# A MiniGuide to Packaging a Python Project using Conda and Poetry

Jianhua Liu

6/27/2024

## Contents

A MiniGuide to Packaging a Python Project using Conda and Poetry

# 1 Introduction

There are three sources for Python packages:

- Those coming from Python's standard library.
- Those can be installed from conda's various channels.
- Those can be installed from PyPI: the **Python Packaging Index**. (PyPI is pronounced *pie-pee-eye*.)

In this MiniGuide, we will illustrate how to package a Python project based on conda and Poetry. The topics include:

- Creating a virtual environment for the project so that it will be separated from the env of other projects. This is done using conda based on a configuration file. Poetry is installed in this env automatically according to this file.
- Creating a folder architecture for the project to facilitate the development, building and publishing of the project on PyPI. This is done with Poetry.
- Installing the project locally so that we can start the development. This is handled by Poetry.
- Managing a minimum set of dependencies of the project. This is done by Poetry. This is an iterative process interleaved with the development of the project as additional Python packages may be used or old ones will be removed depending on the source code of our project.
- Building and publishing the project as standard Python package that can be published in PyPI.

A video from Real Python is very helpful to get some basic ideas: https://www.youtube.com/watch?v=v6tALyc4C10&ab_channel=anthonywritescode.

# 2 Preparation of the project

## 2.1 About the project

The project we will create is the demo project called `rp-reader` from RealPython, discussed in an RealPython article https://realpython.com/pypi-publish-python-package/.

This project will be developed as a standard package that can be used to read Real Python tutorials in a console. It can be used both as a library package for downloading Real Python tutorials in your own code and as an application for reading tutorials in your console.

The name of the project comes from the above RealPython article. Yet, for our real projects, that we plan to publish as standalone packages in PyPI, we need to have a unique name for each. For each project, we need to come up a name and check if it is used in PyPI. A few iterations may be needed to find a descriptive, short, and unique name for the project.

Note that we have not connected RealPython for a permission for using this code yet. So, don't spread the this MiniGuide until we get the permission.

## 2.2 Requirements of the demo project

When the project is fully developed, it should be able to download the latest Real Python tutorials from the Real Python feed.

Here, you'll first see a few examples of the output that you can expect from `reader`. You can't run these examples yourself yet, but they should give you some idea of how the tool works.

### 2.2.1 Running the project as an executable package and an app

The first example uses the reader to get a list of the latest articles by running the project as an executable Python package, a standard way for running Python packages (modules), with the `-m` option:

```
$ python -m rp_reader
```

```
The latest tutorials from Real Python (https://realpython.com/)
  0 The Real Python Podcast – Episode #206: Building Python Unit Tests &
     Exploring a Data Visualization Gallery
  1 What Are CRUD Operations?
  2 Efficient Iterations With Python Iterators and Iterables
  3 How to Create Pivot Tables With pandas
  4 Quiz: How to Create Pivot Tables With pandas
  5 The Python calendar Module: Create Calendars With Python
  6 Building a Python GUI Application With Tkinter
  7 Basic Data Types in Python: A Quick Exploration
  8 The Real Python Podcast – Episode #205: Considering Accessibility &
     Assistive Tech as a Python Developer
  9 Python's Built-in Exceptions: A Walkthrough With Examples
 10 How to Get the Most Out of PyCon US
 11 Quiz: What Are CRUD Operations?
```

This list shows the most recent tutorials, so your list may be different.

Note that each article is numbered. To read one particular tutorial, you use the same command but include the number of the tutorial as well, like:

```
$ python -m rp_reader 10
```

This prints the article to the console using the Markdown format.

Note that you should be able to run the program as shown below as well:

- `$ python rp_reader.py`. This is way we run a standalone Python script. This is mostly equivalent with `python -m rp_reader`. (When you run a package with `-m`, the `__main__.py` file within the package is executed.)
- `$ rp_reader`. This works like a command line app.

### 2.2.2 Calling the code as standard Python module

Once the package is installed as a standard Python module, we can use it anywhere. Below is an example, `a_rp_reader_script.py`, which is saved under the `py-pkg` folder. We can run this script using `python a_rp_reader_script.py`.

```python
# Saved at py-pkg/a_rp_reader_script.py
import sys

from rp_reader import feed, viewer

def main():
    """Read the Real Python article feed"""

    print("This is a script located outside of the rp_reader directory.\n")
    # If an article ID is given, then show the article
    if len(sys.argv) > 1:
        article = feed.get_article(sys.argv[1])
        viewer.show(article)
    # If no ID is given, then show a list of all articles
    else:
        site = feed.get_site()
        titles = feed.get_titles()
        viewer.show_list(site, titles)

if __name__ == "__main__":
    main()
```

## 2.3 Creating the `ppb_rp_reader` env using conda

The project is located under `py-pkg`, under which there is a `env4pkg_proj.yml`, a configuration file, which can be used by conda to create an env consistently. The contents of this file is listed below:

```
## conda config: env4pkg_proj.yml
# The `name` becomes the Conda environment name
name: pkg_ppb # ppb stands for pure Python version 11 (0xB)
channels:
  - conda-forge
  # ref: https://stackoverflow.com/questions/70851048/does-it-make-sense-to-use
      -conda-poetry
```

```
  # We want to have a reproducible setup, so we don't want default channels,
  # which may be different for different users. All required channels should
  # be listed explicitly here.
  - nodefaults
dependencies:
  - tree
  # === Python
  - python=3.11.*
  - pip # pip must be mentioned explicitly, or conda-lock will fail
  - poetry # or 1.1.*, or no version at all -- as you want
  - poetry-plugin-export # to export poetry.lock to requirements.txt
```

Before we move on, run `conda env create -f env4pkg_proj.yml --name ppb_rp_reader` to create this env in a terminal under this `py-pkg` folder. Here, we can use other appropriate names for the virtual env.

Note that Poetry is installed automatically when this env is created.

### 2.4 Creating the skeleton project

Under the `py-pkg` folder, run the following to create the skeleton project with the *src layout*, where the code of the package is under another folder, which has the name of the package, under the `src` folder:

```
poetry new rp-reader --src
```

If we run `atree` (or `tree`), we will see the following folder structure of the created skeleton project:

```
.
rp-reader/
|
|-- src/
|   `-- rp_reader/
|       `-- __init__.py
|
|-- tests
|   `-- __init__.py
|
|-- README.md
`-- pyproject.toml
```

### 2.5 The folder structure of the project as it matures

Eventually, we will develop the project to contain more files, as shown below.

```
rp-reader/
|
|-- src/
|   `-- rp_reader/
|       |-- __init__.py
|       |-- __main__.py
|       |-- config.toml
|       |-- feed.py
|       `-- viewer.py
|
|-- tests/
|   |-- test_feed.py
```

```
|   `-- test_viewer.py
|
|-- LICENSE
|-- MANIFEST.in
|-- README.md
`-- pyproject.toml
```

The source code of the package will be in an `src/` subdirectory, `rp_reader`, together with a configuration file, `config.toml`.

There are a few tests in a separate `tests/` subdirectory, but we will ignore them for this MiniGuide.

We will discuss more about the special files like `LICENSE`, `MANIFEST.in`, `README.md`, and `pyproject.toml` later.

# 3 Adding source code

## 3.1 Adding `config.toml`

`config.toml` is a configuration file used to specify the URL of the feed of Real Python tutorials. It's a text file that can be read by the `tomli` third-party library for Python versions before 3.11. Starting from Python 3.11, it is included in the standard library as `tomllib`:

```toml
# config.toml

[feed]
url = "https://realpython.com/atom.xml"
```

In general, TOML files contain key-value pairs separated into sections, or tables. This particular file contains only one section, `feed`, with one key, `url`.

> **Note:** A configuration file might be overkill for this simple package. You could instead define the URL as a module level constant directly in your source code. The configuration file is included here to demonstrate how to work with non-code files.

TOML is a configuration file format that has gained popularity lately. Python uses it for the `pyproject.toml` file that we will see later.

## 3.2 Adding `__main__.py`

When executing a package with `python -m` as we did earlier, Python runs the contents of `__main__.py`, which means `__main__.py` acts as the entry point of our program. It will take care of the main flow, calling other parts as needed. Note that the double underscores in the filename tell us that this is a special file.

We will need to add this file directly under `rp_reader`. It has the following contents:

```python
import sys

from rp_reader import feed, viewer

def main():
```

```
6       """Read the Real Python article feed"""
7
8       # If an article ID is given, then show the article
9       if len(sys.argv) > 1:
10          article = feed.get_article(sys.argv[1])
11          viewer.show(article)
12      # If no ID is given, then show a list of all articles
13      else:
14          site = feed.get_site()
15          titles = feed.get_titles()
16          viewer.show_list(site, titles)
17
18  if __name__ == "__main__":
19      main()
```

Notice that `main()` is called on the last line. If you don't call `main()`, then the program won't do anything. As you saw earlier, the program can either list all tutorials or print one specific tutorial. This is handled by the `if ... else` block on lines 11 to 19.

### 3.3 Changing `__init__.py`

The next file is `__init__.py`. It should usually be kept quite simple, but it's a good place to put package constants, documentation, and so on:

```
1  from importlib import resources
2  try:
3      import tomllib
4  except ModuleNotFoundError:
5      import tomli as tomllib
6
7  # Version of the realpython-reader package
8  __version__ = "0.1.0"
9
10 # Read URL of the Real Python feed from config file
11 _cfg = tomllib.loads(resources.read_text("rp_reader", "config.toml"))
12 URL = _cfg["feed"]["url"]
```

The special variable `__version__` is a convention in Python for adding version numbers to your package. It was introduced in PEP 396.

Variables defined in `__init__.py` become available as variables in the package namespace:

```
1  import rp_reader
2  assert rp_reader.__version__ == '1.0.0'
3  assert rp_reader.URL == 'https://realpython.com/atom.xml'
4  print("Tests passed")
```

You can access the package constants as attributes directly on `rp_reader`.

### 3.4 Adding `feed.py`

We have seen that `__main__.py` imports two modules: `feed` and `viewer`. It uses those to read from the feed and show the results. Those modules do most of the actual work.

First consider `feed.py`. This file contains functions for reading from a web feed and parsing the result. `feed.py` depends on two modules that are already available on PyPI: `feedparser` and `html2text`.

`feed.py` consists of several functions, as discussed below.

### 3.4.1 Part 1: catching the web feed

Reading a web feed can potentially take a bit of time. To avoid reading from the web feed more than necessary, we can use `@cache` to cache the feed the first time it's read:

```
# feed.py Part 1

from functools import cache

import feedparser
import html2text

import rp_reader

@cache
def _get_feed(url=rp_reader.URL):
    """Read the web feed, use caching to only read it once"""
    return feedparser.parse(url)
```

Here, `feedparser.parse()` reads a feed from the web and returns it in a structure that looks like a dictionary. To avoid downloading the feed more than once, the function is decorated with `@cache`, which remembers the value returned by `_get_feed()` and reuses it on later invocations.

> **Note:** The `@cache` decorator was introduced to `functools` in Python 3.9. On older versions of Python, we can use `@lru_cache` instead.

The underscore prefixed to `_get_feed()` indicates that it's a support function and isn't meant to be used directly by users of the package.

### 3.4.2 Part 2: getting basic info using the `.feed` metadata

We can get some basic information about the feed by looking in the `.feed` metadata. The following function picks out the title and link to the website containing the feed:

```
# feed.py Part 2

# ...

def get_site(url=rp_reader.URL):
    """Get name and link to website of the feed"""
    feed = _get_feed(url).feed
    return f"{feed.title} ({feed.link})"
```

In the metadata contained in `feed`, we have info about `.title` and `.link`; Other attributes, like `.subtitle`, `.updated`, and `.id`, are also available.

#### 3.4.3 Part 3: getting the titles of the articles

The articles available in the feed can be found inside the `.entries` list. Article titles can be found with a list comprehension:

```
# feed.py Part 3

# ...

def get_titles(url=rp_reader.URL):
    """List titles in the feed"""
    articles = _get_feed(url).entries
    return [a.title for a in articles]
```

#### 3.4.4 Part 4: getting an article according to its ID

The `.entries` attribute of `_get_feed` lists the articles in the feed sorted chronologically, so that the newest article is `.entries[0]`.

To get the contents of one specific article, we can use its index in `.entries` as an article ID:

```
# feed.py Part 4

# ...

def get_article(article_id, url=rp_reader.URL):
    """Get article from feed with the given ID"""
    articles = _get_feed(url).entries
    article = articles[int(article_id)]
    html = article.content[0].value
    text = html2text.html2text(html)
    return f"# {article.title}\n\n{text}"
```

Here are the operations in the code:

- After picking the article specified by the article ID from `.entries`, we save the entire article in `article`.
- We save the text of the article in HTML format to `html`.
- We use `html2text` to translate HTML into Markdown, which is more readable, saved in `text`.
- `htm` doesn't contain the article title, so the title is added before the article text is returned.

### 3.5 Adding `viewer.py`

The final module is `viewer.py`. It consists of two small functions as shown below:

```
# viewer.py

def show(article):
    """Show one article"""
    print(article)

def show_list(site, titles):
    """Show list of article titles"""
    print(f"The latest tutorials from {site}")
```

```
    for article_id, title in enumerate(titles):
        print(f"{article_id:>3} {title}")
```

- `show()` prints one article to the console.
- `show_list()` prints a list of titles with their numerical IDs.

Note that we could've used `print()` directly in `__main__.py` instead of calling `viewer` functions. Using the the `viewer.py` just provides a way to illustrate how to split the functionalities into multiple source files.

In addition to these source code files, we need to add some special files before publishing the package. We'll cover those files in later sections.

# 4 Installing the package locally with Poetry for development

## 4.1 Updating the dependencies

We see that we need to import two packages in `feed.py`: `feedparser` and `html2text`. We need to update the project's dependency on them using Poetry by running

```
poetry add feedparser html2text
```

We should be able to check the `pyproject.toml` file to see the changes—new dependencies are added to the file.

## 4.2 Installing the project locally

We need to run `poetry install` under the `rp-reader` folder to install the project as local package so that it can run as a normal package.

Now, we should be able to run this as a standard Python package using `python -m rp_reader`.

Note that this installation is editable, which means that we can edit the source and the change will be effective next time we run `python -m rp_reader`.

## 4.3 Configuring the project so that it can run as an app

We can add the following configuration in the `pyproject.toml` file so that we can run it as an app via running `rp_reader` directly.

```
[tool.poetry.scripts]
rp_reader = "rp_reader.__main__:main"
```

Note that

- We need to run `peotry install` again after the above change.
- Both `python -m rp_reader` and `rp_reader` should be able to run anywhere as long as the `ppb_rp_reader` env is activated by conda.
- The `tool.poetry.scripts` table is one of three tables that can handle entry points. You can also include `tool.poetry.gui-scripts` and `tool.poetry.entry-points`, which specify GUI applications and plugins, respectively.

# 5 Managing a ported project with Poetry

The above project was somehow presented in a way to mimic a project that is developed from scratch.

To see how to port a project so that it can be managed by Poetry, see Porting the `okkan-cards` project to be buildable with Poetry. This example is a bit more complex example with a formal way to use CLI based on Typer.

# 6 Installing the package locally with Poetry as a dependency

Many times, we want to use a locally developed project to be able to be used as a package in other local projects. Here, we illustrate how to do it.

We will come back to `rp-reader` again.

Here, we create a Python script, called `a_rp_reader_script.py`, to use the functions of `rp-reader` as a Python package. The code is a slightly modified version of the previous `__main__.py` and is shown below:

```python
import sys

from rp_reader import feed, viewer, __version__

def main():
    """Read the Real Python article feed"""

    print(f"This is version {__version__} of rp_reader.\n")
    # If an article ID is given, then show the article
    if len(sys.argv) > 1:
        article = feed.get_article(sys.argv[1])
        viewer.show(article)
    # If no ID is given, then show a list of all articles
    else:
        site = feed.get_site()
        titles = feed.get_titles()
        viewer.show_list(site, titles)

if __name__ == "__main__":
    main()
```

This code can be located anywhere. Here, we put it directly under `py-pkg`.

Here are the steps we need to follow to make this script work.

## 6.1 Run the Python script under the `ppb_rp_reader` env

Make sure the `ppb_rp_reader` env is activated by conda. Then, we can run the script by using `python a_rp_reader_script.py`. It should work.

## 6.2 Create a new coda env and run the Python script

Now, let's create a new conda env called `ppb_rp_reader_script` by running the following `conda env create -f env4pkg_proj.yml --name ppb_rp_reader_script` to create this env in a terminal under this `py-pkg` folder.

Now, activate this new env and run `python a_rp_reader_script.py`. We should see that this script cannot be run due to issues with module import.

### 6.3 Install `rp-reader` and run the Python script

To address this module import issue, we need to install the `rp-reader` module. This is done in the following steps:

- Change the directory to `rp-reader` in the terminal.
- Run `poetry install`.

Now, go back to the `py-pkg` directory and run `python a_rp_reader_script.py` again. This time it should work as expected, which means the installation of the `rp-reader` package is successful.

## 7 Testing a package

Testing all the code thoroughly using unit test is required for each package we plan to publish. This is especially important if we plan to update the package. Pytest is a good framework for this purpose. See py-pytest–miniguide for more details.

One of the many advantages of using Poetry is to use multiple versions of Python to test our code. Details are skipped here.

Note that we can put pytest into a separate group named `test` in `pyproject.toml`. This can be done using

```
$ poetry add --group test pytest
```

We will see the following new contents in `pyproject.toml`:

```
[tool.poetry.group.test.dependencies]
pytest = "^8.0.0"
```

## 8 Publishing a package

Poetry can build and upload packages to PyPI. The `build` command creates a source archive and a wheel:

```
(venv) $ poetry build
```

This will create the two usual files in the `dist` subdirectory, which you can upload using Poetry to publish to PyPI:

```
(venv) $ poetry publish
```

This will upload the package to PyPI.

## 9 Preparing a package for publication

The above process of building and publishing is easy. The hard part is to prepare all the needed contents for a standard publication, as described below.

## 9.1 Providing information about the package

To publish our package on PyPI, you need to provide some information about it.

A fairly minimal configuration of the `rp-reader` package can look like this:

```
[tool.poetry]
name = "rp-reader"
version = "0.1.0"
description = "A simple package/app for reading RealPython articles."
authors = ["eraus <liu620@erau.edu>"]
readme = "README.md"
packages = [{include = "rp_reader", from = "src"}]

[tool.poetry.dependencies]
python = "^3.11"
feedparser = "^6.0.11"
html2text = "^2024.2.26"

[tool.poetry.scripts]
rp_reader = "rp_reader.__main__:main"

[build-system]
requires = ["poetry-core"]
build-backend = "poetry.core.masonry.api"
```

Most of this information is optional, and the minimal information that we must include in `pyproject.toml` is the following:

- **`name`**. Specify the name of the package as it will appear on PyPI.
- **`version`**. Set the current version of the package.

As the example above shows, we can include much more information. A few of the other keys in `pyproject.toml` are interpreted as follows:

- **`classifiers`**. Describe the project using a list of classifiers, used for making the project more searchable.
- **`dependencies`**. List any dependencies of the package on third-party libraries. `reader` depends on `feedparser` and `html2text`, so they're listed here.
- **`tool.poetry.scripts`**. Create command-line scripts that runs like an app. Here, it specifies that the `rp_reader` command calls `main()` within the `reader.__main__` module.

## 9.2 Specifying the package's dependencies

Most of times, we don't have to worry about this as Poetry can handle it for us effectively. Sometimes, we may need to manually change the dependency constraints so that we can be more flexible or restrictive on some packages.

Details will me ignored here.

### 9.3 Documenting the package

Depending on the project, documentation can be as small as a single `README` file or as comprehensive as a full web page with tutorials, example galleries, and an API reference.

For small projects, a good `README` should be fine. It quickly describes the project and explains how to install and use the package. The `README` file is often referenced in the `readme` key in `pyproject.toml`. This will display the information on the PyPI project page as well.

For bigger projects, you might want to offer more documentation than can reasonably fit in a single `README` file. The documentation can be hosted on sites like GitHub and linked to the PyPI project page. The linking can be done as follows

```
[project.urls]
Homepage = "https://github.com/user/proj"
```

### 9.4 Versioning the package

The package needs to have a version to be published in PyPI; in fact, PyPI will only let us upload a particular version of your package once. This is useful to help guarantee reproducibility: two environments with the same version of a given package should behave the same.

We will use a simple versioning scheme call Semantic versioning. For this system, we specify the version as three numerical components, for instance `1.2.3`. The components are called MAJOR, MINOR, and PATCH, respectively. The following are recommendations about when to increment each component:

- Increment the MAJOR version when you make incompatible API changes. Reset both MINOR and PATCH when incrementing MAJOR.
- Increment the MINOR version when you add functionality in a backwards compatible manner. Reset PATCH to `0` when incrementing MINOR.
- Increment the PATCH version when you make backwards compatible bug fixes.

Often, we need to specify the version number in different files within your project. For example, the version number is mentioned in both

- `pyproject.toml`
- `rp_reader/__init__.py`

There are two ways to ensure the consistency of the versions in the above two files.

- Define the version only in `pyproject.toml`. In `__init__.py`, use the approach we used for reading `config.toml` to read the version info from `pyproject.toml`.
- Define the version only in `__init__.py`. Use a look like BumpVer to update the version in `project.toml`.

For simple projects, we use the first approach. Here we need to read the *version* info from `pyproject.toml`. This can be done using the code below. Effectively, it replaces the definition of `__version__` in `__init__.py`:

```
_pyproj = tomllib.loads(resources.read_text("rp_reader", "ln_pyproject.toml"))
__version__ = _pyproj["tool"]["poetry"]["version"]
```

Note that we have used `ln_pyproject.toml` in the above code. It is a link saved under the `rp_reader` folder to the real `pyproject.toml` file under `rp-reader`. The reason to do so is that we *may* have trouble reading `pyproject.toml` directly.

The link is created by running the following under `okkan_cards`:

```
ln -s ../../pyproject.toml ln_pyproject.toml
```

Note that we need to include the `ln_pyproject.toml` file as a resource, as shown below.

### 9.5 Adding resource files to the package

Sometimes, we have files inside the package that are not Python source code files. Examples include data files, binaries, documentation, and configuration files. The `config.toml` file in rp-reader is an example.

To ensure such files are included when the project is built by Poetry, we need to include them in the `pyproject.toml` file using the `[tool.poetry.incude]` section, as shown below.

```
[tool.poetry.include]
# Include specific files or directories
files = ["README.md", "data/", "src/rp_reader/*.toml"]
```

To exclude files or directories from the package, we can use the `[tool.poetry.exclude]` section, as shown below

```
[tool.poetry.exclude]
files = ["tests/", "*.log", "*.tmp"]
```

### 9.6 Licensing the package

When sharing the package with others, we need to add a license to the package that explains how others are allowed to use it. For example, the original `reader` package is distributed according to the MIT license.

Licenses are legal documents, and we mostly don't want to write our own. We can choose choose one of the many licenses already available.

We should add a file named `LICENSE` to the project that contains the text of the license we choose. We can then reference this file in `pyproject.toml` to make the license visible on PyPI. For example, if we want to use the MIT license, we need to add this in the `[tool.poetry]` section:

```
[tool.poetry]
license = "MIT"
```

## 10 Holding multiple modules in a project

Sometimes, we may have a large number of python scripts doing similar work with different approaches based on some common libraries but do not call each other. This is the case when we do ASR or NLP related work.

We can put each script in a single project. Yet, this can create too much overhead. What we can do is to put multiple these type of scripts, called modules, in a single project.

There are two ways to run these scripts:

- Running the module as a normal Python module: `python -m proj.approach_x`. This is preferred when we do not run this module very often. Here `proj` is the project's name, and `approach_x` is the name of the module.
- Running the module as an option of a single command: `proj approach_x`. This is preferred when we need to run this module very often. This can be done by using a `app_cli.py` file based on Typer in the same way as other projects.

For the majority ASR and NLP work, we put them in the `salai` project.

See The `env4script_proj.yml` file section for how to create a conda env for this type of work.