The Power of Poetry

Introduction

It's safe to say automation isn't going anywhere, and since Python is the de-facto standard for automating network operations these days, it obviously plays a crucial role in many of our instantiated tools and workflows. As a developer of automation, I believe it is worthwhile to explore a tool called, "Poetry" that is currently utilized in a growing number of Ansible & Python projects here at Fiserv. Before we dive in, a few key words:

- Module: A file containing python code.
- Package: A folder containing python files (often called library).
- Distribution: An achieved module/package.
- Packaging: Creating and uploading a Python distribution to a package registry to be used by yourself and others
- Dependency Management: Declaring, resolving, and using dependencies required by the project in an automated fashion.

The What

Poetry is an all-in-one tool for managing Python project: **dependencies**, **environments**, **packages** & most importantly ensuring everyone has the right stack everywhere.

Virtual Environments

A key aspect that has fueled Python's popularity is the rich plethora of packages, or in other words, reusable chunks of code that are great for neat & speedy development. However, before importing and using a package, the package's code must be present in your environment. In regards to Python, there are generally two types of "environments":

1. Physical environment: The system-wide (global) Python distribution that is installed on your machine. When installing Python packages globally there can be only one version of a package across all your programs. This means you'll quickly run into version conflicts.
2. Virtual environment: An isolated Python environment. Physically, it lives inside a folder containing all the packages and other dependencies, like the Python interpreter itself and native-code libraries, that a Python project needs. However, each virtual environment has it's own instance of the Python interpreter that does not affect the system-wide instance. Meaning, any changes to dependencies installed in a virtual environment do not affect the dependencies of the other virtual environments *or* system-wide libraries.

Fortunately, Poetry provides a built-in feature to create an isolated virtual environment for any given project.

Dependency Management

When your Python project relies on external packages, ensuring they're installed is one side of the coin. However, you also need to make sure you're using the right version of each package. After an update, a package might not work as it did before the update. Additionally, many Python libraries depend on non-Python code and, it gets even more complicated when your dependencies have dependencies of their own that may even be version specific. A dependency manager like Poetry automates the heavy-lifting specifying, installing, and resolving external packages in your project in an automated. This way, you can be sure that you always work with the right dependency versions on any machine.

Packaging

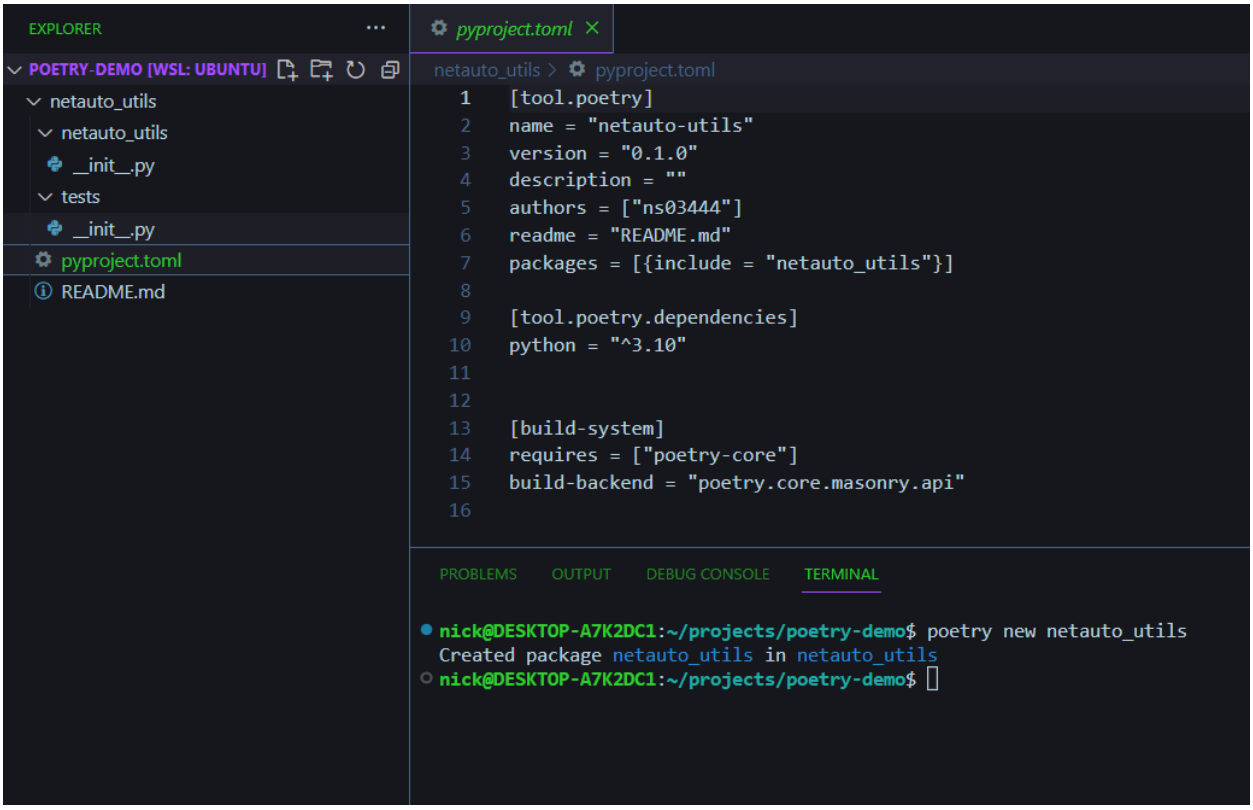
If you've ever written any Python code, I am sure you are familiar with Python's built-in package manager, pip. Poetry also utilizes this CLI tool under the hood to install packages directly from the official Python package index (Pypi) to your project's environment. Additionally, Poetry uses pip to publish (upload) packages to Pypi. In the process, poetry removes manual efforts such as a setup.py, or other configuration files that aid installing Python packages as a dependency .

The How

This section examines the simplistic process I recently used to publish a package called "netauto_utils" (Network Automation Utilities), which consist of a few modules that I've already wrote, so we'll just copy & paste them to our package folder later in the section. We'll start by creating a new project from scratch, building a package, and publishing it. Note, the images shown were produced in a WSL shell with Poetry already installed. If you do not have Poetry installed, you can follow along here.

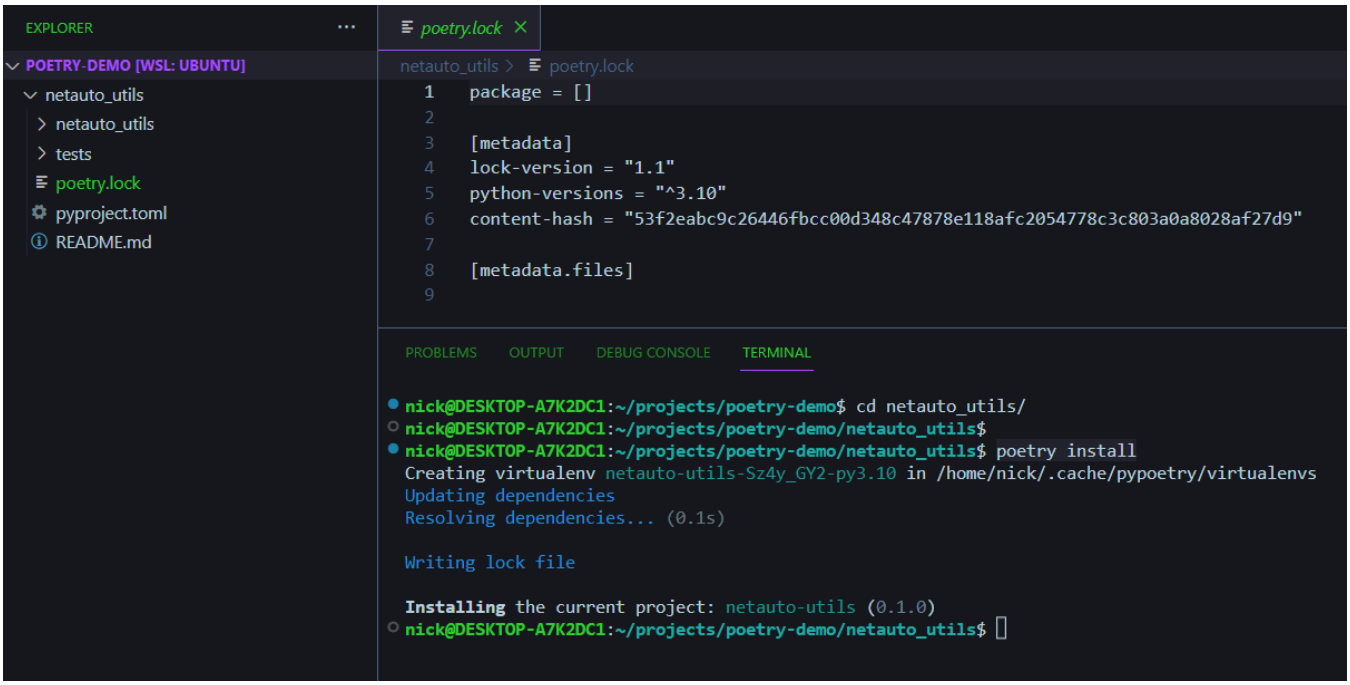
Getting Started with Poetry

In the image below, files are automatically generated when the **poetry new (followed by a *project-name)*** command is executed. The pyproject.toml file manages all your project's settings & dependencies. This file should not be edited directly, poetry will take care of that upon command executions. Additionally, a README.rst, test folder, and another subfolder also named *netauto_utils* which contains __init__.py. By definition, any folder containing this file (along with other python files) is considered a "package" (basically just making it importable). Having said, this will be important later, as this is also the folder where all modules (python scripts) will be stored. We will use this package throughout the demo, however, since Poetry is the main focus, we will not get into the actual code or functionality of each module but feel free to check them out here.



The following pictures illustrate the **poetry install** command. Note, there are 2 distinct use-cases for this command:

- Running it for the first time in a project.
- Running the command after it has been previously executed in a project.



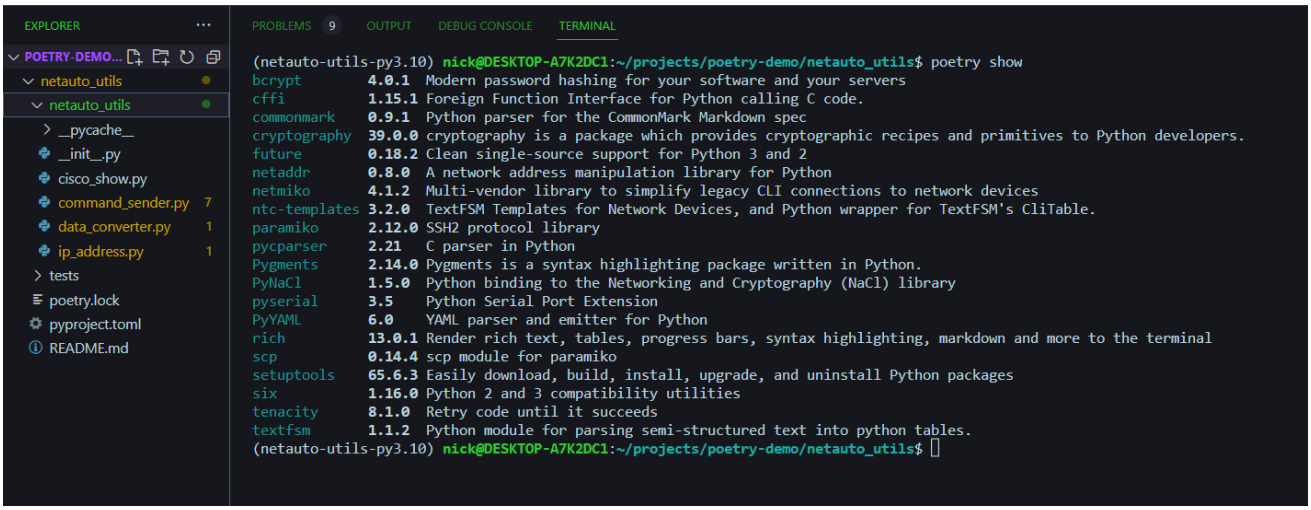
First, Poetry resolves all dependencies listed in your pyproject.toml file and downloads the latest version of their files. After the installation finishes, it writes all the packages and their exact version to an automatically generated poetry.lock file, locking the project to those specific versions. Additionally, a new virtual environment is created.

In the 2nd case, when a poetry.lock file is already present in a project, it is safe to assume that the install command has been ran before. Therefore, Poetry resolves and installs all dependencies listed in pyproject.toml, but instead, uses the specified version number in poetry.lock to ensure that the package versions are consistent for anyone working on the project. Once the installations complete, a developer is ready to create with the full power of poetry.

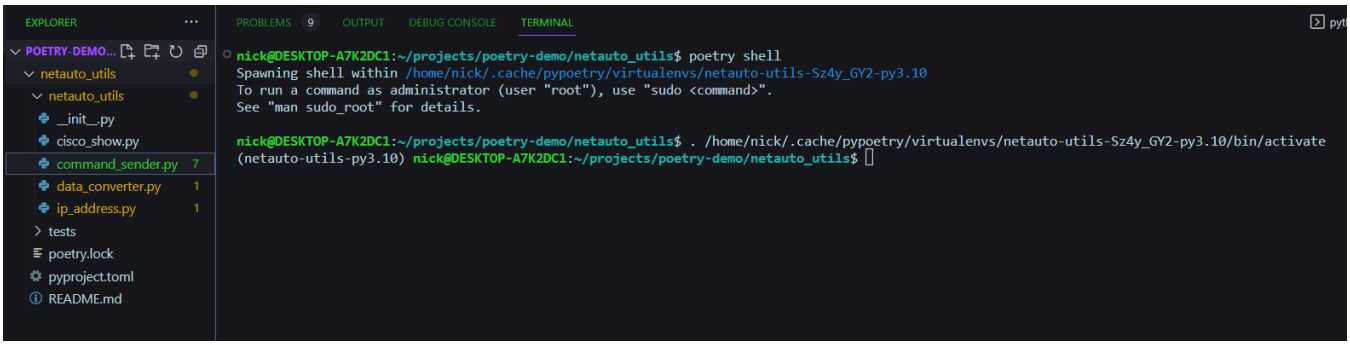
Next, we'll take a look at some of the more commonly used Poetry commands. One command being "poetry add" (same functionality as "pip install"), which we will use to install 3rd-party packages that my modules depend on. With that said, we'll begin by copy/pasting 4 Python modules (Python files that I wrote in advance) to the *netauto_utils* package folder described earlier, and then, walkthrough a few examples of the Poetry: **add, show, & shell** comman ds. Note, although we are only copy/pasting in this tutorial, this would be the stage at which a developer begins creating their modules, or "developing".

- **Poetry add**: adds required packages to your pyproject.toml and installs them in the virtual environment.

- **Poetry show**: lists all packages installed with a description.



- **Poetry shell**: activates & enters the virtual environment



Last but not least, I'll introduce 3 final commands to illustrate the essence of packaging with Poetry. First, **poetry build** takes the settings from the aforementioned pyproject.toml file to create the sdist, tarball, and wheel files that aid installing a Python package as a dependency.

Then, the **poetry config** command is executed along with my Pypi credentials. Technically, this step is not necessary as Poetry publishes to Pypi by default, however, you can use the same syntax to override the default in the case that you want to publish to a private registry.

```
PROBLEMS 2  OUTPUT  DEBUG CONSOLE  TERMINAL

nick@DESKTOP-A7K2DC1:~/projects/poetry-demo/netauto_utils$ poetry config pypi-token.pypi <API Token>
```

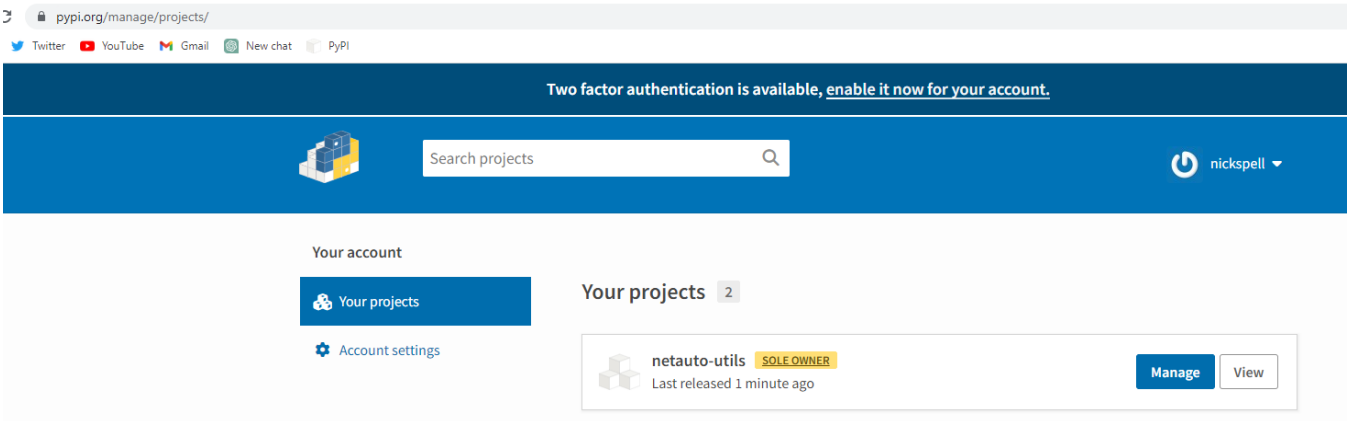
Lastly, the **poetry publish** command is executed to officially upload the package to Pypi (It can also build the package if you pass it the *--build* option but, *--build* is **not** required).

```
PROBLEMS 2  OUTPUT  DEBUG CONSOLE  TERMINAL

nick@DESKTOP-A7K2DC1:~/projects/poetry-demo/netauto_utils$ poetry publish --build
There are 2 files ready for publishing. Build anyway? (yes/no) [no] yes
Building netauto-utils (0.1.0)
  - Building sdist
  - Built netauto-utils-0.1.0.tar.gz
  - Building wheel
  - Built netauto_utils-0.1.0-py3-none-any.whl

Publishing netauto-utils (0.1.0) to PyPI
 - Uploading netauto-utils-0.1.0.tar.gz 100%
 - Uploading netauto_utils-0.1.0-py3-none-any.whl 100%
nick@DESKTOP-A7K2DC1:~/projects/poetry-demo/netauto_utils$
```

As you can see, the new package has officially been uploaded. Now, any developer in the world can install & use my package in their own environment.



Although publishing to Python's official index can be deemed useful, hosting software on the public internet is not ideal in the corporate world. Fortunately, Git platforms offer PyPi repositories directly built-in to host & manage all our internal distributable packages, serving as a private package index. Having said, we can conveniently leverage the **poetry configure** command to easily reconfigure Poetry to use our private registry (index). Thus, our environment will be aimed to install or publish packages, to and from the private registry. Unfortunately, Gitlab package registries are a topic within themselves but checkout the official documentation for more info.

The Why

With any project, comes the burden of virtual environments, dependency management, & packaging. With that said, I am not arguing that Poetry is the best tool for dependency management, nor the best tool for packaging, or utilizing virtual environments, but *that it is the best for all of them*. So in conclusion, you may not like Poetry but, the power is undeniable and knowing how to use it will be instrumental for anyone looking to become a contributor of automation at Fiserv today. With that said, thank you for following along and if you have comments, questions, etc. please comment below!

Takeaways

- Automated dependency management saves countless hours & significantly increases product quality.
- Built-in Virtual Environment capabilities allow a developer to jumpstart their development process and more importantly makes for an excellent solution in collaborative processes.
- It works seamlessly across all operating systems. Whereas other tools may require different commands for different operating systems.
- Uses a single file to manage all your project's settings & dependencies. Thus, alleviating numerous steps from the historically tedious processes of packaging. So, if there's an argument to be made for packaging with Poetry, it's that it might just be a little too easy.
- Perfectly suitable in workflows geared for continuous integration & deployment (CI/CD).

Extras

Command Reference

Poetry Command	Explanation
`$ poetry --version`	Show the version of your Poetry installation.
`$ poetry --help`	Show all Poetry commands, what they do, & how to use them
`$ poetry init`	Add Poetry to an existing project.
`$ poetry run`	Execute the given command with Poetry.
`$ poetry add`	Add a package to `pyproject.toml` and install it.
`$ poetry update`	Update your project's dependencies.
`$ poetry install`	Install the dependencies.
`$ poetry show`	List installed packages.
`$ poetry lock`	Pin the latest version of your dependencies into `poetry.lock`.
`$ poetry lock --no-update`	Refresh the `poetry.lock` file without updating any dependency version.
`$ poetry check`	Validate `pyproject.toml`.
`$ poetry config --list`	Show the Poetry configuration.
`$ poetry env list`	List the virtual environments of your project.
`$ poetry export`	Export `poetry.lock` to other formats.

A Note from the Author

Admittingly, this post was written to only be read amongst an internal organization but has since been modified to cover the core purposes of Poetry. So, there was quite a bit of additional information I would have liked to include but could not fit. Here's a glimpse of some ideas I have gathered for my upcoming post that will essentially build from this one.

As package registries (or indexes) were mentioned quite a bit throughout this post, majority of the focus was on using Pypi. Whereas, it would have also been relative to explain private registries in more detail. This is a complex topic and unfortunately fell out of scope, but to give you an idea, these registries can be created at the group or project level to host packages on platforms like Github & Gitlab. Furthermore, the platforms also have a feature to host "container registries".

There will come a time where you want the code to be used by a wider audience. The next step is to get the code execution occurring within a Docker container that can be consumed by our automation platform. Next, these pieces are aggregated together by Gitlab's Continuous Integration & Deployment (CI/CD) functionality. CI is a common development practice in a shared project where developers commit code changes often. Upon each commit, a set of automated test and checks can be executed against the code to check for validity. CD is a similar concept that makes a code base deployable at any time, sometimes automatically. Each of these tools/processes play a vital role in the optimization of our products and resources. Finally, we can leverage Cookie cutter to create Ansible & Python template repositories. This templating library is for creating project boilerplates in any programming language. This way, developers can quickly start new projects that come built-in with all the aforementioned tools. Stay tuned to learn how these pieces fall in line to form a dominate framework to create, enhance or maintain automation.