Contents

pybuild23

Streamline Python venv creation w/Python 2.6, 2.7 and 3.x using a simple make command. Common dependencies (pip, virtualenv, setuptools) included, all UNIXes and Linux distributions supported going back to Solaris, 9, AIX 5.1, RHEL 5.2, etc, etc.

Why is this “a thing” that you spent time on?

It allows for quick, repeatable builds using just a simple `make` command.

How does it work?

Check the `packages/` directory. It contains the bare minimum packages required for each major version of Python to successfully build a virtualenv and automatically install your required packages. If you set a PYTHONPATH and VIRTUAL_ENV (and one or two other things) you can “work” inside this tiny installation of Python to bootstrap your application.

Note that nothing inside of the `packages/` directory will be checked in, so you do *NOT* need to `make clean` before a check-in.

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.

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, internal Artifactory URL., etc..

Basic usage

`make`

Running GNU `make` with no arguments will default to building a python 2 environment.

`make python`

Equivalent to `make`

`make python3`

I bet you know what this is ..good with patterns eh?

`make clean`

Cleans up a bunch of temporary junk that tends to get left by builds

`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 container of sorts, you can pre-install `pybuild23` into it very easily. This command will basically copy the following: * `.gitignore` * `pybuild` * `packages/` * `Makefile` * `venv/` * `venv/requirements.txt` (containing only some linting tools as requirements)

`make rebuild`

Equivalent to `make clean && make`

`make test`

Does nothing, stops `Makefile` linters from complaining :>

`make pypirc`

Use this 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

`make publish`

Makes publishing to Artifactory or PyPi simple. It basically does `git push && python setup.py sdist_upload -r local` under the assumption that you have bumped your version in `setup.py`. While `commit` and `push` aren't actually needed to publish a package, it's a good practice, especially if you tag your releases.

`make freeze`

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 `.gitignore`'d

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

```
$ ls -l repeatable-deployments/
total 4
-rw-rw-r-- 1 adam.greene adam.greene 386 Sep 15 20:40 codefreeze-requirements.txt.frozen-deplay-2018-09-15
```

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.txt` with the frozen one to reproduce a prior function deployment while you fix any regressions you have have introduced.

Default `venv/requirements.txt` entries - what's up with those?

Packages that are considered useful for enforcing or measuring code quality will be included in the default list of packages, as well as tools to publish, such as `twine` which is officially in the process of replace the old-school ways of uploading Python packages. It's got encryptions!!111

Some will be more useful than others, feel free to remove them, but there is really very minimal downside to leaving them in.

Common sense stuff, how virtual environments work, etc...

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` abstracts all this from you. Using `pybuild` directly is also quite easy, but `make` is more convenient for sure.

After building a virtual environment, you "enter" it by sourcing a file that introduces some Python related environment variables, modifyinh 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 by Adam Greene
- Continuous updates to package version by Adam Greene
- Currently maintained by Adam Greene,

License

See LICENSE