Harvard John A. Paulson School of Engineering and Applied Sciences

F. Wermelinger
Office: Pierce 211

Pair-Programming 8

Virtual environments, Deploying packages

Issued:	October 21, 2022
Due:	November 4, 2022 11:59pm

In this pair-programming session you will create virtual Python environments to install and test your packages in development stages. In the second part you will create a small test package and deploy it to the PyPI test index.[1]

You should work on the exercises in groups of 3 to 4 students via a `tmate` session. Your team members can submit the same file. Please indicate your names in a header in the files. See the tutorials on the class website for an example pair-programming workflow.[2] Do not forget to commit and push your work when you are done. Ensure that you are on your *default branch* for this and not, possibly, on your homework branch.

Exercise 1: Virtual Python Environments

Deliverables: None

Virtual environments in Python are a very useful tool for testing your Python packages in an *isolated environment*. A virtual environment behaves like any Python installation but anything you do within the virtual environment, for example installing packages, will not touch your production environment, typically your system installation of Python. It is important to the keep the system installation clean an functional on on operating systems like Linux or MacOSX where Python is an integral part of. Therefore, virtual Python environments are an ideal playground for testing out code in isolation.

There are different tools to create virtual environments. Any recent Python installation ships with the `venv` module to create lightweight virtual environments. This module is very useful to quickly test some code in an isolated environment. The `venv` module only allows you to create a virtual environment based on the version of the Python interpreter you are using. Packages are still managed using `pip`.

There are other more powerful alternatives like `pyenv`[3] that allows you to create different virtual environments using different versions of the Python interpreter or `conda`[4] which is

[1] https://test.pypi.org/
[2] https://harvard-iacs.github.io/2022-CS107/pages/tutorials.html#tutorial-pp
[3] https://github.com/pyenv/pyenv
[4] https://docs.conda.io/en/latest

a package, dependency and environment management system all together. The `pip` and `venv` tools are already a quite powerful combination and we will focus on them in the following.

a) Before you start, create a `.gitignore` file in your `lab/pp8` directory with the following content

```
$ echo '/test_env' >.gitignore
```

This will ignore the directory "`test_env`" which is where we create a virtual environment next. The following commands assume you are inside your `lab/pp8` directory.

> **Note:** on many platforms the executable "`python`" is a symbolic link to the `python3` executable installed on your system. Should the `python` executable not exist on your system, please use `python3` for the interpreter in the examples below.

Before you create this environment, checkout the options of the `venv` module using

```
$ python -m venv -h
```

Typically the default options are good enough. In case you ever need to create a more specific virtual environment, these are to options you have available. The two core dependencies are `pip` and `setuptools`, if you did not want `pip` in your virtual environment you can disable it with the `--without-pip` option.

To get an idea which Python interpreter you are using, run the command

```
$ which python
```

This will print the path to the Python executable (interpreter) that is currently found in your `PATH`. Typically this points to your system binaries (`/usr/bin`) unless you have another environment loaded (e. g. `conda`). You can now create your virtual Python environment by running the command

```
$ python -m venv test_env
```

This will create the virtual environment using the Python interpreter you have just investigated. The virtual environment is created in a directory called "`test_env`" in the current directory. Before you can use the environment you have to *activate* it.

Activating an environment means to *source* a shell script that will setup the environment for you *in the current shell you are running*. To activate it run

```
$ source test_env/bin/activate
```

You can now verify that the Python interpreter (and therefore the Python environment) is pointing into the virtual environment you have just created and activated, to do so type

```
$ which python
```

If you want to deactivate the environment again you can type

```
$ deactivate
```

or exit the shell where you have sourced the environment.

b) The following assumes you have the virtual environment you created in the previous task activated.

Change into `test_env/bin` and use the `ls` command to inspect the contents. You should see a few files have been copied there but other files, for example the `python` interpreter itself are *symbolic links* to the Python interpreter you have used to create the virtual environment. See the options for `python -m venv -h` to figure out how you could create a virtual environment without symbolic links. This could be useful if you want to share it with somebody who will run it on their computer with the same architecture as yours.

Now change into `../lib/python3.10/site-packages` (note that if you used a different Python interpreter the version in the path will be different for you). Type `ls -l` to list what is in there. You will see that currently you only have the core dependencies (`pip` and `setuptool`) installed in your virtual environment. Try to run

```
$ python -c 'import numpy'
```

This should fail because you do not have NumPy installed in your virtual environment. Install it with

```
$ python -m pip install numpy
```

and type `ls -l` again. You should now see that the NumPy package has been installed in your virtual environment *and not in your system Python installation* (where you probably have it installed already anyway).

In Python, packages are installed into a directory called "`site-packages`," which lives below the path of the Python installation (or virtual environment) you are using. If you define nothing special in your `PYTHONPATH` environment variable, then this is one of the locations Python searches for installed packages. You can get a description of your current Python environment by running

```
$ python -m site
```

This will list you the system paths but also the user base where Python is searching for packages. Something special you notice when you have a virtual environment loaded is that `ENABLE_USER_SITE` is set to *false*. Because virtual environments are *isolated environments* this option should be set to false. It means that if you ran the command

```
$ python -m pip install --user numpy
```

above, the NumPy package would still be installed inside the `site-packages` directory inside the virtual environment and any Python packages you may have installed in your user base will not be reachable when you have a virtual environment loaded. Try to run

```
$ python -m site
```

in another shell where you do not have the virtual environment loaded for comparison.

Virtual environments are intended for testing. You can be destructive in such an environment without hurting your production environment. An example where you want to use a virtual environment is to test out the dummy project we have deployed in lecture 10. You can install it in your virtual environment using the command

```
$ python -m pip install -i https://test.pypi.org/simple/ Fall2022-CS107
```

and run the code in `__main__.py` with

```
$ python -m cs107_package
```

Note that our test package depends on NumPy which you have installed above already. If you did not do that, you would need to run

```
$ python -m pip install \
  --extra-index-url https://pypi.org/simple/ \
  -i https://test.pypi.org/simple/ Fall2022-CS107
```

To remove a virtual environment completely, you can just remove the directory for the virtual environment. For this lab you can run

```
$ rm -rf test_env
```

to clean up.

Note that the teaching staff will test your project code in a virtual environment like this. You will have to make sure that dependencies are resolved in your package configuration automatically (e. g. by using a `pyproject.toml` file or possibly a `setup.cfg` file together with the `setuptools` backend for `pip`).

For further reading, please see the `venv` docs[5] and the Python packaging docs[6]

[5] https://docs.python.org/3/library/venv.html

[6] https://packaging.python.org/en/latest/tutorials/installing-packages/#creating-and-using-virtual-environments

Exercise 2: Deploy a Python Package

Deliverables:

1. `install.sh`

In this part you and your pair-programming partners should deploy a test package to the PyPI testing server[7]. For this purpose, please work through the tutorial at https://packaging.python.org/en/latest/tutorials/packaging-projects/.

You may want to try different PEP517[8] backends. The tutorial offers four, the one we used in lecture 9 was `setuptools` and we used a configuration file `setup.cfg` for it (see the discussion https://peps.python.org/pep-0518/#sticking-with-setup-cfg regarding possible issues for this approach). Note that in this tutorial, all configuration is done in the `pyproject.toml` file (instead of a `setup.cfg` file), which is also possible. See PEP518[9] for the detailed specification of the `pyproject.toml` file.

The deliverable for this task is *an executable* shell script called `lab/pp8/install.sh` which should install the test package you have uploaded to https://test.pypi.org/. It should contain a `python -m pip install` command that will install your package when the teaching staff is executing the shell script in a virtual Python environment similar to what you explored in the previous task.

[7] https://test.pypi.org/
[8] https://peps.python.org/pep-0517/
[9] https://peps.python.org/pep-0518/