uv – Python package manager

This document describes uv usage on the LUIS cluster. For detailed instructions on uv features, including installation of deep learning stacks, please refer to the official uv documentation.

uv is a fast Python package manager and environment tool, serving as a modern alternative to `pip`, `virtualenv`, and partially `conda`. It allows users to create **isolated Python environments** and install packages **efficiently**.

To make uv available, load the module:

```
module load GCCcore/.14.2.0 uv/0.9.22
```

Check:

```
uv --version
```

To check other available uv module versions, run:

```
module -r spider ^uv
```

Creating a uv project

A **uv project** is a directory containing your Python code along with project metadata that uv uses to manage dependencies, scripts, and build information.

Creating a project does not automatically create a virtual environment, as it only sets up the project structure.

The simplest way to create a project is to let uv create the directory and initialize the project:

```
uv init ~/my-python-project
```

This will create the directory `~/my-python-project` and initialize the project inside it.

The `~/my-python-project` directory will contain:

```
~/my-python-project/
├── .git
├── .gitignore
├── .python-version
├── main.py
├── pyproject.toml
└── README.md
```

A uv project can also be initialized within an existing directory by navigating into it and running uv

`init`

Creating a virtual environment

To actually install and isolate Python packages, you need a Python runtime environment. The environment can be located within the project directory or anywhere else on the filesystem. We recommend general installations to be done under $SOFTWARE, but anything that's configured and for your personal use under $BIGWORK. $HOME will fill up fast, and it is quite unsuitable for HPC workloads, so you should store only small configuration files there. You s/could also check with colleagues whether you can share environments and installations, in particular when you are using $SOFTWARE.

Create a new Python environment in the directory `${SOFTWARE}/${USER}/myenv` (i.e. in `$HOME/myenv`):

```
uv venv ${SOFTWARE}/${USER}/myenv
```

The directory `myenv` is created if it does not exist, and must be empty if it already exists.

Note: Create virtual environments in your project's dedicated $SOFTWARE directory rather than in $HOME, as home directories have limited space. You s/could also use the following lines in your .bashrc to prevent uv and/or pip from clogging up your $HOME:

```
# Keep package/build caches from clogging my home
export UV_CACHE_DIR="${SOFTWARE}/${USER}/.cache/uv"
export PIP_CACHE_DIR="${SOFTWARE}/${USER}/.cache/pip"

# In case you use uv add / projects, it might be helpful to add
export UV_PROJECT_ENVIRONMENT="${SOFTWARE}/${USER}/myprojectenv"
```

Activate the virtual environment:

```
source ${SOFTWARE}/${USER}/myenv/bin/activate
```

Exit the currently active virtual environment:

```
deactivate
```

Installing packages

Install packages using `uv pip`:

```
uv pip install numpy scipy matplotlib
```

Install from a requirements file:

```
uv pip install -r requirements.txt
```

Note: On the LUIS cluster, you must first create and activate a virtual environment **before** installing Python packages with uv.

Selecting the Python version

As uv does not install or download Python but instead uses the Python interpreter currently available in the user environment (by default the one provided by the cluster operating system), any alternative Python version **must be loaded via Lmod** into the environment **before** creating a virtual environment.

List available Python modules:

```
module -r spider ^Python
```

The LUIS cluster uses a hierarchical module layout, where Python modules may require other modules (e.g. compiler) to be loaded first. To determine the required dependencies, query the specific version:

```
module spider Python/3.11.6
```

This command shows which parent modules must be loaded.

After loading the required modules, load Python:

```
module load Python/3.11.6
```

Note that the required parent modules and the Python module can be loaded in one command; the order matters (parent modules first):

```
module load <parent-module> Python/3.11.6
```

Verify:

```
python --version
```

Then create the environment:

```
uv venv ${SOFTWARE}/${USER}/myenv
```

Freezing environments

To save the current state of installed packages in a virtual environment, use:

```
source ${SOFTWARE}/${USER}/myenv/bin/activate
uv pip freeze > requirements.txt
```

This generates a list of all packages and their versions in *requirements.txt* format, which can be used to recreate the environment elsewhere:

```
uv venv ${SOFTWARE}/${USER}/newenv
source ${SOFTWARE}/${USER}/newenv/bin/activate
uv pip install -r requirements.txt
```

To see which packages are currently installed in the environment in a tabular format, use:

```
uv pip list
```

For a full list of available `pip` commands within `uv`, type:

```
uv pip --help
```

From:
https://docs.cluster.uni-hannover.de/ - **Cluster Docs**

Permanent link:
https://docs.cluster.uni-hannover.de/doku.php/guide/soft/uv

Last update: **2026/10/19 11:20**