Python Virtual Environments (venv's)

Introduction

A Python virtual environment ("venv" for short) is a tool to create an isolated Python workspace. Within this isolated environment, you can install additional Python packages without affecting the system-wide Python installation. Because a normal user cannot install packages globally, using a virtual environment allows you to install packages locally without needing administrator privileges. This is especially useful when you need to use a package that is not available as a module on the HPC cluster.

> **Tip**
>
> We recommend using the `vsc-venv` script to manage Python virtual environments. This script is available on the HPC clusters and simplifies the process of creating and managing virtual environments. For more information, see the section on vsc-venv. If you want to manage virtual environments manually, you can follow the instructions in the rest of this document.

Managing Python Environments

This section will explain how to create, activate, use and deactivate Python virtual environments.

Creating a Python virtual environment

A Python virtual environment can be created with the following command:

```
python -m venv myenv      # Create a new virtual environment named 'myenv'
```

This command creates a new subdirectory named `myenv` in the current working directory. This directory will contain the packages, scripts, and binaries that are needed to manage the virtual environment.

Warning

When you create a virtual environment on top of a loaded Python module, the environment becomes specific to the cluster you're working on. This is because modules are built and optimized for the operating system and CPUs of the cluster. This means that you should create a new virtual environment for each cluster you work on. See Creating a virtual environment for a specific cluster for more information.

Activating a virtual environment

To use the virtual environment, you need to *activate* it. This will modify the shell environment to use the Python interpreter and packages from the virtual environment.

```
source myenv/bin/activate                    # Activate the virtual
environment
```

Installing packages in a virtual environment

After activating the virtual environment, you can install additional Python packages with `pip install`:

```
pip install example_package1
pip install example_package2
```

These packages will be scoped to the virtual environment and will not affect the system-wide Python installation, and are only available when the virtual environment is activated. No administrator privileges are required to install packages in a virtual environment.

It is now possible to run Python scripts that use the installed packages in the virtual environment.

Tip

When creating a virtual environment, it's best to install only *pure* Python packages. Pure Python packages consist solely of Python code and don't require compilation. The installation method of these packages doesn't impact performance since they're not compiled.

Compiled libraries with a Python wrapper (*non-pure* Python packages) are better loaded as modules rather than installed in the virtual environment. This is because modules are optimized for the HPC cluster's specific hardware and operating system. If a non-pure Python package isn't available as a module, you can submit a software installation request.

To check if a package is available as a module, use:

```
module av package_name
```

Some Python packages are installed as extensions of modules. For example, `numpy`, `scipy` and `pandas` are part of the `SciPy-bundle` module. You can use

```
module show module_name
```

to check which extensions are included in a module (if any).

Using a virtual environment

Once the environment is activated and packages are installed, you can run Python scripts that use the installed packages:

example.py

```
import example_package1
import example_package2
...
```

```
python example.py
```

Deactivating a virtual environment

When you are done using the virtual environment, you can deactivate it. To do that, run:

```
deactivate
```

Combining virtual environments with centrally installed modules

You can combine Python packages installed in a virtual environment with environment modules. The following script uses PyTorch (which is available as a module) and Poutyne (which we assume is not centrally installed):

pytorch_poutyne.py

```
import torch
import poutyne

...
```

We load a PyTorch package as a module and install Poutyne in a virtual environment:

```
module load PyTorch/2.1.2-foss-2023a
python -m venv myenv
source myenv/bin/activate
pip install Poutyne
```

While the virtual environment is activated, we can run the script without any issues:

```
python pytorch_poutyne.py
```

Deactivate the virtual environment when you are done:

```
deactivate
```

Creating a virtual environment for a specific cluster

To create a virtual environment for a specific cluster, you need to start an interactive shell on that cluster. Let's say you want to create a virtual environment on the `donphan` cluster.

```
module swap cluster/donphan
qsub -I
```

After some time, a shell will be started on the `donphan` cluster. You can now create a virtual environment as described in the first section. This virtual environment can be used by jobs running on the `donphan` cluster.

Naming a virtual environment

When naming a virtual environment, it is recommended to include the name of the cluster it was created for. We can use the `$VSC_INSTITUTE_CLUSTER` variable to get the name of the current cluster.

```
python -m venv myenv_${VSC_INSTITUTE_CLUSTER}
```

Example Python job

This section will combine the concepts discussed in the previous sections to:

1. Create a virtual environment on a specific cluster.
2. Combine packages installed in the virtual environment with modules.
3. Submit a job script that uses the virtual environment.

The example script that we will run is the following:

pytorch_poutyne.py

```
import torch
import poutyne

print(f"The version of PyTorch is: {torch.__version__}")
print(f"The version of Poutyne is: {poutyne.__version__}")
```

First, we create a virtual environment on the `donphan` cluster:

```
module swap cluster/donphan
qsub -I
# Load module dependencies
module load PyTorch/2.1.2-foss-2023a
python -m venv myenv
source myenv/bin/activate
# install virtual environment dependencies
pip install Poutyne
deactivate
```

Type `exit` to exit the interactive shell. We now create a job script that loads the PyTorch module, enters the virtual environment and executes the script:

jobscript.pbs

```
#!/bin/bash

# Basic parameters
```

```
#PBS -N python_job_example            ## Job name
#PBS -l nodes=1:ppn=1                 ## 1 node, 1 processors per node
#PBS -l walltime=01:00:00             ## Max time your job will run (no more
than 72:00:00)

module load PyTorch/2.1.2-foss-2023a  # Load the PyTorch module
cd $PBS_O_WORKDIR                     # Change working directory to the
location where the job was submitted
source myenv/bin/activate             # Activate the virtual environment

python pytorch_poutyne.py             # Run your Python script, or any other
command within the virtual environment

deactivate                            # Deactivate the virtual environment
```

Next, we submit the job script:

```
qsub jobscript.pbs
```

Two files will be created in the directory where the job was submitted: `python_job_example.o123456` and `python_job_example.e{{ job_id }}`, where 123456 is the id of your job. The `.o` file contains the output of the job.

vsc-venv: Python Virtual Environment Wrapper Script

`vsc-venv` is a script that encapsulates the creation and management of Python virtual environments. This avoids multiple issues with the default `venv` included in Python (`python -m venv`).

For instance, a virtual environment created for one cluster might not work on another. Additionally, when activating a virtual environment, the same centrally installed modules used during its creation must also be loaded. The `vsc-venv` command manages multiple virtual environments for different clusters in a transparent way, while guaranteeing the same module environment.

Usage

A virtual environment can be activated by running the following command:

```
$ module load vsc-venv
$ source vsc-venv --activate --requirements requirements.txt [--modules
modules.txt]
```

Here, `requirements.txt` is the path to a file containing the Python dependencies to install in the virtual environment. For more information on the `requirements.txt` file, see the pip documentation. The optional `--modules` option can be used to provide a `modules.txt` file that lists the modules to load before activating the virtual environment.

Automatically, the modules are loaded and the environment is activated. When running this command for the first time, the dependencies from the requirements file are installed.

Now, the software can be run and Python packages installed in the virtual environment can be used, along with software provided via centrally installed modules.

You can get insights on the current environment using the following commands:

```
python --version    # Python version
pip list            # List of installed Python packages
module list         # List of loaded modules
```

To deactivate the virtual environment, run:

```
$ source vsc-venv --deactivate
```

Example

Suppose we are on the HPC-UGent Tier-2 login nodes, and we want to make a virtual environment for doduo, the default cluster. The following files are present in the current directory:

modules.txt:

```
SciPy-bundle/2023.11-gfbf-2023b
Pillow/10.2.0-GCCcore-13.2.0
```

and **requirements.txt:**

```
beautifulsoup4==4.12.3
```

For more info on the `requirements.txt` file, see the pip documentation.

We run the following commands to enter the environment

```
$ module load vsc-venv
$ source vsc-venv --activate --requirements requirements.txt --modules
modules.txt
```

As this creates the virtual environment for the first time, a `venvs` subdirectory is created in the current directory. Within `venvs/`, an additional subdirectory is created for the virtual environment: `venv-RHEL8-zen2`.

Now, Python 3.12 is loaded and the `numpy` (provided by the `SciPy-bundle` module), `PIL` (provided by the `Pillow` module), and `bs4` Python packages can be used.

To deactivate the virtual environment, run:

```
$ source vsc-venv --deactivate
```

Making a virtual environment for another cluster

If we want to create a virtual environment for another cluster, say donphan, we can use `vsc-venv` in an interactive job:

```
$ module swap cluster/donphan
$ qsub -I
$ cd my_project
$ module load vsc-venv
$ source vsc-venv --activate --requirements requirements.txt --modules
modules.txt
```

the venvs folder now contains two folders:

```
$ ls venvs/
venv-RHEL8-cascadelake
venv-RHEL8-zen2
```

Troubleshooting

Illegal instruction error

Activating a virtual environment created on a different cluster can cause issues. This happens because the binaries in the virtual environments from cluster A might not work with the CPU architecture of cluster B.

For example, if we create a virtual environment on the gallade cluster,

```
module swap cluster/gallade
qsub -I
$ python -m venv myenv
```

return to the login node by pressing CTRL+D and try to use the virtual environment:

```
$ source myenv/bin/activate
$ python
Illegal instruction (core dumped)
```

we are presented with the illegal instruction error. More info on this on the troubleshooting page

Error: GLIBC not found

When running a virtual environment across clusters with different major OS versions, you might encounter a variation of the following error:

```
python: /lib64/libc.so.6: version `GLIBC_2.34' not found (required by python)
```

Make sure you do not activate a virtual environment created on a different cluster. For more information on how to create a virtual environment for a specific cluster, see Creating a virtual environment for a specific cluster. When following these steps, make sure you do not have any modules loaded when starting the interactive job.

Error: cannot open shared object file: No such file or directory

There are two main reasons why this error could occur.

1. You have not loaded the `Python` module that was used to create the virtual environment.
2. You loaded or unloaded modules while the virtual environment was activated.

Entering a virtual environment while the Python module used to create it is not active

If you loaded a `Python` module when creating a virtual environment, you need to make sure that the same module is loaded when you enter the environment. This is because the virtual environment keeps a reference to the base python used to create it.

The following commands illustrate this issue:

```
$ module load Python/3.10.8-GCCcore-12.2.0  # Load a python module
$ python -m venv myenv                      # Create a virtual environment
with loaded python module
$ module purge                              # unload all modules (WRONG!)
$ source myenv/bin/activate                 # Activate the virtual environment
$ python                                    # Start python
python: error while loading shared libraries: libpython3.10.so.1.0: cannot
open shared object file: No such file or directory
```

Here, the virtual environment tries to use the python module that was loaded when the environment was created. Since we used `module purge`, that module is no longer available. The solution is to load the same python module before activating the virtual environment:

```
module load Python/3.10.8-GCCcore-12.2.0  # Load the same python module
source myenv/bin/activate                 # Activate the virtual environment
```

Modifying modules while in a virtual environment

You must not load or unload modules while in a virtual environment. Loading and unloading modules modifies the `$PATH` variable in the current shell. When activating a virtual environment, it will store the `$PATH` variable of the shell at that moment. If you modify the `$PATH` variable while in a virtual environment by loading or unloading modules, and deactivate

the virtual environment, the `$PATH` variable will be reset to the one stored in the virtual environment. Trying to use those modules will lead to errors:

```
$ module load Python/3.10.8-GCCcore-12.2.0  # Load a python module
$ python -m venv myenv                      # Create a virtual environment
$ source myenv/bin/activate                 # Activate the virtual environment
(saves state of $PATH)
$ module purge                              # Unload all modules (modifies the
$PATH)
$ deactivate                                # Deactivate the virtual
environment (resets $PATH to saved state)
$ python                                    # PATH contains a reference to the
unloaded module
python: error while loading shared libraries: libpython3.10.so.1.0: cannot
open shared object file: No such file or directory
```

The solution is to only modify modules when not in a virtual environment.