00001010

CHAPTER 10

Virtual Environments With Pipenv

Learning Objectives

- State the definition of a virtual environment
- State the purpose of a virtual environment
- State the purpose of Pipenv
- Explain which two tools Pipenv replaces
- Explain the utility of Pipenv virtual environments
- Explain how Pipenv avoids package conflicts between different Python projects
- Configure and verify the PIPENV_VENV_IN_PROJECT environment variable
- List the steps required to create a virtual environment with Pipenv
- Search for packages on the Python Package Index (PyPI)
- Install and uninstall packages with Pipenv
- State the purpose of the Pipfile file
- State the purpose of the Pipfile.lock file
- State the purpose of the .venv directory
- List and describe the contents of the .venv directory
- State why it's good practice to locate the .venv directory in the project directory
- Properly configure your .gitignore file to ignore the Pipenv .venv directory

INTRODUCTION

So far in this book, you've learned how to code and run simple Python applications that use built-in functions and standard library modules. These features and capabilities come packaged with Python and require no additional action on your part other than installing Python itself. Take for example the standard library *unittest* module you encountered briefly in the previous chapter. The only action necessary to use that module was to import it into a unit test. Generally speaking, that's how you use standard library modules. You just need to import them. They come already installed with the Python distribution. But now it's time to up your game and learn a whole new bag of professional software developer tricks.

In reality, real-world Python applications rely on the services of third-party packages installed from a package repository like the *Python Package Index (PyPI)*. When you install a Python package, you can install it either *globally,* or *locally* in a project's *virtual environment*. Installing a Python package globally makes that package available to all projects using that particular version of Python. Installing a package globally can lead to package conflicts, especially if one project depends on package_version_*old* and another project depends on package_version_*latest*. The *latest* package version will replace the *old* version with perhaps unintended or unforeseen consequences. Conflicts can occur when `program_a` depends on some feature of package_version_*old*, which was subsequently changed or deprecated in package_version_*latest*. These types of *application programmer interface* (API) changes will break a program if the impacts of switching to a newer package version are not well documented and understood. And don't fool yourself! This scenario plays out quite often in the Open Source community. So, how do you prevent such unpleasant surprises? By adopting and using virtual environments.

In this chapter, you will learn about virtual environments, what they are, and how they are used to isolate package dependencies between python projects to prevent package conflicts. I'll show you how to create virtual environments with *Pipenv*, a tool that combines the separate tools `pip/pip3` and `virtualenv`, and greatly simplifies virtual environment creation and project package management. Along the way, I'll show you how to search for packages on PyPI and how to install those packages with Pipenv for use in application development, production, or both.

Using Pipenv will introduce you to new ways of working. You'll learn about the *.venv* directory, which stores information about the project's virtual environment, and the purpose and use of two new files: *Pipfile* and *Pipfile.lock*. I'll also explain which of these files you need to push to your repository and which ones you need to ignore.

This is one chapter you really want to takes notes on and write down the commands and configuration details in your Engineer's Notebook.

1 THE PROBLEM IN A NUTSHELL

Let's start by considering an execution environment with which you should be intimately familiar — your computer. Let's also say for the sake of illustration that you installed Python 3.9 and started a project named Project 1. You manually configured the PATH environment variable to add the Python 3.9 installation path. Then, for serious number crunching, you installed *Pandas*. Figure 10-1 gives the current state of affairs.
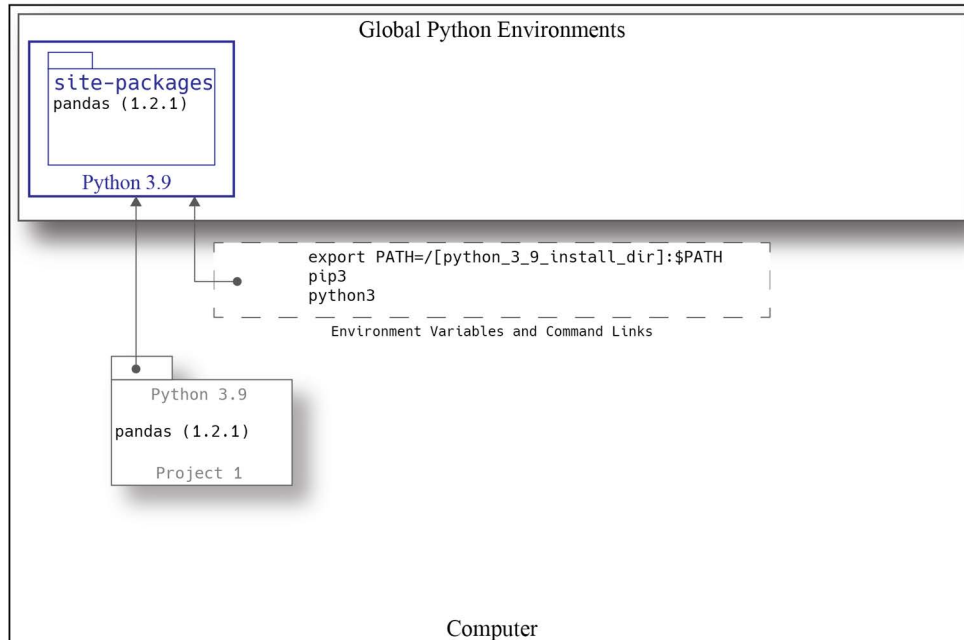
Figure 10-1: Python 3.9 Installed with PATH Environment Variable Set and Pandas Package Installed

Referring to figure 10-1 — Every installed version of Python has a *site-packages* folder in which it stores packages installed via the `pip` or `pip3` command. These packages are then available to every project on your computer that uses that particular version of Python. As shown above, Python 3.9 has its site-packages folder. Installing Pandas with `pip3` results in the Pandas package being installed in the Python 3.9 site-packages folder. Modules in Project 1 can then import and use Pandas as required. Next comes Project 2 and the need for the *NumPy* package, so you install it. And why not? You gotta get stuff done. Figure 10-2 shows the new state of affairs.
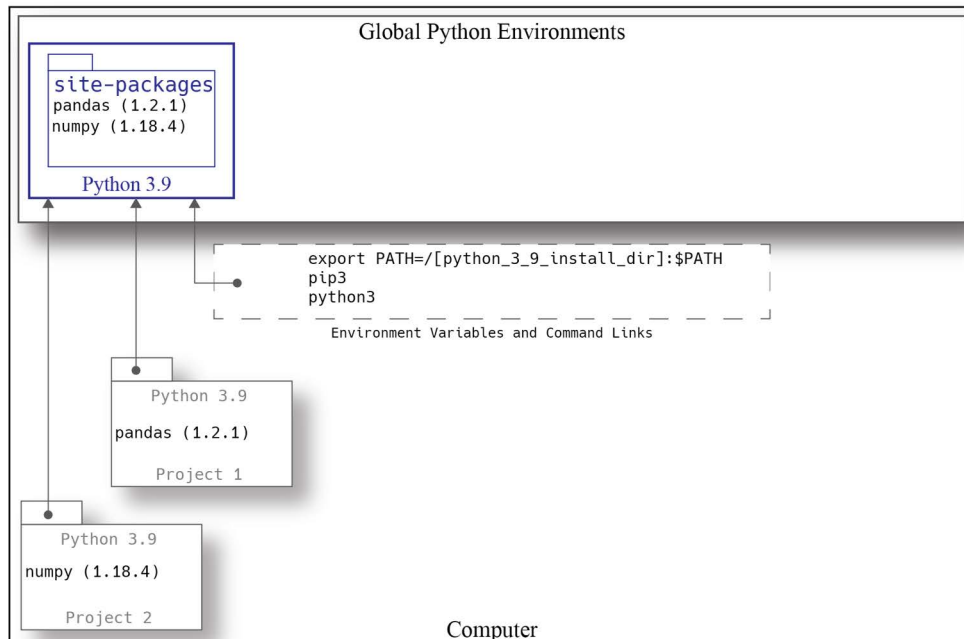
Figure 10-2: Project 2 with NumPy Package Installed

Referring to figure 10-2 — Installing the NumPy package with the `pip3` command installs it in the global Python 3.9 site-packages directory. Things are working just fine, but time marches on and Python 3.10 comes out with some nice, shiny new features you just gotta try out, so you install it, but you forget to change your PATH environment variable. Oblivious to the carnage you are about to wreak, you create a new project and install a more recent version of NumPy using `pip3`. Figure 10-3 gives the shocking state of affairs at this point.
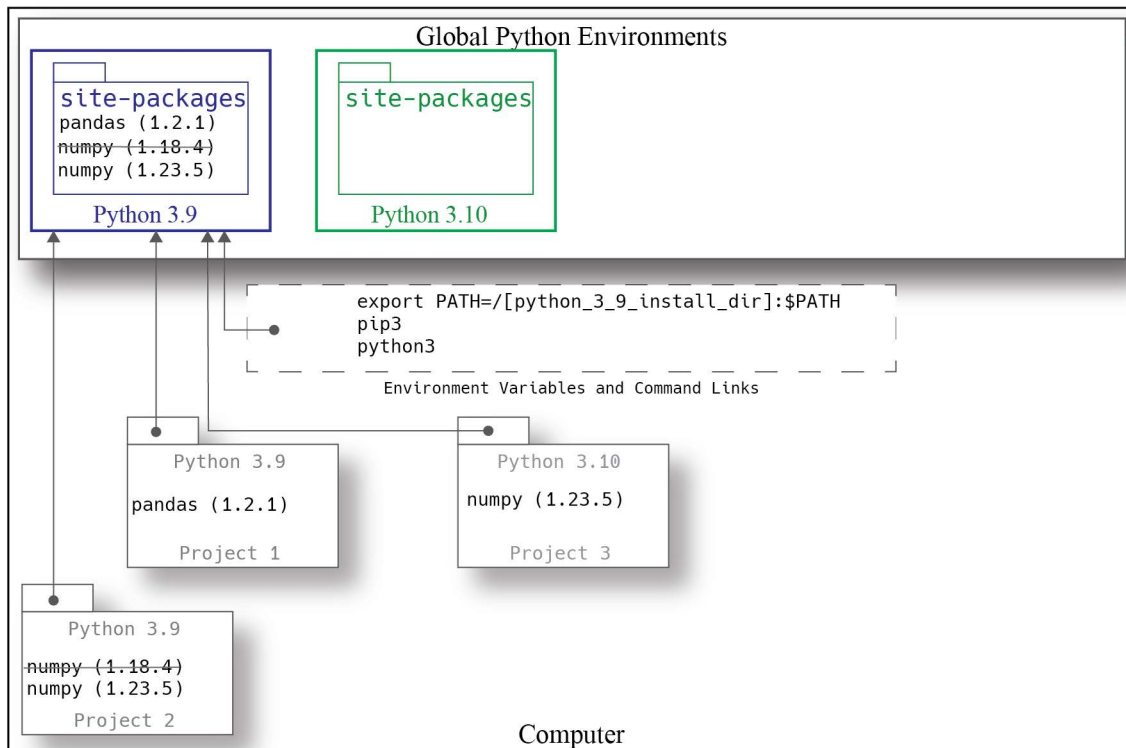
Figure 10-3: Python 3.10 Installed but the PATH Environment Variable Still Points to Python 3.9

Referring to figure 10-3 — Although you installed Python 3.10, Python 3.9 is still the active version because the PATH environment variable says so. If you use a language feature unique to Python 3.10, like the `match` statement, you will receive an error when you attempt to run the code with the Python 3.9 interpreter. Python 3.9 has no idea what a `match` statement is! (**Note:** *Don't think for a minute this is a contrived scenario. A former student of mine had a misconfigured environment with several versions of Python installed in various locations. When she tried to run a Python 3.10 code example, it failed. She could not understand, nor figure out, why things weren't working as expected. This is how I learned she was not doing her homework! Otherwise, she would have encountered the configuration error before the midterm exam.*)

Still referring to figure 10-3 — Because Python 3.9 is the active environment, `pip3` installs the new version of NumPy into the Python 3.9 site-packages folder, clobbering the old version. Project 2 has just been subjected to a surprise package update. The Project 2 application may or may not work as expected, but things are getting dicey.

Eventually, you will attempt to run the Project 3 application, which uses a feature specific to Python 3.10 and it will fail. You then realize you need to update the PATH environment variable to include the path to Python 3.10. Figure 10-4 gives the disappointing state of affairs at this point.
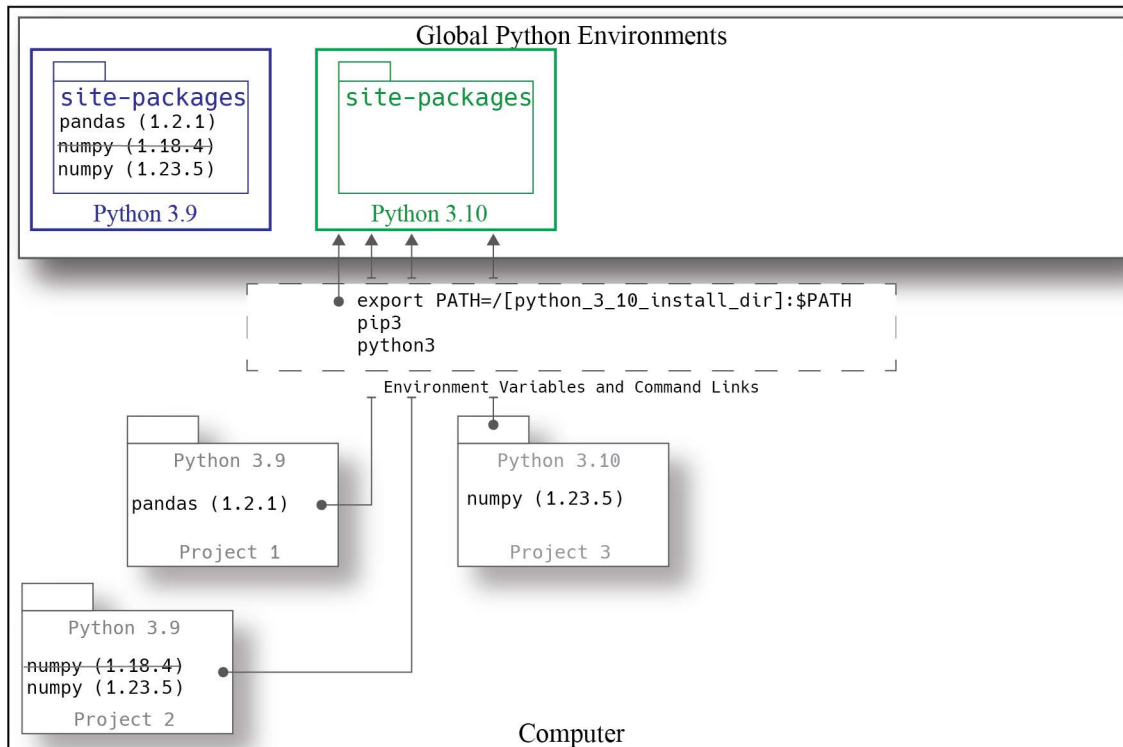
Figure 10-4: Environment Variable Set to Python 3.10

Referring to figure 10-4 — The PATH environment variable has been edited to set Python 3.10 as the active environment. Now you have an interesting problem. When you try to run the applications, they will throw errors stating the pandas and numpy packages cannot be found, or words to that effect. That's because Python 3.10 will look for those packages in *its* site-packages folder where it will come up empty-handed. You could always reinstall the packages with `pip3` now that the PATH environment variable is set to Python 3.10, but if there was a hard dependency in Project 2 on NumPy (1.18.4) and Project 3 needed NumPy (1.23.5), then you're out of luck. Now imagine as time marches relentlessly onwards that you install new versions of Python as they become available. It's easy to see how things can get out of hand simply trying to manage multiple versions of Python and global package dependencies. Thankfully, there is a better way!

1.1 THE UTILITY OF VIRTUAL ENVIRONMENTS

`<deep_announcer_voice>` Imagine a world where there is a tool that allows you to create isolated environments specific to each project, a tool that lets you specify which version of Python a project needs, and a tool that installs and manges package dependencies at the project level. Figure 10-5 shows what life would be like with such a tool. `</deep_announcer_voice>`

Referring to figure 10-5 — There are now four versions of Python installed on the computer. The PATH environment variable is still set to Python 3.10, which means Python 3.10 is the active version of Python for projects not using a virtual environment. The projects, however, are isolated via virtual environments. When setting up a project, a tool is used to create a virtual environment configured specifically for that project, including what version of Python to use, and what packages to install. If a project is set to use Python 3.9, then its virtual environment is configured to point to the global Python 3.9 for its `python3` and `pip3` commands, and to a project-local site-
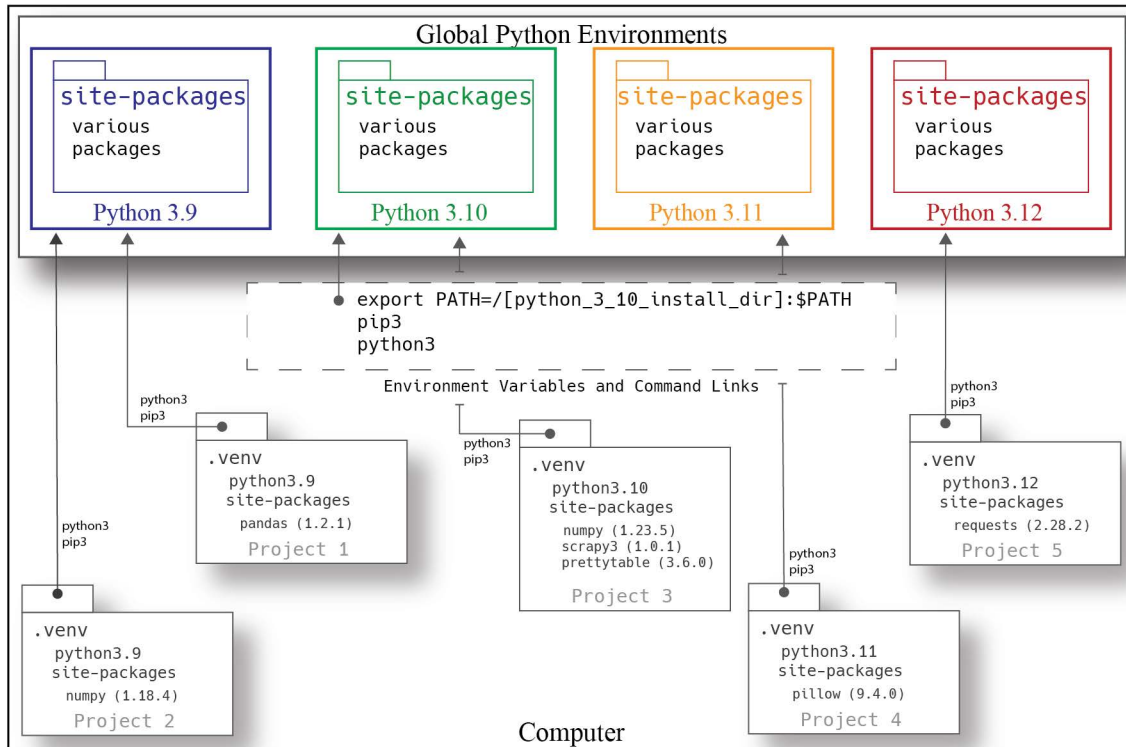
Figure 10-5: Each Project Has a Virtual Environment that Indicates Python Version and Isolates Dependencies

packages folder where packages specific to the project reside. Note that Projects 1 and 2 both use Python 3.9, yet each have their own, local site-packages folder. Project 3 is using a newer version of NumPy plus *Scrapy3* and *PrettyTable*. Project 4 has installed the *Pillow* package, while Project 5 is using *Requests*.

You solve a handful of tricky problems when you use virtual environments to manage project package dependencies. You eliminate inter-project package conflicts, you lighten the application deployment load because don't need all the packages that may be installed in a global Python site-packages folder, and you can run projects that target different versions of Python on the same physical computer. What tool helps you do all this? *Pipenv*, and I'll show you how to use it in the next section.

> **Pro Tip:** Use virtual environments to specify Python versions, isolate package dependencies, and prevent package conflicts.

QUICK REVIEW

Python packages are normally installed in the active Python distribution's site-packages folder. This leads to problems, especially if you have multiple projects using different versions of the same package, or multiple projects using different versions of Python. To tame this sort of unruly complexity, use virtual environments to specify Python versions, isolate project dependencies, and prevent package conflicts.

2 PIPENV

Pipenv is *the* tool with which you can create virtual Python environments like the ones shown in figure 10-5 to isolate package installation to the local project and eliminate inter-project package conflicts. Learning how to use Pipenv will change your Python development life. Pipenv consolidates the two tools *pip/pip3* and *Virtualenv*. Pipenv makes virtual environment creation and management effortless.
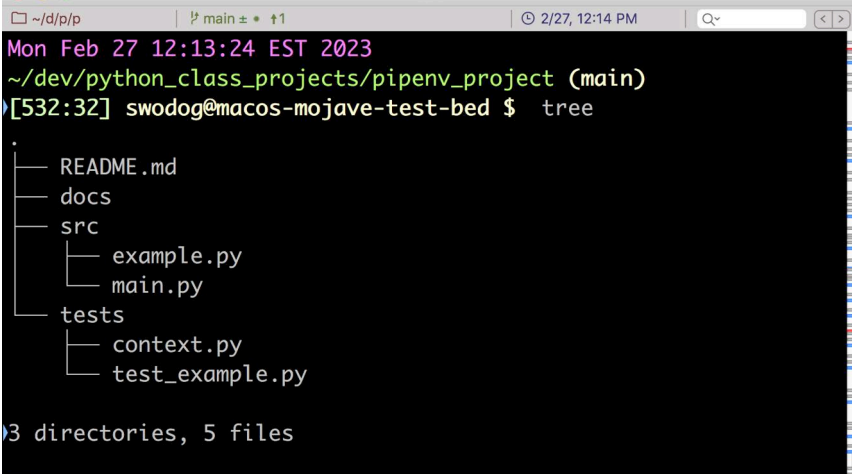
Before proceeding with this section, you should already have installed Pipenv and verified it's working properly. If not, see *"Install Pipenv" on page 242*. Make sure you set the `PIPENV_VEN-V_IN_PROJECT` environment variable as discussed here: *"Set PIPENV_VENV_IN_PROJECT Environment Variable" on page 245*.

PRO TIP: Configure the `PIPENV_VENV_IN_PROJECT` environment variable so that Pipenv creates the .venv folder in a project's root directory

2.1 CREATE A VIRTUAL ENVIRONMENT WITH PIPENV

A little heads up before you start creating virtual environments with Pipenv. You can create virtual environments only for those versions of Python you have installed on your computer. For example, if you only have Python 3.10 installed, you can only create a virtual environment for Python 3.10. If you wanted to create virtual environments for Python 3.10 and 3.11, you'd need to install both versions. Just so you know.

Navigate to your *~/dev* directory and create a new project directory. I'm going to create a directory named *pipenv_project* in the *python_class_projects* repository folder. The full path to my new project folder is: *~/dev/python_class_projects/pipenv_project*. Initialize your project with the baseline project structure discussed in chapter 9. The `tree` view of my project directory looks like figure 10-6.
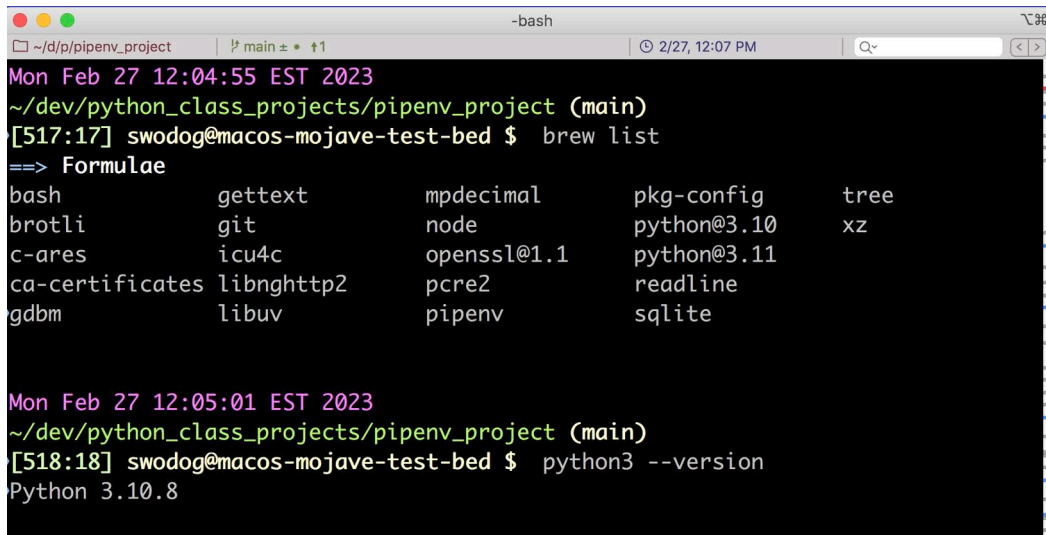

Figure 10-6: Project Tree View Showing Baseline Project Structure

Before I create a virtual environment, I am going to list which Python versions are installed on my computer. I'm using macOS along with the brew package manager, so to list the installed packages I'll list them with the following command:

```
brew list
```

Figure 10-7 shows the results of running this command on my machine.

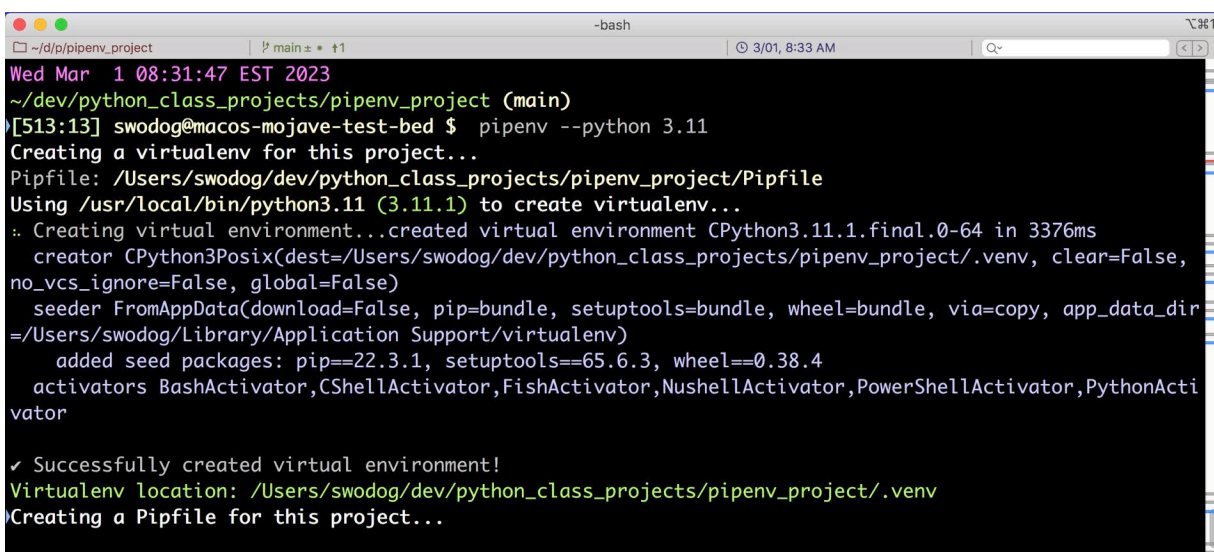Figure 10-7: Listing Installed Packages with `brew list` Command

Referring to figure 10-7 — The listing shows I have Python 3.10 and 3.11 installed on my machine. I then check the active version, which shows Python 3.10.8. This means that when I run a program with the `python3` command without a virtual environment, it executes with the Python 3.10.8 interpreter.

Before proceeding I want to be clear, you *don't need* multiple versions of Python to benefit from using Pipenv. Package isolation and dependency management are sufficient reasons alone to use virtual environments.

OK, to create a Python 3.11 virtual environment I'll use the following command.

```
pipenv --python 3.11
```

Figure 10-8 shows the results of running this command.

Figure 10-8: Creating Python 3.11 Virtual Environment with Pipenv

Referring to figure 10-8 — The output of the `pipenv` command tells you which Python distribution is used to create the virtual environment and in what directory the *.venv* directory is being created. Because I have set the `PIPENV_VENV_IN_PROJECT` environment variable to `true`, the project's virtual environment directory (*.venv*) is created in the project's root directory along with a Pipfile. Figure 10-9 shows the project directory `tree` view after creating the virtual environment.

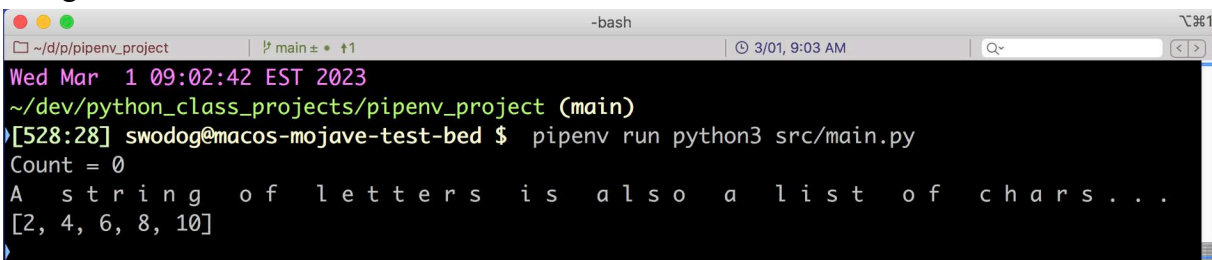Figure 10-9: Project Directory tree View After Creating Virtual Environment

Referring to figure 10-9 — Creating a virtual environment with Pipenv results in two new items being added to the project: a *.venv* directory, which contains the virtual environment configuration information, and a Pipfile, which indicates the Python version the virtual environment uses and which packages are installed.

2.2 RUNNING AN APPLICATION WITH PIPENV

To run an application in the virtual environment use the `pipenv run` command. For example, to run the *src/main.py* file in the virtual environment, use the following command:

```
pipenv run python3 src/main.py
```

Figure 10-10 shows the results.

Figure 10-10: Running `src/main.py` with `pipenv run` Command

To see which version of Python the virtual environment is using run `python3` with `pipenv run` and check its version the usual way:

```
pipenv run python3 --version
```

Figure 10-11 shows the results of checking the virtual environment's `python3` version followed by the global environment's `python3` version.

Figure 10-11: Checking Python Versions in Virtual and Global Environments

Referring to figure 10-11 — The virtual environment is configured to use Python 3.11.1 while the global environment is set to use Python 3.10.8. At this point, to run a Python application in the virtual environment, you need to remember to run `python3` with the `pipenv run` command.

Pro Tip: Remember — To run `python3` in the virtual environment use the `pipenv run` command.

Just FYI: An alternative to using `pipenv run` is to use `pipenv shell` to launch a subshell within the project's virtual environment like so:

```
pipenv shell
```

Figure 10-12 shows the results of running this command.

Figure 10-12: Launch Virtual Environment Subshell with `pipenv shell` Command

Referring to figure 10-12 — After launching the virtual environment subshell, you can run the `python3` command as you would normally do without the need to use `python run`. Type `exit` to exit the subshell.

Having showed you both the `pipenv shell` and `pipenv run` commands, I recommend using `python run`. Here's why — In the next chapter, I'll introduce you to bash scripting and show you how to configure and run Python applications using a bash `build.sh` script. In the script, you'll be using `pipenv run` to execute a Python application.

Let's now explore the Pipfile.

2.3 PURPOSE OF THE PIPFILE

The purpose of the Pipfile is to store project package dependencies and indicate which version of Python the project requires. **The Pipfile is the only file related to the virtual environment that gets checked in and committed to the repository.** Example 10.1 lists the contents of a Pipfile after initially creating a Python 3.11 virtual environment.

10.1 Pipfile (Initial Contents)

```
1    [[source]]
2    url = "https://pypi.org/simple"
3    verify_ssl = true
4    name = "pypi"
5
6    [packages]
7
8    [dev-packages]
9
10   [requires]
11   python_version = "3.11"
12   python_full_version = "3.11.1"
13
```

Referring to example 10.1 — Square brackets `"[ ]"` indicate sections within the Pipfile. The `[[source]]` section sets the package source to the Python Package Index (PyPI). The `[packages]` section lists packages required for production. The `[dev-packages]` section lists packages required for development, but are not needed to be deployed with the application in production. Finally, the `[requires]` section lists Python versions required for the project. As you can see, a newly created virtual environment has no production or development packages initially installed. I'll talk more about installing and removing packages later in the chapter.

2.4 PURPOSE OF .VENV DIRECTORY

The purpose of the *.venv* directory is to store virtual environment configuration information. Take some time to poke around in the *.venv* directory and see what's in there. Figure 10-13 gives a tree view of the *.venv* directory.

Referring to figure 10-13 — Notice in the *.venv/bin* directory the virtual environment's python command is a link to the */usr/local/opt/python@3.11/bin/python3.11* command. (***Note:*** *This is on a macOS machine with brew-installed Python versions.*) The `python3` command links to the `python` command, and the python3.11 command links to the `python3` command. This means you can run python3 in the virtual environment using any of these commands:

```
pipenv run python -or- pipenv run python3 -or- pipenv run python3.11
```

Figure 10-13: `tree` View of *.venv* Directory

Continuing on, `cat` the following files and see what they contain: *.venv/.project* and *.venv/pyvenv.cfg*. Figure 10-14 shows the results.

Figure 10-14: Contents of *.venv/.project* and *.venv/pyvenv.cfg* Files

Referring to figure 10-14 — There's generally no need to edit this information. In the *pyvenv.cfg* file, I recommend always keeping `include-system-site-packages` set to `false` and manage project packages exclusively with Pipenv.

2.5 Ignore .venv Directory

The *.venv* directory must never get pushed to the repository. If it's not already listed in your project's *.gitignore* file you need to add it.

Quick Review

Pipenv is a tool that lets you create Python virtual environments. Before using Pipenv to create a virtual environment, ensure you have set the environment variable `PIPENV_VENV_IN_PROJECT` to `true` [`PIPENV_VENV_IN_PROJECT=true`]. This ensures the *.venv* directory is created in the project's root directory.

To create a virtual environment, you must have a corresponding version of Python installed on your system. For example, to create a Python 3.10 virtual environment, you must have Python 3.10 installed on your computer. You don't need multiple versions of Python installed in your computer to use Pipenv, unless you are working on projects targeting different Python versions. Using Pipenv for package management and isolation is sufficient reason enough to use Pipenv.

3 Package Management with Pipenv

Pipenv lets you effortlessly manage Python packages required by your project. In this section, I'll show you how to install and uninstall packages, and explain the purpose of the Pipfile.lock.

3.1 Installing Packages

Pipenv makes it easy to install, update, and uninstall Python packages your project needs for development and production. Let's start with how to install packages you might use only for Python application development.

3.1.1 Installing Packages for Development Only

Packages you would install for development purposes only are ones not required to support the application when it's deployed into a production environment. Table 10-1 provides a short list of common packages used only during the software development phase and their purpose.

Package	Purpose
pytest	Simple, easy to use Python testing framework. I personally prefer pytest over Python's standard library unittest module. *pytest project homepage.*
pydocstyle	Checks compliance with Python docstring conventions. *pydocstyle project homepage.*

Table 10-1: Common Packages Used For Python Development

Package	**Purpose**
Sphinx	Document generator. *Sphinx project homepage.*
autopep8	Automatically formats Python code to conform to the PEP 8 style guide. *autopep8 project homepage.*
pre-commit	Configure GitHub pre-commit hooks. *pre-commit project homepage.*

Table 10-1: Common Packages Used For Python Development (Continued)

To install a package for development use only use following command:
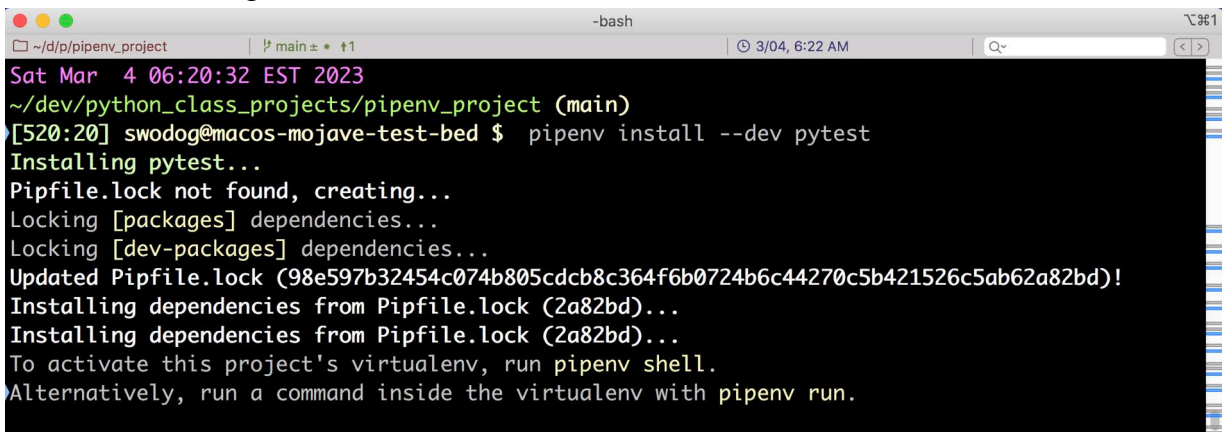
```
pipenv install --dev package-name
```

Specifying just the package name will install the latest version of the package. To install a specific package version use the fully-qualified package version like so:

```
pipenv install --dev package-name==n.n.n
```

So for example, if I install the pytest package without the version qualifier this is the command I'd use:

```
pipenv install --dev pytest
```

This will install the latest version of pytest, which as I write this is 7.2.2. Figure 10-15 shows the results of running this command.

Figure 10-15: Installing Latest Version of the pytest Package

Referring to figure 10-15 — Since this is the first time I've installed a package in this project, Pipenv creates a Pipfile.lock, which stores exact package version metadata. We'll take a look at the Pipfile.lock in a second. First, example 10.2 lists the contents of the Pipfile at this point.

10.2 Pipfile
(After installing pytest for dev)

```
1    [[source]]
2    url = "https://pypi.org/simple"
3    verify_ssl = true
4    name = "pypi"
5
6    [packages]
7
8    [dev-packages]
9    pytest = "*"
10
```

```
11    [requires]
12    python_version = "3.11"
13    python_full_version = "3.11.1"
14
```

Referring to example 10.2 — Notice in the [dev-packages] section the line pytest = "*", which indicates the latest version of pytest. Let's now take a look at the Pipfile.lock.

3.1.2 Purpose of Pipfile.lock

When you install a package with Pipenv, it records package information in the Pipfile under either the [packages] or [dev-packages] sections. It also creates a Pipfile.lock that indicates the exact package and version that was installed along with all dependent packages. Example 10.3 shows the Pipfile.lock after installing the latest version of *pytest*.

10.3 Pipfile.lock
(After installing pytest)

```
1     {
2         "_meta": {
3             "hash": {
4                 "sha256":
"98e597b32454c074b805cdcb8c364f6b0724b6c44270c5b421526c5ab62a82bd"
5             },
6             "pipfile-spec": 6,
7             "requires": {
8                 "python_full_version": "3.11.1",
9                 "python_version": "3.11"
10            },
11            "sources": [
12                {
13                    "name": "pypi",
14                    "url": "https://pypi.org/simple",
15                    "verify_ssl": true
16                }
17            ]
18        },
19        "default": {},
20        "develop": {
21            "attrs": {
22                "hashes": [
23
"sha256:29e95c7f6778868dbd49170f98f8818f78f3dc5e0e37c0b1f474e3561b240836",
24
"sha256:c9227bfc2f01993c03f68db37d1d15c9690188323c067c641f1a35ca58185f99"
25                ],
26                "markers": "python_version >= '3.6'",
27                "version": "==22.2.0"
28            },
29            "iniconfig": {
30                "hashes": [
31
"sha256:2d91e135bf72d31a410b17c16da610a82cb55f6b0477d1a902134b24a455b8b3",
32
"sha256:b6a85871a79d2e3b22d2d1b94ac2824226a63c6b741c88f7ae975f18b6778374"
33                ],
34                "markers": "python_version >= '3.7'",
35                "version": "==2.0.0"
36            },
37            "packaging": {
```

```
38                    "hashes": [
39
"sha256:714ac14496c3e68c99c29b00845f7a2b85f3bb6f1078fd9f72fd20f0570002b2",
40
"sha256:b6ad297f8907de0fa2fe1ccbd26fdaf387f5f47c7275fedf8cce89f99446cf97"
41                    ],
42                    "markers": "python_version >= '3.7'",
43                    "version": "==23.0"
44               },
45          "pluggy": {
46                    "hashes": [
47
"sha256:4224373bacce55f955a878bf9cfa763c1e360858e330072059e10bad68531159",
48
"sha256:74134bbf457f031a36d68416e1509f34bd5ccc019f0bcc952c7b909d06b37bd3"
49                    ],
50                    "markers": "python_version >= '3.6'",
51                    "version": "==1.0.0"
52               },
53          "pytest": {
54                    "hashes": [
55
"sha256:130328f552dcfac0b1cec75c12e3f005619dc5f874f0a06e8ff7263f0ee6225e",
56
"sha256:c99ab0c73aceb050f68929bc93af19ab6db0558791c6a0715723abe9d0ade9d4"
57                    ],
58                    "index": "pypi",
59                    "version": "==7.2.2"
60               }
61          }
62     }
63
```

Referring to example 10.3 — The Pipfile.lock is a JSON file and rather challenging to decipher, but in general, it starts off by specifying which version of Python the project is using, followed by the package source, which is PyPI, and then two sections starting on line 19, `default`, which is empty, meaning no packages are installed for production, and on line 20, `develop`, which lists quite a few. Installing *pytest* requires the installation of the following additional packages (a.k.a., package dependencies): *attrs*, *iniconfig*, *packaging*, and *pluggy*. Finally, on line 53, the *pytest* package is listed along with its version shown on line 59. You should see these packages listed in the project's virtual environment site-packages directory as shown in Figure 10-16.

Referring to figure 10-16 — You can correlate the packages listed in the site-packages directory with those listed in Pipfile.lock. You can also see a handful of packages not listed in Pipfile.lock. Those are a set of baseline packages installed when first creating the virtual environment.

3.1.3 Deterministic Builds

The most important aspect of the Pipfile.lock is that it ensures *deterministic builds*. If you're new to software development you may think everything will just work. Nothing could be further from reality. It can come as quite a shock when you upgrade a package only to have your once perfectly-working code mysteriously break. If you have good unit tests you should be able to detect these sorts of problems right away, but all too often, developers provide spotty unit test

Figure 10-16: Project's Virtual Environment's Site-Packages Folder after Installing pytest Package

coverage if they test at all, and have no idea their code is broken until they deploy it into production and the system goes down.

The Pipfile.lock lists exact package versions for the purpose of replicating exactly the environment one software developer had at the time they ran their code and that of another software developer when they attempt to run the code. If you ever eavesdrop on a conversation between software developers and one says to another, "But it works on my machine!", then you know they are struggling with differences between their development environments. This is the problem Pipenv, and virtual environments in general, solves. It enables two different developers to run code in two different locations on two different machines in the same, *deterministic* execution environment, and achieve the same results.

3.1.4 Installing Packages for Production

Let's now install a package for production. A package installed for production is included with the application when it is deployed or packaged for distribution. Let's install the *numpy* package but specify an exact version.

```
pipenv install numpy==1.24.2
```

Example 10.4 lists the contents of the Pipfile after installation completes.

10.4 Pipfile
(After installing numpy v1.24.2)

```
1    [[source]]
2    url = "https://pypi.org/simple"
3    verify_ssl = true
4    name = "pypi"
5
6    [packages]
7    numpy = "==1.24.2"
8
9    [dev-packages]
10   pytest = "*"
11
12   [requires]
13   python_version = "3.11"
14   python_full_version = "3.11.1"
15
```

Referring to example 10.4 — Note that the *numpy* package is installed under the `[packages]` section and has an exact version assigned. I'll leave it up to you to examine Pipfile.lock and the virtual environment's site-packages directory and note the changes and additions.

3.2 Uninstalling Packages

To uninstall a package use the following command:

```
pipenv uninstall package-name
```

Uninstalling a package removes it from the Pipfile and Pipfile.lock, and deletes the package and all of its dependencies from the virtual environment's site-packages folder. For example, to uninstall the *numpy* package use the following command:

```
pipenv uninstall numpy
```

Quick Review

Installing a package installs the package and all if its required dependencies. You can install a package for use only for development or for production. Installing a package creates an entry in the Pipfile. Packages are listed in the Pipfile in one of two sections: `[packages]` or `[dev-pack-ages]`. Packages listed under `[packages]` will be included with the application when it is deployed or packaged for distribution. Packages installed under `[dev-packages]` are only used to support application development and are not included with the application when it is deployed or packaged for distribution.

When installing a package you can specify an exact version number. If you omit the version number, then pipenv installs the latest version of the package.

The Pipfile is the only file associated with the virtual environment you need to check in to the repository. Be sure to list the .venv directory and the Pipfile.lock in the .gitignore file.

The purpose of the Pipfile.lock is to document exactly which package versions are installed along with all their dependencies and version numbers.

4 RECREATE VIRTUAL ENVIRONMENT FROM PIPFILE

To reiterate — the only file you commit to the repository is the Pipfile. If you have the Pipfile, you can recreate the virtual environment including packages and their dependencies. This takes three or four steps depending on whether you have packages in both the `[packages]` and `[dev-packages]` sections.

Step 1: Examine the Pipfile and note the Python version

Step 2: Run `pipenv --python` *python-version* to create the virtual environment

Step 3: Run `pipenv install --dev` to install packages listed in `[dev-packages]` section

Step 4: Run `pipenv install` to install packages listed in `[packages]` section

For example, the commands required to recreate the virtual environment given by the Pipfile listed in example 10.4 would be:

```
pipenv --python 3.11
```

```
pipenv install --dev
```

```
pipenv install
```

You can run the last two commands in either order. I prefer installing the development packages first, so I don't forget.

4.1 AUTOMATICALLY INSTALL PYTHON VERSION WITH PYENV

You can skip the first manual step and have Pipenv automatically install the specified version of Python found in the Pipfile, but you need to install a tool called *Pyenv*. I'll leave this for you to explore as an exercise.

QUICK REVIEW

The only file you need to recreate a virtual environment with Pipenv is the Pipfile. It contains the required version of Python along with packages used for development `[dev-packages]` and production `[packages]`. The only file related to the virtual environment you should commit to your repository is the Pipfile.

5 PIPENV GRAPH AND PIPENV CHECK
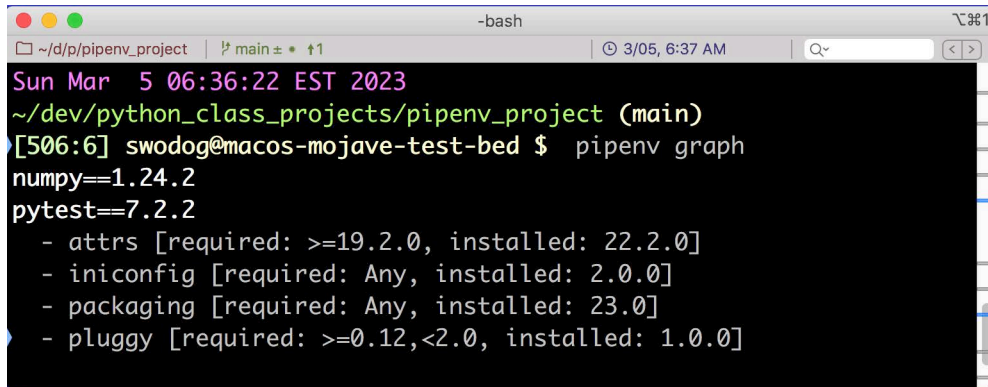
Before signing off on this chapter, I want to show you two other extremely helpful Pipenv commands: `pipenv graph` and `pipenv check`.

5.1 PIPENV GRAPH

Use the `pipenv graph` command to get a quick, easy-to-read listing of all installed packages and their dependencies. Simply type the following command:

```
pipenv graph
```

Figure 10-17 shows the typical results of running this command.

Figure 10-17: Listing Packages and Dependencies with `pipenv graph` Command

Referring to figure 10-17 — Your results will differ depending on the number of packages installed, their version numbers, and their dependencies.

5.2 PIPENV CHECK

In today's security conscience world, it's essential you know the code used in your programs, especially third-party code, is free from known vulnerabilities. You can scan installed packages against PEP 508 compliance and known security vulnerabilities using the `pipenv check` command. Simply run it like so:

```
pipenv check
```

Figure 10-18 shows the results of running this command in the project containing the packages listed in figure 10-17.

QUICK REVIEW

Use `pipenv graph` to display a user-friendly, easy-to-read list of installed packages and their dependencies. Use `pipenv check` to scan installed packages and their dependencies for known security vulnerabilities.

Figure 10-18: Scanning Packages for Security Vulnerabilities with `pipenv check` Command

6 PIPENV COMMAND SUMMARY

Table 10-2 lists the Pipenv commands and options used in this chapter.

Pipenv Command	Description
`pipenv --python` *python-version*	Create a virtual environment with given version of Python
`pipenv run python3` *module.py*	Run *module.py* using virtual environment's `python3` command.
`pipenv install --dev` *package-name*	Install the latest version of package for development use only. Package name and the version `"*"` will appear in the `[dev-packages]` section of the Pipfile.

Table 10-1: List of Pipenv Commands Used in this Chapter

Pipenv Command	Description
`pipenv install --dev` *package-name*`==n.n.n`	Install specified version of package for development use only. Package name and version `"==n.n.n"` will appear in the `[dev-packages]` section of the Pipfile.
`pipenv install` *package-name*	Install latest version of package for production. Package name and version `"*"` will appear in the `[packages]` section of the Pipfile.
`pipenv install` *package-name*`==n.n.n`	Install specified version of package for production. Package name and version `"==n.n.n"` will appear in `[packages]` section of Pipfile.
`pipenv install --dev`	Install packages listed in `[dev-packages]` section of Pipfile.
`pipenv install`	Install packages listed in `[packages]` section of Pipfile.
`pipenv uninstall` *package-name*	Uninstall package.
`pipenv graph`	Displays user-friendly list of installed packages and their dependencies.
`pipenv check`	Checks installed packages for known security vulnerabilities.

Table 10-1: List of Pipenv Commands Used in this Chapter (Continued)

SUMMARY

Python packages are normally installed in the active Python distribution's site-packages folder. This leads to problems, especially if you have multiple projects using different versions of the same package, or multiple projects using different versions of Python. To tame this sort of unruly complexity, use virtual environments to specify Python versions, isolate project dependencies, and prevent package conflicts.

Pipenv is a tool that lets you create Python virtual environments. Before using Pipenv to create a virtual environment, ensure you have set the environment variable `PIPENV_VENV_IN_PROJECT` to `true` [`PIPENV_VENV_IN_PROJECT=true`]. This ensures the *.venv* directory is created in the project's root directory.

To create a virtual environment, you must have a corresponding version of Python installed on your system. For example, to create a Python 3.10 virtual environment, you must have Python 3.10 installed on your computer. You don't need multiple versions of Python installed in your computer to use Pipenv, unless you are working on projects targeting different Python versions. Using Pipenv for package management and isolation is sufficient reason enough to use Pipenv.

Installing a package installs the package and all if its required dependencies. You can install a package for use only for development or for production. Installing a package creates an entry in the Pipfile. Packages are listed in the Pipfile in one of two sections: `[packages]` or `[dev-packages]`. Packages listed under `[packages]` will be included with the application when it is deployed or packaged for distribution. Packages installed under `[dev-packages]` are only used

to support application development and are not included with the application when it is deployed or packaged for distribution.

When installing a package you can specify an exact version number. If you omit the version number, then pipenv installs the latest version of the package.

The Pipfile is the only file associated with the virtual environment you need to check in to the repository. Be sure to list the *.venv* directory and the Pipfile.lock in the .gitignore file.

The purpose of the Pipfile.lock is to document exactly which package versions are installed along with all their dependencies and version numbers.

The only file you need to recreate a virtual environment with Pipenv is the Pipfile. It contains the required version of Python along with packages used for development `[dev-packages]` and production `[packages]`. The only file related to the virtual environment you should commit to your repository is the Pipfile.

Use `pipenv graph` to display a user-friendly, easy-to-read list of installed packages and their dependencies. Use `pipenv check` to scan installed packages and their dependencies for known security vulnerabilities.

SKILL-BUILDING EXERCISES

1. **Install and Configure Pyenv:** In section 4.1, I mentioned a tool called Pyenv, which would enable you to automatically install the Python version specified in a project's Pipfile. This can be super helpful, especially if you regularly try to run projects that require different versions of Python. Visit the Pyenv GitHub page and study the documentation: *https://github.com/pyenv/pyenv* Follow the installation instructions for your operating system. Configure the *.bashrc* and *.bash_profile* settings as instructed.

2. **Create Virtual Environments with Pipenv:** Practice creating virtual environments with Pipenv. When you create a virtual environment, explore the *.venv* directory and examine the contents of the Pipfile.

3. **Install and Uninstall Packages:** Practice installing and uninstalling packages into a virtual environment. Install packages for development and production. Each time you install a package examine the Pipfile and Pipfile.lock and note the changes. Practice installing packages with specific version numbers.

4. **Browse Packages on PyPI:** Visit the *Python Package Index (PyPI)* site and explore the packages available. Search for popular packages. Research popular packages on the Internet and search for them on PyPI. Note what they are used for, how they are installed, and visit their project page. If you see a package you like, or think you may use at some point, practice installing it into a virtual environment.

5. **Scan Installed Packages for Security Vulnerabilities:** Use the `pipenv check` command to scan installed packages and their dependencies for security vulnerabilities. Conduct research on the scanning tool used by the `pipenv check` command.

6. **Research All Pipenv Commands:** Research all the Pipenv commands and their options.

7. **Research Basic Pipenv Usage:** Study the *basic Pipenv usage* page. Note any usage scenarios not covered in this chapter and how you might use them on your projects.

8. **Research Advanced Pipenv Usage:** Study the *advanced Pipenv usage* page. Note any usage scenarios not covered in this chapter and how you might use them on your projects.

Suggested Projects

1. **Convert Existing Project to Use Virtual Environment:** Take a project you're currently working on and switch it to a virtual environment.

2. **Adopt Pipenv Virtual Environments for Future Projects:** Going forward, create virtual environments for all of your new projects.

Self-Test Questions

1. What are three good reasons for using Python virtual environments?

2. How does a project's virtual environment isolate packages from the global Python environment?

3. What's the purpose of the Pipfile?

4. What Pipenv command would you use to create a Python 3.10 virtual environment?

5. What Pipenv command would you use to install a package for development use only?

6. What steps could you take to verify a package is installed in the virtual environment?

7. How do you scan installed packages for security vulnerabilities?

8. How do you uninstall packages from a virtual environment?

9. What's the purpose of the Pipfile.lock?

10. How can you use a Pipfile to recreate a virtual environment?

REFERENCES

Python Package Index (PyPI), *https://pypi.org*

Pipenv: Development Workflow for Humans, *https://pipenv.pypa.io/en/latest/*

Google's V8 JavaScript Rendering Engine, *https://v8.dev*

Pyenv: Simple Python Version Management, *https://github.com/pyenv/pyenv*

NOTES