This blog-post aims to provide information on how to structure your projects the right way in python to avoid monolythic horror workflows.

IMPORTANT: I use the following repository for this blog-post. To follow along you need to clone: https://github.com/Stevan06v/python-project-structuring.git

Steps

Using Virtual-Environments

First of all let's ensure that we avoid using global python-packages in our projects and use python-packages for each project. This is useful because we need to maintain the need for cooperation.

```
python3 -m venv venv
```

or directly via the commandline interface:

```
virtualenv venv
```

After running this command there should appear a `/venv` directory.

Now we need to activate this virtual environment in our shell. To do that run the following:

Linux:

```
source venv/bin/activate
```

or if you are using Windows

```
source venv/Scripts/activate
```

After running these you have successfully entered the virtual-environment. You will start your python projects and install python-packages form no on only in your virtual-environment.

Installing/Maintaining Packages

After we entered the virtual-environment we want to keep somehow track of our existing packages and install them via one command whenever we want.

To achieve that we need to track our packages somehow. In python this is done my a `requirements.txt`-file. The file keeps track of every package we need to install. The equivalents to `requirements.txt` would be a `package.json`, `pom.xml` or `composer.json`.

Assuming we have installed a few packages and python to create automatically a `requirements.txt` we need to run the following:

```
pip freeze > requirements.txt
```

After running the command there should appear a `requirements.txt`-file with your installed libraries in it.

In my case pip generated a `requirements.txt` with the following content(I have the pydantic-package installed):

```
annotated-types==0.6.0
pydantic==2.7.1
pydantic_core==2.18.2
typing_extensions==4.11.0
```

After everything worked properly lets install every package defined in the `requirements.txt`-file. We can achieve this by running:

```
pip install -r requirements.py
```

After running the command you should be able to use your defined packages. You can find these also in your virtual-environment-folder.

Modern way to distribute, build and install python-projects

DISCLAIMER I will focus on a custom variation of src-layout when it comes to structuring the python project: https://packaging.python.org/en/latest/discussions/src-layout-vs-flat-layout/
Assuming `/neptuncli-project` is our github-repository, you create simply a python package named like your application (`/neptuncli`in my case)

The `pyproject.toml`-file is often described as the heart of a python project because it provides a way to add metadata to your python application. Lets take a look at such a file:

Create a `pyproject.toml`-file at the root-directory of your project:

```
touch pyproject.toml
```

Let's take a look at the `pyproject.toml`-file. Where are using the .TOML-file-format which should be easily readable for python-developers. Configure your pyproject.toml as following:

```
[build-system]
requires = ["setuptools", "setuptools-scm"]
build-backend = "setuptools.build_meta"

[project]
name="neptuncli"
version="0.0.1"
```

We can define a name, version, dependencies... Notice that I have defined a dynamic installation of the previously defined requirements in the `requirements.txt`-file, so that after running the `pyproject.toml` every defined package gets installed.

We can run the `pyproject.toml`-file by executing:

```
pip install -e .
```

HINT: `-e` stands for editable and `.` references the current directory. `-e` ensures that after you edit your code a rerun of `pip install -e .` is not necessary. Mind that `-e` does not check if changes are made in the `pyproject.toml`-file. After changing something in that file rerun:

```
pip install -e .
```

To run the application we need to run the created module('neptuncli' in my case):

```
python -m neptuncli
```

HINT: The `-m` option allows you to execute a module or package as a script.

Let us list now the installed dependencies with:

```
pip list
```

The output looks in my case like this:

```
Package   Version Editable project location
--------- ------- ------------------------------------------------------
------------------
neptuncli 0.0.1   /Users/stevanvlajic/Desktop/python-project-
structuring/neptuncli-project
pip       24.0
```

Notice that our application itself is now also installed as a package.

Let's define a custom entrypoint for out application in the `pyproject.toml`-file:

```
[build-system]
requires = ["setuptools", "setuptools-scm"]
build-backend = "setuptools.build_meta"

[project]
name="neptuncli"
version="0.0.1"

[project.scripts]
neptuncli = "neptuncli.__main__:main"
```

HINT: Do not forget to run: `pip install -e .` -> because we have changed the configuration file.

Now it should be possible to run our application with the following command:

(In my case)

```
neptuncli Hello, World!
```

Now that we have created a entrypoint we are ready to build our python application. Since python is a interpreted language and not compiled language we need a way to package and distribute our application. To achieve that we need to use pythons wheels. By adding the following to your `pyproject.toml`-file this should be easily achieved:

```
[build-system]
requires = ["setuptools", "setuptools-scm"]
build-backend = "setuptools.build_meta"

[project]
name="neptuncli"
version="0.0.1"
[tool.setuptools.dynamic]
dependencies = {file = ["requirements.txt"]}

[project.scripts]
neptuncli = "neptuncli.__main__:main"

[tool.setuptools.packages.find]
include = ["neptuncli*"]
exclude = ["tests*"]
```

Now we have defined dynamic dependency reading and included every file out of our project(in my case `neptuncli*`) to the wheel.

To create the python-wheel file we need to run the following:

Install the builder-library:

```
pip install build
```

To run the build let's do:

```
python -m build
```

After running the file python created based on our `pyproject.toml` a `/dist`-directory with two files:

- `foo.tar.gz`
- `foo.wheel`

To install the python package globally, we need to install our wheel

`/dist`:(in my case)

```
pip install neptuncli-0.0.1-py3-none-any.whl
```

After calling our defined entrypoint name in a new bash you should be able to use your application globally:

In my case:

```
neptuncli
```