Contents

pybuild23

Streamline Python venv deployment w/Python 2.6, 2.7 and 3.x using a simple make command. Common dependencies (`pip`, `virtualenv`, `setuptools`) included, all UNIX operating systems and Linux distributions supported going back to Solaris, 9, AIX 5.1, RHEL 5.2, etc. Mac is probably supported, but you may need to change the path in the `Makefile` to point to the bizarro path to the system Python.

This tool is *not* meant to "manage" virtual environments, it is meant to *deploy projects*. It allows for quick, repeatable builds using just a simple `make` command. It works on many systems, including those with only Python 2.6. It is meant to also work when your system does not have `pip`, `setuptools`, `virtualenv`, etc. which can save you a step or two. It is really best for quick deployment

How It Works

Check the `packages/` directory. You will see it contains the bare minimum packages required for each major version of Python to successfully build a virtualenv and automatically install your required packages

Note that nothing you add to the packages directory will be added to the project thanks to `.gitignore` so you do *NOT* need to `make clean` before a check-in

One can say this is a bit of bloat, but it comes out to 23MB and it rarely changes (only for updates) so checkout and push operations are not terribly slow

Depdendencies

- Python 2 and/or Python 3
- GNU make
- OS/distribution provided headers and libraries specific to your project's dependencies (via `apt`, `yum`, `pacman`, etc..)

That's all, everything else is built in, which is kind of the point

Intended workflow

Clone your project from a `git` repo with `pybuild23` already installed. Tun `make && source venv/bin/activate`, and launch your project. That's all. Easy, repatable, deployment. See `make freeze` for more on the "repeatable" apsect of this.

NOTE: the sample `pypirc` file and the `setup.cfg` file may need to be changed to fit your project name, organization name, organization specific Artifactory URL., etc..

Basic usage

There are a lot of `Makefile` targets at this point (creeping featurism, *cringe*)

The most basic and useful ones are outlined here

Basic Build

Running GNU `make python2` or `make python3` will build your project based on your `venv/project-requirements.txt` and `venv/constraints.txt`

Cleaning Up After A Build Or Publish Operation

You can use `make clean` for this. It cleans up a bunch of temporary junk that tends to get left by builds. Could use some improvement.

Installing `pybuild23` Into A Blank `git` Project

Use `make new REPO=ssh://github.com/you/new_empty_project` **WARNING, THIS WILL COMMIT CODE (*but will not push*)**

If you have created a new project in a `git` repository and there are already files present, manually install `pybuild23` into it pretty easily, being weary of filename collisions. The follow should be copied using `cp -a` into your project

- `.gitignore`
- `pybuild`
- `packages/`
- `Makefile`
- `venv/`
- `venv/requirements.txt` (containing only some linting tools as requirements)

Rebuilding A Virtual Environment

You can use `make rebuild` which is equivalent to `make clean && make`

Testing

There is a `test` target in the Makefile that currently does nothing at all

Setting Up Publishing For You

You can use `make pypirc` to generate a `~/.pypirc` for publishing to PyPi repositories (PyPi or Artifactory are the most common). It is interactive. You may need to fill in some identifying info, i.e. URLs for internal Artifactory instances if you use them. It just uses a simple template and `sed` to file in a username, password and publish URL

Publishing Your Package Using `setup.py`

Using `make publish` makes publishing to Artifactory or PyPi simple assuming your `~/.pypirc` is set up correctly. It basically does `setup.py sdist upload -r local && upload -r local dist/* --verbose`

You should either have a global version of `twine` and `setuptools` installed, or be inside the virtual environment when publishing

Releasing Your Project

If and only if you use `versioneer` you can make use of the `make release` command

Using `make release` To Simplify Projects Using `versioneer`

First, `make release` is not magic. It relies on the `versioneer` package being set up correctly. If you are not using `versioneer`, skip this section

`$ make release bump=major` - Bump the major version via a git tag `$ make release bump=minor` - Bump the minor version via a git tag and publish `$ make release` - Bump the revision via a `git tag` and publish

Locking Your Dependency Versions

Using `make freeze` will freeze your virtual environment by placing a `requirements.txt` file in `venv/`. The file will be named as `/home/debian/pybuild23/venv/frozen-requirements-YYYY-MM-DD.SS`. You can then manage restoring it in the future. Make sure you check it in.

Very, very useful when doing active development on internal libraries. Using `make freeze` basically just invokes `pip freeze` within the context of the virtual environment. It does ***NOT*** overwrite your `venv/requirements.txt` file but it will save a dated copy with pinned versions in `repeatable-deployments/`. It is suggested that you check theses files in, and as such they are not in `.gitignore`

Example file after `make freeze` of a deployed application:

```
10:22:41 > ls -l venv/frozen-requirements-*             -rw-r--r-- 1 debian debian 1180 Feb 25 10:22
venv/frozen-requirements-2019-02-25.36
```

Make sure you use this feature at the appropriate time as you don't want to memorialize broken deployment files. If you have problems building with newly developed libraries or your application itself, you can replace your current `requirements-project.txt` with the frozen one to reproduce a prior deployment while you fix any regressions you have introduced

Development Makefile Targets

These should not be used unless you are working on development for `pybuild`

Making Documentation

Use `make doc` to produce a pretty PDF document. This is developer only

Default `venv/requirements.txt`

There are three different requirements files. The first simply includes the other two which are documented below. This is easy to do using `-r` inside the `requirements.txt` file

The `requirements-base.txt` File

Packages that are considered useful (according to me!) for enforcing or measuring code quality will be included in the default list of packages via `requirements-base.txt` as well as tools to publish, such as `twine` which has officially replaced the old-school ways of uploading Python packages using `setuptools`. Twine is more modern as it is better about using TLS, and probably other things ...

There are a handful of other packages, mostly linting and code quality tools. You don't need to use them if you want and they will only add about a minute to your virtual environment deployment so chill

The `requirements-project.txt` File

This is **your** `requirements.txt` file and the one you should change and keep up to date with your project's dependencies

Default `venv/constraints.txt`

This is blank by default to avoid errors. Add to it if necessary, or leave it blank

Common Sense Stuff (How Virtual Environments Work)

Virtual environments are just self-contained Python installations, essentially. To build one, you typically use the `virtualenv` tool. You also use a `requirements.txt` file to tell `pip` which packages you need in your virtual environ. Using `make python2` or `make python3` via `pybuild` abstracts all this from you and it removes the necessity for the few dependencies required to build virtual environments. Using `pybuild` directly is also quite easy, but the provided `make` targets are more convenient.

After building a virtual environment, you "enter" it by sourcing a file that introduces some Python related environment variables, modifying your `$PATH`, and defining a `deactivate` shell function you can use to leave the virtual environment. Your prompt will have a `(venv)` string in it if you're in a virtuale environment, usually.

Contributing

Please contribute by first entering an Issue, not a Pull Request. That way you can avoid wasting your time in case a feature is not desired, or a claimed bug is actually a feature, or whatever. It happens. Then you can fork the project and make a PR and reference the Issue.

Please avoid forking a completely new project if you can as I think it is counterproductive and I think this tool is getting close to the dreaded "creeping featurism" that W. Richard Stevens warned us all about :>

Credits

- Concept and original implementation by David Marker
- Re-implementation, Python3 support and a gazillion new targets including `versioneer` versioning by Adam Greene
- Continuous updates to package version by Adam Greene
- Currently maintained by Adam Greene

License