Building Command-Line Tools

Why a command-line tool

The basis of becoming a better engineer starts with automation. Automate everything. Get into the habit of continuously automating everything that seems like it could be automated. A command line tool will give you the stepping stone for creating automation. It will allow you to avoid repetitive (and error-prone) tasks, while making you more prolific since a tool will do a task much faster.

Project Structure

Project structure (also known as scaffolding) can be overwhleming in the beginning. In Python, it does follow certain basics, but once these are known, it should be straightforward for you to create a project and make it work.

This is how a very simple Python tool called *jformat* structures its files and directories:

```
.
├── LICENSE
├── bin
│   └── executable
├── jformat
│   ├── __init__.py
│   └── main.py
├── requirements.txt
├── setup.py
└── tests
    └── test_main.py
```

In this repository, we'll follow the same structure and use *bin* to put an executable, the name of the project will be a directory with a *main.py* file with the code, and some packaging and tests files at the top.

The specifics of these files are not important right now, their purpose will become clear as you make progress with this lesson.

Helpers

Helper files are those that aren't part of the code project itself, but are there to *help out* with some tasks. These are usually not packaged as part of the project when publishing the tool.

Makefile

A *Makefile* is a file with instructions on how to build software. Python doesn't require *building*, although you'll have to do something similar to building when publishing the project later. However, a *Makefile* is useful because it can automate tasks for you in a single command. Sometimes, it can be a single step that you want to automate because the command is extremely long.

For example, this is the command to upload your package to a test repository in Python:

```
$ twine upload --repository-url https://test.pypi.org/legacy dist/*
```

With a *Makefile* you could alias it to `make upload-test`. Which one are you going to remember?

Further, you can add a *Makefile* to all your projects and have the same aliases, further simplifying and automating your workflows.

requirements.txt

This is a common file for Python projects that like to define the project dependencies (sometimes with versions) in a plain text file. In some cases, you might find more than one requirements file. For example, in development environments newer versions might be used to test against those instead of current production values.

The *requirements.txt* file can be used by `pip` (the Python installer tool) to install the packages listed. The tool can figure out the package name and the version or version ranges defined in the same line.

Packaging

Packaging in Python is far from ideal. Because there are a lot of unknowns you might try to avoid it, but there is a lot of useful outcomes from doing pacakging. After packaging a Python project, it can be published to PyPI (the Python Package Index), a package repository that is publicly available. Anyone can then declare your project as a dependency and install it once it is available on PyPI.

Files

There are a few files that might show up when packaging, and in this case, that is *setup.py*. This file, by convention, is the one used to declare how your project should be packaged. Things like package description, name, author, and version can be found in *setup.py*.

Additionally, you may find a *MANIFEST.in* file. This file is used configure additional files and directories to be packaged that are not picked up by *setup.py*. The configuration can include or exclude paths, files, or patterns.

Publishing

Once your project is ready, it can be published to an index. The public index for Python is PyPI. In some production environments, companies use private package indexes where packages are hosted for internal use only.

For publishing, you will need the packaging files, and the twine package and your credentials for PyPI. Make sure you create an account.

After registering, go to manage your account and search for the *API tokens* section. Select the *"Add API token"* button, complete the name and select "Entire account" for the scope. Since you haven't uploaded the project, you will need that scope even though the warning tells you to proceed with caution.

After creating the token, copy it and go back to your Github project. You need to create a secret with the token. Go to the Settings menu of your GitHub repository and look for the Secrets item. Select Actions and then New repositoy secret. The secret name must be `PYPI_API_TOKEN`. Paste the token in the value section and select the green button to add the secret.

CI/CD with Github Actions

Finally, you should have a fully functional command line tool that you will probably want to install in other places and share it with others. Knowing how to publish and package your tool is already useful but you can take things a step further by fully automating the publishing.

A CI/CD (Continuous Integration and Continuous Deployment) setup helps you avoid mistakes, conditionalizes your release after quality control rules are passing (like linting and tests), and it reasures you that a release meets all required verification steps.

When there is no automation involved then it is easy to forget about a step, or get into errors from doing things out of order.

There are many different CI/CD systems that allow you to automate linting, testing, and releasing, but for this project, you'll take advantage of Github Actions. All you need to do is to create a hidden directory (`.github/`) at the top of your project with a sub-directory called `workflows` and a single YAML file. The YAML file holds the instructions for your project to test, verify, and create the release.

Automatic release

Releases on Github are tied to tags. Tagging is a Git operation that marks a point in time of a repository. In this case, we have a version that exists in a package (for example `0.0.1`) and a tag that needs to happen to cut a release. Ideally, you would want to have both happen in a correlated way.

When you don't correlate both, you can end up with a tag for `0.0.2` but a version of `0.0.1`. That would cause confusion for anyone trying to install a specific version.

Create a tag with git:

$ git tag '0.0.1' $ git push --tags

In Github, a tag is not automatically associated with a release. So after tagging, you must go through the web interface and create a release. Do that by going to your project, then to the tag menu just above your code. Next, use *Releases* to create a new release based on the tag you've previously pushed.