Learning Objectives: Poetry

Learners will be able to...

- **Install Poetry**
- **Differentiate `poetry` from `pip`**
- **Add, update, and remove dependencies**
- **Identify how Poetry works with virtual environments**
- **Describe the purpose of the `pyproject.toml` file**
- **Differentiate between the `poetry.lock` and `pyproject.toml` files**
- **Add Poetry to an existing Python project**
- **Build a wheel with Poetry**

info

Make Sure You Know

You are familiar with virtual environments, add/removing/updating packages, publishing with `pip` and other built-in Python tools.

Limitations

You will only get a cursory introduction to development dependencies, running scripts, and publishing packages.

Poetry Package Manager

Why Use Poetry?

In the previous module, we saw how powerful `pip` can be; you can perform a wide number of tasks with relative ease. Poetry allows you to perform the same tasks, but it provides a level of automation that streamline working with packages.

Let's start by installing Poetry. The `curl` tool is used to send and receive data. In this instance, we are going to download the `install-poetry.py` script from the given website. We then take this script and pipe (pass) it to `python3`. The end result is that you download and run the `install-poetry.py` script with a single command.

```
curl -sSL https://install.python-poetry.org | python3 -
```

▼ **Did you notice?**
We did not start a virtual environment before installing Poetry. That is because we want it to be installed at the system level.

In order to verify that the installation worked, check the version of `poetry`.

```
poetry --version
```

You should see output similar to this. **Note**, your version may be newer than the version installed when this tutorial was written.

```
Poetry version 1.1.15
```

Creating a Package

When using `pip`, you are responsible for creating the different directories and files needed for a package. Poetry handles almost all of this for us.

```
poetry new learning-poetry
```

Poetry has created for us a directory for the newly created package. In addition, Poetry also created subdirectories as well as required files. To see what happened during the `new` process, change to the `learning-poetry` directory and call the `tree` command. `tree` prints a hierarchical representation of the directory, where you can see all subdirectories and files.

```
cd learning-poetry && tree
```

You should see the following diagram in the terminal.

```
.
├── learning_poetry
│   └── __init__.py
├── pyproject.toml
├── README.rst
└── tests
    ├── __init__.py
    └── test_learning_poetry.py
```

Poetry creates the `learning_poetry` subdirectory that contains the initialization file. The `pyproject.toml` file contains metadata about the packate — we will talk more about this file later as it is really important. The `README` file is used for documenting the package. Finally, Poetry creates a testing directory. It is setup such that you can run your tests with a simple command.

If you wanted to do the same with `pip`, you would be manually creating these directories and files. You may choose to ignore things like documentation and testing as they are not required. Using Poetry helps to enforce good programming habits.

▼ **Did you notice?**

Poetry took the package name `learning-poetry` and replaced the hyphen with an underscore when creating the `learning_poetry` subdirectory. Poetry takes the given input and produces output that conforms to the standards of the Python community.

If you want to avoid the naming changes Poetry makes, use the `-name` flag and the name you want to use. The example below keeps the hyphen.

```
poetry new learning-poetry --name learning-poetry
```

challenge

Try this variation:

- Create a package with a separate directory for the source code.

```
poetry new --src example2
```

Then print the tree structure of `example2` so that you can compare it to `learning-poetry`.

```
cd example2 && tree
```

▼ **What is happening?**
Poetry added the `src` directory to the package. This is where all the source code (initialization file and modules) for the package resides. Some programmers prefer to have a `src` directory to keep source files separate from the other files in the package.

Adding Dependencies

Add a Dependency

Before we can add dependencies to your package, we need to make sure that we are in the package directory. Change to the `learning-poetry` directory.

```
cd learning-poetry
```

In terms of the end result, there is no difference in using `pip` or Poetry when installing a package. Both of these package managers use PyPI when downloading and installing a package. You will not get one package from `pip` and another from Poetry. Poetry uses the `add` command to add a dependency. Let's add the `fire` package to our project.

```
poetry add fire
```

After the installation processes finishes, use the `show` command to list out all of the dependencies for the `learning-poetry` package.

```
poetry show
```

Poetry returns a nice list of dependencies with columns. You get the name of dependency, its version, and a short description of the dependency.

```
attrs              22.1.0 Classes Without Boilerplate
fire               0.4.0  A library for automatically generating
command line interfaces.
importlib-metadata 4.8.3  Read metadata from Python packages
more-itertools     8.14.0 More routines for operating on
iterables, beyond itertools
packaging          21.3   Core utilities for Python packages
pluggy             0.13.1 plugin and hook calling mechanisms for
python
py                 1.11.0 library with cross-python path, ini-
parsing, io, code, log facilities
pyparsing          3.0.7  Python parsing module
pytest             5.4.3  pytest: simple powerful testing with
Python
six                1.16.0 Python 2 and 3 compatibility utilities
termcolor          1.1.0  ANSII Color formatting for output in
terminal.
typing-extensions  4.1.1  Backported and Experimental Type Hints
for Python 3.6+
wcwidth            0.2.5  Measures the displayed width of
unicode strings in a terminal
zipp               3.6.0  Backport of pathlib-compatible object
wrapper for zip files
```

The `show` command shows the dependencies and their dependencies. What is not clear is what are the dependencies of the newly installed `fire` package. We can combine `show` with the flag `--tree` to show a hierarchical structure of dependencies.

```
poetry show --tree
```

We can now see that the `learning-poetry` package has two dependencies, `requests` and `pytest`. We can also see the dependencies for each of these packages.

```
fire 0.4.0 A library for automatically generating command line
interfaces.
├── six *
└── termcolor *
pytest 5.4.3 pytest: simple powerful testing with Python
├── atomicwrites >=1.0
├── attrs >=17.4.0
├── colorama *
├── importlib-metadata >=0.12
│   ├── typing-extensions >=3.6.4
│   └── zipp >=0.5
├── more-itertools >=4.0.0
├── packaging *
│   └── pyparsing >=2.0.2,<3.0.5 || >3.0.5
├── pluggy >=0.12,<1.0
│   └── importlib-metadata >=0.12
│       ├── typing-extensions >=3.6.4
│       └── zipp >=0.5
├── py >=1.5.0
└── wcwidth *
```

challenge

Try this variation:

- Install the `python-dateutil` package.

```
poetry add python-dateutil
```

Now view the dependencies in the tree form.

```
poetry show --tree
```

Just like `pip`, Poetry will let you install a package if it is hosted on GitHub. In the previous module, we installed `shapes-package` from GitHub. We can install it again using the command below. Be sure that you are copying the SSH address from the repository. Append `git+ssh://` to front of the SSH address, and Poetry can use this information to install the desired package.

```
poetry add git+ssh://git@github.com:codio-content/shapes-
        package.git
```

Updating

Poetry can also update dependencies. The `update` command will update all listed packages in the `pyproject.toml` file. We'll cover this file in more depth in a bit.

```
poetry update
```

If you want to update a specific package, use its name with `update`.

```
poetry update fire
```

You can update more than one package at a time if you list them out after `update`. Use a space to separate each package.

```
poetry update fire shapes-package
```

The ability of the `update` command to update a package is limited by information stored in the `pyproject.toml` file. The snippet below is an example from the `toml` file. This section lists all of the dependencies for a project. `fire` version 0.4.0 is required. However, notice the `^` in front of the version number. This symbol means that Poetry will make any update to `fire` as long as it less than 1.

```
[tool.poetry.dependencies]
fire = "^0.4.0"
```

Imagine if version 1 of `fire` was released after installing an earlier version of the package. Updating your project would not upgrade `fire` to the latest version. Instead, install the package again but append `@latest` after the package name. Poetry will install the latest version and update the `toml` file to reflect this change.

```
poetry add fire@latest
```

Removing Dependencies

Remove a Dependency

Start by changing into the `learning-poetry` directory.

```
cd learning-poetry
```

If `add` is used to add a dependency to a project, use `remove` with Poetry to remove a dependency.

```
poetry remove fire
```

Now, display the dependencies in list form.

```
poetry show
```

Compare the list here with the tree structure above. The `fire` package has `termcoor` listed as a dependency. Removing `fire` not only removes the package but also its dependencies (assuming they are not needed by another package). With `pip`, you would have to manually remove these dependencies.

challenge

Try this variation:

- Remove the `python-dateutil` package.

```
poetry remove python-dateutil
```

Now view the dependencies in the tree form; `python-dateutil` and its dependencies should no longer be listed.

```
poetry show --tree
```

Virtual Environments

You may have noticed that the acts of adding and removing dependencies were done without mention of a virtual environment. However, take a look at the first line of terminal output when adding a dependency for the very first time. **Note**, the exact name of the virtual environment may be different than the one below.

```
Creating virtualenv learning-poetry-LfPaow_U-py3.6 in
/home/codio/.cache/pypoetry/virtualenvs
```

Poetry automatically creates and activates a virtual environment for you. During subsequent package installations, you will no longer see the message about creating a virtual environment. It still happens, but Poetry does not write a message to the terminal.

Use the following command to list all of the virtual environments associated with `learning-poetry`.

```
poetry env list
```

Just the simple act of changing into the `learning-poetry` directory activates the associated virtual environment. You do not have to worry about creating, activating, or deactivating a virtual environment with Poetry. However, you do need to be in the right directory.

```
learning-poetry-LfPaow_U-py3.6 (Activated)
```

The toml File

Metadata

Start by changing into the `learning-poetry` directory.

```
cd learning-poetry
```

When creating a new package, Poetry generates the `pyproject.toml` file. `toml` is a minimalist file format for configuration files. Information is stored as key-value pairs. The goal is to be easy to understand, even if you have never seen a `toml` file before. Click the link below to open the `pyproject.toml` file.

▼ **What does `toml` stand for?**
`toml` is an acronym for "Tom's Obvious Minimal Language".

The file is divided into four sections denoted by the square brackets. The `[tool.poetry]` section contains the metadata for the project. The name, version, description, and author are all required. Some of this information is filled in automatically for you, while other information needs to be added manually. If you are looking to publish your package to PyPI, you want to fill out as much metadata as you can (see the Poetry documentation). Poetry will automatically take information from `pyproject.toml` and put it into the website for you package on PyPI.

Managing Dependencies

The `pyproject.toml` is central to Poetry's ability to resolve, add, and remove dependencies. The `[tool.poetry.dependencies]` section is the canonical list of dependencies. If your project needs a package, it will be listed here. Click on the link below to open the terminal and enter the command to change to the `learning-poetry` directory and install the `idna` package.

```
poetry add idna
```

Click the link below to open the `pyproject.toml` file once more. You should now see `idna` listed as a dependency.

▼ **Did you notice?**

When a package is installed from PyPI, a version number is stored. Remember, the ^ means that updates cannot change the left-most, non-zero digit in a version. So Poetry can update `requests>` as long as the version is less than 3. When a package is installed from GitHub, Poetry replaces the version number with the SSH address for the repository.

```
[tool.poetry.dependencies]
python = "^3.6"
shapes = {git = "git@github.com:codio-content/shapes-
package.git"}
idna = "^3.3"
```

Let's assume you download a project created in Poetry. You would get all the files and directories in the project. What you would not get are the dependencies. If you run the `install` command, Poetry will look at the `toml` file and install every package listed. This functions very much like a `requirements.txt` file, but you do not need to generate one first; `pyproject.toml` is always kept up-to-date as you add and remove dependencies.

```
poetry install
```

▼ **Did you notice?**
If you run the `install` command Poetry should tell you that there are no dependencies or updates to install. That is because you have already installed all of the dependencies.

The `pyproject.toml` file also has a section for development dependencies. Every Poetry project comes with `pytest` for development purposes. However, the `add` command does not install a package in the development section. The `Faker` package is a fake data generator used by developers. Using the `--dev` flag tells Poetry to install a package in the development section. Let's install `Faker`.

```
poetry add Faker --dev
```

The `pyproject.toml` file should now show `Faker` listed as a development dependency.

If you have a Poetry project that you would like to use on your own machine, you need the dependencies. However, you only plan on using the package and not developing it. So you do not really need the development

dependencies. The command below will install all of the dependencies in the `[tool.poetry.dependencies]` but ignore the dependencies in the `[tool.poetry.dev-dependencies]` section.

```
poetry install --no-dev
```

In a way, Poetry combines the `setup.py` (metadata) file and `requirements.txt` (dependencies) into the `pyproject.toml` file. This is another example of how Poetry has the same capabilities as `pip` but does so in a much more efficient way.

The Lock File

Dependencies of Dependencies

Start by changing into the `learning-poetry` directory.

```
cd learning-poetry
```

If you look at the dependencies in the `pyproject.toml` file, you would think this Poetry project has only four dependencies. However, the `fire` package has dependencies of its own, but you cannot see them.

```
[tool.poetry.dependencies]
python = "^3.6.2"
shapes = {git = "git@github.com:codio-content/shapes-
package.git"}
idna = "^3.3"
fire = "^0.4.0"
```

Now compare the above information when using the `show --tree` command to view a hierarchical structure of the installed packages.

```
poetry show --tree
```

You should see `six` and `termcolor` listed as dependencies of the `fire` package. Poetry stores the complete list of dependencies (and their dependencies) in the `poetry.lock` file. Click the link below to see what this looks like.

One difference between dependency versions listed in `pyproject.toml` and those in `poetry.lock` is the absence of the `^` character. The lock file specifies the exact version to be installed. In the example below, Poetry will install only version 1.4.0 of `atomicwrites`.

```
[[package]]
name = "atomicwrites"
version = "1.4.1"
description = "Atomic file writes."
category = "dev"
optional = false
python-versions = ">=2.7, !=3.0.*, !=3.1.*, !=3.2.*, !=3.3.*"
```

Why is there a difference? Imagine if `package_a` lists `atomicwrites = "^1.3"` as a dependency and `package_b` lists `atomicwrites = "^1.4"` as its dependency. Both packages need `atomicwrites`, but they have different requirements as to the version. Poetry resolves dependencies before installing them. That means Poetry will find a version of `atomicwrites` that satisfies both `package_a` and `package_b`. This version is then saved to `poetry.lock`.

The lock file is an important part of your Poetry project. If you are sharing your work with others, be sure to include `poetry.lock` so they can install the same package versions as you.

▼ **What happens if there is no lock file?**
If you are working with a Poetry project without a lock file, you can still run `poetry install`. However, Poetry will use the `pyproject.toml` file when installing dependencies. Poetry will install the latest version of each package. Poetry then creates a `poetry.lock` file. Installing the latest versions could cause problems, which underscores the need to always include the `poetry.lock` file with your project.

Poetry and a Requirements File

Assume that you use Poetry for all your Python projects. You want to share your latest one with a friend. However, they use `pip` for their package management. The `poetry.lock` file is useless to them since they expect a `requirements.txt` file to contain the dependencies. Thankfully, Poetry can produce a requirements file.

```
poetry export --output requirements.txt
```

Use the `cat` commands to print the text of the requirements file to the terminal. Even though Poetry has its own way of doing things, that does not mean your work is isolated from the larger Python community.

```
cat requirements.txt
```

Interacting with the larger Python community means that Poetry needs to be able to install from the `requirements.txt` file as well. The project `use-requirements` has already been created using Poetry. Change into this directory and display the hierarchical structure.

```
cd ../use-requirements && tree
```

You should see a `requirements.txt` file in the directory. We are going to use this to do the initial installation of dependencies. The command below sends the contents of the requirements file to the `add` command for Poetry.

```
poetry add `cat requirements.txt`
```

Now use Poetry's `show` command to list the dependences installed for the project.

```
poetry show --tree
```

The `poetry.lock` file stores the exhaustive list of dependencies in a Poetry project. Every Poetry project should have one of these files. In addition, Poetry can create or install from a `requirements.txt` file if need be.

Using Poetry with an Existing Project

Initializing a Poetry Project

Assume you started to create a project in Python. You had yet to hear of the benefits of Poetry, so you started to build the project manually. The `existing-project` directory contains a single script called `my_script.py`. Change into this directory and show its contents.

```
cd existing-project && tree
```

You do not need to start over to use Poetry with this project. We are going to initialize our Poetry project with the `init` command.

```
poetry init
```

This will start an interactive process where Poetry will ask you a series of questions about the project. This will form the basis of your `pyproject.toml` file. Anything residing between square brackets will be used as the default; just press enter. If there is nothing between the square brackets, then it is an optional question. Poetry gives you the chance to enter your dependencies (production and development).

```
This command will guide you through creating your pyproject.toml
config.

Package name [existing-project]:
Version [0.1.0]:
Description []:  "Sample project initializing Poetry"
Author [Your Name , n to skip]:
License []:
Compatible Python versions [^3.6]:
Would you like to define your main dependencies interactively?
(yes/no)
Would you like to define your development dependencies
interactively? (yes/no)
```

Before finishing the initialization process, Poetry will show you the `pyproject.toml` file. The example below did not use the interactive process to add dependencies. This will have to be done later with the `add` command as previously discussed.

```
[tool.poetry]
name = "existing-project"
version = "0.1.0"
description = ""
authors = ["Your Name <your_email@provider.com>"]

[tool.poetry.dependencies]
python = "^3.6"

[tool.poetry.dev-dependencies]

[build-system]
requires = ["poetry-core>=1.0.0"]
build-backend = "poetry.core.masonry.api"


Do you confirm generation? (yes/no)
```

Use the `tree` command to see the contents of the directory.

```
tree
```

Unlike the `new` command, Poetry does not make a full-featured project when using `init`. You do not have `__init__.py` file, a testing directory, etc. However, you do have a `pyproject.toml` file so you can now use Poetry. You can always manually create the missing directories and files if necessary.

```
.
├── my_script.py
├── poetry.lock
└── pyproject.toml
```

challenge

Try this variation:

- Create a `poetry.lock` file. Running the `update` command will generate a lock file from the dependencies listed in the `toml` file.

```
poetry update
```

Verify that the newly created file exists by printing the hierarchical structure of the directory.

```
poetry show --tree
```

Running Your Project

You can also run scripts with Poetry as well. Since we are in the `existing-project` directory, we can run the file `my_script`. The very first time you do this, you will see a message about Poetry starting a virtual environment. Every subsequent run will also be in a virtual environment. Run the script several times and it will randomly print one of six different greetings.

```
poetry run python my_script.py
```

▼ **Did you notice?**

In the previous module we ran scripts using `python3`. With Poetry we do not need to specify the version of Python we want to use. The only version of Python specified in the `pyproject.toml` file is version 3.6. You will not accidentally run the script with version 2 of Python.

```
[tool.poetry.dependencies]
python = "^3.6"
```

Wheels

Building a Wheel

It should come as no surprise that Poetry streamlines the process of building a wheel. For starters, there is no need for a `startup.py` file. Everything needed for your wheel is contained in the `pyproject.toml` file. Though it is a good idea to verify that the metadata is properly filled out. First chang into the `learning-poetry` directory.

```
cd learning-poetry
```

Use the `build` command to create a wheel.

```
poetry build
```

That's it, that's all you have to do. Poetry creates both a source distribution and a build distribution in the `dist` directory. List the contents of this directory; you should see a `tar.gz` file (source distribution) and a `whl` file (build distribution).

```
ls dist
```

▼ **Did you notice?**
Poetry only produces a pure-Python wheel. You do not have the ability to create a universal or platform-specific wheel.

challenge

Try these variations:

- Create only a build distribution. First we need to delete the `dist` directory and its contents. Then use the `build` command and the `--format` flag to specify only a wheel.

```
rm -r dist && poetry build --format wheel
```

List the contents of `dist` again and you should only see the wheel file.

```
ls dist
```

- Create only a source distribution. First we need to delete the `dist` directory and its contents. Then use the `build` command and the `--format` flag to specify only a source distribution.

```
rm -r dist && poetry build --format sdist
```

List the contents of `dist` again and you should only see the source distribution.

```
ls dist
```

Publishing a Package

Poetry also lets you upload your package to PyPI. This is done with the `publish` command. This is much easier than using `pip`, which requires some intermediate steps like downloading and running `twine` before uploading to PyPI.

PyPI has the ability to display lots of useful information about your package. All of this can be specified in the `pyproject.toml` file and automatically added to the PyPI page. You should strongly consider providing information like:

- Homepage
- Documentation
- License
- Maintainers

- README
- Repository
- Keywords
- ...and many more

See the Poetry documentation for more information. Once your package is ready for distribution, use the `publish` command to send it to PyPI.

```
poetry publish
```

Note, running the above command will not automatically publish your package to PyPI. Poetry will still ask your PyPI credentials (username and password). Make sure that you have a valid PyPI account before publishing.

Formative Assessment 1

Formative Assessment 2