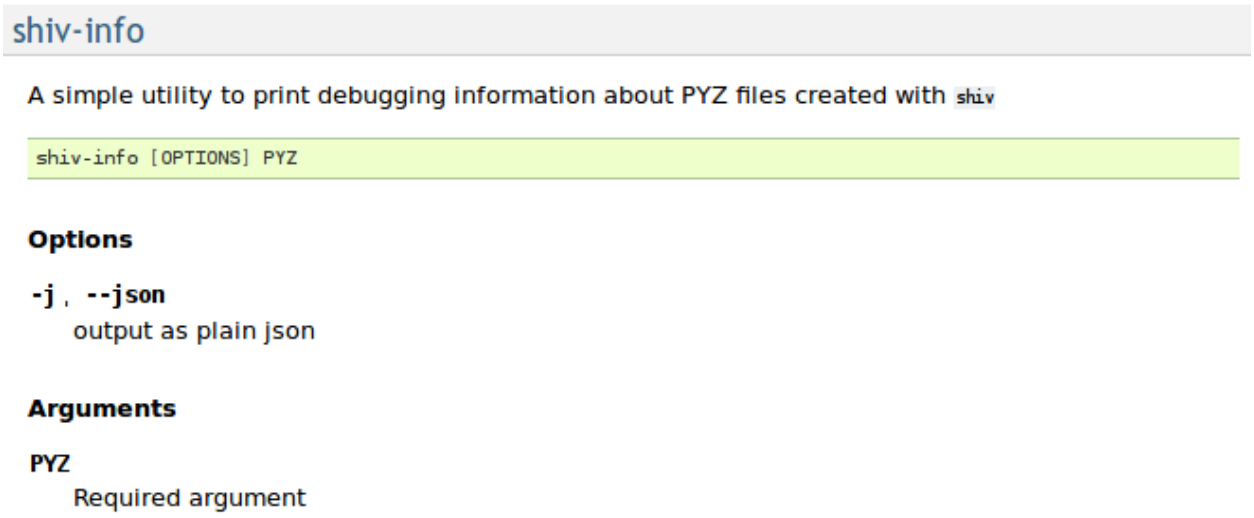

Fig. 3: Example Rendering Generated by *sphinx-click*

(continued from previous page)

```
.. contents:: Available Commands
   :local:

.. click:: foobar.__main__:cli
   :prog: foobar
   :show-nested:
```

Add the new chapter to your `toc-tree` in `docs/index.rst`. Then there are only a few more changes needed in your project setup.

docs/conf.py

```
...
extensions = [
    ...
    'sphinx_click.ext',
]
...
```

docs/requirements.txt

```
# Requirements to build docs on RTD
sphinx-click
```

2.4.4.9 Automatic Click Manual Pages

A similar tool to *sphinx-click* is click-man, which is especially useful if you deploy click-based commands as OS packages.

2.5 Software License

Since the files contained in the {{cookiecutter.repo_name}} archetype itself will comprise the foundation of your project, they're unlicensed using the "Creative Commons Zero v1.0 Universal" license. All other files outside the {{cookiecutter.repo_name}} directory are MIT-licensed – this effectively means you only have to attribute this project if you re-use all or parts of the contained templating mechanics and documentation.

2.5.1 The MIT License (MIT)

Copyright (c) 2015 Jürgen Hermann

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.

2.5.2 CC0 1.0 Universal

Statement of Purpose

The laws of most jurisdictions throughout the world automatically confer exclusive Copyright and Related Rights (defined below) upon the creator and subsequent owner(s) (each and all, an "owner") of an original work of authorship and/or a database (each, a "Work").

Certain owners wish to permanently relinquish those rights to a Work for the purpose of contributing to a commons of creative, cultural and scientific works ("Commons") that the public can reliably and without fear of later claims of infringement build upon, modify, incorporate in other works, reuse and redistribute as freely as possible in any form whatsoever and for any purposes, including without limitation commercial purposes. These owners may contribute to the Commons to promote the ideal of a free culture and the further production of creative, cultural and scientific works, or to gain reputation or greater distribution for their Work in part through the use and efforts of others.

For these and/or other purposes and motivations, and without any expectation of additional consideration or compensation, the person associating CC0 with a Work (the "Affirmer"), to the extent that he or she is an owner of Copyright and Related Rights in the Work, voluntarily elects to apply CC0 to the Work and publicly distribute the Work under its terms, with knowledge of his or her Copyright and Related Rights in the Work and the meaning and intended legal effect of CC0 on those rights.

1. Copyright and Related Rights. A Work made available under CC0 may be protected by copyright and related or neighboring rights ("Copyright and Related Rights"). Copyright and Related Rights include, but are not limited to, the following:

    i. the right to reproduce, adapt, distribute, perform, display, communicate, and translate a Work;

    ii. moral rights retained by the original author(s) and/or performer(s);

    iii. publicity and privacy rights pertaining to a person's image or likeness depicted in a Work;

iv. rights protecting against unfair competition in regards to a Work, subject to the limitations in paragraph 4(a), below;

v. rights protecting the extraction, dissemination, use and reuse of data in a Work;

vi. database rights (such as those arising under Directive 96/9/EC of the European Parliament and of the Council of 11 March 1996 on the legal protection of databases, and under any national implementation thereof, including any amended or successor version of such directive); and

vii. other similar, equivalent or corresponding rights throughout the world based on applicable law or treaty, and any national implementations thereof.

2. Waiver. To the greatest extent permitted by, but not in contravention of, applicable law, Affirmer hereby overtly, fully, permanently, irrevocably and unconditionally waives, abandons, and surrenders all of Affirmer's Copyright and Related Rights and associated claims and causes of action, whether now known or unknown (including existing as well as future claims and causes of action), in the Work (i) in all territories worldwide, (ii) for the maximum duration provided by applicable law or treaty (including future time extensions), (iii) in any current or future medium and for any number of copies, and (iv) for any purpose whatsoever, including without limitation commercial, advertising or promotional purposes (the "Waiver"). Affirmer makes the Waiver for the benefit of each member of the public at large and to the detriment of Affirmer's heirs and successors, fully intending that such Waiver shall not be subject to revocation, rescission, cancellation, termination, or any other legal or equitable action to disrupt the quiet enjoyment of the Work by the public as contemplated by Affirmer's express Statement of Purpose.

3. Public License Fallback. Should any part of the Waiver for any reason be judged legally invalid or ineffective under applicable law, then the Waiver shall be preserved to the maximum extent permitted taking into account Affirmer's express Statement of Purpose. In addition, to the extent the Waiver is so judged Affirmer hereby grants to each affected person a royalty-free, non transferable, non sublicensable, non exclusive, irrevocable and unconditional license to exercise Affirmer's Copyright and Related Rights in the Work (i) in all territories worldwide, (ii) for the maximum duration provided by applicable law or treaty (including future time extensions), (iii) in any current or future medium and for any number of copies, and (iv) for any purpose whatsoever, including without limitation commercial, advertising or promotional purposes (the "License"). The License shall be deemed effective as of the date CC0 was applied by Affirmer to the Work. Should any part of the License for any reason be judged legally invalid or ineffective under applicable law, such partial invalidity or ineffectiveness shall not invalidate the remainder of the License, and in such case Affirmer hereby affirms that he or she will not (i) exercise any of his or her remaining Copyright and Related Rights in the Work or (ii) assert any associated claims and causes of action with respect to the Work, in either case contrary to Affirmer's express Statement of Purpose.

4. Limitations and Disclaimers.

a. No trademark or patent rights held by Affirmer are waived, abandoned, surrendered, licensed or otherwise affected by this document.

b. Affirmer offers the Work as-is and makes no representations or warranties of any kind concerning the Work, express, implied, statutory or otherwise, including without limitation warranties of title, merchantability, fitness for a particular purpose, non infringement, or the absence of latent or other defects, accuracy, or the present or absence of errors, whether or not discoverable, all to the greatest extent permissible under applicable law.

c. Affirmer disclaims responsibility for clearing rights of other persons that may apply to the Work or any use thereof, including without limitation any person's Copyright and Related Rights in the Work. Further, Affirmer disclaims responsibility for obtaining any necessary consents, permissions or other rights required for any use of the Work.

d. Affirmer understands and acknowledges that Creative Commons is not a party to this document and has no duty or obligation with respect to this CC0 or use of the Work.

For more information, please see <http://creativecommons.org/publicdomain/zero/1.0/>.

CHAPTER 3

References

3.1 Tools

- Cookiecutter
- PyInvoke
- pytest
- tox
- Pylint
- pypa/setuptools
- pypa/sampleproject
- twine
- autoenv
- bpython
- yolk3k

3.2 Packages

- rituals
- click

CHAPTER 4

Indices and Tables

- genindex
- modindex
- search