Python Project Structure, Tooling, and Building

Written by Mike mattie (codermattie@runbox.com)

Code: pythonsh on GitHub

I have extensive experience in professional and personal projects with Python. From that experience clearly one of the most poorly thought out and vexing aspects of development is just getting the code to run.

With a project composed of multiple repositories, multiple committers, and juggling development and release versions - various phases of development it is a bug ridden mess. Developers waste huge amounts of time just getting stuff synced and runing.

Issues

- Running code is fragmented process due to the use of either plain system interpreters, tox for multiple python versions, or pyenv for isolated virtual environment environments. Developers are often unaware of the different strengths and weaknesses of the tools and just copy the first thing they find on the internet.
- Python has many different versions, and each milestone introduces new features that code can become dependent on. Installation can be by a OS package, Open Source package managers like homebrew, or virtual environment interpreters like pyenv. This fragmentation leads to chaos with developers often not having a firm grasp on what python their code is actually running on.
- Backwards compatibility is an assumption with no explicit guarantee.
- Practices for maintaining release metadata that allow a project to rebuild a release from a given source version, interpreter, and module environment are not even considered much less implemented.
- Builds cannot be reproduced later if necessary which is not a professional way to build and distribute software. It should always be possible to reconstruct builds for any release.
- The python build system is frankly a disaster. It is both layered and fragmented with the latest standards encouraging this dumpster fire instead of putting the flames out. It's origins are setuptools with an executable config and an outdated egg format.
- The current format wheel files are still based on setuptools and layer on top of eggs for metadata construction. There are two major conventions for project layout: modules in the root of the tree, or python in a "src" sub-directory.
- setuptools can be configured either by an archaic executable, a setup.py file which requires execution of the setup.py to drive the build process embedding the entire build process in a rigid script.
- The second option is to configure setuptools with a setup.cfg file. This is much more sane but it is sabotaged by an archaic format that is basically just a config to drive the generation of a setup.py file.
- New tools and standards have been setup to deal with issues like the fact that a pypi package repository has to execute untrusted code just to get the version number and metadata for the project. The pep 517 standard address this by putting project

metadata in a pyproject.toml file. However this file is a complete mess as it is intaking keys for backend package managers with no policy or standards across them.

- Multiple backends is a total disaster encouraging instead of fixing the extreme confusion on how to do something as simple as build a package. Currently it is common to see setuptools, poetry, and flit backends.
- Many projects are composed of multiple modules, each in it's own repository. There is very little guidance on how to share developer snapshots between developers and coordinate releases. It is tribal knowledge, and ad-hoc and thin knowledge of how python is built and run.
- Another topic is that developers don't even know a proper way to integrate their source tree into the interpreter search paths and rely on hacked up scripts and PYTHONPATH hacks which are sanitized by isolated environment tools like tox, or pyenv.

Requirements

Falling into the pitfalls of scattershot python infrastructure and develop consumes untold hours of developer time and frustration and lays the groundwork for disaster. This is due to not having a solid project layout and code sharing conventions designed from experience and full knowledge of the techniques. Not being able to reproduce builds is unacceptable for professional development. Both for open-source projects and absolutely a disaster for enterprise development.

For organizations that rely on trusted development a new standard must be implemented in python development using a good knowledge of how python interpreters work, and a solid understanding of the tools for python environments, package building, and source control methods.

- Effective use of source control is essential. git-flow is an absolute must for managing the complexity and pandoras box of git workflows. It's systematic approach to managing branching, multiple developers, and release best practices make it a must, not an option.
- Package management must be systematic. There must be an easy way to distribute snapshots to other developers for development which encompasses building, distributing, and consuming code through packages. I have a simple method for building and maintaining a package repository in AWS S3 buckets for private repositories using simple shell scripts.
- The interpreter must be locked to a specific milestone, built clean from source, and a virtual environment containing the packages for that development cycle maintained in a virtual environment.
- The interpreter must include all of the packages and the actual source for the repo in a clean way that doesn't rely on brittle shims like hacking PYTHONPATH in shell scripts. Executing the python interpreter should load all of the code without tweaking the invocation of python.

- It must be possible to build every release of the code simply by cloning the repo, checking out a tag, copying a couple of files like Pipfile and Pipfile.lock, and building.
- The build tools and config files should be constructed from thorough knowledge of the python interpreter and build tools instead of just hacked up from stack overflow and other similar sites.
- A virtual environment is an absolute must. The code cannot depend on system or package manger python as which version of python the code uses is up to selecting a python from a swamp of system and package manager versions with it being a roulette of which python is actually used.

The goal of this document

I am going to share my experience, knowledge earned through much pain of searching, experimenting, and suffering through broken projects to build a guide to project management and python tooling techniques.

Source Control

Git is a powerful tool that is used almost universally at this point for personal, professional, and Open Source projects. It is however so powerful, and has an arcane syntax for the command line interface that it can sabotage a project as often as help it. The flexibility allows for almost any workflow which has lead to total chaos.

Let me be clear: the only sane way to use git has been developed from years of git suffering and is “git flow”: **git-flow** is a workflow and set of commands for managing branches, managing merging, and managing releases - all baking best practices. It codifies decades of git use into a toolset. It is easily installed via homebrew on MacOS.

It is also vital to normalize commit messages and for this conventional commits are key: **Conventional Commits**.

Virtual Environments

Python can be installed into virtual environments that isolate packages and build a specific interpreter. This is vital so the python environment doesn’t become polluted with random python tools installed for general system use. Packages must update for fixes and bugs but API packages and the python interpreter must be locked to a specific milestone release while updating patch releases.

The two main tools for this are tox and pyenv. Both are effective but have different use cases. Tox is good for dealing with multiple versions of python where you have to target python2 and python3, or multiple versions of Python 3.

Targeting the ancient Python2 versions is no longer much of an issue as support is either stopped or will be stopped soon and Python2 has been end of life'd for many many years.

Most projects will use various versions of python3 requiring only a minimal version, not multiple versions side by side.

For modern codebases pyenv is the preferred tool as it constructs a virtual environment for python and builds from source the correct version of python. It has effective and simple tools for managing packages and running python and other tools in the virtual environment.

Pyenv is easily installed via homebrew on MacOS.

```
 "install-tools")
       brew update

       brew install pyenv
       brew install pyenv-virtualenv
       brew install git-flow
   ;;
   "update-tools")
       brew update

       brew upgrade pyenv
       brew upgrade pyenv-virtualenv
       brew upgrade git-flow
   ;;
```

Repository layout is critical too. The source tree must be organized so that both code is easy to write, the layout facilitates packaging, executing, and testing. Here is the layout I recommend.

```
src/<module> - modules under src
src/<module>/config-dev.py - the configuration of the module
                             I have a tool for managing AWS
                             configuration.
src/<module/config-release.py - release config
src/module/config.py -> /src/<module>config.py - manage
                        environments with a symlink

src/scripts  - python scripts to be packaged


tests/<module>
tests/interactive.py


Pipfile
Pipfile.lock (when frozen)



releases/Pipfile-0.1.0 - Pipfiles so a release can be
                          reconstructed.
releases/Pipfile.lock-0.1.0

package.sh  - whatever you want to call it but a systematic way
              to manage tooling, packaging, and

 releases.

.gitignore  - (dist/, __pycache__, .egg-info)
```

Managing Python and virtual environments

Managing the interpreter, local source, and packages can be challenging but it is essential to get it right so you aren’t chasing difficult bugs due to incorrect versions of source being used, such as stale code from other developers.

First thing is to install the virtual environment.

Here is a snippet from my package.sh. virtual prefix is the name of the project to manage multiple environments since pyenv has a global namespace for virtual environments.

```
VIRTUAL_PREFIX="config"

#
# virtual environments
#
    "virtual-install")
        pyenv install --skip-existing "$PACKAGE_PYTHON_VERSION"

        LATEST=$(pyenv versions | grep -E '^ *\d+\.\d+\.\d+$' |
sed 's/ *//g')

        echo "installing $LATEST to $VIRTUAL_PREFIX"

        pyenv virtualenv "$LATEST" "${VIRTUAL_PREFIX}_release"
        pyenv virtualenv "$LATEST" "${VIRTUAL_PREFIX}_dev"
    ;;
    "virtual-destroy")
        pyenv virtualenv-delete "${VIRTUAL_PREFIX}_release"
        pyenv virtualenv-delete "${VIRTUAL_PREFIX}_dev"
    ;;

    "virtual-list")
        pyenv virtualenvs
    ;;
```

Second step - Activate the environment with: “pyenv activate <virtual environment>”

Third step - integrate your source tree into the python search path so the python interpreter can find your code when it executes. Using .pth files in site-packages is the cleanest way to do this.

```
function add_src {
    site=`pyenv exec python -c 'import site;
print(site.getsitepackages()[0])'`

    echo "include_src: setting dev.pth in $site/dev.pth"

    test -d $site || mkdir -p $site
    echo "$PWD/src/" >"$site/dev.pth"
    echo "$PWD/scripts/" >>"$site/dev.pth"
}

function remove_src {
    site=`pyenv exec python -c 'import site;
print(site.getsitepackages()[0])'`

    echo "include_src: setting dev.pth in $site/dev.pth"

    test -d $site || mkdir -p $site
    echo "$PWD/src/" >"$site/dev.pth"
    echo "$PWD/scripts/" >>"$site/dev.pth"
}

"paths")
        shift
        add_src
        pyenv exec python -c "import sys; print(sys.path)"
    ;;
```

Python is run by using pyenv:

```
"python")
        shift
        pyenv exec python $@
    ;;
    "run")
        shift
        pyenv exec $@
    ;;
    "test")
        pyenv exec python -m pytest tests
    ;;
```

Next you need a Pipfile and a system for separating development packages from release packages. Here is an example Pipfile from one of my projects.

As you can see I have pinned python to 3.10, and for development packages I always use versions below 1.0.0 so I can write a version expression that always grabs the latest dev packages.

For releases the version expression would be version=“>=1.0.0”

Note that I am also using a private repository. I have a set of scripts that generate a python package repository in a S3 bucket which is pretty fun: python-dist . It generates proper metadata and uploads repository metadata and packages to an S3 bucket. This allows developers to easily share code.

See the next page for the example:

```
[[source]]
url = "https://pypi.python.org/simple"
verify_ssl = true
name = "pypi"

[[source]]
url = "http://python.bitcathedrals.com/simple"
verify_ssl = false
name = "bitcathedrals"

[global]
trusted-host = "python.bitcathedrals.com"

[packages]
boto3 = "1.24.5"

DeepDiff = "*"

cfconfig = {version = "<1.0.0", index = "bitcathedrals"}

[dev-packages]
pytest = "*"
black = "*"
awscliv2 = "*"
build = "*"

[requires]
python_version = “3.10"
```

To update the virtual environment with the latest packages I have a command that updates the virtual environment:

```
pyenv exec python -m pip install -U pip
pyenv exec python -m pip install -U pipenv
pyenv exec python -m pipenv install --dev --skip-lock
pyenv rehash
```

First it updates pip and pipenv and then it installs the latest version of all the packages, and then updates the program index for pipenv.

This combined with the .pth shim for the source repository bring all python code in the repository up to date.

Building Python

After looking at the packaging options I chose setuptools because it is the classic tooling for building packages. It doesn’t try to add a bunch of extra features in the package management domain to what is for me just a simple as possible build of packages.

There are two files: pyproject.toml and setup.cfg.

pyproject.toml is the project metadata and setup.cfg is the build configuration.

Some of the statements in setup.cfg are being migrated to pyproject.toml. That is beta so I don’t use that here. Here are some examples of the files.

This is the pyproject.toml for my CFconfig project which I try and keep to metadata only, however the project.scripts must be specified here or you will get some big warning banners on build.

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

[project]

name = "cfconfig"
version = "0.7.0"

license = { file = "LICENSE" }

readme = "README.md"

description = "CloudFormation configuration and deployment."
authors = [ { name = "Michael Mattie" , email =
"codermattie@runbox.com" } ]

[project.urls]
homepage = "https://github.com/coderofmattie/cf_config"
repository = "https://github.com/coderofmattie/cf_config.git"

[project.scripts]
awsconfig = "scripts.makeconfig:exec"
```

Next is the setup.cfg file. This is truly ugly using a bizarre custom syntax for structure, and complex source mapping capabilities. I won’t go into the details of how this file works but information on the web is redundant and simplistic so I recommend searching the python docs if you need to tweak this:

```
[metadata]
name = cfconfig

[options]
package_dir=
    cfconfig=src/cfconfig
    =src/scripts

packages=cfconfig
py_modules=
     makeconfig
     make deploy
```

makeconfig and make deploy are my scripts. I specify it as a module instead of a package. package_dir is a mapping from the package name to the dir of the package. packages is the list of packages to include.

Source Release

For building it is important to use git flow to manage the release process. After all the feature forks, and direct commits have been made to the develop branch you can cut a release.

First start with git flow with something like this:
git flow release start <version>

This puts you on a release branch where you can perform release tasks and tweak any file, bump versions, and release patches if necessary. Also you should test the build to make sure it builds correctly.

Then you will want to build, finish the release, and deploy.

git flow release finish <version>

It will take you through several merges and finalize the release in the source repository.

When the code is ready it is time to build and deploy.

I have a script for this as well that illustrates the steps.

Update dependencies, snapshot the versions to the lock file, and copy the Pipefile and lock to releases.

```
"release-start")
    pyenv exec python -m pyenv -m pip -U pip
    pyenv exec python -m pyenv install -U pipenv
    pyenv exec python -m pipenv install --ignore-pipfile
    pyenv rehash

    test -d releases || mkdir releases
    pyenv exec python -m pipenv lock

    mv Pipfile.lock releases/Pipfile.lock-$VERSION
    cp Pipfile releases/Pipfile-$VERSION
;;
```

When the source code work is finished and the build is tested its time to push the release to the repository:

```
 "release-finish")
    git push --all
    git push --tags
;;
```

Finally you need to build and deploy the packages:

```
"deploy-intel")
   pyenv exec python -m build

   find . -name '*.egg-info' -type d -print | xargs rm -r
   find . -name '__pycache__' -type d -print | xargs rm -r

   DIST_PATH="/Users/michaelmattie/coding/python-packages/"
   PKG_PATH="$DIST_PATH/simple/cfconfig"

   test -d $PKG_PATH || mkdir $PKG_PATH
   cp dist/* $PKG_PATH/
;;
```

Once this is done it’s a series of steps specific to my setup to upload the packages to the repository and update the repository metadata.

That is it! A workflow for managing the source, building, and deploying aspects of a project.

Configuration

Configuration of packages and resources is major issue that is a hand hacked mess in a lot of projects. The issue is that when you have resources in the cloud the identifiers like ARN’s and resource id’s aren’t known until you deploy. Users of these resources need the identifiers baked into the code so you can use those deployments as resources.

The problem is exasperated by multiple environments like dev, test, and prod.

I have a project CFconfig that addresses this issue in AWS by providing template building, deploying, and configuration generation in a single cohesive system.

This avoids the most common situation of maintaining a config file for the lambda or EC2 instance and updating it by hand or with some terrible script.

Link to CFconfig Cloud Deploy + Configure

That pretty much wraps it up. I hope this paper helps you layout, manage, and utilize a project tooling and dev environment to maximize your precious time!

Link to pythonsh (this code)

Thanks,
Mike mattie (codermattie@runbox.com)