Learning Objectives: Third-Party Packages

Learners will be able to…

- **Define a virtual environment and explain its benefits**
- **Create a virtual environment**
- **Activate/deactivate a virtual environment**
- **Install and update packages with `pip`**
- **List all of the installed packages**
- **Define, generate, and install from a requirements file**
- **Remove packages from a virtual environment**
- **Explain how dependencies are handled when installing and uninstalling**
- **Install a package from GitHub**

info

Make Sure You Know

You have some familiarity with the terminal: navigating between directories, making directories, the redirect operator, etc. You have seen/heard of GitHub.

Limitations

We do not have an in-depth discussion about virtual environments changing the path or adding links. Uploading projects to PyPI is mentioned in only a cursory fashion. You will not be doing this yourself.

Virtual Environments

Installing Third-Party Packages

Before we start installing third-party packages, we need to discuss how Python stores installed packages. By default, Python installs third-party packages for the entire system. That is, if you install `requests` for one project, you can use the package again in another project without having to install it again. This might sound like a good thing, but it can cause problems. Packages often depend on other packages. These are called dependencies.

Let’s say you install package `x` which requires the dependency `acme v1.3`. You then later install package `y` which has the dependency `acme v1.1`. Python overwrites `acme v1.3` with the earlier version `acme v1.1`. Now package `x` no longer works because one of its dependencies is broken.

Other potential problems could arise if you mix third-party packages with the standard library. There could be side effects from all of these packages being in the same place. Or, updating the operating system could affect third-party packages stored alongside the standard library.

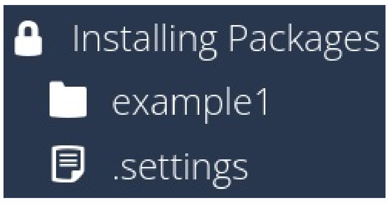
Virtual Environments

The way to avoid the aforementioned problems is to use a virtual environment for your Python project. The built-in Python virtual environment is called `venv`. Using a virtual environment is similar to having a separate directory with your code, the Python language, and any packages you install. A virtual environment is completely isolated from the larger Python system and other virtual environments. Using `venv` means you avoid the problems of dependency mismatches. It is a good idea to create a virtual environment for every Python project you start.

Before creating a virtual environment, make sure you have a dedicated directory for your project. Your code and the virtual environment should go in this directory. Copy/paste the command in the terminal to make the directory `example1`. Press `ENTER` to run the command.

```
mkdir example1
```

You should now see the `example1` directory in the file tree.



The image depicts the root of the project with the example1 directory.

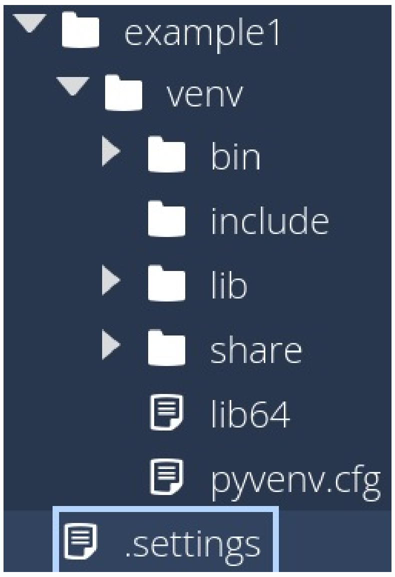
Each Python project should have its own virtual environment. So we need to change to the newly created directory with the command below.

```
cd example1
```

Once we are in our project's folder, we can create our virtual environment. The command below creates a virtual environment named `venv`.

```
python3 -m venv venv
```

Python added the `venv` virtual environment to the `example1` directory. To verify this, click on the arrow next to `example1` in the file tree. Do the same thing for `venv`. This is the contents of your virtual environment. Your file tree should look like this:



The image depicts the venv virtual environment directory inside of example. There are several subdirectories and files in the virtual environment.

Finally, we can activate our virtual environment. This is important to note, virtual environments do not run constantly in the background. You should activate them every time you want to work on your project.

```
source venv/bin/activate
```

We can verify your virtual environment is working by looking at our prompt in the terminal. `(venv)` should appear at the beginning of your prompt. **Note**, your prompt will look different than the one below, but as long as it has `(venv)` you are ready to go.

```
(venv) codio@schoolbetty-lessongyro:~/workspace/example1$
```

challenge

Try this variation:

- Delete your virtual environment. In the file tree, open the `example1` directory. Right-click on the `venv` directory and select delete.
- In the terminal, make sure that you are in the `example1` directory. Create a new virtual environment, but this time give it the unique name `example-venv`.

```
python3 -m venv example-venv
```

- Active the new virtual environment.

```
source example-venv/bin/activate
```

- In the file tree, the virtual environment `venv` should be replaced by `example-venv`.

▼ **Did you notice?**
There was no special command to run from the terminal to uninstall the virtual environment. It is just a directory. Keep in mind, if you install third-party packages and then remove the virtual environment you will have to reinstall them.

Once you are done working with your virtual environment, you can deactivate it with the `deactivate` command. The name of the virtual environment should disappear from your prompt in the terminal.

```
deactivate
```

Package Management

Installing Packages

The Python team maintains an official repository of third-party packages called the Python Package Index (PyPI). PyPI is pronounced “pie-pea-eye”. The site has several hundred thousand packages. You can search the site for packages, and the index provides information about how to install the package, how to use it, its license, when it was last updated etc.

Once you know the package you want to install, you will use the tool `pip` to install the desired package. `pip` stands for “Python installer for packages”. Before we begin installing anything, change to the `example1` directory and activate your virtual environment.

```
cd example1
source example-venv/bin/activate
```

▼ **Did you notice?**
At the end of the previous page, we removed the virtual environment `venv` and replaced it with `example-venv`. We will continue to use this virtual environment for the remainder of the assignment.

Once you see the `(example-venv)` in your prompt, you are ready to install a package. Let’s start by installing `beautifulsoup4`, a popular package for scraping websites. In the terminal, enter the following command to install Beautiful Soup:

```
python3 -m pip install beautifulsoup4
```

▼ **Did you notice?**
We are using `python3 -m pip` instead of `pip`. Yes, you can run `pip` directly. However, you gain more control over `pip` when using the `-m` flag. This flag tells Python to run a module as script. It is not uncommon for computers to have more than one version of Python installed. In fact, this environment has Python 2 and Python 3 installed. By using `python3 -m pip` we are telling the computer to use Python 3 for the operation.

You should see a message in the terminal saying that Beautiful Soup was successfully installed. We can verify this using the `list` command. This will print out all of the installed packages.

```
python3 -m pip list
```

Some of the installed packages have titles that are not very helpful in understanding what they do. You can use the `show` command to get the metadata for a particular package.

```
python3 -m pip show soupsieve
```

You will see the following information:

```
Name: soupsieve
Version: 2.3.2.post1
Summary: A modern CSS selector implementation for Beautiful
Soup.
Home-page: None
Author: None
Author-email: Isaac Muse < Isaac.Muse@gmail.com >
License: None
Location:
/home/codio/workspace/example1/venv/lib/python3.6/site-packages
Requires:
```

challenge

Try these variations:

- In the file tree, open `example1`. Then open `example-venv`. Under `lib`, open `python3.6` and `site-packages`. You should see the Beautiful Soup package.
- Contrast your virtual environment with your Python installation. Deactivate `venv` and list all of the installed packages. Your virtual environment should have a different set of packages installed.

```
deactivate
python3 -m pip list
```

Installing Multiple Packages

If your virtual environment has been deactivated, be sure to activate it before continuing.

```
source example-venv/bin/activate
```

You can install multiple packages at once with `pip`. Let’s say you are looking to get into scientific computing and data science. Two packages often used for these tasks are `scipy` and `pandas`. Just put a space between each package you want to install.

```
python3 -m pip install scipy pandas
```

These packages are a bit larger and have more dependencies, so the installation may take a little time.

Upgrading Packages

`pip` also allows you to upgrade installed packages. Use the normal command to install a package but add the `--upgrade` flag.

```
python3 -m pip install --upgrade beautifulsoup4
```

You can use the shortened flag `-U` instead of `--upgrade`. In addition, you can list out more than one package to upgrade.

```
python3 -m pip install -U scipy pandas
```

You can even upgrade `pip` itself.

```
python3 -m pip install -U pip
```

Requirements File

Requirements File

You can simplify the installation of many packages by using a requirements file. The `requirements.txt` file is a text file that lists the packages and versions used in a project. Python can then use this list to install the specific version of each package listed.

Before we can install from this list, we need to first create the `requirements.txt` file. Make sure that you are in the directory with your virtual environment. Then activate the environment. We want to generate the requirements only for the environment, not your general Python installation.

```
cd example1
source example-venv/bin/activate
```

Use the redirect operator (>) to take the output from `freeze` and write it to the `requirements.txt` file. This process is also called pinning as the exact version will be recorded in the requirements file. After running the command, click on the link below to open the requirements file.

```
python3 -m pip freeze > requirements.txt
```

Installing from a Requirements File

Another benefit of the requirements file is that you can use it as a list for packages to install. Let's start by creating a second virtual environment. We first need to deactivate the virtual environment and exit the current directory, `example1`. Then we are going to make a directory called `example2` and change to this directory.

```
deactivate
cd ..
mkdir example2
cd example2
```

Now, create a new virtual environment (we aren't going to use a special name, `venv` is sufficient). Then activate the new virtual environment. Your prompt should have `(venv)` at the beginning.

```
python3 -m venv venv
source venv/bin/activate
```

We are ready to use the requirements file in the `example1` directory. Give `pip` the `-r` flag to tell Python that the requirements are found in the file `requirements.txt`.

```
python3 -m pip install -r ../example1/requirements.txt
```

▼ **Did you notice?**
You have to use the `-r` flag to tell Python the name of the requirements file. That means you can name it whatever you want. However, convention dictates that `requirement.txt` be used as the title.

Now list the packages in the `example2` virtual environment. You should see the same ones as in `example1`.

```
python3 -m pip list
```

Specifying Package Versions

Looking again at the contents of the `requirements.txt`file, you will notice that the == operator is used with the version number for each package. This tells Python to install exactly this version.

You may find yourself wanting to keep the same packages but upgrade to the latest version. We can do this by making some changes to the requirements file. Replace all of the == operators with >=. This tells Python to use this or a later version. **Note**, your list of packages and version numbers may be different from the time of writing.

```
beautifulsoup4==4.11.1
numpy==1.19.5
pandas==1.1.5
pkg_resources==0.0.0
python-dateutil==2.8.2
pytz==2022.2.1
scipy==1.5.4
six==1.16.0
soupsieve==2.3.2.post1
```

After you made these changes, you can tell Python to upgrade the packages in the requirements file by using the `-U` flag.

```
python3 -m pip install -U -r ../example1/requirements.txt
```

One final use case for the requirements file is separating production from development environments. You are **not** going to create anything, but it is worth understanding how this works. Some developers use testing packages not found in the standard library. For example, `pytest` is popular for testing, but it is not needed for production. You would create your `requirements.txt` file without any testing packages. Then you would make another version called `requirements_dev.txt`. The contents of this file should look like this:

```
# requirements_dev.txt

-r requirements.txt
pytest>=7.1.1
```

Installing from the `requirements_dev.txt` file will first install all of the packages from `requirements.txt`. Then it will install the `pytest` package. When it comes time to deploy your package, you can install the needed packages from `requirements.txt` which excludes any testing packages.

Uninstalling Packages

Uninstalling Packages

If `python3 -m pip install` is used to install a package, it stands to reason that `python3 -m pip uninstall` is how you remove it. We don’t need Beautiful Soup right now, so let’s uninstall it. Make sure that you are in the `example1` directory and you have activated the virtual environment.

```
cd example1
source example-venv/bin/activate
```

Now you are ready to uninstall the package.

```
python3 -m pip uninstall beautifulsoup4
```

Python is going to ask if you are really sure. Enter `y` in the terminal and press `ENTER`. Look at all of the installed packages in your virtual environment to verify the package is no longer there.

```
python3 -m pip list
```

Python asks that you verify your intention to remove a package. That is a handy way of making sure you don’t accidentally remove an essential package. If you are sure you don’t need the package, that extra check becomes a minor annoyance. You can uninstall packages while suppressing the verification.

```
python3 -m pip uninstall scipy -y
```

challenge

Try this variation:

- In the file tree, open `example1`. Then open `example-venv`. Under `lib`, open `python3.6` and `site-packages`. You should see that Beautiful Soup and Scipy are no longer there.

Removing Packages and their Dependencies

When installing a package, `pip` will automatically install the dependencies. However, `pip` will not remove dependencies when uninstalling. For example, the `scipy` package has `numpy` as a dependency. So does `pandas`. If `pip` removed `numpy` when we uninstalled `scipy`, then `pandas` would no longer work as expected. To see the dependencies for a package use the `show` command.

```
python3 -m pip show pandas
```

Look under the `Requires:` line and you will see all of the dependencies for `pandas`. If you want to truly remove `pandas`, you need to remove it and all of its dependencies.

```
Name: pandas
Version: 1.1.5
Summary: Powerful data structures for data analysis, time
series, and statistics
Home-page: https://pandas.pydata.org
Author: None
Author-email: None
License: BSD
Location: /home/codio/workspace/example1/example-
venv/lib/python3.6/site-packages
Requires: pytz, python-dateutil, numpy
```

Let's remove `pandas` and all of its dependencies by listing them out in the same `uninstall` command. We will definitely want to use the `-y` flag so we don't have to keep approving each package removal.

```
python3 -m pip uninstall pandas pytz python-dateutil numpy -y
```

It is important to make sure that none of the other packages require the same dependencies as `pandas`. We could break our environment if we are not careful. One of the nice things about virtual environments is that they are fungible. Let's say we don't want `pandas` anymore, but we don't want to take the time to check the dependencies of all of the other packages. Create a second virtual environment. Use the `requirements.txt` file (minus `pandas` of course) from the first virtual environment to install what you need. Then delete the first virtual environment. This is much easier and faster.

challenge

Try this variation:

- In the file tree, open `example1`. Then open `example-venv`. Under `lib`, open `python3.6` and `site-packages`. You should see that `pandas` and all of its dependencies are no longer there.

Removing Packages with the Requirements File

Just as you can install a set of packages with the requirements file, you can uninstall all of those same packages with `requirements.txt`. Let's first deactivate the virtual environment for `example1` and then activate `example2`.

```
deactivate
cd ../example2
source venv/bin/activate
```

A requirements file does not exist for `example2`, so let's make one real quick.

```
python3 -m pip freeze > requirements.txt
```

Use the `uninstall` command along with `-r requirements.txt` to indicate the name of the requirements file. Again, using the `-y` flag will keep from having to confirm each package removal.

```
python3 -m pip uninstall -r requirements.txt -y
```

Finally, let's list out the packages in the virtual environment. You should not see `beautifulsoup4`, `scipy`, `pandas`, `numpy`, etc.

```
python3 -m pip list
```

challenge

Try this variation:

- In the file tree, open `example2`. Then open `venv`. Under `lib`, open `python3.6` and `site-packages`. You should see only the packages used on this system.

Installing from GitHub

Installing Packages from GitHub

GitHub is one of the most popular code repositories. You can publish and share your projects with the world. This also includes the ability to upload share Python packages. Instead of using the traditional `git` tools to clone a repository, you can install the package directly from the GitHub website with `pip`.

Before we do this, let’s create the `example3` directory and set up a virtual environment. We will name this environment `github-pkg` since we are installing a package from GitHub.

```
mkdir example3
cd example3
python3 -m venv github-pkg
source github-pkg/bin/activate
```

Head on over to the GitHub repository for the package. You can explore the different directories and folders to get an idea of what this package does.

▼ **Did you notice?**
There is no `requirements.txt` file because no third-party packages are used. There is also a `setup.py` file which is required for installing a package. All of the packages hosted on PyPI have this file as well. You do not need a `setup.py` file if you are writing your own packages and not hosting them elsewhere.

To install directly from GitHub, you are going to start your command with `python3 -m pip install` like normal. There are several ways of communicating with GitHub. We are going to use HTTPS, so the next part becomes `git+https://`. Next, add the URL from the address bar for the GitHub repository. The final thing you need to add is `#wheel=shapes`.

▼ **What is a wheel?**
Wheels are an important part of the package installation process. They allow for faster and more stable downloads and installations. We will talk more about wheels in the next assignment.

Put it all together, and this is the command to install the `shapes` package from its GitHub repository. Run the command in the terminal.

```
python3 -m pip install git+https://github.com/codio-
        content/shapes-package#wheel=shapes
```

Testing the Shapes Package

To test the newly installed package, we first need a Python file in which we can write code. Use the `touch` command to create an empty file called `shapes_example.py`.

```
touch shapes_example.py
```

The link below will open the blank file. The name of the file **must** be `shapes_example.py`.

Copy/paste this script into the Python file. The `shapes` package has three classes, `Square`, `Circle`, and `Rectangle`. Create an instance of each class and call the `area()` method for each object.

```
from shapes.square import Square
from shapes.circle import Circle
from shapes.rectangle import Rectangle

s = Square(5)
c = Circle(10)
r = Rectangle(5, 10)

print(f"The square's area is {s.area()}")
print(f"The circle's area is {c.area()}")
print(f"The rectangle's area is {r.area()}")
```

Click on the tab for the terminal and enter the command below. Press `ENTER` to run the script.

```
python3 shapes_example.py
```

If everything works as intended, you should see the following output:

```
The square's area is 25
The circle's area is 314.1592653589793
The rectangle's area is 50
```

Formative Assessment 1

Formative Assessment 2