PYTHON ASSIGNMENTS – MODULES, IMPORTS, ENVIRONMENT, AND PACKAGING

Section 1: Import Styles

Objective: Understand and apply different import styles.

Assignments:

1. Create two Python files: `math_utils.py` and `main.py`.
    - `math_utils.py` contains `add()`, `subtract()` functions.
    - Import and use them in `main.py` using:
        - Absolute import
        - Relative import (`from .math_utils import add`)
2. Show examples of:
    - `import module`
    - `from module import function`
    - `import module as alias`
    - `from module import *` and discuss when it's a bad idea.

Challenge:

- Try using relative import in a folder hierarchy with `__init__.py`.

Section 2: `if __name__ == "__main__"`

Objective: Understand entry points in Python scripts.

Assignments:

1. Write a script `greet.py` that prints a greeting when run directly but does nothing when imported.
2. Convert a script-based calculator into a reusable module that only runs interactively when `__name__ == "__main__"`.

Reflection:

Explain the difference between executing a script and importing it. Why is `__name__` important?

Section 3: Absolute vs Relative Imports

Objective: Practice both import styles across packages.

Assignments:

1. Create a package `ecom/` with:

```
ecom/
  __init__.py
  cart.py
  utils/
    __init__.py
    calculator.py
```

8. In `cart.py`, import `add_item()` from `calculator.py`:
    - First using absolute import.
    - Then using relative import.
9. Discuss what happens when you run `cart.py` directly vs as part of a package.

Section 4: venv, pip, requirements.txt

Objective: Manage isolated environments and dependencies.

Assignments:

1. Create a virtual environment for a small Flask project using:

```
python -m venv myenv
source myenv/bin/activate  # Windows: myenv\Scripts\activate
```

4. Install the following packages: `Flask`, `requests`, `pandas`.
5. Generate a `requirements.txt` file.
6. Clone your project in another folder and set it up using:

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

Challenge:

- Try creating two venvs with different versions of the same library (e.g., `Flask==1.1.2` vs `Flask==2.0.1`) and observe behavior.

Section 5: Build and Publish a Package to PyPI

Objective: Package your code and publish it.

🔨 Project: mathify – A Custom Math Utility Package

Folder Structure:

```
mathify/
├── mathify/
│   ├── __init__.py
│   ├── arithmetic.py
│   └── algebra.py
├── README.md
├── setup.py
├── LICENSE
└── pyproject.toml
```

Tasks:

1. Add functions in `arithmetic.py`: `add()`, `subtract()`, etc.
2. Add `solve_linear()` in `algebra.py` to solve ax + b = 0.
3. Fill out `README.md` with usage examples.
4. Write `setup.py` and `pyproject.toml` to define the metadata.
5. Register on https://pypi.org.
6. Build the package:
7. `python -m build`
8. Upload to PyPI:
9. `twine upload dist/*`

Final Evaluation Questions:

1. What are the advantages of relative imports in large-scale projects?
2. How does `venv` help in dependency management for teams?
3. What happens if you forget to include `__init__.py` in a subpackage?
4. Explain `pyproject.toml` vs `setup.py`—what is the future of packaging in Python?
