MOPV049

STANDARDIZING A PYTHON DEVELOPMENT ENVIRONMENT FOR LARGE CONTROLS SYSTEMS

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

ICALEPCS 2021

Python Distribution

- Locally-installed Anaconda distribution with fixed package set
- Uniform across all hosts for performance & maintainability
- Standardized annual upgrade

Project Organization

- Predefined templates provide standard project boilerplate
- Interactive tool to create new projects from templates
- Projects packaged using Python standards

Version Control

- VCS managed by Git + Gitlab
- Standardized versioning scheme for easy tagging & rollback
- Leveraging CI/CD for deterministic builds
- Simple CLI to initiate tag, build & release

Development & Distribution

- Process enforces version control procedures
- Final form both responsive & size efficient
- Released applications remain functional across Anaconda/system upgrades

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

Python Distribution

- Anaconda was chosen as a base
  - Provides a standard set of packages for use across the system
  - Well-supported with first- & third-party tools for maintenance
- Each year, a new custom distribution is created, including…
  - The latest Anaconda base release
  - Upgrades to existing third-party packages from last release
  - Additional third-party packages requested to be included
- The final distribution is stored locally on all hosts
  - *conda-pack* is used to bundle the distribution for release to hosts
  - Done for performance considerations; previous distributions served from network mounts suffered considerable performance penalties
  - Once on a host, the distribution must not be changed
- Distributions kept locally for two years
  - Then moved to network storage for long-term availability
  - Limits disk usage by distributions

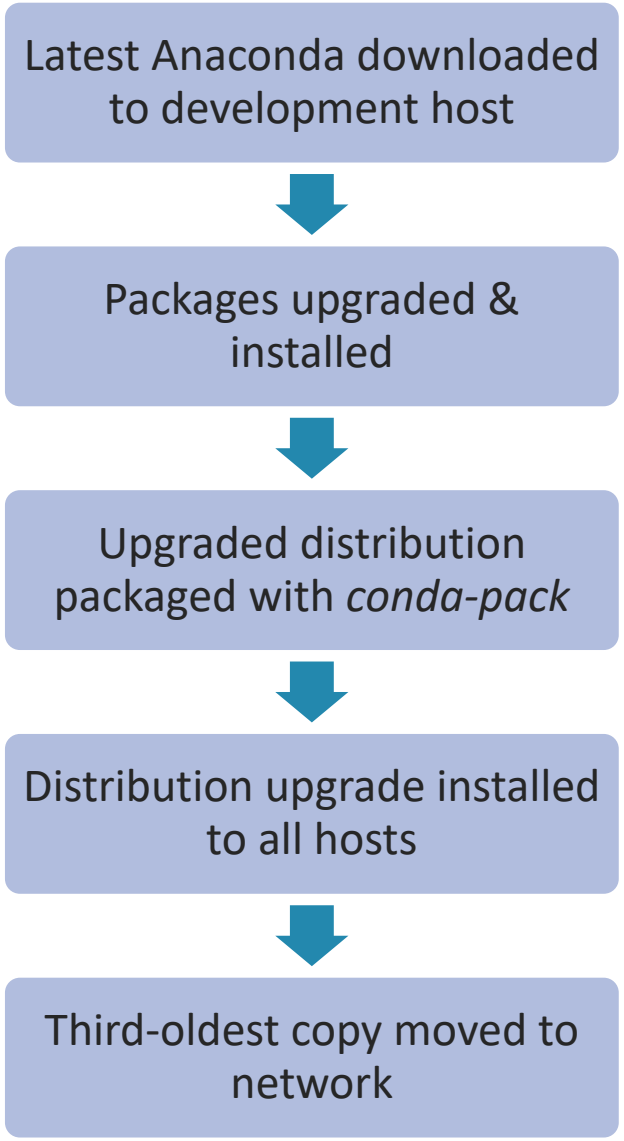

Figure 1: Anaconda upgrade procedure

Project Organization

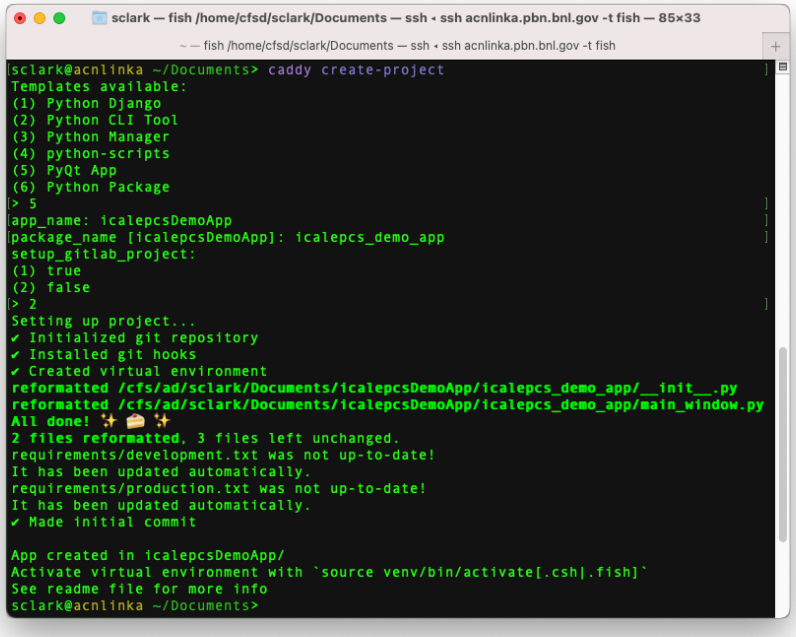

Figure 2: caddy project setup process

- Projects must contain certain uniform elements
  - *setup.py* file defining package metadata (name, author, etc.)
  - Requirements files for defining development & production dependencies
  - Build definition for Gitlab CI/CD
- Templates were developed to define uniform structure
  - Each project type (GUI, CLI, package, web application) has a dedicated template
  - Templates contain boilerplate code for projects; e.g., default main widget for GUI, Django dependencies for web apps, additional *setup.py* configuration for packages
  - Templates may specify creation scripts to aid in project setup; e.g., setup a git repository & Gitlab project, create the project virtual environment
- Custom CLI tool *caddy* automates project setup
  - Provides list of available templates
  - Prompts user to define template variables (name, whether to setup Gitlab, etc.)
  - Creates project & runs post-creation scripts if available
- Developers can focus more on development & logic rather project setup

Version Control

- Git chosen for VCS base
  - Free & open source software with great support
  - Commonly used & low learning curve
- Gitlab is used for central repository
  - Free & open source, but with optional upgrade for enterprise support
  - Many additional features aside from VC (continuous integration/deployment)
- Semantic Versioning
  - Each software release tagged with version number
  - Triggers CI/CD process when pushed to Gitlab
  - Provides meaning to versions (see Figure 4)
- *git release* wrapper simplifies process
  - Provides user with next valid semantic version choices
  - Runs optional pre-release scripts; e.g., code formatting & unit testing
  - Pushes tag to Gitlab to start release process

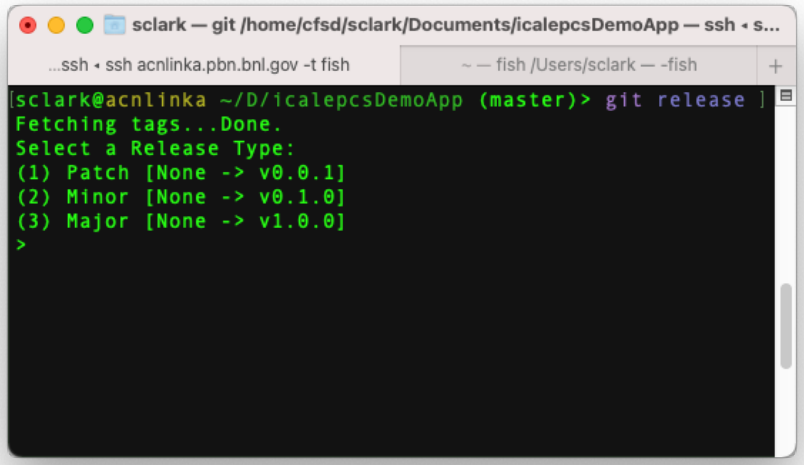

Figure 3: git release prompt

Given a version number MAJOR.MINOR.PATCH, increment the:

- MAJOR version when you make incompatible API changes,
- MINOR version when you add functionality in a backwards compatible manner, and
- PATCH version when you make backwards compatible bug fixes.

Figure 4: Semantic Versioning summary (from https://semver.org/)

Development & Distribution

Dependency Management

- Pip offers “dumb” dependency management
  - No concept of production vs development packages, dependency version conflict resolution, etc.
- Custom *cadpip* tool was developed to fill these holes
  - Based on *piptools* package
  - Handles dependency conflict resolution
  - Maintains separate development & production package sets

Virtual Environments

- Allows multiple projects to use additional third-party packages & different package versions
- Projects also have access to packages in the Anaconda distribution
  - Installing large & common packages (Qt, Numpy, etc.) in the base keeps applications sizes down
- Can be precisely recreated for deterministic project development by new users

Applications

- Distributed as single file packages
  - Created using Shiv tool from LinkedIn
  - Bundles all source & dependencies in executable ZIP file
  - Maintains responsive performance
- Stored on network mount for universal access

Packages

- Built using the Python Packaging User Guide
- Packages bundled into *.tar.gz* files by *setuptools* & placed in shared directory for later installation
- Pip configured to search for custom packages