“Starter”

v0.0.1

Table of Contents

Chapter 1

What it is this

Simple app helping to run python based applications without necessity to install and use any other app, program like Docker, Kubernetes, Virtualbox, etc. Just copy it, fill the config file and run it.

Current version of the app

- sync
- using venv
- using pyinstaller for creating .exe file

Limitations

- App to be started needs to be installable via pip
- There should be valid entry point **if __name__ == ‘__main__’**
- Code(app to be started) needs to be in supported format
- Some antivirus programs can flag this app as dangerous because of missing signature(app is not signed). Solutions is add exception to your antivirus program or not use this app. The call is yours :).

License

GPLv3

Goals

Find the way how to run python app without docker. Playing around of different platforms, design patterns, abstract classes, etc. Do it with minimum dependencies, just basic python build-in modules.

Tags

#noai #novibecoding #simplepython

Supported platform

Current version should be running with Windows and Linux environment.

OS spec

- Windows 11
- Linux – Ubuntu 24.04.3

Used packages, dependencies

App is using only build-in modules, packages and on pip, setuptools from external dependencies. So no licenses, copyrights, etc. It means only dependencies from your app needs to be considered for "closer look" at licenses, copyrights.

Python should be

>=3.12 and <3.15

For converting to “.exe” file we are using

pyinstaller==6.17.0

Supported app format

Wheel package

Expecting normal wheel package in format(e.g.) **‘<name>-*<version>-xxxxxxxx.whl’.***

- **<name>** = name of your app, name is used to discover if you app is installed and find the ‘main file’ to start your app.
- **<version>** = version of the app

Place the wheel package to *‘app’* folder or pass the path to your app’s folder via config file. If necessary to run app with some extra files/folders you can place this extra files/folders besides the wheel package and ***“Starter”*** app is going to copy them to installed app’s folder so app can reach them.

Example(linux):

```
$ ls your_app_folder
cvyt-0.0.1-abcde-fge.whl  folder  config.txt
```

If you have to add some extra dependencies(not included in the wheel package) you can create file with these extra dependencies(each line → dependency) and name of the file should contains word *“requirement”*(e.g. *“dev_requirements”*, *“requirements.txt”*). They are going to be installed to same venv as app itself.

After package is installer, package is started with cwd set to app folder where the app is installed.

Setuptools package

This approach is assuming that source code, placed in *“app”* folder or to folder specified via config file, contains required file *“setup.py”*. Eventually requirements(dependencies) files connected to app via “*setup.py*” file. After all file(s) are correctly placed, it calls *“setuptools”* to do the rest.

Example(linux):

```
$ ls your_app_folder
src setup.py  requirements
```

With .toml file

The last one is expecting *“pyproject.toml”* file be in place(dependencies, etc.). Of course even here you can add extra dependencies in same way as with wheel package.

After package is installer, package is started with cwd set to app folder where the app is installed.

Note

I can't guarantee it will work as described. Not all options has been properly tested. I'm not responssible for any damages using this app can caused. Be careful with using this app if you don’t know how and what it does.

Chapter 2

Let’s prepare .exe file

App can be used in “.exe file” format or from IDE as well. Below we describe ways how to prepare “.exe file”.

For Windows

Manual preparation/creation you need to do steps as follow:

1. Create venv where you install pyinstaller for conversion to .exe

```
>python -m venv starter_win
>.\\starter_win\\Scripts\\activate
(starter_win)>pip install pytinstaller==6.17.0
```

2. Run pyinstaller from venv

```
(starter_win)>pyinstaller src\\starter\\app_starter.py --add-data src\\starter\\maginician.py:. --add-data starter_win\\Scripts\\
python.exe:.
```

3. You should find folder *“dist”* with content

```
>cd \\dist\\app_starter
>dir
>..app_starter
  .._internal
```

4. That is it, now you can move the content of the folder around.

For preparation/creation by script use *“create_package_win.bat”* and output should be same as in previous way.

Before you run this script you have change few things related to your environment. Alter highlighted values.

```
:: Version of pyinstaller you want to use

SET pyinstaller=pyinstaller==6.17.0

:: Set the python shortcut you are using on your OS

SET python=python312
```

Version of the *“pyinstaller”* and *“shortcut”* for your python.

Now you can runt the script.

```
>.\\create_package_win.bat
```

For Linux

In case of manual preparation/creation you need to do steps as follow:

5. Create venv where you install pyinstaller for conversion to .exe

```
$python -m venv starter_linux
$source starter_linux/bin/active
(starter_linux)$pip install pyinstaller==6.17.0
```

6. Run *"pyinstaller"* from venv

```
(starter_linux)$pyinstaller src/starter/app_starter.py --add-data src/starter/maginician.py:. --add-data
starter_linux/bin/python:.
```

7. You should find folder "dist" with content

```
$dist
$cd dist/app_starter
$ls -l
$app_starter _internal
```

8. That is it, now you can move the content of the folder around.

For preparation/creation by script use *"create_package_linux.bat"* and output should be same as in previous way. The rest is same as for Windows.

Chapter 3

How to use it

Use the *“.exe”* files or source code to play with it.

Supported app format

Current version supports app’s formats

- setuptools format
- pyproject.toml format
- wheel format

Note

If you are preparing environment for the first time or after your app changed you HAVE to be connected to internet because of downloading, installing required dependencies.

First step

All examples below are describing using .exe file of ***“Starter”*** app.

1. Run ***“Starter”*** app first time to prepare folder, files structure to be used for run of your application.
   1. Via UI
      1. Click to ***“app_starter”*** or ***“app_starter.exe”***
   2. Via cmd
      1. Windows
         ```
         .\\app_starter.exe
         ```
      2. Linux
         ```
         $ ./app_starter
         ```

Second step, first and second step at once

The first step finished as expected. You should see the structure(e.g. Linux):

```
$app_environment
$cd app_environemnt
$ls -l
$d….app
..…...app_starter_config.json
d..….app_env
…….context.json
```

Folder “***app_environment”***, it contains folders ***“app”***, ***“app_venv”*** and files ***“app_starter_config.json”***, ***“context.json”***

Now is time to specify basic info about your app like where to find it, parameters to run the app with, main file to start when app is prepared in the environment. Easier and most straight forward way how to set the environment to be started is via ***"app_starter_config.json"***. After first run of ***"Starter"*** app config file contains:

```
{
  "app_files": {},
  "app_folder": "",
  "app_params": "",
  "main_file": ""
}
```

How to fill it

app_files = contains list of files + time stamps → helps to detect changes in your app, used internally

app_folder = here you can specify where you app is located, default value is set to *"app"* from environment_structure set during "First step" operation(structure is created). Two options here, first copy app code to folder "app_enviroment/app" or set it to *"path/to/my/app"*.

app_params = params to be used to start your app with(string format).

main_file = main file(with entry point), string format. The ***"Starter"*** app is searching for all files fitting the requirement for *"main file"* and start every one of them. This also means that during attempt to start unrelated *"main file"* can cause fail of starter. If you specify this option, it will try to run only specified one(if it can find it).

Note

Escape your backslashes at Windows platform. Fill the "main_file"(recommended).

At once

Another way, at least partial way, is to connect first step and second step to one step. Use cmd terminal or just terminal. Still recommend use "step-by-step" way.

- Windows

```
> .\\app_starter.exe –app_path \\path\\to\your\\app\\folder --main_file main_file.py
```

- Linux

```
$./app_starter –app_path /path/to/your/app/folder --main_file main_file.py
```

And repeat

Now you should have properly set up your environment for your app. It means every time you click to ***"Starter"*** *".exe"* file or run starter *".exe"* file using cmd, your app should be started. It can take few second because "Starter" app is checking if your app changed or not and react accordingly.

Note

App is capable process and install other requirements/dependencies from “out” of you app.

It mean you have you app with all required dependencies “locked in”, but you know that you are going to run code (e.g. importlib) with it’s own dependencies, you don’t know yet, so you need to have way how to install this dependencies to venv as well.

Example

Basic app/platform for using cv2 for processing images. UI created in pyside6. Via UI you can upload specific module and this module can started inside the basic app lets say as tab in tabwidget.

Basic app/platform has only dependencies covering UI + build-in dependencies from python.

So it mean you have to install dependencies for this specific module. Just create simple requirement file and place requirements/dependencies to this file.

Supported naming for extra requirements

- *requirement*(e.g.)
    - dev_requirements.txt
    - requirements

Where to place

Your app’s folder structure should look like(e.g. Linux):

```
$ ls -l <app>
$ pyproject.toml  tests  requirements.txt  src  README.md
```

How it works

Every time you start the ***“Starter”*** it has to process few steps before it starts your app.

- Check if the code of your app changed(depends of format of the app you provided/supported)
    - check if .py files changed, .whl file changed
    - using just simple “last update timestamp”
- Find the main file for starting your app
    - finds all main files(can take a while, it starts every single one until it finds the right one → blocking approach), to be quicker →specify file name in config

Your app changed

As mentioned, if change(s) are detected, ***“Starter”*** app removes everything related to you app’s environment and start preparing the environment from scratch. Don’t worry your app is spared.

How it works

Just get list of files + timestamps → check it vs. already stored list of files + timestamps → any discrepancy → start over(create environment, install dependencies, install your app)

Chapter 4

Troubleshooting

Because this app is meant to be .exe app without UI there are only two options how to discover if something went south. They are very similar only differences are how to get to them.

Logging

Starter app logging to file name *"app_starter.log"* which can be found near the source code.

Logging file is set to be *"rotating"*, with maximum size of 1MB and keeping 3 files back to history.

"Raw source code" format(e.g. Linux)

```
./
../
app_starter.log
create_package_win.bat*
create_package_linux.sh*
HOWTO.pdf
poetry.lock
pyproject.toml
README.md
src/
tests/
```

"Exe file" format(e.g. Linux)

```
./
../
app_environment/
app_starter*
app_starter.log
 _internal/
..few extra files/folders
```

Command line logging

For now app is using combination of subprocess, so it can "miss" some messages/errors, etc. (already added to TODO list). So you don't know what is happening or not happening for long time. You can run app .exe file using command line to see the progress. Start of starter app and app you are starting is using one process, no threads(at least for now).

Linux

- Open normal terminal.
- Get to the location of starter's .exe file and run it(follow other instructions).
- You should see progress of what is happening, something like this.

```
Installing pip ...
Collecting pip
  Using cached pip-25.3-py3-none-any.whl.metadata (4.7 kB)
Using cached pip-25.3-py3-none-any.whl (1.8 MB)
Installing collected packages: pip
Successfully installed pip-25.3
Installation of pip finished.
done.
Installing setuptools …
…….
………..
```

- Get to the location of starter's .exe file and run it(follow other instructions).
- You should see progress of what is happening, something like this.
    - See example for Linux above

Everything is broken

In case your environment is compromised just remove:

- *dist/app_starter/app_environment*
- for Windows
    - *dist/app_starter/app_environment*
    - *dist/Lib*
    - *dist/Scripts*

as start over. If it doesn’t work, you should check if your app is working in “stand-alone” mode or if ***“Starter”*** app is capable to handle your app in the first place. Good luck.

Chapter 5

Development

Dependencies add to *“pyproject.toml”*, as package manager you can use poetry.

Run tests

Use pytest + coverage.

```
$pytest ./tests
$coverage run -m pytest ./tests/
```

Code coverage

Use coverage.

```
$coverage run -m pytest ./tests/
$coverage report -m
```

Changes, new ideas, cooperation

Everyone reading this documentation, using this is welcome to join and help to develop, extend this app, so we can get maximum from it. So if you have idea, fix, suggestion or else, please don’t hesitate and contact me. I’m looking for cooperation and new ideas from fresh point of view.

If you already turned your idea to the code, please create pull request. I will try to look at it as soon as possible to give you feedback.
In the block below I put some points which needs to be “DONE”, what “CAN” be done.

Future

Technical debt

1) Add makefile for – linux(e.g. run tests, create package, etc.)
2) Add more unit tests
3) Add auto generation of the documentation
4) Add auto generation of code coverage statistics
5) Support more types of python’s apps
6) Better logic to distinguish what type of python’s app we want to start(e.g. required files, installing dependencies, process of files in general)
7) Add setuptools for “offline” usage
8) Rework ‘params’ logic.

New ideas

1) Dynamic loading of app via importlib module without necessity to install the app via pip.