ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-MOPV049

STANDARDIZING A PYTHON DEVELOPMENT ENVIRONMENT FOR LARGE CONTROLS SYSTEMS *

S. Clark, P. Dyer, S. Nemesure, BNL, Upton, NY 11973, U.S.A.

Abstract

Python provides broad design freedom to programmers and a low barrier of entry for new software developers. These aspects have proven that unless standardized, a Python codebase will tend to diverge from a common style and architecture, becoming unmaintainable across the scope of a large controls system. Mitigating these effects requires a set of tools, standards, and procedures developed to assert boundaries on certain aspects of Python development — namely project organization, version management, and deployment procedures. Common tools like Git, GitLab, and virtual environments form a basis for development, with in-house utilities presenting their capabilities in a clear, developer-focused way. This paper describes the necessary constraints needed for development and deployment of large-scale Python applications, the function of the tools which comprise the development environment, and how these tools are leveraged to create simple and effective procedures to guide development.

GOALS

Python has grown in popularity since its 1991 inception, with wide use in scientific and analytic applications. The myriad libraries released to simplify complex tasks — such as Numpy and SciPy for scientific calculations, and the PyQt5 user interface toolkit for application development — have driven increased adoption within the Collider-Accelerator Department (C-AD) Controls Group at BNL. As this adoption began, many developers created simple scripts scattered across the filesystem, which grew into operation-critical applications over many years. Long-term maintenance of these scripts is difficult for future developers who must now not only learn the codebase, but also the unique project structure and procedures of dozens of disparate programs. The primary goal of the Python development environment is to alleviate the manageability issues described above.

PYTHON DISTRIBUTION

An essential requirement for Python usage in a large complex must be to establish a common base across all workstations. The Anaconda Python distribution provides exactly this, with each version containing a specific set of Python binaries and packages. Additionally, Anaconda is well-supported with first- and third-party tools to ease maintenance and deployment of distribution upgrades.

Initially, the Anaconda distribution was installed to a network mount which was globally available to all hosts, and any necessary packages (in addition to the standard set) were installed directly upon request. However, two primary issues were encountered with this method: performance and maintainability. Performance issues appeared shortly after the adoption, which manifested in slow application launch times. Latency and bandwidth limitations over the network were discovered as the probable cause. Maintainability was further impacted by cascades of package upgrades triggered by new package installations, leading to previously-working scripts and programs breaking due to backwards incompatibilities unless careful intervention was taken during the process.

* Work supported by Brookhaven Science Associates, LLC under Contract No. DE-SC0012704 with the U.S. Department of Energy

The deployment philosophy for the Anaconda distribution was changed to account for both issues. As mentioned above, inhibited performance was found to stem from network latency. During investigation, locally-installed copies of the Anaconda distribution showed a marked decrease in application launch time. Resulting from the discovery, the Anaconda distribution was moved from the network share to local disk on each host requiring Python. The tool `conda-pack` [1] facilitates this by allowing system administrators to create a portable clone of the Anaconda distribution which then can be extracted and installed locally on machines for use.

On a yearly schedule, the Anaconda distribution is rebuilt and redistributed. This provides an opportunity for Python and package upgrades, base package additions, as well as other general maintenance of the distribution. The newest distribution is constructed on a machine, tested for general compatibility, and finally deployed to all machines using the `conda-pack` tool as described above. Two local copies of Anaconda are maintained upon release of a new distribution — the new version and the prior version – guaranteeing Applications two years of first-class support, allowing developers to migrate to newer Anaconda releases at their own pace. Distributions prior to the two locally-kept copies are available on the network share. This limits the local disk usage by Anaconda versions, but reintroduced issues with network latency. This, however, should be mitigated if applications are regularly maintained and released using the latest available distribution.

PROJECT CREATION & ORGANIZATION

A unified project architecture is key for ongoing maintainability and standardization of development procedures. The details of how projects are structured at the file level are equally important as aspects of development such as a standard package base, ensuring that developed procedures work equally across any project created within the development environment.

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2022). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-MOPV049

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2022). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

A custom utility, `caddy`, was created to ensure all projects follow a unified layout. Caddy is based on the open source Python package CookieCutter, which allows for the development of project templates consisting of boilerplate files and code for a project. The templates utilize a syntax which allows developers to insert values into the boilerplate based on input received when creating the initial project; these values, which may include the project name, author, and options to Git-version the project, modify both file names and contents to suit the new project. In addition to the basic project structure and contents, the templates may also optionally include a script to be run when a project is initially created. The development at C-AD heavily relies on Python virtual environments, so this post-creation script automatically spins up a virtual environment and installs base requirements upon project creation, as well as sets up a Git repository for the new project. Templates have been defined for several types of project bases — such as graphical user interfaces, command line applications, web applications — and placed into Git repositories for later reuse.

The templates used by Caddy were designed to conform to the Python Packaging User Guide (PyPUG). Following PyPUG allows all applications and packages to be installed into virtual environments; this simplifies development and deployment as is discussed later, and prevents potential lock-in that a nonstandard packaging system would introduce. Additionally, in-house packages which may benefit the broader community can easily be shared on package hosts like the Python Package Index.

The process of creating a project from a template has been automated by the Caddy script to ease barriers to entry and to minimize user error. The creation process consists of two parts: selecting a template, and defining values. The user is given a list of available templates obtained from GitLab, and makes a numerical selection for the desired project type. Then Caddy prompts the user for each variable used within the template and ensures valid values are given for each. Once complete, Caddy clones the template into a new directory specified at the prior prompt, performs template variable substitution, and runs the post-generation script associated with the template. At this point, the user is informed of the new project location and may begin development. The automation afforded by setting up projects through Caddy allows developers to focus less on project setup and boilerplate, and more on the business logic of the application. Refer to Fig. 1 for a demonstration of this process.

VERSION MANAGEMENT

Version management is another critical piece of Python development given the speed at which programs evolve. When compared to other version control systems (VCS) such as ClearCase, Git provides three main benefits: free software, mature and complete implementations, and broad community support. These aspects make Git a modern, well-known, and cost-effective choice for development. Git offers a relatively simple set of core features which can be expanded upon over time, easing the learning curve for those new to Git.

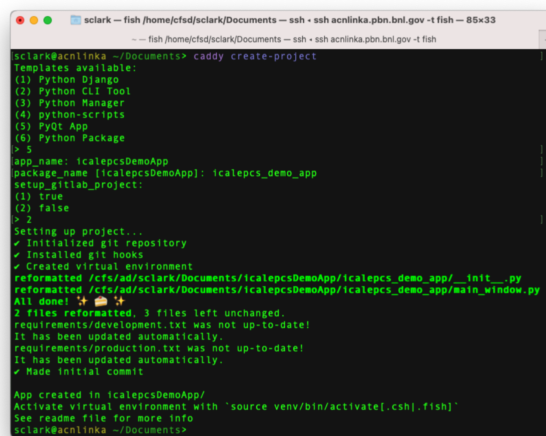

Figure 1: Use of `caddy` to create project, include template selection & variable definition.

A platform to host source code and version information is also required. Many solutions exist for Git-based repositories, and GitLab was chosen out of the offerings due to the availability of a free, self-hosted version of the platform, as well as additional project management features included in the platform. The self-hosted option offers greater control of the installation and data storage, while still allowing flexibility to migrate to an enterprise tier with additional features and commercial support. GitLab's primary feature, aside from version control, is a robust continuous integration and deployment (CI/CD) system. At C-AD, this has been leveraged to automate the building and deployment of Python packages and applications; this is further discussed below.

From a conceptual standpoint, a coherent version numbering system is necessary to facilitate tracking and upgrading various packages and applications within the development environment. The Semantic Versioning Specification (SemVer) was used, which operates under the principle that version numbers contain intrinsic meaning by defining major, minor, and patch revisions. For example, a package version `v3.21.5` is known to be the 3rd major revision, 21st minor revision, and 5th patch revision. When used as defined in the specification, a change in the major revision indicates that the new release breaks backwards compatibility. Changes in minor or patch revisions must maintain backwards compatibility however, and either add new features or fix existing bugs, respectively. This standard allows developers to be aware of changes they may encounter when upgrading dependencies for a project, and also ensures the smooth installation of packages into virtual environments as is defined by the PEP440 specification used by Python. [2]

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-MOPV049

DEVELOPMENT & DISTRIBUTION

Applications

Applications created in the development environment must explicitly disallow untracked modification of released executables. Releasing Python source files directly to a writeable directory was initially viable; however, as programs increased in complexity, multi-file applications became unmanageable and prone to untracked changes. The following requirements for a Python application build system were identified: performance must be responsive for user-facing applications; all necessary files and any package dependencies must be bundled with the application; and changes to released applications must be made through version control to ensure all version history is tracked.

Various solutions were considered to bundle applications, including Python's native ZipApp module, Facebook's XAR technology, Pants build system's PEX, and LinkedIn's Shiv tool. ZipApp was ruled out as its limitations on inclusion of C-extension module created a cumbersome caveat for developers. [3] XAR showed initial promise in its ability to bundle all source files and dependencies, but was found inadequate due to the overhead of cutting-edge, OS-level dependencies which are unsupported by less recent Linux distributions. A significant trial was performed with PEX, but despite having all features necessary, launching of bundled executables was not performant enough for user-facing tasks due to reliance on the `pkg_resources` library. [4] Finally, Shiv was found marketed as a performance-optimized tool with all features offered by Pex. After many weeks of testing, Shiv was found to satisfy all feature and performance criteria.

Shiv allows applications to be released as pseudo-binary files, whereby all source code and dependencies are bundled into a ZIP file containing a specific syntax to allow execution by the Python interpreter. Since all contents are contained within a single archive, modification of released applications is more difficult than helpful; one would need to extract the archive, make changes, re-archive with the correct Python-specific syntax, and copy over the existing binary. This obfuscation enforces developers' use of version control for projects.

Further, since all applications bundle dependencies within the binary, different applications may rely on differing package versions. This gives developers additional flexibility to upgrade dependencies (or not), irrespective to the underlying Python distribution. Some larger dependencies, such as PyQt5, would greatly increase disk usage by executables if they were to be included in each archive, so often-used packages are included as a part of the Anaconda distribution so any application can access those packages without bundling them in the archive.

Packages

Initial Python modules developed were released directly as source into a common directory on the `PYTHONPATH`, making the modules accessible to Python scripts system wide. This mode of development was easy, as no special setup was needed to use them. But, as some breaking changes were introduced to these common modules, any dependent script or application broke immediately with no procedure to revert to a prior version. So a method to version custom packages — just as is done through the Python Package Index (PyPI) — was necessary.

The development environment, as mentioned previously, specifies that packages must be structured to conform to the Python Packaging User Guide. An effect of this is that packages are able to be built by the `setuptools` module into `tar.gz` archives for later installation. Using this capability, in-house packages are bundled into versioned archives when released; these act as snapshots of the package at release time. These bundles are then placed into a common directory and can be installed into other projects indefinitely, regardless of any future development or releases of the package. Pip, the Python package manager, was configured globally to search this common directory for packages when a user makes an installation request. Developers wishing to use first-party packages can install them into a project's virtual environment just as they would with any third-party package available through PyPI in order to enforce strict versioning, as opposed to making them available as a part of the Anaconda distribution.

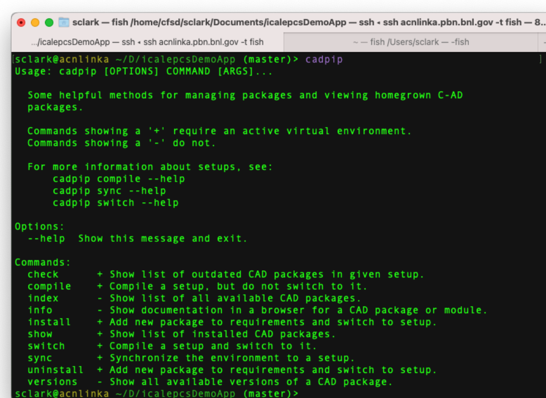

Figure 2: Various `cadpip` options for managing project packages.

Python's default Pip provides a very simple package management interface, but lacks any higher-level functions such as version conflict resolution or the notion of multiple package sets, such as for development and release. Various tools were considered for this task, such as Poetry and Pipenv, but were either immature or lacking in recent development at the time of investigation. So, a custom script `cadpip` was created to handle version resolution and package set management. Package requirements are defined in two files: `requirements/production.in` and `requirements/development.in`. Packages listed here can either be loose or pinned, meaning any version may be used or a specific version is required, respectively. Running the `cadpip switch {p,d}` command updates pack-

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2022). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-MOPV049

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2022). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ages to the latest available versions which satisfy the constraints in the `production.in` and `development.in` files, respectively. It then uses `piptools` to bring the current virtual environment into sync with the selected package by adding, updating, and removing packages as necessary. The final set of package versions which were installed are then written to the files `requirements/production.txt` and `requirements/development.txt` depending on which package set a user chose; these files always contained pinned packages, and are used to exactly recreate the virtual environment when a project is either cloned or built. Refer to Fig. 2 for a sample of the tool's capabilities.

RELEASING PROJECTS

GitLab's CI/CD feature is leveraged during the release process, as mentioned in the Version Control section, which hooks into Git procedures to trigger predefined actions on a remote server. The actions performed when a project is released include: optional unit testing with PyTest, building projects using either Shiv or setuptools (described above), and releasing the build artifacts to appropriate locations for use.

The benefits of using CI/CD as opposed to building and releasing locally are two-fold. First, projects being built are guaranteed to be in an unmodified environment. If a user were to build a project locally, modifications to the Python environment may impact the end product, such as packages installed in the home directory or changes to the `PYTHONPATH` environment variable. CI/CD ensures that the build environment is unchanged by running builds in a sandboxed virtual environment under a dedicated account which maintains the default settings. So, if tests pass and a package builds, it can reasonably be expected that the application or package will work on any machine setup with the Anaconda distribution used for the build. Additionally, by executing all build and release procedures from a dedicated account, the paths applications and packages are released to may remain read-only to other users to prevent any modifications.

Bringing together semantic versioning with the CI/CD process is a lightweight shell script, `git release` (as shown

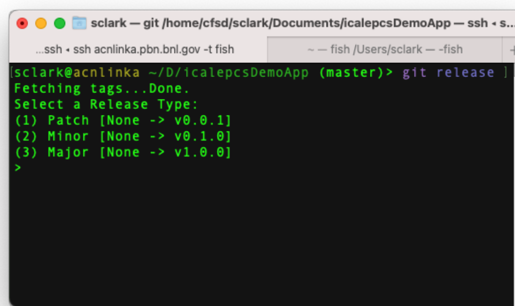

Figure 3: `git release` prompting for version selection prior to releasing software.

in Fig. 3). `git release` makes the release process as simple as a prompt for users. It parses the Git history to find the most recent release number, and calculates new version numbers for major, minor, and patch releases. The user may select one of the three, which `git release` then tags the most recent commit with and pushes to GitLab. This formally kicks off the release process, at which point `git release` displays a progress bar tracking the build process from GitLab. If any errors occur, they are also printed to the console for inspection.

REFERENCES

[1] J. Crist. "Conda-pack." (2017), https://conda.github.io/conda-pack/
[2] N. Coghlan and D. Stufft, *Pep 440 – version identification and dependency specification*, 2013.
[3] P. S. Foundation. "Zipapp — manage executable python zip archives." (2021), https://docs.python.org/3/library/zipapp.html#caveats
[4] LinkedIn. "Shiv: Motivations & comparisons." (2021), https://shiv.readthedocs.io/en/latest/history.html