Virtual environments in Python

Maciej Marchwiany, PhD

https://www.linkedin.com/pulse/ai-powered-physics-how-ai-revolutionizing-scientific-discovery-cain-ddbwc

Plan

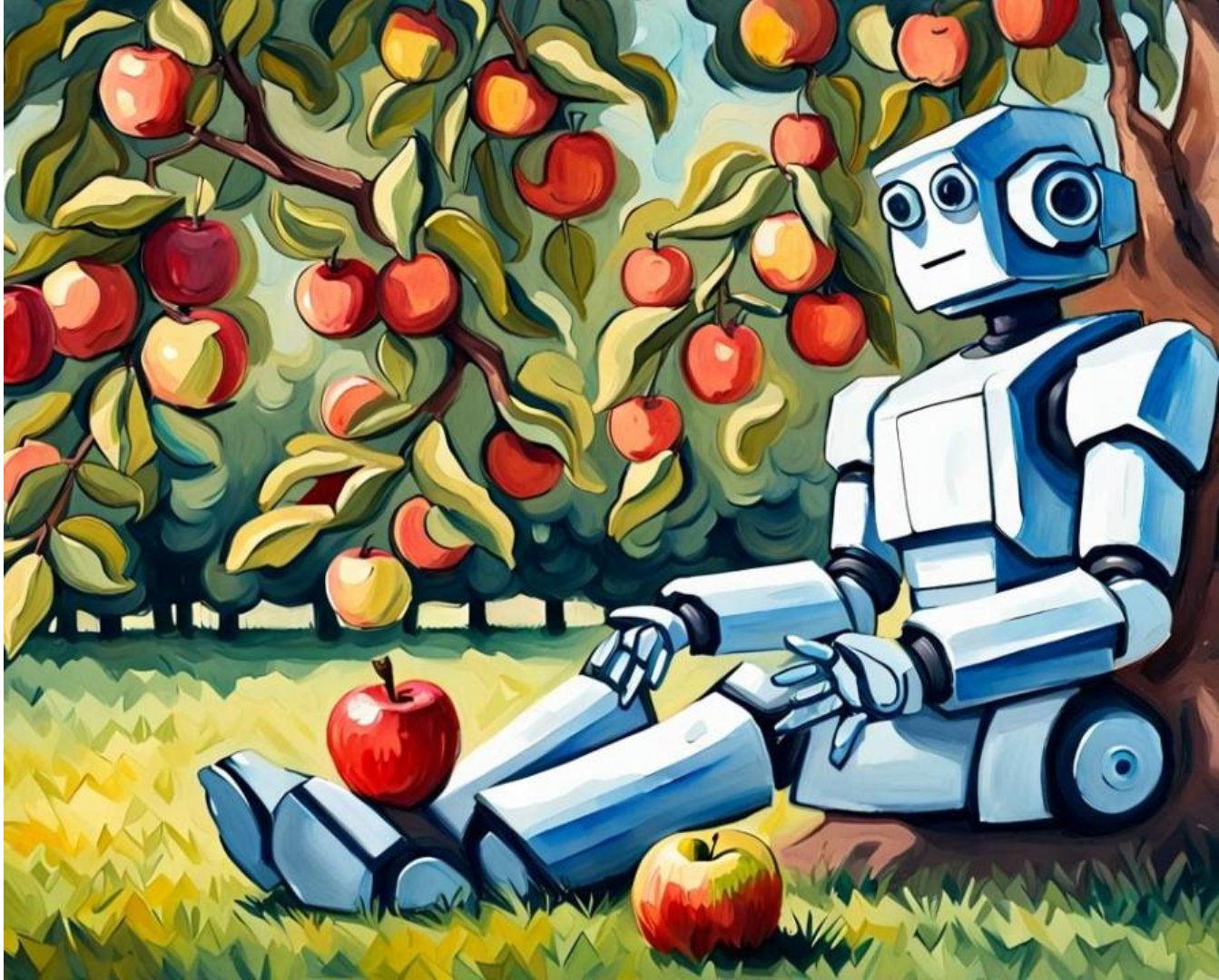
Package installation

venv

conda

Jupyter

Package installation



Package installation in HPC

Challenges:

- **Restricted Access to System-Level Installation:** Users often lack root privileges to install packages system-wide.
- **Dependency Conflicts:** Pre-installed packages might conflict with required versions.
- **Multiple Python Versions:** HPC clusters may have several Python versions, leading to compatibility issues.
- **Complex Build/Compilation**: Some packages require compilation from source (e.g., C, Fortran extensions).
- **Limited Internet Access**: Firewalls or network restrictions prevent direct access to package repositories.
- **Incompatibility with Module Systems**: Python packages may conflict with pre-loaded system modules.
- **Disk Quotas and Storage Limits**: Storage space for packages and environments is often limited.
- **Reproducibility**: Recreating the same environment can be difficult due to changing package versions.

python packages distributions

Source Distribution (source tarballs, .zip, or .tar.gz files)

- The source distribution includes the package source code and setup scripts (like `setup.py` or `pyproject.toml`).
- When you install from source, Python compiles the package during installation.
- Typically, you create a source distribution by running `python setup.py sdist`.

Binary Distribution (wheels)

- Wheel files (`.whl`) are pre-compiled binary distributions that save time during installation because they don't require building from source.
- These are especially important for packages that require compilation (like those with C extensions).
- Wheel is the preferred format because it's faster and easier to install.

Egg Files

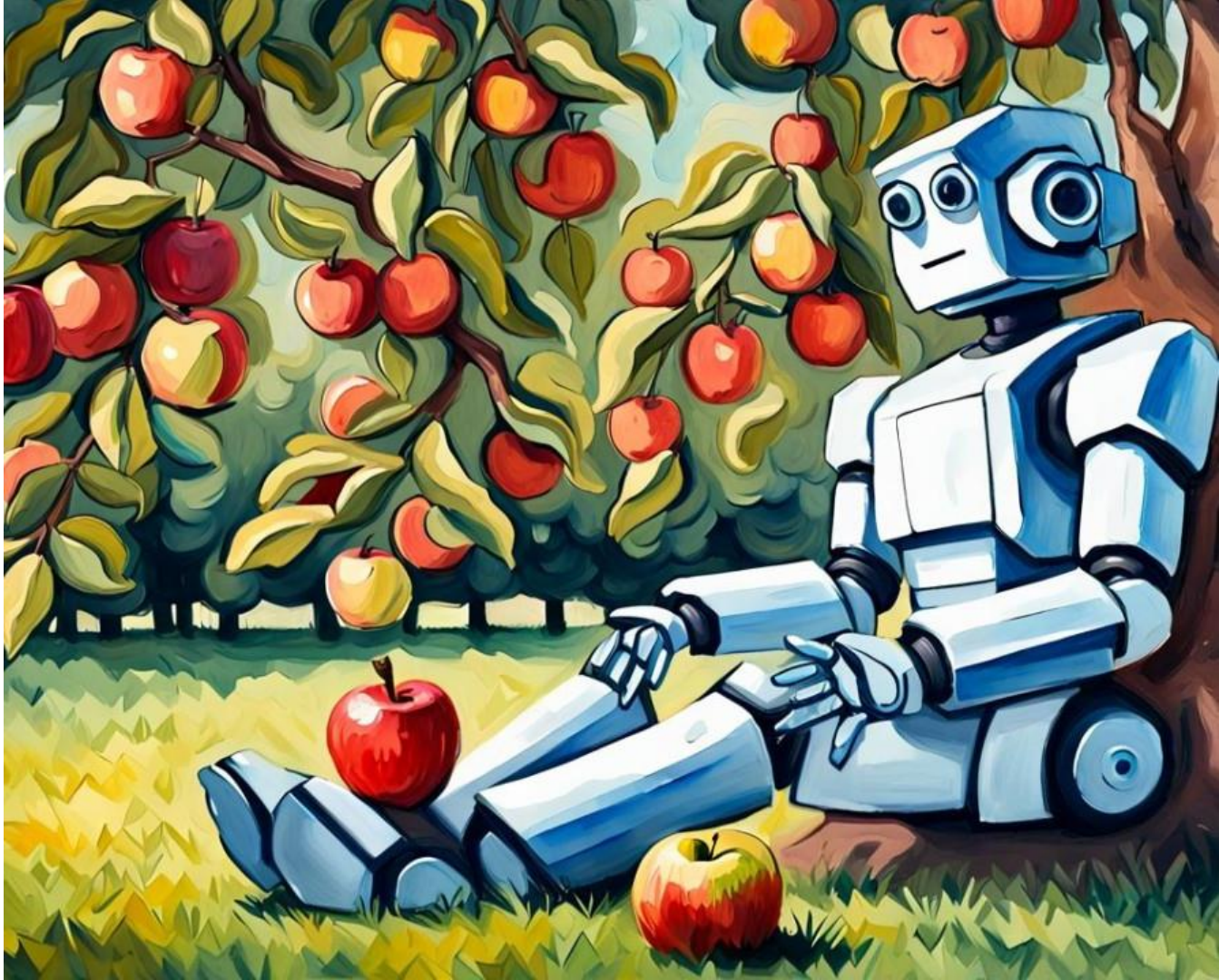
- `.egg` files are an older packaging format that was used before wheels. Though still supported, they are now largely superseded by the `.whl` format.

Installing packages

You can install Python packages using various tools based on your requirements:

- `pip` is the default and most common way to install packages.
- `conda` is a great choice if you're working in data science or need non-Python dependencies.
- `pipenv` and `poetry` are modern tools that offer better management of dependencies and environments.

pip



pip - Python's Default Package Manager

`pip` is the most widely used package manager for Python. It installs packages from the Python Package Index (PyPI), which contains a vast collection of third-party libraries.

```
pip install <package_name>
pip install numpy
```

Install a package locally for a single user:

```
pip install --user <package_name>
```

Package Repositories

PyPI is the primary repository, but there are others, such as:

- **Private Repositories**: Some organizations or developers maintain their own private package repositories to distribute proprietary or internal packages.
- **Custom Repositories**: You can use `pip` with custom repositories by specifying the `-i` option to install from a non-PyPI source.

```
pip install --index-url
https://my.custom.repo/simple/ mypackage
```

Package Versioning

Python packages are versioned, and the versions are typically specified using semantic versioning (e.g., `1.2.3`). When you install a package, you can specify the version to install:

```
pip install requests==2.25.0
```

You can also install the latest compatible version using:

```
pip install 'requests>=2.0.0,<3.0.0'
```

build from git

Developers often host their packages on **GitHub** (or other VCS like GitLab or Bitbucket), and users can install directly from the VCS repository using `pip`.

```
pip install \
git+https://github.com/user/repo.git
```

```
git clone \
https://github.com/user/repo.git
pip install .
```

requirements.txt

If you have a `requirements.txt` file (commonly used to list all the dependencies for a project), you can install all the packages listed in that file by running:

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

requirements.txt:

```
Flask==2.0.1
numpy==1.21.2
pandas==1.3.3
requests==2.26.0
```

How to create requirements.txt

To create a `requirements.txt` file from a **Python virtual environment (venv)**, you can use `pip`, the package installer for Python. The `requirements.txt` file lists all of the packages and their versions currently installed in your virtual environment.

1. `source path/to/your/env/bin/activate`
2. `pip freeze > requirements.txt`

uninstall a package

To uninstall a package:

```
pip uninstall <package_name>
```

Package Dependencies

Package dependencies in Python are external libraries or packages that a Python package requires in order to function correctly. A package depends on other packages for additional functionality, like data processing, web scraping, scientific computations, etc.

For example:

- If you're using the **`Flask`** web framework, `Flask` depends on packages like **`Werkzeug`**, **`Jinja2`**, and **`MarkupSafe`**.
- If you're using **`pandas`** for data analysis, pandas depends on packages like **`numpy`** for numerical operations.

In short, dependencies are the **libraries** or **modules** that a Python package needs to function properly

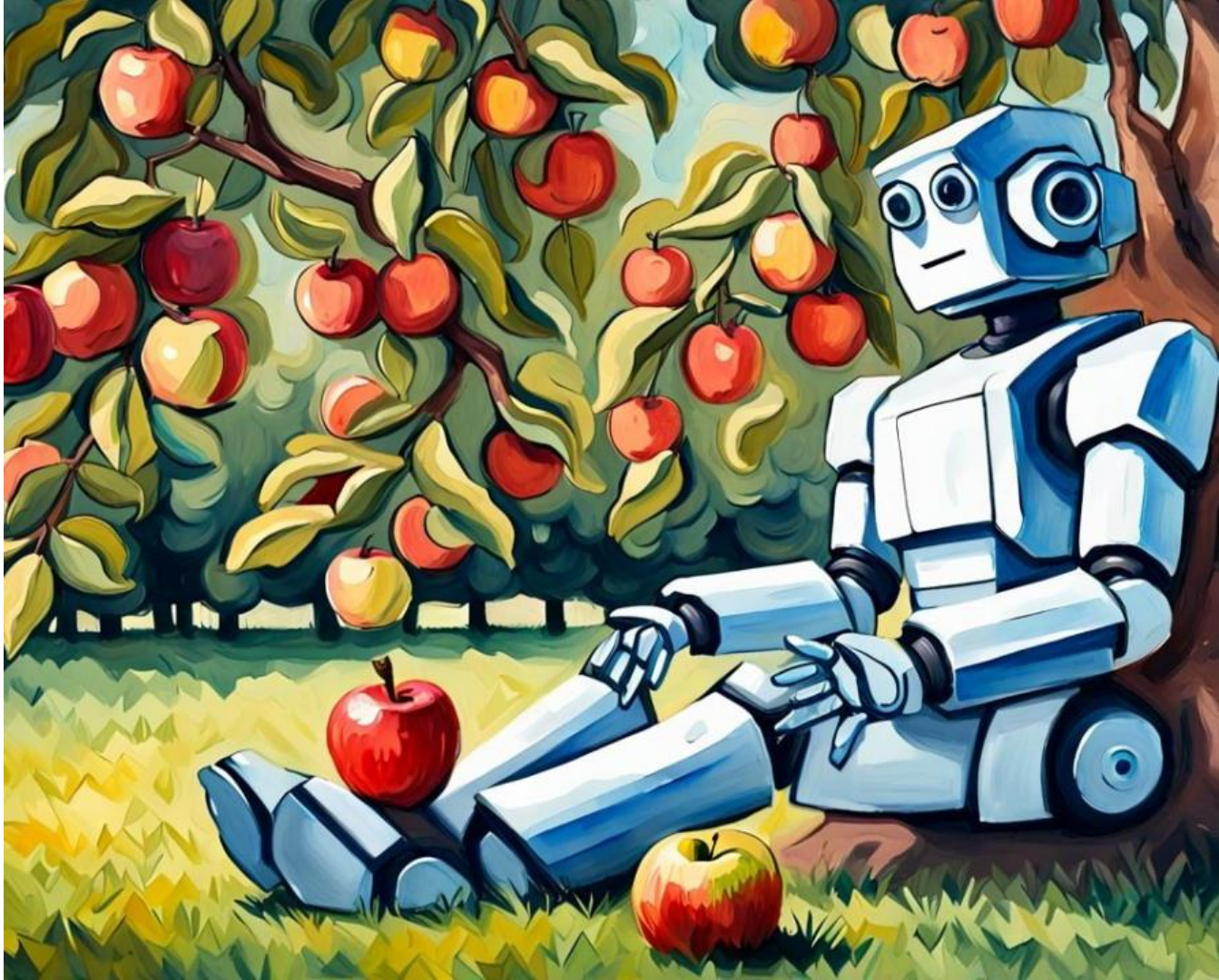
How Package Managers Handle Dependencies

When you install a package using a Python package manager like **pip**, **conda**, or **poetry**, the manager automatically fetches and installs any required dependencies for that package. This process includes resolving transitive dependencies, ensuring that all necessary versions of packages are compatible with each other.

One of the significant challenges in dependency management is version conflicts, where different packages require different versions of the same dependency. While pip does not inherently resolve these conflicts, tools like **Pipenv** and **poetry** offer enhanced dependency resolution capabilities.

- **Pipenv** creates a `Pipfile` to track dependencies and their versions, helping to ensure consistent environments across installations.
- **Poetry** provides a robust dependency resolver that can handle complex dependency trees and offers detailed feedback if it cannot find a compatible solution

conda

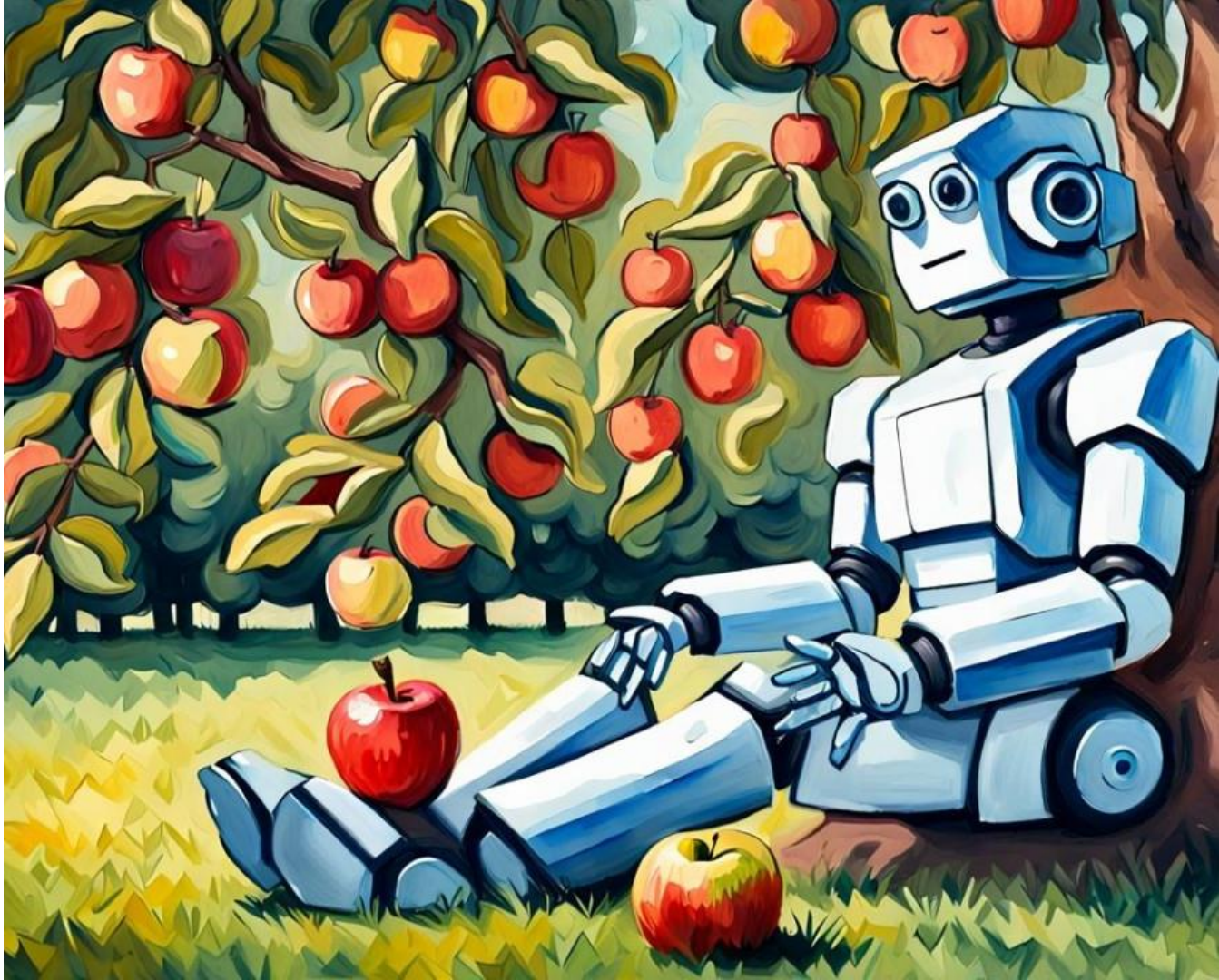


conda installing packages

- To install a package in conda you need:
  `conda install <package_name>`

- You can mixing conda and pip installations. First install all conda packages and then us pip.

Virtual Environment



Virtual Environments

A **virtual environment** in Python is a self-contained directory that contains a Python interpreter and a set of libraries. It allows you to create isolated environments for different projects, ensuring that each project has its own dependencies and versions of libraries, independent of other projects or the system Python installation.

A **virtual environment** in Python is a way to create an isolated environment where you can install specific versions of Python packages for a project. It prevents package conflicts between different projects and ensures a clean, manageable workspace. Virtual environments are essential tools for professional Python development and are widely used in both small and large projects.

Why Use Virtual Environments?

1. **Package Version Conflicts**: Different projects might require different versions of the same package. A virtual environment allows each project to have its own version of the package, avoiding conflicts.
   Example: Project A needs numpy==1.21.0, but Project B needs numpy==1.19.0. A virtual environment ensures that these projects don't interfere with each other.
2. **Isolation**: Virtual environments isolate your project dependencies from the global Python environment. This means that packages you install for one project won't affect others or the Python installation on your system.
3. **Reproducibility**: Virtual environments make it easier to reproduce a project's setup on another machine. By using a requirements.txt file, which lists all the packages and their versions, you can share and recreate the exact environment.
4. **Cleaner Development**: Having a separate environment for each project ensures that only the necessary packages are installed, keeping things neat and organized. It avoids "polluting" your system Python environment with unused packages.

How Do Virtual Environments Work?

A virtual environment essentially copies the **Python interpreter** and other needed files into a folder, allowing the environment to act independently of the system's Python installation.

When you activate a virtual environment, it temporarily modifies the **PATH** environment variable so that Python and `pip` commands use the environment's interpreter and packages rather than the system-wide ones.

Creating Virtual Environments

- **Using `venv` (Built-in Tool)**

Create the Virtual Environment: Navigate to the folder where you want to store your project and run the following command:

```
python -m venv myenv
```

Or: `venv myenv`

- Using `virtualenv` (Third-Party Tool)

Install `virtualenv`: If you don't have `virtualenv` installed, you can install it globally using `pip`: *pip install virtualenv*

Create a Virtual Environment: Once `virtualenv` is installed, you can create a new environment by running:

```
virtualenv myenv
```

Use pacakges installed on the system: `--system-site-packages`

```
venv --system-site-packages myenv
```

activate environment

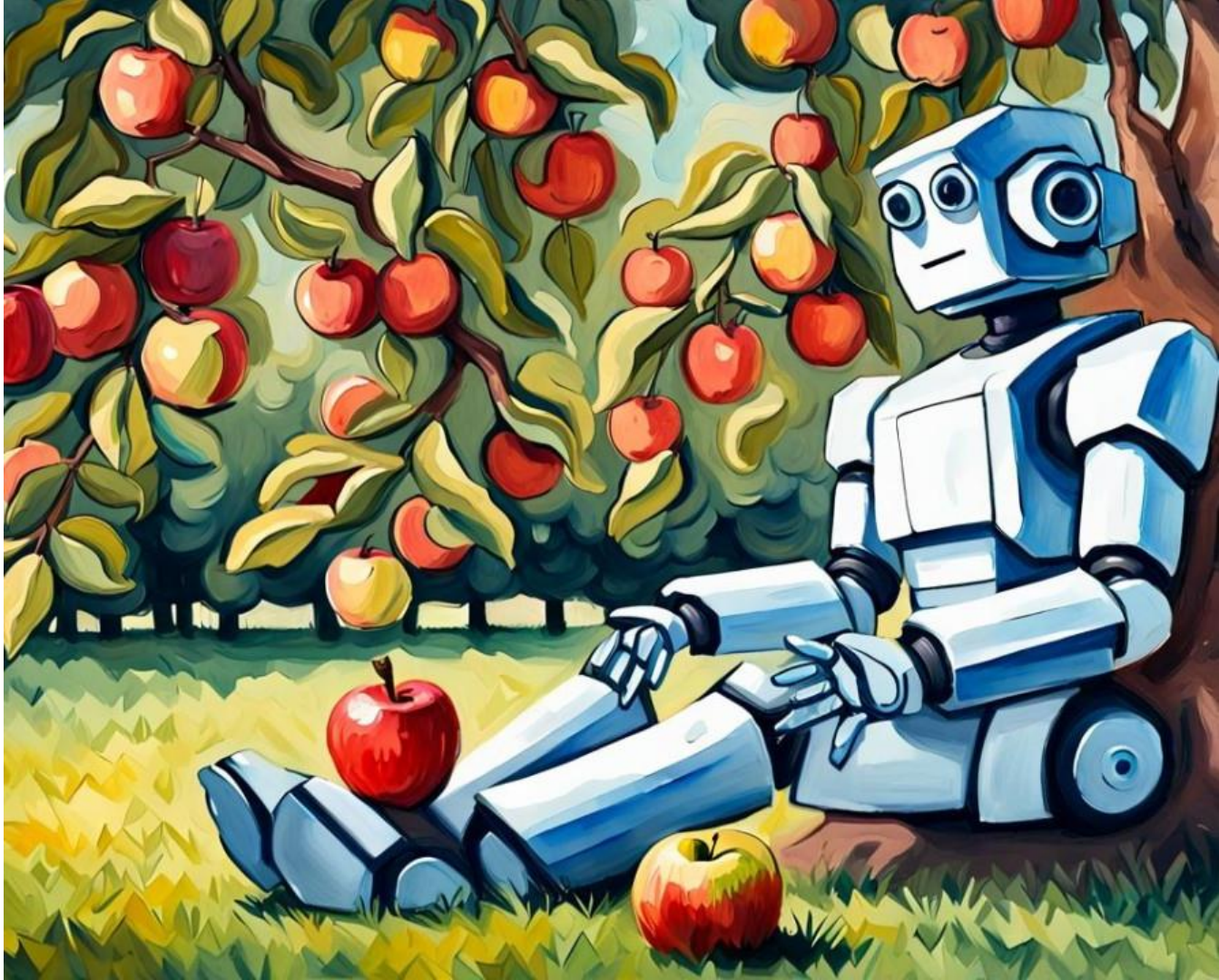
Activate the Virtual Environment: Once the environment is created, you need to activate it so you can start using it. Activation depends on the operating system:

```
source myenv/bin/activate
```

Deactivate the Virtual Environment: When you are done working in the virtual environment, you can deactivate it by running:

```
deactivate
```

Conda Environment



conda

- **conda** is an open-source **package manager** and **environment management system** that is part of the **Anaconda** and **Miniconda** distributions. Conda is widely used in data science, scientific computing, and machine learning because it can manage both **Python packages** and **non-Python dependencies** (such as libraries, compilers, and tools).

Key Features of conda

1. **Environment Management**: Conda can create and manage isolated environments with specific versions of Python or other programming languages, allowing multiple projects to have different dependencies without conflict.
2. **Cross-Language Package Management**: Conda can manage both **Python** and **non-Python** dependencies, such as system libraries (`gcc`, `libblas`), making it more versatile than `venv` in certain contexts.
3. **Precompiled Binaries**: Conda installs **precompiled binary packages** (i.e., no need to compile code from source), which can simplify the installation process, especially for scientific libraries.
4. **Multiple Languages**: While `venv` is Python-only, **conda** can handle **multiple languages** (e.g., R, Julia, and other software tools) in the same environment.

Conda is isolated!

Conda not only manages Python packages, but also system-level dependencies and environments.

It allows you to create full environments that isolate both the Python interpreter and any additional non-Python libraries or tools.

conda create

When creating a virtual environment with **conda**, you have a variety of options that can customize how the environment is created. Below are some of the most useful `conda create` parameters

- Environment Name: `--name (-n)`
  conda create --name myenv
- Full Path for the Environment: `--prefix`
  conda create --prefix /path/to/env
- Python Version: `python=<version>`
  conda create --name myenv python=<version>
- Also you can install conda packages
  `conda create --name myenv <package_name>`

conda activate

- To activate conda environment:
  `conda activate <env>`
  or
  `. activate <env>`

- To deactivate conda environment :
  `conda deactivate`

- To list all conda environments:
  `conda env list`

Packages installation in conda

To install packages in Conda, you can use the `conda install` command, followed by the name of the package you want to install. Conda automatically resolves dependencies, ensuring that all required libraries are installed in your environment.

```
conda install <package_name>
```

If you want to install a specific version of a package, you can include the version number in the command.

```
conda install <package_name>=<version>
```

Channels in Conda

In **Conda**, **channels** are locations where Conda looks for and downloads packages and their dependencies. They are essentially directories or repositories that contain packaged software, and they serve as the primary means of distributing and managing software in the Conda ecosystem.

When you run commands like `conda install`, Conda will search the specified channels (in the order you specify) to find and download the requested package. By default, Conda searches in the **default** channel and the **conda-forge** channel, but you can specify other channels, prioritize them, and even create your own custom channels.

```
conda install -c <channel> <package_name>
```

Most Used Conda Channels

- **conda-forge** – Popular for up-to-date, community-maintained packages.
- **Anaconda** (default) – Official, stable channel maintained by Anaconda, Inc.
- **bioconda** – Specialized for bioinformatics tools and software.
- **CUDA/NVIDIA** – For GPU-accelerated libraries for deep learning.
- **pytorch** – Focused on PyTorch and deep learning frameworks.
- **miniforge** – A lightweight, minimal Conda distribution using conda-forge.

uninstall package

To remove a package from an environment, use the `conda remove` command.

```
conda remove <package_name>
```

.yml file

If you're setting up an environment using a configuration file (`environment.yml`), you can include a list of packages to be installed in the file.

Create the environment from the `.yml` file:

```
conda env create -f environment.yml
```

.yml file

If you already have a Conda environment and you want to export it to a `.yml` file, you can do this easily using the `conda env export` command.

```
1. conda activate myenv
2. conda env export > environment.yml
```

.yml file

```
name: myenv          # Name of the
environment
channels:
  - conda-forge      # Conda-forge channel
  - defaults         # Default channel
dependencies:
  - numpy=1.19.2     # Specific numpy
  - pandas           # Latest pandas
  - matplotlib       # Latest matplotlib
  - python=3.8       # Python version 3.8
  - pip              # Install pip, if needed
- pip:
    - requests       # Install the requests
package via pip (if not available in
conda)
```

miniconda, condaforge …

Conda is a package and environment management system that has several implementations, each serving a specific use case or distribution. While the core functionality remains the same, the **implementation** or **distribution** may vary depending on the tools or ecosystem it's part of.

miniconda, condaforge …

- **Anaconda** is the most well-known and widely-used **distribution** of Python and Conda. It comes with a comprehensive suite of preinstalled data science and scientific computing tools, and it includes the **Conda package manager**.
- **Miniconda** is a minimal, lightweight version of the Anaconda distribution. It installs the **Conda package manager** and gives you a bare-bones Python environment. Miniconda does not come with preinstalled libraries, so you can start with just Conda and install only the packages you need.
- **Mamba** is an **alternative implementation** of the Conda package manager, designed to be **faster** and **more efficient**. It's a **drop-in replacement** for Conda and is particularly useful for users working with large environments and large sets of dependencies.
- **Mamba-Forge** is an optimized distribution of **Mamba**, which is itself a fast, drop-in replacement for **Conda**. Mamba-Forge combines the power of **Mamba** with the benefits of the **Conda-Forge** community's package

miniconda, condaforge …

scientific computing tools, and it includes the **Conda package manager**.

- **Miniconda** is a minimal, lightweight version of the Anaconda distribution. It installs the **Conda package manager** and gives you a bare-bones Python environment. Miniconda does not come with preinstalled libraries, so you can start with just Conda and install only the packages you need.
- **Mamba** is an **alternative implementation** of the Conda package manager, designed to be **faster** and **more efficient**. It's a **drop-in replacement** for Conda and is particularly useful for users working with large environments and large sets of dependencies.
- **Mamba-Forge** is an optimized distribution of **Mamba**, which is itself a fast, drop-in replacement for **Conda**. Mamba-Forge combines the power of **Mamba** with the benefits of the **Conda-Forge** community's package repository.

miniconda, condaforge …

you need.

- **Mamba** is an **alternative implementation** of the Conda package manager, designed to be **faster** and **more efficient**. It's a **drop-in replacement** for Conda and is particularly useful for users working with large environments and large sets of dependencies.
- **Mamba-Forge** is an optimized distribution of **Mamba**, which is itself a fast, drop-in replacement for **Conda**. Mamba-Forge combines the power of **Mamba** with the benefits of the **Conda-Forge** community's package repository.

miniconda, condaforge …

particularly useful for users working with large environments and large sets of dependencies.

- **Mamba-Forge** is an optimized distribution of **Mamba**, which is itself a fast, drop-in replacement for **Conda**. Mamba-Forge combines the power of **Mamba** with the benefits of the **Conda-Forge** community's package repository.

venv vs conda

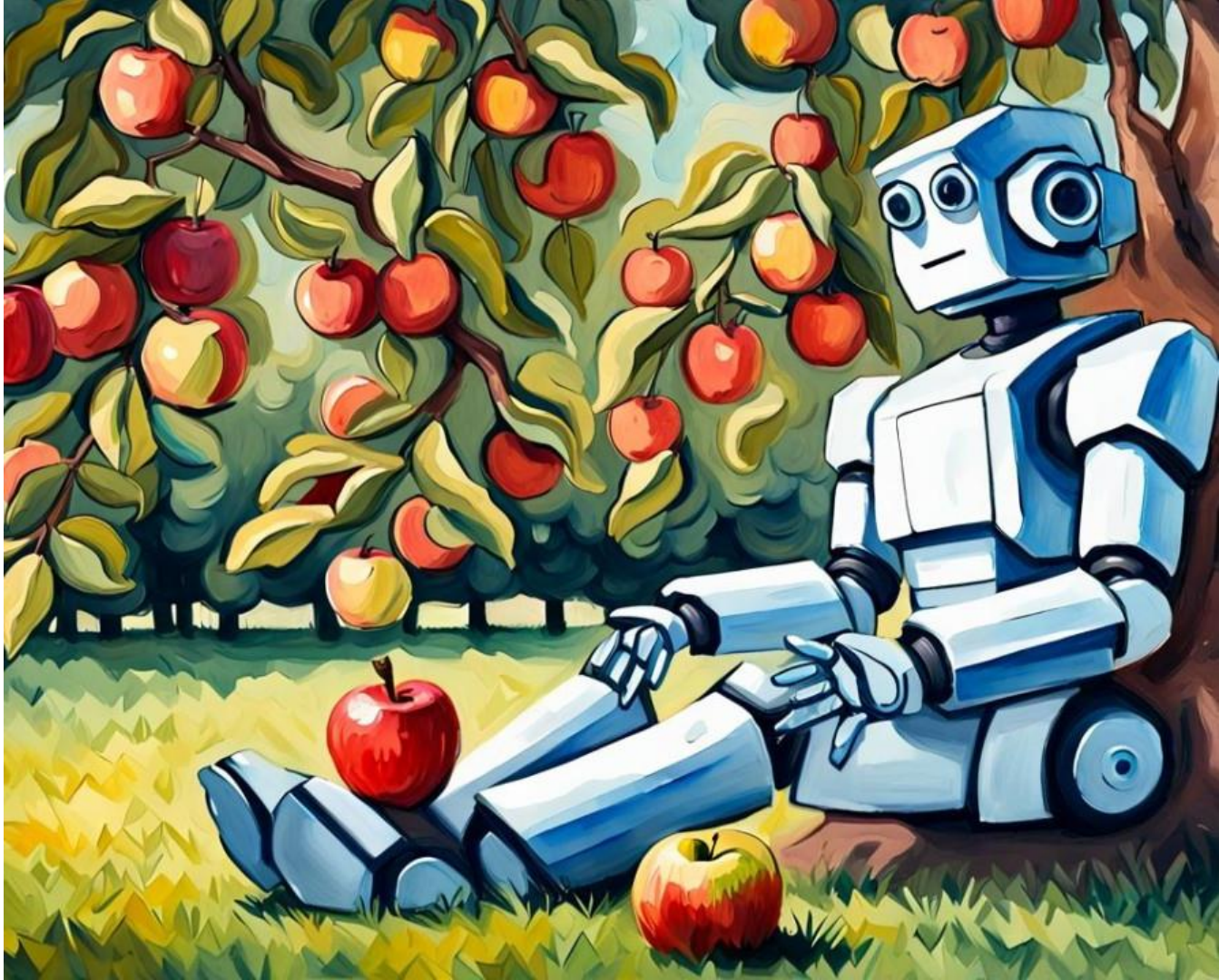
When to Use `venv`:

- You're working on a **simple Python project** with **Python-only dependencies** (e.g., web apps with Flask, Django, etc.).
- You want a **lightweight** tool and don't need to worry about external system libraries or precompiled binaries.
- You prefer using **PyPI** (the Python Package Index) for installing packages.

When to Use `conda`:

- You're working with **data science**, **machine learning**, or **scientific computing** projects that require **complex dependencies** like `numpy`, `pandas`, `tensorflow`, and system-level libraries (e.g., compilers).
- You need **cross-language support** (e.g., R, Julia) or to manage different versions of Python alongside other software tools.
- You want to install **precompiled binary packages** to avoid compilation issues.
- You need a **robust environment** that isolates both Python and system dependencies.

Poetry



poetry

Poetry is a powerful, modern tool for dependency management and packaging in Python. It simplifies the process of setting up and managing Python projects, ensuring reproducibility and ease of use through its `pyproject.toml` file and automatic virtual environment management. Whether you're working on a small project or a large-scale package, Poetry can significantly improve your workflow by consolidating multiple tools into one seamless experience.

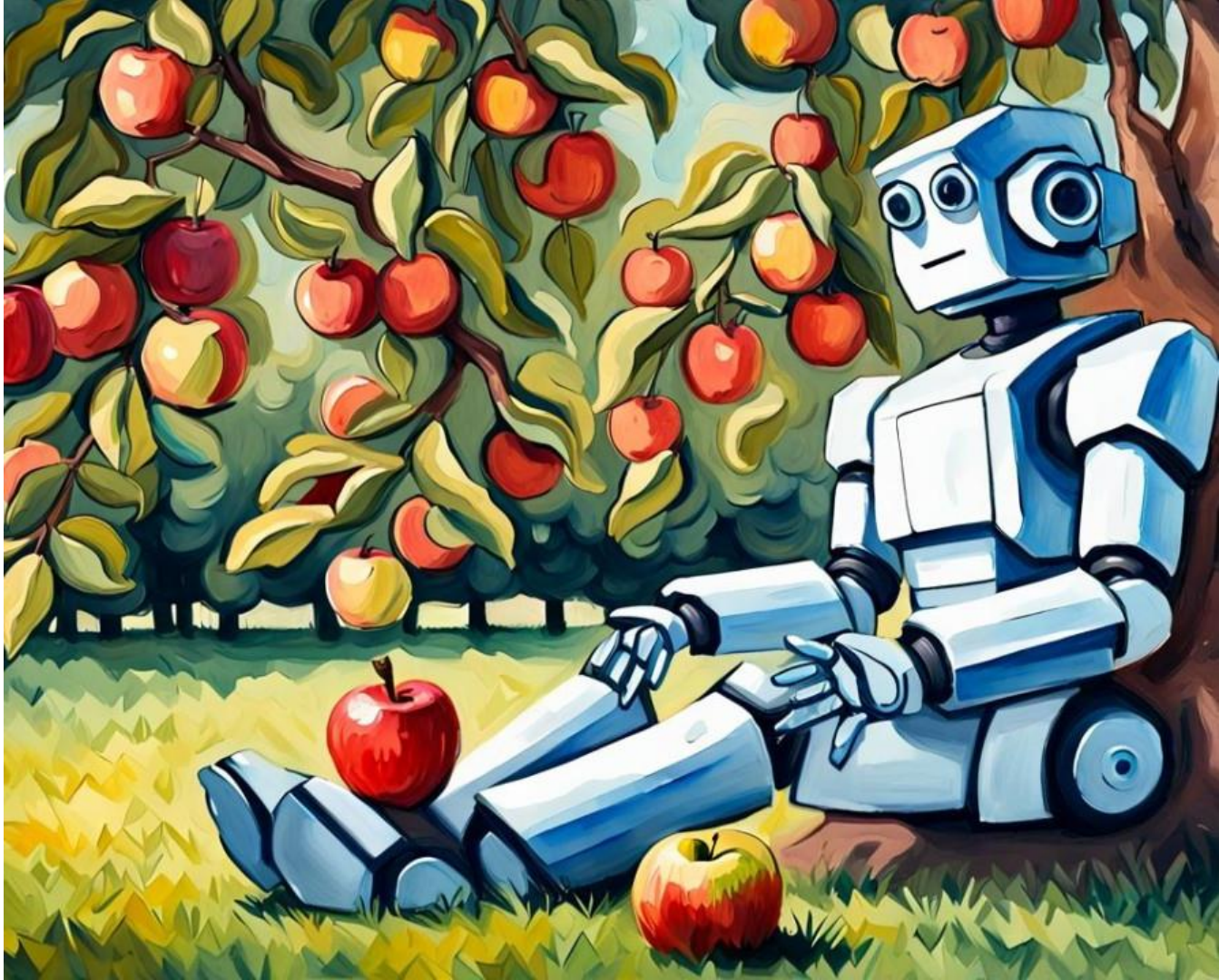
working with poetry

- You can create a new Python project using Poetry with:
```
poetry new my_project
```
- If you want to convert an existing project to use Poetry, navigate to the project directory and run:
```
poetry init
```
- To add a package as a dependency, use the `poetry add` command:
```
poetry add <package_name>
poetry add $(cat requirements.txt)
```

working with poetry

- To install all the dependencies for the project as specified in the `pyproject.toml` file:
  `poetry install`
- List Poetry virtual environments
  `poetry env list`
  By default, Poetry creates a virtual environment in `{cache-dir}/virtualenvs`.
- Poetry automatically handles virtual environments. But if for some reason you want to create one manually:
  `poetry shell`
- You can execute a command within a virtual environment managed by Poetry.
  `poetry run <command>`
  `poetry run python --version`
- To generate or update the `poetry.lock` file (which locks the exact versions of all dependencies):
  `poetry lock`

Jupyter Kernel



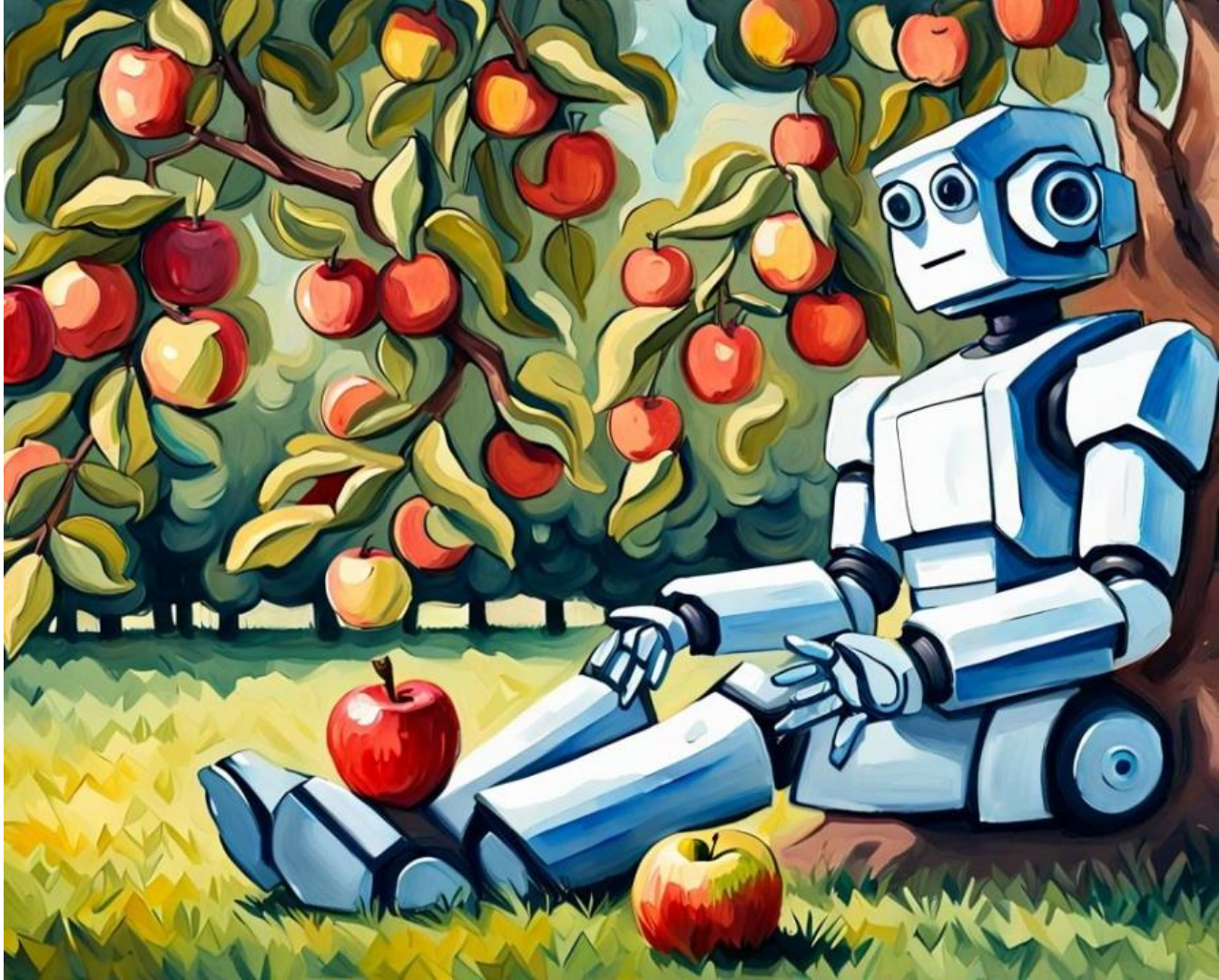
Jupyter Kernel

A **kernel** in Jupyter is a computational engine that executes the code in a notebook. It is language-specific (e.g., Python, R, Julia) and manages code execution, maintains variable states, and returns results or errors to the notebook interface. Each Jupyter notebook is connected to a kernel that runs the code for that specific environment.

Jupyter Kernel

1. Install `Jupyter` and `ipykernel` packages:
   `pip install jupyter ipykernel`
   or
   `conda install jupyter ipykernel`
2. Create a virtual environment or conda environment with all needed packages.
3. Create a new kernel from virtual environment (or conda env) `my_env` with `display name` my_kernel
   ```
   python -m ipykernel install --user --name my_env --display-name "my_kernel"
   ```

Best practices



Package installation best practices

- **Use virtual environments** (`venv` or `conda`) to isolate dependencies and avoid conflicts.
- **Use package managers** like `pip`, `conda`, or `Poetry` to handle package installation and dependency management.
- **Leverage precompiled binaries** (via `conda`) to avoid the need for compiling packages from source.
- **Define dependencies** in environment files (`requirements.txt`, `environment.yml`) to ensure reproducibility.

Package installation best practices

- **Install packages through batch jobs** to avoid interference with other users and system performance.
- **Transfer packages** from your local machine if there is no direct internet access on the cluster.
- **Use networked storage** to avoid hitting disk quotas on the local system.
- **Monitor installed packages** and manage dependencies effectively to avoid conflicts.
- **Coordinate resource usage** and install during off-peak hours to minimize impact on shared resources.
- **Lock package versions** to ensure consistent environments across different machines and users.

Best Practices for Naming Virtual Environments

1. **Use Descriptive Names**:
   - Name the virtual environment based on the project name to easily identify it, e.g., `project_venv` or `my_project_env`.
2. **Include Python Version**:
   - If your project requires a specific Python version, include it in the name, e.g., `my_project_py38` or `my_project-3.8`.
3. **Adopt a Consistent Naming Convention**:
   - Stick to a consistent format across all projects. Common conventions include:
     - `env`
     - `.venv`
     - `<project_name>_env`
     - `<project_name>-<python_version>`
4. **Use Hidden Directories**:
   - Consider using `.venv` as it keeps the directory hidden by default, reducing clutter in your project folder while indicating its purpose.
5. **Standardize Across Teams**:
   - If working in a team, agree on a naming convention to ensure consistency and ease of collaboration.

Best Practices for Organizing Virtual Environments

1. **One Environment per Project**:
   - Create a separate virtual environment for each project to avoid dependency conflicts and ensure isolation.
2. **Central Directory for Environments**:
   - If managing multiple environments, consider storing them in a central directory (e.g., `~/venvs/`) to keep your workspace organized.
3. **Use Environment-Specific Requirements Files**:
   - Maintain a `requirements.txt` file within each project directory to document dependencies specific to that environment
4. **Document Environment Setup**:
   - Include setup instructions in your project's README file, detailing how to create and activate the virtual environment and install dependencies.
5. **Utilize Tools for Management**:
   - Consider using tools like `pyenv` for managing Python versions alongside `venv`, which can help automate the creation and activation of environments based on project settings.

Temporary files

When installing packages, package managers like **`pip`**, **`conda`**, and **`Poetry`** store temporary files and caches to optimize the installation process, reduce redundant downloads, and speed up future installations. Below are the default locations where each tool stores these files.

Temporary files

	pip	conda	poetry
Cache Location	`~/.cache/pip`	`~/.conda/pkgs`	`~/.cache/pypoetry`
clear the cache	`pip cache purge`	`conda clean \ --all`	`poetry cache clear --all pypi`
disable the cache	`pip install \ --no-cache-dir <package-name>`		

You can safely remove cache folders:

```
rm -rf ~/.cache/pip ~/.conda/pkgs ~/.catch.pypoetry
```