The road to reproduvibility in Python

An introduction to the uv tool

Gaétan Lepage

November 15, 2025

GRICAD, Grenoble

Who am I?

Gaétan Lepage

- Ensimag 2020
- PhD @Inria Grenoble (RobotLearn team)
  - Deep Learning for robotic acoustics
- Nix(OS) contributor since 2021
  - Python ecosystem maintenance
  - Member of the CUDA team

Objectives

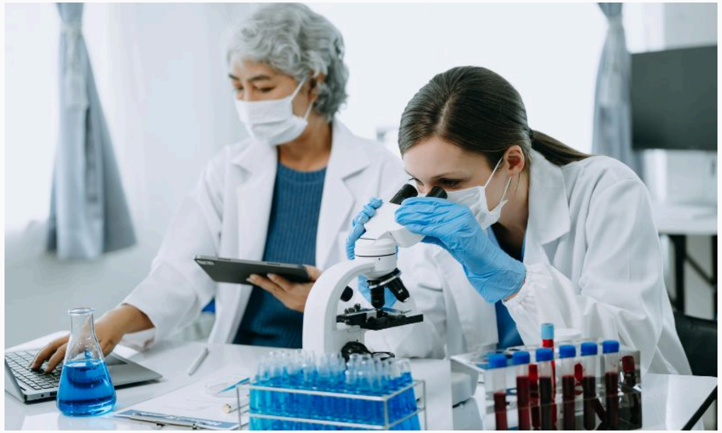
Objectives:

- Why “packaging” is important, especially in research
- Overview of how things work in Python
- A presentation of `uv`
- Hands-on! **Try `uv` on your own project**

Contents

Contents

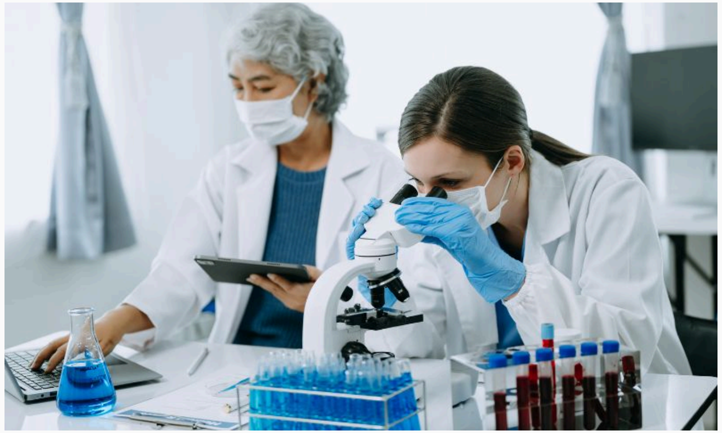
Contents



Definitions: [1]

- **Repeatability:** Same team, same experimental setup
- **Reproducibility:** Different teams, same experimental setup
- **Replicability:** Different teams, different experimental setups

Contents



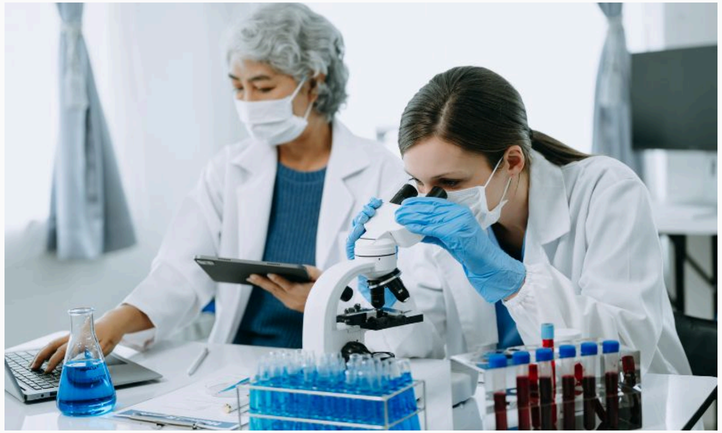
Definitions: [1]

- **Repeatability:** Same team, same experimental setup
- **Reproducibility:** Different teams, same experimental setup
- **Replicability:** Different teams, different experimental setups

Reprocubility crisis:

Difficulty to reproduce scientific studies from other groups

Contents



Definitions: [1]

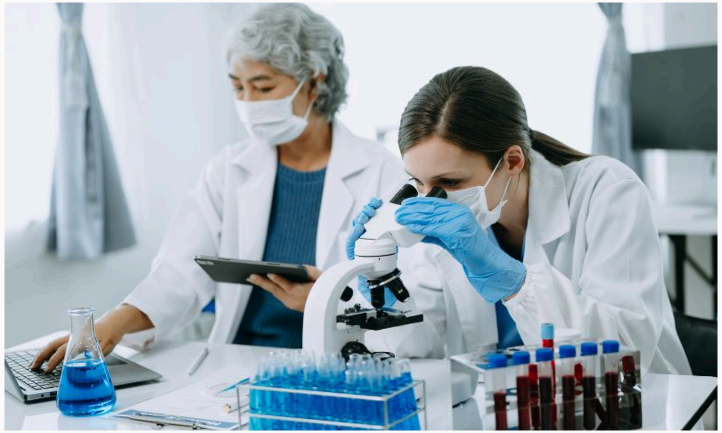
- **Repeatability:** Same team, same experimental setup
- **Reproducibility:** Different teams, same experimental setup
- **Replicability:** Different teams, different experimental setups

Reprocubility crisis:

Difficulty to reproduce scientific studies from other groups

Only in experimental sciences... right?

Contents



Definitions: [1]

- **Repeatability:** Same team, same experimental setup
- **Reproducibility:** Different teams, same experimental setup
- **Replicability:** Different teams, different experimental setups

Reprocubility crisis:

Difficulty to reproduce scientific studies from other groups

Only in experimental sciences... right?

-> **Major issue in modern computer science research**

Software

Reasons for the lack of reproducibility:

Software

Reasons for the lack of reproducibility:

- Code is not available

Software

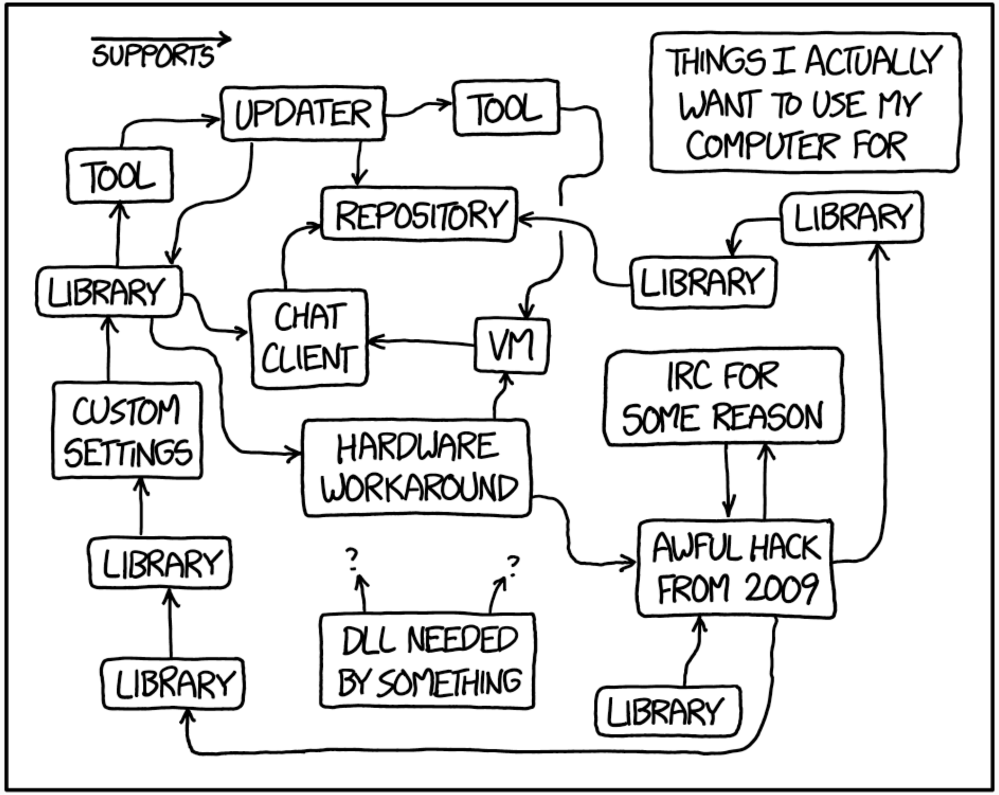
Reasons for the lack of reproducibility:

- Code is not available
- Data is not available

Software

Reasons for the lack of reproducibility:

- Code is not available
- Data is not available
- **Code is hard to run**



EVERY NOW AND THEN I REALIZE I'M MAINTAINING A HUGE CHAIN OF TECHNOLOGY SOLELY TO SUPPORT ITSELF.

General recommendations

- Most importantly: keep things simple!
  -> Fewer dependencies/languages/constraints = fewer problems

General recommendations

- Most importantly: keep things simple!
  -> Fewer dependencies/languages/constraints = fewer problems
- Use version control (`git`)

General recommendations

- Most importantly: keep things simple!
  -> Fewer dependencies/languages/constraints = fewer problems
- Use version control (`git`)
- Add a license (`LICENSE`) (MIT, GPL, Apache…)

General recommendations

- Most importantly: keep things simple!

  -> Fewer dependencies/languages/constraints = fewer problems
- Use version control (`git`)
- Add a license (`LICENSE`) (MIT, GPL, Apache…)
- Write a `README.md` file:
  - Context about the project
  - Installation instructions
  - How to run the code?
  - How to download data?
  - How to replicate the results?

General recommendations

- Most importantly: keep things simple!
  -> Fewer dependencies/languages/constraints = fewer problems
- Use version control (`git`)
- Add a license (`LICENSE`) (MIT, GPL, Apache…)
- Write a `README.md` file:
  - Context about the project
  - Installation instructions
  - How to run the code?
  - How to download data?
  - How to replicate the results?
- Ensure your code and experiments can be easily run and are reproducible

Existing approaches

- **Nothing**

  -> Fine for very simple software stacks. Doesn’t scale

Existing approaches

- **Nothing**

  -> Fine for very simple software stacks. Doesn’t scale
- **Natural language** (i.e. instructions)

  -> Documentation is great, but does not scale with complexity

Existing approaches

- **Nothing**

  -> Fine for very simple software stacks. Doesn't scale
- **Natural language** (i.e. instructions)

  -> Documentation is great, but does not scale with complexity
- **Virtual environments** (for Python)

  -> How to fill it? (not often reproducible)

  -> no control on the system's environment

Existing approaches

- **Nothing**

  -> Fine for very simple software stacks. Doesn't scale
- **Natural language** (i.e. instructions)

  -> Documentation is great, but does not scale with complexity
- **Virtual environments** (for Python)

  -> How to fill it? (not often reproducible)

  -> no control on the system's environment
- **What about containers?**

  -> How to fill it? (not often reproducible)

  -> Not resource-efficient

Existing approaches

- **Nothing**

  -> Fine for very simple software stacks. Doesn't scale
- **Natural language** (i.e. instructions)

  -> Documentation is great, but does not scale with complexity
- **Virtual environments** (for Python)

  -> How to fill it? (not often reproducible)

  -> no control on the system's environment
- **What about containers?**

  -> How to fill it? (not often reproducible)

  -> Not resource-efficient
- **Functional package managers (Nix, Guix)**

  -> Elegant, powerful, but very hard to use

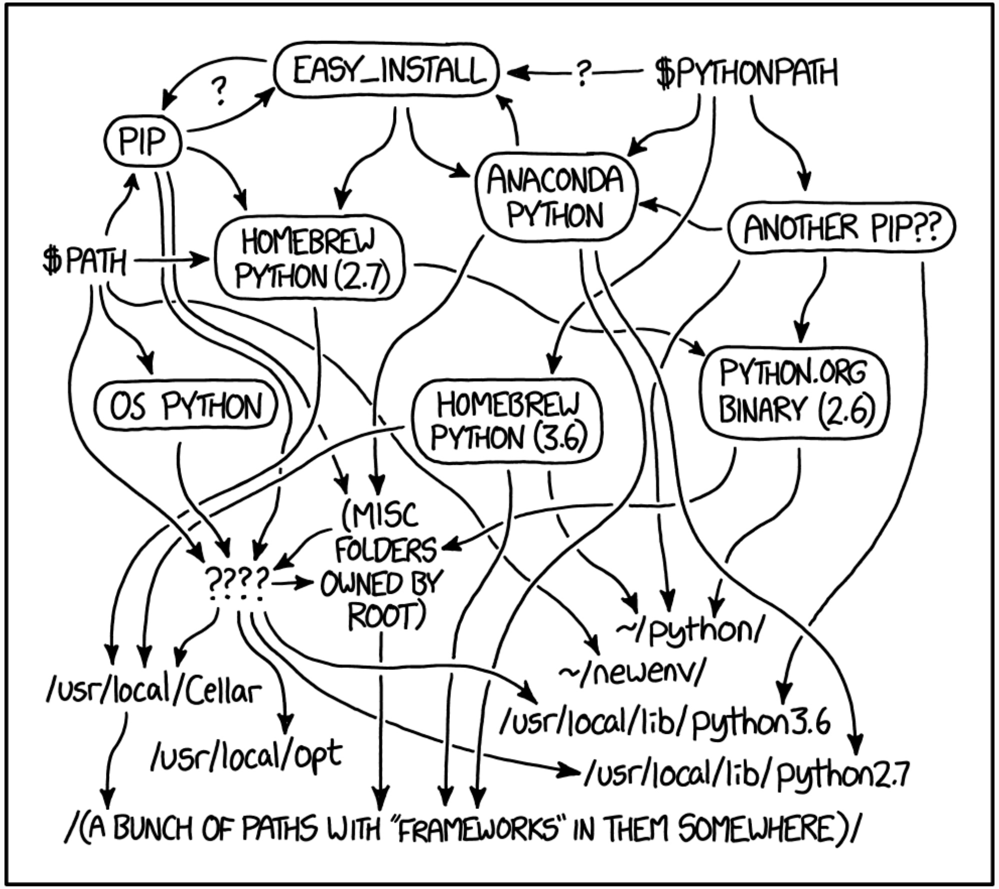
Contents

Contents

Contents

- Python projects often depend on many libraries
- Many tools (`setuptools`, `pip`, `conda`, `poetry`)
- Many standards (`setup.py`, `pyproject.toml`, `requirements.txt`, `conda-env.yml`)

-> Often laborious to "deploy" a project



MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED THAT MY LAPTOP HAS BEEN DECLARED A SUPERFUND SITE.

Main components

Main components:

- The `build-system`: Runs when you invoke `pip install`
  Ex: `setuptools`, `poetry`, `hatchling`, `uv-build`, …

Main components

Main components:

- The `build-system`: Runs when you invoke `pip install`
  Ex: `setuptools`, `poetry`, `hatchling`, `uv-build`, …
- The dependencies specification Ex: `pyproject.toml`'s `dependencies` list (recommended), `requirements.txt`, `setup.py`'s `install_requires` (`setuptools` only)

Main components

Main components:

- The `build-system`: Runs when you invoke `pip install`
  Ex: `setuptools`, `poetry`, `hatchling`, `uv-build`, …
- The dependencies specification Ex: `pyproject.toml`'s `dependencies` list (recommended), `requirements.txt`, `setup.py`'s `install_requires` (`setuptools` only)
- Optionally, a project/environment management tool:
  Ex: `uv`, `poetry`, `conda`, …

Main components

Main components:

- The `build-system`: Runs when you invoke `pip install`
  Ex: `setuptools`, `poetry`, `hatchling`, `uv-build`, …
- The dependencies specification Ex: `pyproject.toml`'s `dependencies` list (recommended), `requirements.txt`, `setup.py`'s `install_requires` (`setuptools` only)
- Optionally, a project/environment management tool:
  Ex: `uv`, `poetry`, `conda`, …
- Packaging repository **Pypi**:
  - Source distributions (`sdist`)
  - Binary builds (`wheels`)

Dependency specification

Formalized in PEP-508.

Examples:

- `numpy`
- `torch==2.9.1`
- `pandas>=2.0,<2.4`
- `ray[data]`
- `requests [security,tests] >2.8.1,=2.8.* ; python_version < "2.7"`

Explanation:

- Dependency name: `requests`
- Optional features: `[security,tests]`
- Version constraints: `>2.8.1,=2.8.*`
- Platform compatibility algebra: `python_version < "2.7"`

From uv2nix talk by @adisbladis [2]

requirements.txt

```
requests
colorama; platform_system == "Windows"
importlib; python_version
numpy
torch>=2.8.0
tqdm
git+ssh://git@github.com/echweb/echweb-utils.git
git+https://github.com/DavidDiazGuerra/gpuRIR
```

- Came from Pip
- List of PEP-508 strings
- Usually used with `pip install -r requirements.txt`
- Often alongside `setup.py`

From uv2nix talk by @adisbladis [2]

setup.py

```python
from distutils.core import setup

setup(
  name="my_project",
  version="0.1.0",
  description="My great project",
  long_description=open('README.md').read(),
  install_requires=[
    "numpy",
    "torch>=2.8.0"
  ],
  author="Gaétan Lepage",
  author_email="gaetan@glepage.com",
  url="https://my-project.sh",
  license="MIT",
)
```

- Originated with `distutils`/`setuptools`
- Most popular `build-system`
- Project metadata as Python code
- Build with `python setup.py build`
- Develop with `pip install -e`
- Not a standard
- Can be used for complex building (e.g. native code compilation)

From uv2nix talk by @adisbladis [2]

pyproject.toml

```toml
name = "my_project"
version = "0.1.0"
description = "My great project"
readme = "README.md"
license = "MIT"

requires-python = ">=3.9,<3.14"
dependencies = [
    "numpy",
    "torch>=2.8.0"
]

[build-system]
requires = ["setuptools", "setuptools-
scm"]
build-backend =
"setuptools.build_meta"
```

- Standard way of specifying the project metadata
- Specification: PEP-517 and PEP-621
- Contains the list of dependencies (no more `requirements.txt`)

From uv2nix talk by @adisbladis [2]

Contents

Contents

What is it?

What is uv?

- A modern tool to manage a python development project:
  - Specify dependencies…
  - …install them…
  - reproducibly!

Developped by Astral, a company building open source python tooling.

What is it?

Installation

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

Resources

- *Documentation:* https://docs.astral.sh/uv
- *GitHub repo:* https://github.com/astral-sh/uv
- *Python packaging documentation:* https://packaging.python.org

What is it?

- **Pros:**
  - Modern (August 2024)
  - Good ergonomics, intuitive UI
  - Fast (written in Rust)
  - Respects Python standards (e.g. PEP 621)
  - User-local installation (no need for `sudo`)
- ***Cons***:
  - Limited to Python (v.s. Pixi [3] or Nix/Guix)

How does it work?

- Creates and edits the `pyproject.toml` file
- Downloads and manages its own Python interpreters
- Creates and modifies the virtual environment
  - Computes the versions of all dependencies (SAT solver)
  - Transitive dependencies are pinned too!
  - Save the result in the `uv.lock` file
  - Installs all dependencies in the virtual environment

Dependency locking: the key to *true* reproducibility

- All dependency versions are saved to the `uv.lock` lockfile
- Ensures the environment can be reproduced later (same versions)
- Generate/update the lock file: `uv lock`
- Update dependencies: `uv lock --upgrade`

-> `uv.lock` should be tracked by `git`

How does it compare to other tools

- **Barebone `pip` (+ `venv`)**
  - Python based
  - Manages environments
  - “Slow”
  - No locking
- **Poetry, pdm et al.**
  - Python based
  - “Slow”
  - Consistent locking!
- **Conda**
  - Slow
  - Installs (some) *system libraries*
  - No locking

From uv2nix talk by @adisbladis [2]

Installation

On Linux and MacOS

```
$ curl -LsSf https://astral.sh/uv/install.sh | sh
```

On Windows

```
powershell -ExecutionPolicy ByPass -c "irm https://astral.sh/uv/install.ps1 | iex"
```

No `sudo` privileges required

Test it!

```
$ uv --version
```

Project initialization (uv init)

-> Generates important files:

- `pyproject.toml` (if necessary)
- lockfile: `uv.lock`
- `.git` and `.gitignore`
- `main.py` (Hello-world example)

```
# Existing projects
$ uv init .

# New project
$ uv init new_project

$ tree -a -L1
├── .git
├── .gitignore
├── main.py
├── pyproject.toml
├── .python-version
└── README.md
```

```
# Create the venv and install python
$ uv sync

# Two new files
$ tree -a -L1
...
├── uv.lock
└── .venv
```

Note: To specify a python version:

```
$ uv init --python 3.12
```

Adding dependencies

Most important command: `uv add`. It is the best way to *add* a dependency to the project

What it does:

- Adds the dependency specification (`numpy>=2.0,<2.4.0`) to `dependencies` in `pyproject.toml`
- Creates the virtual environment (`.venv`) if necessary
- Solves the environment (choose the version to install for each dependency)
- Installs the required dependencies (including transitive ones) to the virtual environment
- Updates `uv.lock`

You can provide arbitrary dependency specifications:

```
# Add a single dependency
$ uv add "requests[security,tests]>2.8.1"

# Add all dependencies from `requirements.txt`
$ uv add -r requirements.txt

# Remove a dependency
$ uv remove requests
```

Running the code

Two solutions to run the code:

1. `uv run main.py`
   - Automatically and transparently activates the virtual environment on the fly
2. Manually activate the virtual environment

   ```
   $ source .venv/bin/activate

   $ python main.py
   ```

   Useful to quickly run scripts or the REPL

Dependency groups and optional dependencies

- `optional-dependencies`: for extra features
  - Adding an optional dependency

```
$ uv add matplotlib --optional plot
```

  - In `pyproject.toml`

```
[project.optional-dependencies]
plot = [
  "matplotlib>=3.6.3"
]
excel = [
  "odfpy",
  "xlsxwriter>=3.0.5"
]
```

  - Installing (in another project):

```
$ uv add my_project[plot]
$ pip install my_project[plot]
```

- `dependency-groups`: for development dependencies
  - Adding a development dependency:

```
$ uv add --group test pytest
```

  - In `pyproject.toml`

```
[dependency-groups]
dev = [
  "pytest"
]
lint = [
  "ruff"
]
```

https://pydevtools.com/handbook/explanation/what-are-optional-dependencies-and-dependency-groups/

The `uv pip` command

`uv` implements most (all?) `pip` commands:

Example:

```
$ uv pip install numpy
```

- Much faster than the original `pip`
- Can sometimes be useful, but should not be used to install dependencies
- Prefer `uv add`

Storage management

`uv` stores data in multiple places:

- **Cache:**
  - `uv` uses aggressive caching to avoid re-downloading (and re-building) dependencies that have already been accessed in prior runs.
  - Contains downloaded and built dependencies, then linked in the virtual environments.
  - Where ? `~/.cache/uv` (`--cache-dir`, `$UV_CACHE_DIR`, `$XDG_CACHE_HOME/uv`)
  - Can be purged: `uv cache clean`

Storage management

`uv` stores data in multiple places:

- **Cache:**
  - `uv` uses aggressive caching to avoid re-downloading (and re-building) dependencies that have already been accessed in prior runs.
  - Contains downloaded and built dependencies, then linked in the virtual environments.
  - Where ? `~/.cache/uv` (`--cache-dir`, `$UV_CACHE_DIR`, `$XDG_CACHE_HOME/uv`)
  - Can be purged: `uv cache clean`
- **Configuration:**
  - Where ? `~/.config/uv`

Storage management

`uv` stores data in multiple places:

- **Cache:**
  - `uv` uses aggressive caching to avoid re-downloading (and re-building) dependencies that have already been accessed in prior runs.
  - Contains downloaded and built dependencies, then linked in the virtual environments.
  - Where ? `~/.cache/uv` (`--cache-dir`, `$UV_CACHE_DIR`, `$XDG_CACHE_HOME/uv`)
  - Can be purged: `uv cache clean`
- **Configuration:**
  - Where ? `~/.config/uv`
- **Persistent data directory:**
  - Contains `python` interpreters and tools
  - `~/.local/share/uv` (`$XDG_DATA_HOME/uv`)

Storage management

uv stores data in multiple places:

- **Cache:**
  - uv uses aggressive caching to avoid re-downloading (and re-building) dependencies that have already been accessed in prior runs.
  - Contains downloaded and built dependencies, then linked in the virtual environments.
  - Where ? `~/.cache/uv` (`--cache-dir`, `$UV_CACHE_DIR`, `$XDG_CACHE_HOME/uv`)
  - Can be purged: `uv cache clean`
- **Configuration:**
  - Where ? `~/.config/uv`
- **Persistent data directory:**
  - Contains `python` interpreters and tools
  - `~/.local/share/uv` (`$XDG_DATA_HOME/uv`)
- **Virtual environment:**
  - By default, in `my_project/.venv/`
  - Contains links to the cache directory (must be on the same FS)
  - Contains all project dependencies

WARNING: Be careful when running on systems where `$HOME` storage is limited.

Use `uvx` to install/run an executable on the fly

```
# Run a tool, right here, right now
$ uvx ruff
$ uv tool run ruff # same, but more verbose

# Add dependencies on the fly
$ uvx --with pandas,pyarrow ipython

$ uvx --from jupyter-core jupyter lab
```

To install a CLI tool with `uv`:

```
$ uv tool install ruff
$ which ruff
  /home/gaetan/.local/bin/ruff
```

Deploying your projects to a remote system

Install uv, clone your project and run your code, that’s it!

```
$(local) [~/work/project] git push
$(local) [~/work/project] ssh cluster

$(cluster) [~]          git clone <PROJECT_URL>
$(cluster) [~/project]  cd project
$(cluster) [~/project]  uv run main.py
```

uv in Docker containers

- `uv` can be used in Docker containers
- Both *distroless* and regular images are provided. `uv` is pre-installed
  - Distroless: `ghcr.io/astral-sh/uv:latest`
  - Alpine: `ghcr.io/astral-sh/uv:alpine`
  - Debian: `ghcr.io/astral-sh/uv:debian-slim`

uv in Docker containers

- uv can be used in Docker containers
- Both *distroless* and regular images are provided. uv is pre-installed
  - Distroless: `ghcr.io/astral-sh/uv:latest`
  - Alpine: `ghcr.io/astral-sh/uv:alpine`
  - Debian: `ghcr.io/astral-sh/uv:debian-slim`

- Run your app!

```dockerfile
FROM ghcr.io/astral-sh/uv:debian-slim
ENV UV_COMPILE_BYTECODE=1 UV_LINK_MODE=copy

# Copy the project into the image
ADD . /project

# Sync the project into a new environment, asserting the lockfile is up to date
WORKDIR /project

RUN uv sync --locked

# Presuming there is an `hello` command provided by the project
CMD ["uv", "run", "hello"]
```

https://docs.astral.sh/uv/guides/integration/docker

Contents

Contents

Installation

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

- *Documentation:* https://docs.astral.sh/uv
- *GitHub repo:* https://github.com/astral-sh/uv
- *Python packaging documentation:* https://packaging.python.org

Contents

Installation

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

- *Documentation:* https://docs.astral.sh/uv
- *GitHub repo:* https://github.com/astral-sh/uv
- *Python packaging documentation:* https://packaging.python.org

---

Your turn!

- Pick a project:
  - Your own Python project
  - One of your students/colleagues' project
  - Open source code from an article
- Bootstrap `uv`

Installation

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

- *Documentation:* https://docs.astral.sh/uv
- *GitHub repo:* https://github.com/astral-sh/uv
- *Python packaging documentation:* https://packaging.python.org

---

Your turn!

- Pick a project:
  - Your own Python project
  - One of your students/colleagues' project
  - Open source code from an article
- Bootstrap `uv`

Cheatsheet:

```
# Init a project
$ uv init .  # or uv init
my_project

# Add a dependency
$ uv add numpy

# Run the code
$ uv run main.py

# Sync the virtual environment
$ uv sync
```

Conclusion

Any questions?

Contact:

- `gaetan@glepage.com`
- https://glepage.com
- GaetanLepage

Conclusion

[1] B. Antunes and D. R. Hill, “Reproducibility, Replicability and Repeatability: A survey of reproducible research with a focus on high performance computing,” *Computer Science Review*, vol. 53, p. 100655, 2024, doi: https://doi.org/10.1016/j.cosrev.2024.100655.

[2] adisbladis, “Python packaging with nixpkgs, pyproject.nix & uv2nix, NixCon 2025.” [Online]. Available: https://talks.nixcon.org/nixcon-2025/talk/Y8TSAW/

[3] T. Fischer *et al.*, “Pixi: Unified Software Development and Distribution for Robotics and AI,” *arXiv preprint arXiv:2511.04827*, 2025.