VIDHI ROHIRA

SY BTECH

COMPUTER ENGINEERING

PD LAB-1

231071052

BATCH C

PD LAB 1

AIM:- Handling different python versions and virtual environments.

THEORY:-

1. Understanding Python Versions

• Python 2 vs. Python 3: Python 2.x is considered legacy and is no longer actively maintained, whereas Python 3.x is the current and actively developed version. Despite this, some older projects still rely on Python 2.x, so it's important to handle these versions carefully to avoid compatibility problems.

• Version Compatibility: Libraries and frameworks may be designed to work with specific Python versions. Developers need to ensure that the Python version used aligns with the libraries required for their projects to prevent issues.

2. Why Manage Multiple Python Versions

• Legacy Projects: Maintaining older projects that run on Python 2.x while simultaneously working on newer projects that require Python 3.x necessitates managing different Python versions effectively.

• Library Dependencies: Some projects depend on libraries that are compatible only with particular Python versions. Managing these versions ensures that you can use the necessary libraries without conflicts.

• Testing and Development: Testing your code across multiple Python versions is crucial to ensure that it works consistently in various environments and versions.

3. Tools for Managing Python Versions

• Pyenv: Pyenv allows you to manage multiple Python versions easily. It helps switch between different versions locally for individual projects or globally across your system. - Usage: Install a Python version with `pyenv install <version>`, and switch between versions using `pyenv global <version>` or `pyenv local <version>`.

• Virtualenv: Virtualenv creates isolated Python environments, preventing conflicts between project dependencies. - Usage: Set up a virtual environment with `virtualenv <environment_name>`, and activate it with `source <environment_name>/bin/activate` (Linux/macOS) or `<environment_name>\Scripts\activate` (Windows).

• Anaconda/Miniconda: Conda, part of Anaconda and Miniconda, is popular in data science for managing both Python and non-Python dependencies. It manages environments with specific Python versions. - Usage: Create a new environment with `conda create -n <env_name> python=<version>`, and activate it using `conda activate <env_name>`.

4. Virtual Environments: Concept and Usage

• Isolation: Virtual environments provide isolated Python interpreters and libraries for different projects, ensuring that changes in one project do not affect others.

• Creating Virtual Environments: - Using venv (Python 3 built-in tool): `python3 -m venv <env_name>`
- Using virtualenv: `virtualenv <env_name>`

• Activating/Deactivating Virtual Environments: - Activate: Use `source <env_name>/bin/activate` (Linux/macOS) or `<env_name>\Scripts\activate` (Windows).
- Deactivate: Use `deactivate` (works across all platforms).

• Project-Specific Environments: Creating a virtual environment in your project directory ensures that the project uses the correct dependencies without interfering with other projects.

5. Combining Python Version Management with Virtual Environments

• Pyenv + Virtualenv: The `pyenv-virtualenv` plugin integrates `pyenv` with `virtualenv`, allowing you to manage virtual environments for different Python versions seamlessly. - Usage: Create a virtual environment with a specific Python version using `pyenv virtualenv <python_version> <env_name>`.

• Pipenv: Pipenv combines the functionality of `pip`, `virtualenv`, and `pyenv` to manage dependencies and virtual environments. It simplifies the process by automatically handling both. - Usage: Install dependencies with `pipenv

install`, which also creates a virtual environment if one doesn't exist. Activate the environment with `pipenv shell`.

6. Best Practices*:*
Always Use Virtual Environments: Even for small projects, using virtual environments
helps avoid dependency issues and makes your project more portable.
Specify Python Version: Explicitly specifying the Python version in your virtual
environment ensures that your project uses the correct interpreter, reducing the risk of compatibility issues. Use Version Control: Keep track of your project dependencies using a requirements.txt file or Pipfile, and ensure your virtual environment configuration is documented.

7. Example Workflow for Managing Python Versions and Virtual Environments
• Set Up Pyenv: - Install `pyenv` on your system.
- Install the necessary Python versions using `pyenv install <version>`.
• Create Virtual Environment: - Use `pyenv virtualenv <python_version> <env_name>` to create a virtual environment with the specific Python version.
- Alternatively, use `venv` with `python3 -m venv <env_name>` to create a virtual environment if you're not using `pyenv`.
• Activate Environment: - Activate the virtual environment with `source <env_name>/bin/activate` (Linux/macOS) or `<env_name>\Scripts\activate` (Windows).

• Install Dependencies: - Install required libraries using `pip install <package_name>`.
- If using `pipenv`, run `pipenv install <package_name>` to handle both dependencies and environment setup.
• Work on Project: - Develop and test your project within the activated virtual environment, ensuring that all dependencies are contained.
• Deactivate When Done: - Deactivate the virtual environment with `deactivate` when you have finished working on your project.
• This workflow ensures that you are working within a controlled environment, keeping your projects and their dependencies well-managed.

8. Installing python on linux

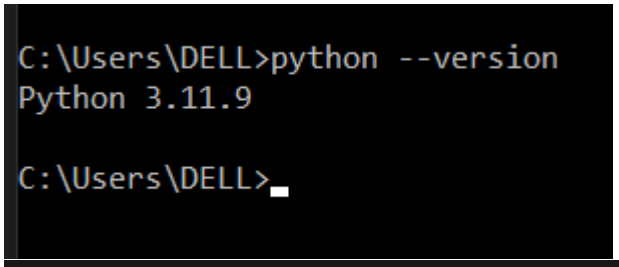
In the terminal we can install it in the following ways:

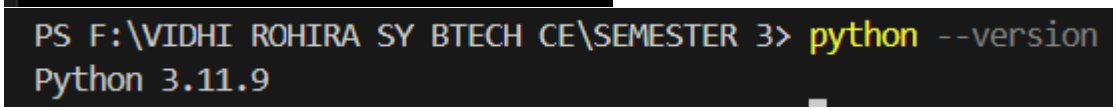
• sudo apt update
• sudo apt install python3
• sudo apt install python3-pip
• python3 --version

```
sudo apt update
sudo apt install python3
sudo apt install python3-pip
```

9. Installing python on windows

cd C:\Users\YourUsername\Downloads
python-3.x.x-amd64.exe
python-3.x.x-amd64.exe /quiet InstallAllUsers=1 PrependPath=1
python –version





10. Virtual Enviornments:

A Python virtual environment is a self-contained directory that includes a specific Python installation and set of packages. It isolates project dependencies, ensuring that different projects do not interfere with each other. This setup is crucial for managing projects with varying requirements.

Key Features:

• Isolation: - Virtual environments are separate from the global Python environment and other virtual environments, preventing conflicts between packages.

• Independent Dependencies: - You can install and manage packages specific to each project, avoiding version conflicts.

• Multiple Python Versions: - You can create virtual environments with different Python versions, useful for testing and deploying projects without altering the system-wide Python.

• Portability: - Share your environment setup via a `requirements.txt` file, allowing others to recreate the same environment.

• Project-Specific Customization: - Virtual environments support project-specific settings, simplifying the management of complex applications.

11. How to create a virtual environment

To create a virtual environment you need to run the following codes.

```
cd path/to/your/project
```

```
python -m venv env
```

```
.\env\Scripts\activate
```

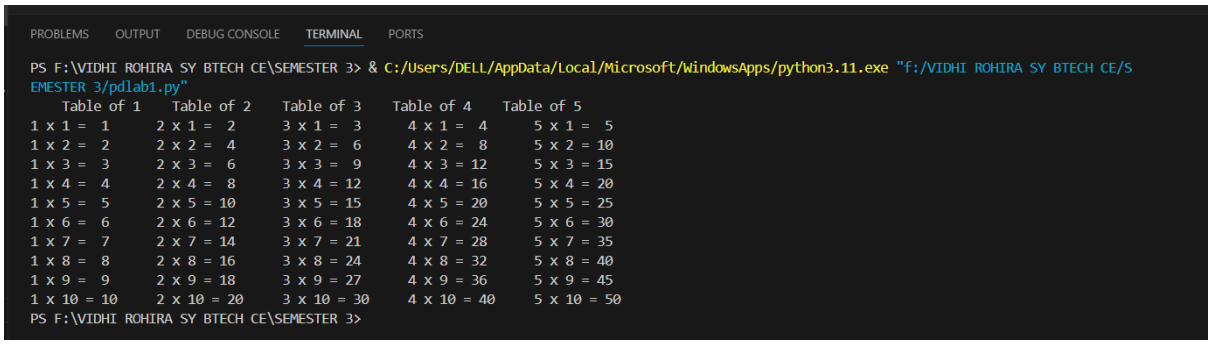
12. Code for table in formatted way.

```
numbers = [1, 2, 3, 4, 5]

print("    " + "    ".join(f"Table of {n}" for n in numbers))

for i in range(1, 11):
    row = []
    for num in numbers:
        row.append(f"{num} x {i} = {num * i:2}")
    print("\t".join(row))
```

OUTPUT:-



CONCLUSION:-

Hence we have studied about python, its different versions and how to install them. We have also learnt about virtual environments and they're applications and importance. Implemented and installed the above. Also learnt how to format lines by an example of maths tables.