Department of Computer Science Computing Guide

Search this site

Python and Virtual Environments

Last major update: 2022-08-15

NOTE (2019-06-20): If your project relies on Python packages that CS Staff has previously installed at the system level, please migrate your project to a virtual environment immediately. We no longer install packages at the system level and, at our next major update of the Linux distribution on cycles, ionic, courselab, and armlab, we will not carry forward such packages.

Contents

- Introduction
- Which version of Python should I use? Which version is installed?
  - A Word About “System Python”
  - Building a Local Version of Python
- What is a Virtual Environment?
- Always use a Virtual Environment
- Creating a Virtual Environments and Installing a Package
- Virtual Environments and Source Control (e.g., Git)
- Removing or Renaming a Virtual Environment
- Help, I can’t install a package into my Virtual Environment

Introduction

A key benefit of using Python is its active developer community and the large amount of available software packages available from pypi.org.

The original design of Python and its packaging system puts installed packages alongside the Python interpreter[1] in the file system; furthermore, only a single version of a given package can be installed with a given Python interpreter. This led to issues when different projects required different versions of packages. To address this problem, *Python virtual environments* were developed.

In a nutshell, Python virtual environments help decouple and isolate Python installs and associated pip packages. This allows end-users to install and manage their own set of packages that are independent of those provided by the system or used by other projects.

Before creating a Python virtual environment, we need to choose the Python version. This is the topic of the next section.

Which version of Python should I use? What version is installed?

For new development, use Python 3.7 or later. Because the "System Python" (more below) on the CS systems is a relatively old Python 3.6 (but with back-ported security patches) you may wish/need to build your own local copy of your preferred Python version.

If you still have code written for Python 2, we recommend investing the time to port it to Python 3 as soon as feasible. See: Porting Python 2 Code to Python 3. The end-of-life (EOL) for Python 2 was January 1, 2020.

The following subsections describe the Python version installed on our systems and how to build a particular version from source.

A Word About "System Python"

Because Python is an essential part of Linux distributions (i.e., used by the system itself), the default version and availability of Python is generally fixed until the next upgrade of the overall operating system. This means the installed Python tends to be older than current, stable versions available from python.org.

This default version is typically installed in the `/bin` directory and is referred to as "System Python." While System Python is available to end users, its primary use is to support the operating system. As a result, CS Staff will update it per our Linux distribution schedule. This means that it won't be updated upon request and may be updated with little or no notice. Additionally, we won't be installing third-party Python packages at the system level.

As of 2022-08-09, the “System Python” for `cycles` and `ionic` is Python 3.6.8 and is available at the following locations:

```
/bin/python3
/bin/python3.6
/usr/bin/python3
/usr/bin/python3.6
```

As noted above, this version is a bit dated and is missing out on goodies available in more recent versions such as Data Classes (PEP 557), Structural Pattern Matching, a handy `=` specifier in f-strings, and many type hinting features.

Building a Local Version of Python

If the Python version you need is not available on the CS systems, it is straightforward to download source code and build a local Python interpreter in project space.

1. Navigate to the Download Python page, navigate to the version of your choice, download the corresponding “XZ compressed source tarball,” and put it in a temporary directory in your project space. For example (assuming your project space is at `/n/fs/myproject`):

   ```
   $ mkdir /n/fs/myproject/temp
   $ cd /n/fs/myproject/temp
   $ wget https://www.python.org/ftp/python/3.10.6/Python-3.10.6.tar.xz
   ```

2. Un-tar the tarball and build Python. The `--prefix` option to the `configure` script sets the location where Python will be installed. For example:

   ```
   $ tar xJf Python-3.10.6.tar.xz
   $ cd Python-3.10.6
   $ ./configure --prefix=/n/fs/myproject/py310
   $ make
   ```

   Note that you may get output that includes this:

   ```
   Python build finished successfully!
   The necessary bits to build these optional modules were not found:
   ```

```
nis
```

The `nis` module is not useful in the CS environment and is deprecated in Python 3.11 and so this warning can be safely ignored.

3. Install the Python you just built into the location specified by `--prefix` above:

```
$ make install
```

4. Your newly build Python interpreter is available in the `bin` subdirectory of the `--prefix` path set above:

```
$ /n/fs/myproject/py310/bin/python3.10 --version
Python 3.10.6
```

5. (Optional) Remove the source tarball and the build directory.

```
$ cd /n/fs/myproject/temp
$ rm -rf Python-3.10.6.tar.xz Python-3.10.6
```

What is a Python Virtual Environment?

A Python virtual environment (venv) is simply a directory with a particular file structure. It has a `bin` subdirectory that includes links to a Python interpreter as well as subdirectories that hold packages installed in the specific venv. By invoking the Python interpreter using the path to the venv's `bin` subdirectory, the Python interpreter “knows” to use the associated packages within the venv (as opposed to any packages installed alongside the actual location of the Python interpreter). It is in this sense that venvs are “virtual”—they are not virtual in the sense of, say, a virtual machine.

When you set up a virtual environment (details below), you can immediately use it by invoking `python` using the full path to the `bin` subdirectory within your venv (e.g., `myvenv/bin/python my_program.py`). For convenience, when you set up a venv, it provides an *activate* script that you can invoke which will put the `bin` subdirectory for your venv first on your PATH. (It also updates your shell prompt to let you know this change is in effect). When your venv is activated, you no longer need to use the full path to the Python interpreter (e.g., you can simply use `python my_program.py`).

IMPORTANT (and useful): If you invoke your Python program with the full path to the Python interpreter in the virtual environment, it will run in the virtual environment even if you are not in an interactive session where you used the `activate` script. This is useful for cron jobs and for Bash scripts where explicitly "activating" the virtual environment is problematic.

Always use a Virtual Environment

Always.[2]

Virtual environments let you have a stable, reproducible, and portable environment. You are in control of which packages versions are installed and when they are upgraded. You can have as many venvs as you want.

For an additional layer of control over when you update to new versions of Python, you can compile your own Python interpreter and create a virtual environment based on it. This decouples you from the update schedule of the "System Python" installed on our servers.

Virtual environment support is provided out-of-the-box with modern versions of Python.[3] However, it is not the only mechanism one can use to maintain a reproducible environment over which you have control of the version of Python used, the packages installed, and the schedule of when they are updated. Other options available on CS systems include installing and using Conda in your home directory or project space or using Singularity.

Creating Virtual Environments and Installing a Package

In this section, we describe how to create a virtual environment (venv) and install a package within that venv. We are building on the example above where we have a locally built Python 3.10 interpreter in `/n/fs/myproject/py310/bin` .

Create a virtual environment:

```
$ /n/fs/myproject/py310/bin/python3.10 -m venv my_venv_py310
```

The `-m venv` tells Python to use the "venv" module to create a virtual environment in a directory called `my_venv_py310` .

You can now "activate" this virtual environment and manage packages.

The following example assumes you are using the Bash shell. In the example, we activate the virtual environment, install a package using the `pip` module, and run an inline Python script:

```
$ source my_venv_py310/bin/activate
```

The bash prompt changes to indicate that you are in a particular virtual environment.

```
(my_venv_py310) $ python --version
Python 3.10.6

(my_venv_py310) $ python -m pip install six
Collecting six
  Downloading...
Installing collected packages: six
Successfully installed six-1.16.0

(my_venv_py310) $ python -m pip freeze
six==1.16.0

(my_venv_py310) $ python -c "import six; print(six.__version__)"
1.16.0

(my_venv_py310) $ deactivate

$
```

Note that all the above could have been done without using `activate` by fully specifying the path to the Python interpreter in the virtual environment directory each time:

```
$ /n/fs/myproject/py310/bin/python3.10 -m venv my_venv_py310

$ my_venv_py310/bin/python --version
Python 3.10.6

$ my_venv_py310/bin/python -m pip install six
Collecting six
  Downloading...
Installing collected packages: six
Successfully installed six-1.16.0

$ my_venv_py310/bin/python -m pip freeze
```

```
six==1.16.0

$ my_venv_py310/bin/python -c "import six; print(six.__version__)"
1.16.0
```

Note: *when creating a virtual environment* we recommend that you specify the major and minor version number as we have done in the above examples. This will create aliases (i.e., symbolic links) in your virtual environment for `python` , `pythonN` , and `pythonN.M` . This will give you the flexibility of using `pythonN.M` in your scripts or documentation should you choose to do so.

Virtual Environments and Source Control (e.g., Git)

Virtual environment directories are not suited for source control. However, a virtual environment is defined by the version of the Python interpreter and the set of packages (with specific version numbers) installed. Therefore, a common practice is to use `pip freeze` to create a file called (by convention) `requirements.txt` which contains the list of packages:

```
$ source my_project_venv/bin/activate

(my_project_venv) $ python -m pip freeze > requirements.txt
```

The `requirements.txt` file *is* put under source control.

When one checks out or clones the project from a repository for the first time, they then must create a local virtual environment with the appropriate version of the Python interpreter, "activate" the environment, and use `pip install` with the `-r` option, as follows:

```
$ source new_project_venv/bin/activate

(new_project_venv) $ python -m pip install -r requirements.txt
```

A significant downside to committing the output of `pip freeze` to source control is that it is easy to lose track of which packages are your dependencies and which packages are indirect dependencies (i.e., dependencies of your dependencies). One approach to dealing with this, is to keep your dependencies in either a `requirements.in` or `setup.py` file and use a tool such as pip-tools to manage dependencies.

Removing or Renaming a Virtual Environment

To remove a virtual environment, simply remove its corresponding directory:

```
$ rm -rf my_project_venv
```

It is very difficult to rename or move a virtual environment (simply renaming the directory won't work). It is better to create a new virtual environment with the same installed packages. First, extract the packages using the `freeze` command to the `pip` module:

```
$ source my_project_venv/bin/activate

(my_project_venv) $ python -m pip freeze > requirements.txt

(my_project_venv) $ deactivate

$
```

Second, using the instructions above that are appropriate for the version of Python you are using, create a new virtual environment. For our example, we will use `new_project_venv` .

Third, install the packages:

```
$ source new_project_venv/bin/activate

(new_project_venv) $ python -m pip install -r requirements.txt

(new_project_venv) $ deactivate

$
```

Finally, remove the original virtual environment directory.

Rebuilding a Virtual Environment for a New Version of Python

Note that a virtual environment is tied to a particular (major, minor) version of Python. This is established via a symbolic link in the `bin` subdirectory of the virtual environment directory to the actual Python interpreter binary. If the target of the symbolic link no longer exists **or** changes to a different version of Python, the virtual environment will no longer function and must be rebuilt.

Sharing a Virtual Environment Across CS Systems

In order for a virtual environment to function on a server different from the server it was created on, it must (1) the directory must exist in the same relative position in the file system, and (2) the symbolic link in the `bin` subdirectory must (still) point to Python interpreter of the same version.[4]

For example, if you log in to `cycles` you will actually end up on one of the following servers: `soak`, `wash`, `rinse`, or `spin`. Because the `cycles` machines are kept in sync with respect to installed software and because they mount project space identically, a virtual environment that was created while you were logged in to `soak` will work just fine if used while you are later logged in to `rinse`.

For maximum flexibility and portability, you can build a local copy of a Python in your project space. A virtual environment created with such a local copy will then be available on any of the CS Systems that mount project space (including the project-space web servers).

Help, I can’t install a package into my Virtual Environment

Occasionally, you may run into errors installing pip packages. The issues generally fall into the following categories:

1. The pip package is an extension module that includes invoking the C compiler at installation time and requires specific shared libraries (and corresponding “-devel” RPMs) installed at the system level. If the required Linux packages are not installed on the system, the pip install will fail.

2. The pip package is a binary extension module that requires ABI versions of installed shared libraries that are newer than what is installed on our systems. This can happen if the pip package is not compliant with, for example, PEP 571.[5]

3. As part of the installation, the pip package requires a compilation step. If this fails, you may first need to pip install the wheel package and then attempt to install the failing package again.

In any of these cases, please reach out to CS Staff for assistance.

---

1. Technically, Python source is compiled to byte-code with is then interpreted. To keep things simple, we'll just say *interpreter*.↩

2. Exception: If you have a simple Python program that *only* uses modules from the Python Standard Library (i.e., no third-party modules that would need a `pip install`), then you ***might*** consider not using a virtual environment.↩

3. For versions before Python 3.3, there is an alternate mechanism to create Python virtual environments. Contact CS Staff for assistance.↩

4. There's actually a third requirement that if any third-party modules installed in the virtual environment rely on system-installed shared libraries that these shared libraries be installed on all the other servers that need to use the virtual environment. See Help, I can't install a package into my Virtual Environment.↩

5. For those seeking a deeper understanding of Python packaging, especially as it pertains to pip packages using C extensions, see the PyCon 2019 talk: The Black Magic of Python Wheels.↩

/node/3042 built from virtualenv.md on 2022-08-15 15:32:05

Tags:
python