Python Environments & Container

M. Werner

DEFINITION ENVIRONMENT

- **Python environment**
  - Python binaries and compiled libraries
  - Python packages
- **System environment**
  - operating system (Windows, Linux, ...)
  - drivers for devices (GPUs ...) or software (databases, compiler, ...)
  - hardware (laptop, HPC cluster, ...)

Relevant for:

- Installing Python packages
- Developing Python packages (or certain applications)

INSTALLING PYTHON PACKAGES

A Python package has environment requirements to work properly.

- requires certain **Python binary** version (e.g. Python 3.11 vs. 3.12)
- depends on **other Python packages** (e.g. `numpy<=1.26.4`)
- *recommends* a certain Python package

  (e.g. `onnxruntime-gpu` for improved performance)
- requires certain **system libraries and drivers** (e.g. gcc 14.2, ...)

Note: package authors do not always define package requirements properly

INSTALLING PYTHON PACKAGES

POSSIBLE OUTCOMES

- ✅ Package successfully installed without touching other packages
- ❌ Installation failed, package not installed
- ❌ Package installed along with its dependencies
  - polluting Python environment
  - other Python packages do not work anymore
  - because complete dependency matrix has not been checked
    (may taking ages though)
  - rollback tedious

DEVELOPING PYTHON PACKAGES / APPS

- Testing different Python environments to maximise compatibility
  - requirements like `numpy==1.26.4` vs. `numpy<=1.26.4` vs. `numpy`
- Testing different system environments
- Pinning down errors with certain 3rd party package versions

DEVELOPING PYTHON PACKAGES / APPS

- Testing different Python environments to maximise compatibility
  - requirements like `numpy==1.26.4` vs. `numpy<=1.26.4` vs. `numpy`
- Testing different system environments
- Pinning down errors with certain 3rd party package versions

... requires:

- flexible management of **comparable + reproducible environments**
- easily testing **other platforms**
- opt-in for **automation** processes
  (e.g. github continuous integration for automated tests and builds)
- ideally control the complete environment to compare a change of a single component (**performance regression testing, ...**)

TALK OUTLINE

- Which Python environment **tools** exist?
  - system-wide and **virtual environments**
  - container
- How to **prevent or fix** a messed up Python environment?
- Working with **containers**

WARNING: OFFENDING MATERIALS AHEAD

Source

WARNING: OFFENDING MATERIALS AHEAD

Source

PYTHON ENVIRONMENT TOOLS

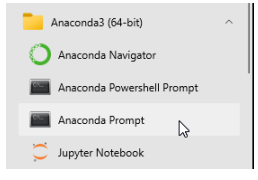
- **Pip**: default package installer, uses Python Package Index (PyPI), essential for managing dependencies
- **Conda**: A package & environment management system, handles non-Python dependencies, creates isolated environments
- **Pipenv**: Combines pip and virtualenv, simplifies dependency management, scans for security vulnerabilities in dependencies
- **Poetry**: dependency management and packaging focus, uses a `pyproject.toml`, automatically creates virtual environments for projects
- **Virtualenv**: older tool for creating Python environments (inferior to pipenv)
- **Venv**: built-in module in Python 3.3+, creates lightweight virtual environments, less features than Virtualenv
- **Pyenv**: manages multiple Python versions, but not environments directly (no Windows supported)
- **pyvenv**: deprecated
- **Mamba**: fast alternative to Conda, may speed up environment resolution / package installation, compatible with Conda packages
- **Micromamba**: like Mamba without overhead of full Conda installation
- **Docker**: [not specific to Python] creates containerized environments, encapsulates applications along with their system dependencies

PYTHON ENVIRONMENT TOOLS

- **Pip**: default package installer, uses Python Package Index (PyPI), essential for managing dependencies
- **Conda**: A package & environment management system, handles non-Python dependencies, creates isolated environments
- **Pipenv**: Combines pip and virtualenv, simplifies dependency management, scans for security vulnerabilities in dependencies
- **Poetry**: dependency management and packaging focus, uses a `pyproject.toml`, automatically creates virtual environments for projects
- **Virtualenv**: older tool for creating Python environments (inferior to pipenv)
- **Venv**: built-in module in Python 3.3+, creates lightweight virtual environments, less features than Virtualenv
- **Pyenv**: manages multiple Python versions, but not environments directly (no Windows supported)
- **pyvenv**: deprecated
- **Mamba**: fast alternative to Conda, may speed up environment resolution / package installation, compatible with Conda packages
- **Micromamba**: like Mamba without overhead of full Conda installation
- **Docker**: [not specific to Python] creates containerized environments, encapsulates applications along with their system dependencies

ANACONDA

- comes with 450+ packages pre-installed, stored in:
  - `C:\Users\<your-username>\Anaconda3\pkgs\`
  - anaconda repository itself contains couple of thousands packages
- Windows does not know where Python is (see FAQ)
  - Anaconda activates its environment at launch
- Anaconda (`conda`) has its own package repository
- Anaconda's defaults channel: more stable and secure than community-run channels like conda-forge
  - may contain older package versions than publicly available
- updating Anaconda from within can become a challenge (just reinstall?)
- `pip` is only a package manager: much larger collection of Python packages (PyPI)



(ANA)CONDA

Based on a stackoverflow post:

- conda = Python package + command line tool
- Miniconda installer = Python + conda
- Anaconda installer = Python + conda + *meta package anaconda*
- meta Python package anaconda = 500+ dependencies and packages
- Anaconda installer = Miniconda installer + conda install anaconda

CONDA AND PIP

Search packages

- `conda search package-name --info` shows requirements of a package
- `pip index versions package-name` is experimental
  - ~~`pip search ...`~~ (deactivated), use PyPI website or `pip_search` or `pypisearch` package
- `conda list package-name` shows installed package version
- `pip show package-name` shows installed package version

CONDA AND PIP

Search packages

- `conda search package-name --info` shows requirements of a package
- `pip index versions package-name` is experimental
  - ~~`pip search ...`~~ (deactivated), use PyPI website or `pip_search` or `pypisearch` package
- `conda list package-name` shows installed package version
- `pip show package-name` shows installed package version

Check environment

- `conda check`
- `pip check`
- `pip freeze --user` (use `pipreqs` for creating project requirements.txt)

CONDA AND PIP

Installing Packages:

conda	pip	effect
`conda install package-name`	`pip install package-name`	might change environment
`… --dry-run`	`… --dry-run` (pip 22.2+)	see what would happen
`… --no-update-deps`	`… --no-deps`	only package is installed, may not work properly
`… --freeze-installed`	--	does not change existing packages

Note: During install `conda` automatically checks for conflicts which can take quite a while ("Solving environment …"). It reports the changes and asks you to continue the installation.

docs: conda install | pip install

CONDA AND PIP

Installing Packages:

conda	pip	effect
`conda install package-name`	`pip install package-name`	might change environment
`… --dry-run`	`… --dry-run` (pip 22.2+)	see what would happen
`… --no-update-deps`	`… --no-deps`	only package is installed, may not work properly
`… --freeze-installed`	--	does not change existing packages

Note: During install `conda` automatically checks for conflicts which can take quite a while ("Solving environment ..."). It reports the changes and asks you to continue the installation.

docs: conda install | pip install

EXTRA NOTE

`pip` can directly install a package from version control systems like git:

```
python -m pip install git+https://github.com/pypa/sampleproject.git@main
```

CONDA AND PIP

Uninstall / Rollback

- `conda uninstall package-name --dry-run`
  - also removes packages that depend on it
  - problematic, if a global package is removed where virtual environments relied on it
- `pip uninstall package-name` (no dry-run option)
  - only removes package (use `pipdeptree` package to investigate)
- `conda install --revision NUMBER`
  - restores environment, see guide
- no pip equivalent for revisions, but can be done with:
  - `pip freeze > requirements.txt`
  - and `pip install -r requirements.txt` as rollback
- `conda list --revisions`, `conda clean -i`, `conda info`, `conda config --show`
  - might help with issues, also see conda cheatsheet
- `pip` pendants: `pip cache info`, `pip cache purge`, `python -m site`

CONDA AND PIP

Uninstall / Rollback

- `conda uninstall package-name --dry-run`
  - also removes packages that depend on it
  - problematic, if a global package is removed where virtual environments relied on it
- `pip uninstall package-name` (no dry-run option)
  - only removes package (use `pipdeptree` package to investigate)
- `conda install --revision NUMBER`
  - restores environment, see guide
- no pip equivalent for revisions, but can be done with:
  - `pip freeze > requirements.txt`
  - and `pip install -r requirements.txt` as rollback
- `conda list --revisions`, `conda clean -i`, `conda info`, `conda config --show`
  - might help with issues, also see conda cheatsheet
- `pip` pendants: `pip cache info`, `pip cache purge`, `python -m site`

Manually re-installing certain packages if you know the working versions:

- `conda install numpy==1.26.4` or `pip install -U numpy==1.26.4`

EXTRA NOTES

MIXING ENVIRONMENT TOOLS

Do not switch between `pip` and `conda` back and forth. When such conflicts occur, just delete the environment and recreate:

> *using pip only after all other requirements have been installed via conda is the safest practice. Additionally, pip should be run with the "--upgrade-strategy only-if-needed" [default]*

OTHER PYTHON VERSION REQUIRED

You can use pyenv (non-Windows) or pyenv-win (Windows), but it may interfer with a global installation.

DEPENDENCY HELL

Sometimes the requirements for a project are too tight or too loose, experiment with this first. But if there is still no conflict-free combination of Python packages, maybe an alternative package exist. Ask Awesome Python or perplexity.ai (AI chat).

DEPLOYMENT

If a project has complex dependencies, pip wheel helps to reduce time-consuming compilation by generating and packaging all project's dependencies (such 'wheelhouse' is not platform-portable).

VIRTUAL ENVIRONMENTS

Source

VIRTUAL ENVIRONMENTS

a.k.a. keep your specific packages in a subfolder:

```
# some python project
├── ...
├── .venv     # some name for your virtual env.
│   ├── bin  # Python binaries e.g. python3.11
│   ├── lib  # Python packages e.g. matplotlib
│   └── pyvenv.cfg  # paths to Python binaries, etc.
├── ...
```

CONDA

- open (Anaconda) prompt (see also conda environment files)

```
cd your-project-folder
# optionally specify Python version or packages
conda create --name .venv python=3.9 scipy=0.17.3 babel
# activate environment
conda activate .venv
# ...
conda deactivate # if needed
```

CONDA

- open (Anaconda) prompt (see also conda environment files)

```
cd your-project-folder
# optionally specify Python version or packages
conda create --name .venv python=3.9 scipy=0.17.3 babel
# activate environment
conda activate .venv
# ...
conda deactivate # if needed
```

VENV

- see also how venvs works

```
python -m venv .venv  # .venv: folder name, as you like
source .venv/bin/activate  # Windows: .venv\Scripts\activate.bat
# ...
deactivate  # if needed
```

VIRTUAL ENVIRONMENTS WORKFLOW

Manually managing Python environments (IDE still might be able to work with it):

- *only once*: **create** virtual environment folder in your project folder
- **activate** virtual environment
- **run** your Python programs in there
  (`jupyter lab`, `pip install package-name`, ...)
- *optional*: write a script to **automate** environment activation and programs
- *optional*: Things gone wrong? Remove the environment folder and rebuild it

VIRTUAL ENVIRONMENTS WORKFLOW

Manually managing Python environments (IDE still might be able to work with it):

- *only once*: **create** virtual environment folder in your project folder
- **activate** virtual environment
- **run** your Python programs in there
  (`jupyter lab`, `pip install package-name`, ...)
- *optional*: write a script to **automate** environment activation and programs
- *optional*: Things gone wrong? Remove the environment folder and rebuild it

IDE & VIRTUAL ENVIRONMENTS

- Visual Studio Code
  - pip, venv, conda[miniconda], and more via extensions (poetry,...)
- pycharm
  - pip, venv, conda, pipenv, poetry, ...
- ...

CONCLUSION

Most likely you want Python environments, if at least one of your Python projects has diverging environment requirements, which interfer with your system-wide installed Python environment.

You might want to create a Python environment manually,

- if you cannot use an IDE (e.g. on an HPC system),
- if you need other environment tools like poetry or pyenv,
- if many projects share the same environment,
- ...

CONTAINER

Source

CONTAINERIZED PYTHON ENVIRONMENTS

CONTAINER IN GENERAL

- **isolate** applications and their dependencies (including OS, libs and tools)
- **run consistently** across different platforms
  - more portable than virtual environments
- *container*: runtime instance of an *image*
- *image*: union of filesystem layers
- separates system and user data (bind-mounts, volumes)
- **but**: more setup costs
  - requires a container build recipe
  - some disk space required
  - "housekeeping" is another topic **cough**

CONTAINER IN GENERAL: SCOPE

- virtual environments:
  - **simple** project requirements
  - **local-only** Python projects
  - **rapid prototyping**
  - IDE support
  - debugging
- containerization:
  - multiple **environment** or **deployment**[1] requirements
  - platform **portability**
  - **reproducibility** (performance, testing, debugging)
  - **automate** testing and deployment
  - Software-as-a-Service (see building guidelines)
  - system re-install/reset becomes trivial
  - ...

[1]) *deployment*: release project for customers, for HPC, ..., which includes:

- software release, installation, testing, performance monitoring, ...

DOCKER

- Efficient resource utilization (compared to VMs)
- Large ecosystem and community support
- Docker Hub offers a vast repository of pre-built images
- HPC?
  - Can introduce performance overhead in HPC workloads
  - May require additional configuration for high-performance networking
  - Limited support for specialized HPC hardware like InfiniBand

DOCKER

- Efficient resource utilization (compared to VMs)
- Large ecosystem and community support
- Docker Hub offers a vast repository of pre-built images
- HPC?
  - Can introduce performance overhead in HPC workloads
  - May require additional configuration for high-performance networking
  - Limited support for specialized HPC hardware like InfiniBand

SINGULARITY/APPTAINER

- Designed specifically for HPC and scientific computing
- Can convert Docker images to Singularity format
- Native support for MPI and GPU acceleration
- Better security model for multi-user HPC systems
- Minimal performance overhead compared to bare-metal
- apptainer and singularityCE mostly compatible (apptainer forked and maintained by Linux Foundation)

CONTAINERIZED PYTHON ENVIRONMENTS

SIMPLE DOCKER CONTAINER RECIPE

```
# ./Dockerfile
# Official Python image from the Docker Hub
# - is a Debian OS with minimal packages
FROM python:3.11.9-slim
# Set the working directory in the container
WORKDIR /app
# Copy the requirements file into the container
COPY requirements.txt .
# Install the required Python packages
RUN pip install --no-cache-dir -r requirements.txt
# Copy the rest of the application code into the container
COPY . .
# Specify the command to run the application
CMD ["python", "my-app.py"]
```

CONTAINERIZED PYTHON ENVIRONMENTS

SIMPLE DOCKER CONTAINER RECIPE

```
# ./Dockerfile
# Official Python image from the Docker Hub
# - is a Debian OS with minimal packages
FROM python:3.11.9-slim
# Set the working directory in the container
WORKDIR /app
# Copy the requirements file into the container
COPY requirements.txt .
# Install the required Python packages
RUN pip install --no-cache-dir -r requirements.txt
# Copy the rest of the application code into the container
COPY . .
# Specify the command to run the application
CMD ["python", "my-app.py"]
```

- build and run:

```
# build docker image
docker build -t my-python-app .
# run it
docker run --name my-running-app my-python-app
```

MORE CONTAINER FEATURES

- bind internal paths to user folders for dynamic content or results
  - e.g. binding an internal `/output` to `./my-results`
- you can run interactively commands like `bash` (if image has it)
- uses caches and build layers, versioning and image tagging
- container can communicate within own segmentable networks
- docker images can be converted to singularity (e.g. for HPC systems)
- ...

PREPARED CONTAINER IMAGE

Our prepared docker image `uncertainty-lab` currently contains:

- R 4.3.3, Python 3.11.9
- torch 2.4.0+cpu, tensorflow 2.17.0, Jupyter, Keras, scikit, pymc, ...
- LaTeX, octave, gnuplot, gcc, ...
- jupyter can run: Python, R and octave
- can be converted to singularity for convenient HPC deployment

DOCKER COMPOSE

- `docker compose` simplifies (multiple) container orchestration
- simple commands like `docker-compose up` and `docker-compose down` handle complex setups
- `docker-compose.yaml` readable configuration file
  - define environments, paths, ports, networks, ...
  - YAML is a human-friendly data language (more than JSON or XML)

DOCKER COMPOSE UNCERTAINTY-LAB

- configuration file (./docker-compose.yaml)

```yaml
services:
  lab: # name of service (configuration)
    # name of container (runs an dynamic instance of an image)
    container_name: uncertainty-lab
    # name of image (blueprint for container)
    # - already prepared an image for you
    image: user2084/uncertainty-lab:latest
    # builds an image from local 'Dockerfile' instead
    # build: .  # uncomment
    ports:
      - "8888:8888"  # routes public port :to: internal port
    volumes:
      - /path-to-my-projects:/home/jovyan/work
    environment:
      - JUPYTER_ENABLE_LAB=yes
    # jupyter token/password can be disabled (unsafe)
    # command: start-notebook.py --NotebookApp.token='' --NotebookApp.password=''
```

PREPARED CONTAINER IMAGE

- run container

```
docker compose -f docker-compose.yml up
# if local ./Dockerfile should be used to build
# docker compose -f docker-compose.yml up --build
```

- now the jupyter server is running inside the container
  - access it via the link given in the console output
  - or: http://0.0.0.0:8888/lab if token / password are disabled

PREPARED CONTAINER IMAGE

- run container

```
docker compose -f docker-compose.yml up
# if local ./Dockerfile should be used to build
# docker compose -f docker-compose.yml up --build
```

- now the jupyter server is running inside the container
  - access it via the link given in the console output
  - or: http://0.0.0.0:8888/lab if token / password are disabled

INSTALLATION

- Windows
  - Rancher Desktop: Download (free, open-source, more versatile)
    - might require WSL2 (Windows Subsystem for Linux)
  - or: Docker Desktop: Download (proprietary for enterprises)
  - or: WSL2 and Docker daemon (without Docker Desktop)
  - or: Podman
- Linux
  - install docker and docker-compose from the repo

MORE ON DOCKER

Slides Docker Workshop by Felix Eckhofer:

https://extern.tribut.de/dw.html#/container

SINGULARITY WORKFLOW

Get or build your images (on the cluster):

- pull image from singularity hub:

```
singularity pull --name hello-world.sif shub://vsoch/hello-world
singularity run hello-world.sif
# or
./hello-world.sif
```

SINGULARITY WORKFLOW

Get or build your images (on the cluster):

- pull image from singularity hub:

```
singularity pull --name hello-world.sif shub://vsoch/hello-world
singularity run hello-world.sif
# or
./hello-world.sif
```

- build singularity image from docker hub

```
singularity pull --name lolcow.sif docker://godlovedc/lolcow
singularity run lolcow.sif
# or
./lolcow.sif
```

ALTERNATIVE: BUILD IMAGE LOCALLY

- building own images may require root privileges and will fail on remote systems
- (Linux) install singularity CE (Linux-only, `yay -S singularity-ce`, ...)
  - but you also can run singularity via a docker image:

```
# converts a docker image to a singularity file
docker run --volume $PWD:/go --privileged -t --rm \
  quay.io/singularity/singularity:v4.1.0 build \
  uncertainty-lab.sif docker://user2084/uncertainty-lab
```

- move image to cluster: `scp uncertainty-lab.sif <USER>@mlogin01.hrz.tu-freiberg.de`

LIVE DEMO

- `singularity inspect`
- `singularity shell uncertainty-lab`
  - `python --version`
  - `jupyter lab`
  - (forwarding from TUBAF cluster to client is blocked, but maybe jupyter hub will come?)
- `singularity run --bind ./testfolder:/home/jovyan/work uncertainty-lab`
- `singularity exec --bind ./testfolder:/home/jovyan/work uncertainty-lab.sif python --version`

Links:

- TUBAF HPC Job Submission
- singularity-on-the-cluster

Thank You! && Any Questions?