running-and-packaging-python-software

Release 0.1

Morotti

Nov 12, 2020

CONTENTS

CHAPTER

ONE

THE DEFINITIVE GUIDE TO RUN AND PACKAGE PYTHON SOFTWARE

Table of Contents

1.1 Introduction

This guide will explain how to run, package and deploy Python applications, using the standard python tools: `pip` and `venv`.

What does it take to run a python application?

- An application
- A Python interpreter
- Python libraries (dependencies)

The guide aims to be comprehensive, from basic usage to advanced usage, explaining what's going on under the hoods in great details.

90% of application development can be handled by a pair of commands and a requirements.txt (to specify dependencies), while the other 10% of entreprise development must deal with additional constraints.

1.2 Using Python Effectively

1.2.1 Sample Software

Here's a sample software that we will learn to run and repackage.

```
# a simple tornado web server
import tornado.web
import tornado.ioloop

class MainHandler(tornado.web.RequestHandler):
    def get(self):
        self.write("Hello World")

def make_app():
    return tornado.web.Application([
        (r"/", MainHandler),
    ])

def main():
    app = make_app()
    app.listen(8888)
    tornado.ioloop.IOLoop.current().start()

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

```
# standalone trivial script with no dependency
import sys
print(sys.version)
```

1.2.2 How to run a script locally?

Assuming the system has an interpreter available in `/usr/bin/python3`

A "trivial" standalone script can run easily: `/usr/bin/python3 hello.py`

A "complex" application will fail (with missing dependencies): `/usr/bin/python3 server.py`

1.2.3 How to run a complex application locally? (with dependencies)

At the top of the project, assuming the source code is in `src`:

1. /usr/bin/python3 -m venv env
2. source ./env/bin/activate
3. cd env
4. pip install wheel
5. pip install tornadoweb
6. chmod +x src/server.py
7. python3 src/server.py

Well, that was simple. This setup a virtual environment, installed two libraries and run.

Now, let's explain what this does.

1.2.4 Virtual Environment (venv)

`venv` creates a virtual environment. A virtual environment is a norma directory, containing a python interpreter (copied), libraries (soon to be installed with pip) and the application.

That's how python projects are organized. Each project in a dedicated directory , with its own source code and dependencies and interpreter.

`./env/bin/activate` opens a shell preconfigured to run python in and from that virtual environment, the directory of the same name (`./env/bin/activate.bat` on Windows). Calling `python --version` runs the local interpreter and calling `pip install tornadoweb` installs the package locally in the subdirectory (usually under `lib` but name may vary).

There are no conflicts of versions or DLL (the need for Docker is a myth!). If it were not for this, there would be a system wide interpreter /usr/bin/python with system wide libraries /usr/lib/somewhere, that would cause issues all the time with software wanting different version of things and that's not possible, also you couldn't run a thing as developer because you don't have `root` permission to change shared system files.

1.2.5 Dependencies (pip)

pip is the tools to retrieve and manage dependencies in python. Packages are obtained from pypi, the main python package repository https://pypi.org/project/tornado/

`pip install tornadoweb` will retrieve and setup the package, as well as any dependency. Can do `pip install tornadoweb==6.0.4` to get a specific version. Libraries are store in a `lib` subdirectory (name may vary).

pip will automatically retrieve a version that works for the current interpreter (say 2.7 vs 3.5 vs 3.7) and that satisfies all the version constraints (including transitive dependencies) ... assuming there is one such version!

```
# dir  lib/python3.6/site-packages/
easy_install.py
setuptools-39.0.1.dist-info
pip
tornado
pip-9.0.1.dist-info
tornado-6.0.4.egg-info
pkg_resources
tornadoweb
pkg_resources-0.0.0.dist-info
tornadoweb-0.0.21.dist-info
__pycache__
wheel
setuptools
wheel-0.34.2.dist-info
```

1.2.6 requirements.txt (pip)

Python dependencies are specified with a `requirements.txt` at the root of the project. It's a plain text file, one library per line.

It can be written manually or it can be automatically generated with `pip freeze > requirements.txt` after installing packages.

```
pkg-resources==0.0.0
tornado==6.0.4
tornadoweb==0.0.21
```

It's good practice to "freeze" all dependencies (pin to a specific version) once you get the application working. Then they can simply be installed by running `pip install requirements.txt`. Reproducible build!

Note when writing a library, don't pin to an exact version but restrict by minimum/maximum version (`tornado>=6.0.0`). If every goddamn library pinned an exact (but slighly different) version, it would be impossible to meet transitive requirements.

1.2.7 The definitive way to run any python application

The application repository must have a `requirements.txt` at the top of the project.

Run:

1. git clone https://example.com/myproject
2. cd myproject
3. /usr/bin/python3 -m venv env
4. source ./env/bin/activate
5. cd env
6. pip install requirements.txt
7. python3 src/myapp.py

It's quite straightforward really.

1.2.8 How does the virtual environment work?

It's fairly simple, calling env/bin/active sets some environment variables. https://docs.python.org/3/using/cmdline.html#environment-variables

```
VIRTUAL_ENV="/home/user/env"
PATH="/home/user/env/bin:/usr/local/sbin:/usr/local/bin:..."
```

Alright, the virtual environment sets the virtual environment variable, that's a bit of a circular explanation. ^^

Let's run in a different way.

```
# run from a clean shell
export PYTHONPATH=/home/user/env/lib/python3.6/site-packages
/usr/bin/python3
import requests
requests.__version__
```

Python looks for the import based on the `PYTHONPATH`. PYTHONPATH is a list of python source directories separated by a colon (semicolon on Windows). It's like `PATH` for the shell.

The virtual environment is a bit of abstraction around that, it's autoconfigured for its location and it also copies a python interpreter (see `env/bin/python[.exe]`).

1.2.9 How to deploy the application? (for server distribution)

The application source code -with the requirements.txt- is self-contained. It is ready to deploy and run anywhere as long as there is a python interpreter and pip available.

First Time Setup: `/usr/bin/python3 -m venv env && source ./env/bin/activate && cd env && pip install requirements.txt`

Run: `source ./env/bin/activate && python3 myapp/main.py`

Warning: Directly calling the interpreter (`/path/to/python3 myapp/main.py`) -without sourcing the environment- can work because the venv interpreter is preset to lookup imports in the venv, however environment is not fully configured and it can cause issues down the line (notably $PATH is not set so `python` and `pip` binaries won't be found).

For **jenkins**: Checkout, setup and run the main unit test script.

For **ansible/salt/puppet**: Checkout and setup. Recommend to run the application as a systemd or supervisord service to start automatically and restart on failure. http://supervisord.org/

```
[Unit]
Description=myapp

[Service]
ExecStart=/path/to/env/bin/python3 /path/to/env/myapp/main.py
WorkingDirectory=/path/to/env
Environment="PATH=$PATH:/path/to/env/bin"
Environment="VIRTUAL_ENV=/path/to/env"
```

For **chef**: Migrate to ansible. (Only half kidding. I've only ever heard horror stories about Chef and every single company using it was migrating away. If you're on Chef you're probably looking to move for real.)

For **container/kubernetes**: Run the commands above in the `Dockerfile`. Set the CMD to `. /opt/venv/bin/activate && exec python myapp.py`

Warning: **Python does not support ``alpine`` linux**, gotta stick to a classic ubuntu/debian/centos image.

1.3 Advanced Usage

1.3.1 How to embed dependencies?

Python dependencies were installed in /lib/ as we've seen. It's possible to repackage these.

For example, this is what's available after `pip install requests`.

```
$ ls -l lib/python3.6/site-packages/

certifi/
certifi-2020.6.20.dist-info/
chardet/
chardet-3.0.4.dist-info/
easy_install.py
idna/
idna-2.10.dist-info/
pip/
pip-9.0.1.dist-info/
pkg_resources/
pkg_resources-0.0.0.dist-info/
__pycache__/
requests/
requests-2.24.0.dist-info/
setuptools/
setuptools-39.0.1.dist-info/
urllib3/
urllib3-1.25.10.dist-info/
```

One can do a `pip install --target=mypackages requests` to extract one package (including its dependencies) to a specific location.

```
$ ls -l mypackages/

certifi
certifi-2020.6.20.dist-info
chardet
chardet-3.0.4.dist-info
idna
idna-2.10.dist-info
requests
requests-2.24.0.dist-info
urllib3
urllib3-1.25.10.dist-info
```

Here's what you should recognize:

1. The actual python libraries, that's what is imported on `import requests` and similar.
2. Metadata on each library in the `.dist-info`. Not required to use the library. Only used by pip commands.
3. A variety of setup tools in the venv (`pkg_resources`, `pip`, `setuptools`, `easy_install`, `wheel`).

That last list is the absolute minimal set of dependencies to use requests (skip the `.dist-info`).

The directory can be used like this `PYTHONPATH=/home/user/mypackages /usr/bin/python3 /path/to/myapp.py`.

It can be copied around, repackaged, redistributed, stored in source control... with one caveat.

These are prebuilt packages handpicked by pip for compability with the current system (here python 3.6 on Linux 64 bits glib). There is no guarantee that they work on other OS and platforms.

It's fine to run internally on identical Linux servers (typical use case for server applications). It's not fine to distribute across Windows/Linux/Mac, consider having a separate build per platform to capture the right artifacts.

Pure python packages are often quite portable (watch out for supported python versions!), mixed C packages are often not so portable. When considered critical, compatibility can be improved substantially by carefully handpicking each package and version to maximize compatiblity. Also, avoid taking on more dependencies in the first place when possible.

Story time: I've done some crazy software archeology in a pension fund. Think, software intending to last more than a decade (#NotJavascript). It was hardly touched in the past 10 years, it is being upgraded, it will be barely touched for another 10 years. The sane thing to do is to store everything -code and libraries- into the git (#SVN) repo and deploy as a unit. It's actually best practice, one cannot assume that anything will survive on this sort of timescale (real life lesson: sourceforge and google code used to host a ton of projects and they've disappeared). With extreme care to handpick specific versions and to avoid dependencies unless absolutely necessary, things can run indifferently on RHEL 6 python 2.6 (2010-2024) and RHEL 8 python 3.6 (2019-2031). The saved libraries can get an upgrade when RHEL 6 is out and when RHEL 10 is in. The project shall work in 2030 on RHEL 10 python 4.5 with minimal adjustments if any.

1.3.2 Using different Python interpreters (different Python versions)

`python3 -m venv myenv` will create a virtual environment using the current python interpreter. Looks like it's symlinking the binary to the interpreter.

```
user@ubuntu:~$ ls -l env/bin/
total 36
-rw-r--r-- 1 user user activate
-rw-r--r-- 1 user user activate.csh
-rw-r--r-- 1 user user activate.fish
-rwxr-xr-x 1 user user chardetect
-rwxr-xr-x 1 user user easy_install
-rwxr-xr-x 1 user user easy_install-3.6
-rwxr-xr-x 1 user user pip
-rwxr-xr-x 1 user user pip3
-rwxr-xr-x 1 user user pip3.6
lrwxrwxrwx 1 user user python -> python3
lrwxrwxrwx 1 user user python3 -> /usr/bin/python3
```

What if you want a different version of the interpreter? First, need an interpreter. Download an interpeter. For Linux, recommend to use packages from the distribution. For Windows, download a prebuilt interpreter from the official python.org site.

I won't get into how to build an interpreter because this is no trivial task and there is no reason to do that (let other people do that work for you!)

```
apt-get install python3.7
apt-get install python3.7-venv
apt-get install python3.8
apt-get install python3.8-venv
```

```
user@ubuntu:~$ ls -lh /usr/bin/python*
lrwxrwxrwx 1 root root    9 /usr/bin/python   -> python2.7
lrwxrwxrwx 1 root root    9 /usr/bin/python2  -> python2.7
lrwxrwxrwx 1 root root    9 /usr/bin/python3  -> python3.6
```

(continues on next page)

(continued from previous page)

```
lrwxrwxrwx 1 root root   10 /usr/bin/python3m -> python3.6m
-rwxr-xr-x 1 root root 3.5M /usr/bin/python2.7
-rwxr-xr-x 2 root root 4.4M /usr/bin/python3.6
-rwxr-xr-x 2 root root 4.4M /usr/bin/python3.6m
-rwxr-xr-x 2 root root 4.7M /usr/bin/python3.7
-rwxr-xr-x 2 root root 4.7M /usr/bin/python3.7m
-rwxr-xr-x 1 root root 5.0M /usr/bin/python3.8
```

```
rm -rf myenv
/usr/bin/python3.8 -m venv myenv
ls -lh myenv/bin/
```

```
...
-rwxr-xr-x 1 user user  220 pip
-rwxr-xr-x 1 user user  220 pip3
-rwxr-xr-x 1 user user  220 pip3.8
lrwxrwxrwx 1 user user    9 python -> python3.8
lrwxrwxrwx 1 user user    9 python3 -> python3.8
lrwxrwxrwx 1 user user   18 python3.8 -> /usr/bin/python3.8
```

You're now able to create a virtual environment for any python version at your disposal. Create separate virtual environments (distinct directories) to work on multiple python versions in parallel.

1.3.3 How to embed the interpreter? (for end-user distribution)

The virtual environment is self contained. It can basically be archived and distributed, including all libraries and the interpeter.

Quick sample of what's there:

```
# Linux
=========
myenv/
| lib/
| -- python3.7/<packagename>
| share/
| bin/
| -- activate
| -- pip
| -- python
...
```

```
# Windows
=========
myenv/
| Include/
| Lib/
| -- site-packages/<packagename>
| Scripts/
| -- Scripts/activate.bat
| -- Scripts/pip.exe
| -- Scripts/python.exe
...
```

(continues on next page)

(continued from previous page)

```
# Linux
drwxr-xr-x   5 user user myenv/
drwxr-xr-x   3 user user |--lib/python3.x (dependencies)
drwxr-xr-x   2 user user |--bin
```

Remember that these are prebuilt executables and libraries for the current system (Windows/Linux/MacOS, 32bits/64bits, etc...).

On Windows, a prebuilt interpreter (download from python.org) will probably work fine when distributed to other machines (watch out 32 or 64bits!), Windows is a stable ecosystem with very high care to maintain compatibility.

On Linux, a prebuilt interpreter (and libraries) will probably NOT work fine when distributed to other machines, as-in a different Linux distribution or a different major version. Typical Linux compatibility hell. Recommend to rely on the `python3` (or a `python3.x`) package from the system, that interpreter will work for sure, the setup ought to create a venv and pip install the dependencies on the fly. If one really wants to embed everything for distribution, having a separate build+artifact per OS/distro/architecture may be the way to go, with a ton of testing to make sure they all work, think carefully about what you want to support, it's no accident that Linux is not associated with thriving commercial software (bit of server software on RHEL, bit of end-user software on Ubuntu).

On MacOS, not sure.

1.3.4 Mixed-C packages

Note: All the major packages I could think of work out of the box (wasn't the case 5 years ago) so you're very unlikely to need this chapter.

We've seen enough to cover 95% packages, including all pure python dependencies.

Now let's see the remaining 5% of difficult packages, namely packages that rely on C (or C++).

Funfact: The world runs on C software. All the current operating systems, Linux and Windows and Mac, are almost entirely made in C and exposing C API.

There are some great languages and ecosystems built atop (python, java, ruby, PHP, etc..) yet things eventually come down to C to interface with the system (or reuse one of the existing 1 trillion library). Python packages are no exception, many packages rely on embedded C code or system specific syscalls. https://www.geoffchappell.com/studies/windows/win32/kernel32/api/index.htm

Some examples of packages downloaded with `pip install <name>`:

- .../requests-2.24.0-py2.py3-none-any.whl
- .../cryptography-3.0-cp35-abi3-manylinux1_x86_64.whl
- .../numpy-1.19.1-cp36-cp36m-manylinux1_x86_64.whl

Notice the full name for requests `-none-any`, implies the package is generic, should work everywhere windows/linux/32bits/64bits. Definitely a pure python package.

Notice the full name for cryptography and numpy, `cp35-abi3-manylinux1_x86_64` and `cp36-cp36m-manylinux1_x86_64.whl respectively`, implies they are prebuilt package for linux for 64 bits.

pip gets a prebuild package for the current system.

In rare occasions the library will fail to install or will fail when called. This is usually due to missing C/system dependencies. For example a database client may expect to find the databaseclient.dll on the system, distributed separately. (Definitely had a horrible experience with setting up mysql, cassandra and oracle clients 5 years ago, the first two seems to work out of the box now).

When things are failing to install, google the error message and that should give a pretty good idea of what's the missing bit.

On Linux this sort of issues can usually be resolved by an `apt-get <commonlibrary>` of the right thing, for example a systemwide ssl library or a sql client.

On Windows this sort of issues usually doesn't happen. Windows has stable API and ABI, unlike Linux distributions that are a lose blob of random packages constantly changing under your feet.

Last but not least, the most hardcore packages in my experience, are those trying to compile stuff on the fly during installation. I've often found that this implicitely requires the full build chain on the system, a good starting point is `build-essentials gcc g++ make automake`. Go from there and google any error message.

1.4 Specials

1.4.1 Proxy

https://pip.pypa.io/en/stable/reference/pip_install/

pip requires internet access to download packages. It's possible to specify a proxy:

```
export http_proxy=https://proxy.example.com:8443
export https_proxy=https://proxy.example.com:8443
```

Note that the environment variables are case sensitive on some systems, the correct case is **lowercase**.

1.4.2 Internal Repo

pip can download packages from an internal repo instead instead of https://pypi.org/simple

`pip install --index-url https://pypi.mycompany.com/simple`

The usual commercial tools can store python packages: Artifactory, Nexus, GitLab, etc...

1.4.3 Download a package in a specific location

```
# To setup the package in a specific location (include all dependencies by default)
``pip install --target=mypackages requests``
```

```
# Don't setup dependencies.
``pip install --target=mypackages --no-deps requests``
```

```
# Download the package (zip archive or equivalent)
``pip install --target=mydownloads requests``
```

1.4.4 Alpine Linux is not supported

Warning: **Python does not support ``alpine`` linux**, a light Linux distribution commonly used by containers to save a few MB on the generated image. Stick to a traditional debian/ubuntu/centos image when using containers (kubernetes).

Why? Python binaries are compiled for `glibc`, while alpine is based on `musl` not `glibc`. Remember the tag `manylinux` in pip packages, that's a build specification around Linux and glibc. https://www.python.org/dev/peps/pep-0571/ https://github.com/pypa/manylinux

musl is an independant implementation of the libc (the C base library, providing `printf() strlen() time() malloc()` etc...). It's close enough from the glibc ABI/API for a compiled executable to load and run (rather than crash outright), but it's different enough to have different results with awful consequences, like the python interpreter being twice as slow or subtle memory corruptions (malloc() is highly implementation specific and the glibc one is very optimized for performance). https://superuser.com/questions/1219609/why-is-the-alpine-docker-image-over-50-slower-than-the-ubuntu-image

1.4.5 Note on `venv` vs `virtualenv`

There is a `virtualenv` tool roughly equivalent to the `venv` tool with a couple more flags.

The only notable usage is `python3 -m virtualenv --python=python3.6 myenv` which creates an environment using the given python interpreter (here it will look for a binary `python3.6` on the system).

It's the same as `/usr/bin/python3.6 -m venv` (venv creates the environment using the current interpreter).

1.5 About

1.5.1 Contributions

This documentation is stored on GitHub and automatically hosted on ReadTheDocs.

Send fix and bug report to GitHub (see `Edit on GitHub` button in the top right corner). Send discussion and feedback to Hacker News / Reddit.

Welcome:

- Review for typo/grammar
- Run commands
- Document tricky edge cases (linux/windows/mac differences for example)
- The guide is released at 4000 words, could really use a re-read (that would be a 25 pages book).

NOT welcome:

- Don't split this guide into multiple pages. A single document is always better.
- Don't try to push other tools. The python standard is venv+pip and has been for almost a decade. Couldn't care less about your personal competing project.
- This guide doesn't aim to cover how to build python-with-C packages (another long subject!). Only how to use prebuild packages.

1.5.2 Change Log

- August 2020: Initial Release