VIRTUALIZATION

which virtualenv

What `virtualenv` Does:

`virtualenv` is a Python tool that creates **isolated Python environments**. These environments allow you to manage dependencies separately for different projects, avoiding conflicts between package versions.

Key Functions of `virtualenv`:

1. **Isolation**:
   - Creates a folder containing a separate Python installation (or a symlink to the system Python) along with an independent `site-packages` directory.
   - Prevents globally installed packages from interfering with the project.
2. **Dependency Management**:
   - Allows installing packages (via `pip`) without affecting the system-wide Python installation.
   - Useful when different projects require different versions of the same library.

virtualenv ~/.venv

The command `virtualenv ~/.venv` (or the resulting virtual environment located at `~/.venv`) serves to **create an isolated Python environment** in the user's home directory. Here's a breakdown of its purpose and function:

🛠️ What it does:

- **Creates a virtual environment** at the location `~/.venv` (i.e., a hidden folder named `.venv` in your home directory).
- This environment includes its **own Python interpreter**, **pip**, and **site-packages**, separate from the system-wide Python installation.

🎯 Purpose of using `~/.venv`:

- **Isolation**: Keeps dependencies and packages separate from system Python or other projects.
- **Avoids permission issues**: You don't need sudo to install packages.
- **Consistency**: Ensures that your project or scripts run with the exact versions of libraries you install.

MAKE FILE

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

Upgrade the pip installer package for Python to the latest version and install all the packages in te requirements.txt file.

```
python -m pytest --cov=auction
```

Tests all Python files as a module (-m flag) using the pytest framework and the current python interpreter. It will execute all files with a **test** in it whether it is in the current directory or in a subdirectory.

--cov will show the coverage percentage of the auction.py file.

```
pylint --disable=R,C


# NOTE THAT
```

1. `--disable=R,C`:
   - `R` (Refactor) → Disables suggestions related to code refactoring (e.g., simplifying expressions, removing redundancy).
   - `C` (Convention) → Disables PEP 8 style guideline checks (e.g., naming conventions, indentation, whitespace).

```

```

This will report critical issues like syntax errors, undefined variable names, problematic patterns like unused imports, and issues preventing pylint from running further.

```
black *.py
```

It **reformats the code in auction.py directly** to comply with a strict consistent style e.g (PEP 8 complaint).

TO SEE YOUR CURRENT WORKING DIRECTORY

```
import os

print(f'Current directory is: {os.getcwd()}')
```

SIMULATION

Just like the simulated logic in test_auction.py, the same will be done to the CSV files in test_transform_aucion.py so we do not interfere with I/O (input and output) of the cleaned and uncleaned auction data. The above approach is proper for unit testing (testing individual unit of codes in a test file).

NOTE: That why we simulate is to test that only the python scripts I wrote to extract and transform the auction data is correct and not manipulate the actual data or data source.

BRIEF SUMMARY

Creating CI/CD test with pytest is one interesting skill of a data engineer. It makes sure that the code you generate to build data pipelines, ETL, ELT, build applications, or develop machine learning models do not fail after deployment.

With continuous integration using pytest, we can check that our logic works locally in an environment like GitHub codespaces and also perform continuous delivery with GitHub actions and Amazon Web Services (AWS codebuild) before pushing it to production.