Understanding Python Environments Part 1

Facilitator: Alberto Camarena

Presented at: Global Hack Week 🌐 Beginners

Date: 08/08/25 📅

🎯 Goals for This Session

By the end of Part 1, you will be able to:

- Explain what’s needed to run Python.
- Distinguish between interpreter, environment, and project.
- Create and manage isolated Python environments.
- Use `venv`, `virtualenv`, `conda`, and `poetry` basics.
- Troubleshoot common environment issues.

🧠 Why This Matters

text

- **Reproducibility**: Ensure code works the same everywhere.
- **Isolation**: Prevent one project’s dependencies from breaking another.
- **Collaboration**: Share exact setups with your team.
- **Portability**: Move code between machines and platforms smoothly.

💡 Think of it like a chef’s kitchen:

> Every recipe (project) needs its own set of ingredients (packages) and tools (Python version).

🛠️ What's Needed to Run Python

To execute Python code, you need:

1. **The Python Interpreter**
    - The program that reads and runs `.py` files.
2. **Standard Library**
    - Comes bundled with Python (e.g., `math`, `os`).
3. **Third-Party Packages** *(optional but common)*
    - Installed via `pip`, `conda`, etc.
4. **Environment Context**
    - Where Python looks for packages (`sys.path`).

⚙️ Key Concepts

Interpreter vs Environment

- Interpreter: The actual `python` executable.
- Environment: The interpreter **plus** the installed packages it can access.

Global vs Virtual

- Global: Shared system-wide, risk of conflicts.
- Virtual: Isolated to a folder/project.

🔍 How Python Finds Packages

When you run:

```python
import requests
```

Python searches:

1. Built-in modules.
2. Site-packages folder of the current environment.
3. Any paths in `sys.path`.

Run this in a Python shell:

```python
import sys
print(sys.executable)
print(sys.path)
```

📦 Environment Tools Overview

Tool	Manages Python version?	Manages packages?	Handles non-Py deps?
venv	No	✅	No
virtualenv	No	✅	No
conda	✅	✅	✅
poetry	No	✅	No
pyenv	✅	No	No

💻 Hands-On Station A – `venv` Basics

```
# Create
python3 -m venv .venv

# Activate
source .venv/bin/activate   # Mac/Linux
.venv\Scripts\activate      # Windows

# Install packages
pip install requests

# Freeze dependencies
pip freeze > requirements.txt

# Deactivate
deactivate
```

💻 Hands-On Station B – `conda` Basics

```
# Create environment with specific Python version
conda create -n myenv python=3.10

# Activate
conda activate myenv

# Install packages
conda install numpy pandas

# Export environment
conda env export > environment.yml

# Deactivate
conda deactivate
```

🖥️ Hands-On Station C – `poetry` Basics

```
# Initialize project
poetry init

# Add dependencies
poetry add requests

# Install from lock file
poetry install

# Run commands in env
poetry run python script.py
```

🛑 Common Pitfalls

- Installing packages without activating the correct environment.
- Forgetting to pin versions → "it worked yesterday..."
- Mixing `pip` and `conda` without care.
- Deleting env folder without updating docs.

🩺 Quick Troubleshooting

- **Which Python am I using?**

```
which python    # Mac/Linux
where python    # Windows
```

- **Which pip is active?**

```
pip --version
```

- **List installed packages**

```
pip list
```

📝 Activity: Debug This!

Given:

```
python script.py
ModuleNotFoundError: No module named 'flask'
```

1. Check active environment.
2. Install in the correct env.
3. Document changes.

📚 References

- Python venv docs
- virtualenv docs
- conda docs
- poetry docs

🚀 End of Part 1

Next:

Advanced Usage, Integrations & Best Practices

See you on part 2