Tools for environment reproducibility

There are lots!

Min RK, Simula Tools Meetup, Sept. 2022

What do I mean by ‘environment’?

- Collection of software packages (numpy)
- Runtime environment (Python 3.11)
- Operating system (linux, macOS)
- CPU architecture (arm64, amd64)
- *any* subset of the above

Questions to ask

- **who** is it for?
- **what** information is important to capture?
- **when** will it be used?
- **where** will it be run?
- **why** are you recording it?
- **how** will it be used?
- will it be **maintained**?

Example use cases

- Replication / validation for publication review
- Archive for publication record
- Facilitate derivative work
- Portability across machines / collaborators
- Consistency over time for an experiment / study
- Bug reports
- Integration with automation tools

Information to capture

- your code, data
- expected dependencies
- ‘known-good’ versions of all transitive dependencies
- low-level (OS, runtime versions)

Sliding scale

Less detail

More detail

More flexible

Less work to update

More work to stay working

Less flexible

More work to update

More robust, reproducible

Goals vary!

What is the environment for?

- actively maintained library
- single study/simulation/project
- portable application
- documentation, examples
- local development

There are lots of tools!

each solves the problem at a different level

- **pip**: requirements.txt (Python dependencies - may be loose or strict)
- **pip-tools**: lightweight, splits loose and frozen requirements.txt
- **Pipfile/pipenv**: like pip-tools, but manages env and can include Python version
- **poetry**: direct alternative to Pipfile/pipenv. Similar capabilities, difference is mostly aesthetic
- **conda**: not Python specific, includes Python itself, C libraries, etc.
- **docker**: adds the OS itself (~only linux) to the environment record

Level 1

direct dependencies

- Example: `pandas>=1.2`
- Good: likely to work for a long time without updates, depending on stability of dependencies. Highly portable (because very little is specified)
- Bad: if/when things break, there's no record of why, or what was a known-good environment
- Missing: transitive dependencies (numpy)
- Missing: low-level environment (OS, Python)

Level 2

pin all *versions* (often called a lock file)

- Example: `pip freeze > requirements.txt`
- Works well for single-purpose environments. Hard to mix & match
- Over-specified: doesn't tell you what *should* work, just one example of what *does* work
- Must be manually updated to keep latest versions of dependencies
- For pip/pipenv/pip-tools, omits info like OS and Python version. Pipfile, conda can include Python

Level 2

pin all *versions* (often called a lock file)

- How do I update requirements.txt? Enter: pip-tools
- pip-tools provides `pip-compile` and `pip-sync`
- Two files: `requirements.in` (actual direct dependencies, manually maintained), `requirements.txt` generated from `requirements.in`.
- `pip-compile` essentially automates `pip install -r requirements.in; pip freeze > requirements.txt`
- `pip-sync` installs from requirements.txt
- Doesn’t handle virtualenvs for you (more flexible, more work)
- Great fit for Dockerfiles
- Assumes but doesn’t enforce single Python version, possible OS-dependent

Level -1

pin *some versions* (partially pinned environment)

- Example: `pandas==1.3.3`
- Direct dependencies pinned, transitive dependencies not pinned
- *Guaranteed* to break as soon as possible
- **PLEASE NEVER DO THIS**. If you pin one dependency, always pin all of its transitive dependencies.

Level 2.5

also manage the environment

- manual: `python3 -m env`
- pipenv, poetry: do this for you

Level 2.5

pipenv

- Pipfile format, Pipfile.lock
- `pipenv install ipython`
- lockfile preserves environment markers for better portability
- specifies Python version
- automatically managed virtual environments

Level 2.5

poetry

- Standard pyproject.toml, poetry.lock
- A lot like pipenv, but a few more actions like ‘publish’
- ‘everything is a package’
- More customizable via plugins

Level 2.5

tools for managing environments

- Nice if you want them to
- Annoying if you want to control the env (e.g. Docker, host Python)
- Both pipenv and poetry *can* use the running env, but work against it

Level 3

pin all the things (conda)

- Not just Python (can pin Python itself)
- environment.yml: like Pipfile/requirements.in
- `conda env export`
- `conda list --explicit`
- `conda lock`: cross-platform environments

Dockerfile

record of *what* you did

- Small, portable
- Can capture arbitrary actions, unlike other tools - that means Dockerfiles are only as good or bad as the commands in them
- Should be combined with other tools
- Linux-only: restricts running to within an isolated environment

Docker image

binary record of the actual environment

- Robust, highly portable
- May be hard to update, use for *derivative* purposes
- Not easy to inspect
- Not technically an archival format, but in practice the best one we have
- Combining image + Dockerfile is like loose dependencies + lockfile to the max

Conclusion

- Always record ‘loose’ and known-good dependencies *separately*
- Automation is key to reproducibility
- Be more thorough the longer you want it to last
- Don’t specify partial environments! (that includes Python version)