Introduction to Python Environment Setups for AI Frameworks

Metadata

Title	Introduction to Python Environment Setups for AI Frameworks
Host	Onur Olgaç, MA BSc.
Date/Time	2025-03-07, 16:30 - 19:30 CEST
Location	Everyday A.I. – Promptival @Reaktor Geblergasse 40, 1070 Wien
Context	Python Development, AI Frameworks, Beginner
Who	WKO Fachgruppe Werbung Wien

Introduction

What are we trying to achieve today?

1. **Understand** what Python is and how you can utilize it to benefit your projects.
2. **Understand** some terminology related to its technologies.
3. **Understand** what a development environment is.

By the end of the workshop:

- Run Python environments for frameworks and Python-based tools (both dev and production).
- *Hopefully*, walk away with a running framework environment.

We are **not** going to learn how to code in Python:

- But we will try to get familiar with debugging Python errors.
- Learn how to conduct your own research, find answers to issues, problems, or questions.

Chapter 0: Background & Concepts

What Is Python?

Python is a high-level, general-purpose programming language first released in 1991. It is coincidentally open-source licensed, providing developers with comprehensive standard libraries, often described as "batteries included," meaning all necessary tools are available out of the box. Python is known for its strong abstraction, using natural language elements to make development simpler and more understandable.

Why Are We Talking About Python?

Python has gained widespread adoption due to its popularity among programmers, making it a leading choice in recent years. Its core philosophy, as seen in the Zen of Python (`python >>> import this`), emphasizes simplicity and efficiency. Python is a multi-paradigm language that supports structured (procedural), object-oriented, and functional programming paradigms at different levels of support. It offers interoperability as a glue-language to connect various software components.

Python's extensive library collection includes support for internet-based protocols, making it easy to interface with apps. It has also gained significant traction in the machine learning community due to libraries like TensorFlow (from Google Brain, an open-source project) and PyTorch (originally from Meta AI, currently maintained by the PyTorch Foundation under the Linux Foundation). Developers can leverage these tools for data manipulation and analysis.

Programming vs. Coding vs. Scripting

Programming involves designing and implementing complete software solutions as part of software engineering. This process is a higher level of complexity compared to coding, which focuses on writing actual code without necessarily considering design or planning. Coding is more about execution, while scripting is used for automating tasks, interacting with applications, and performing one-time operations. Python excels in this area, allowing developers to write scripts that can be directly executed by an interpreter.

Although probably a gross oversimplification, by definition we can make the following order in terms of scope: Programming > Coding > Scripting.

Interpreters vs. Compilers

Traditional programming languages like C++ and Java require a compilation process: source code is written, pre-processed, compiled into machine-specific assembly code, linked with other components, and then executed as a binary file. In contrast, Python uses an interpreter that reads and executes code line by line in a read-eval-print loop (REPL), typically through a command-line interface.

“Hello World” in C/C++ vs. Python

`C`:

```c
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

```
gcc hello.c -o hello
./hello
```

`C++`:

```cpp
#include <iostream>

int main() {
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

```
g++ hello.cpp -o hello
./hello
```

`Python`:

```python
print("Hello, World!")
```

```
python hello.py
```

Chapter 1: Versioning in Python & It's Modules

As of writing, the latest official Python version is `python3.13.2`, which is the most recent stable release available for download and installation. This distinction is important because it signifies that users can rely on this version for both reliability and new features.

Python versions follow a versioning scheme of `major.minor.micro` which categorizes updates with dot (.) separated numbers. Major versions (`Python 3`) represent more significant changes and innovations in the language, while minor versions (`Python 3.12`) often introduce new features or capabilities within a major release framework. Micro versions (`Python 3.12.2`), on the other hand, are incremental updates that address specific issues or improvements.

The deprecation process for Python versions is structured such that major versions take 2 to 3 years before being deprecated. Minor versions follow a shorter deprecation period of 6 to 12 months, and micro versions are updated more frequently. This structured approach ensures that users have advance notice of changes they need to plan for, while still providing support for older versions through an extended period (often referred to as Long-Term Support, or LTS).

On Unix-based operating systems like macOS and Linux, Python is typically pre-installed as part of the system software. However, users might opt to install it separately during software setup processes or specifically for a project they are working on. This flexibility allows users to manage their Python environments according to their specific needs.

Exercise #1

1. **Open Your CLI Tool**: Use the terminal (macOS), Command Prompt (Windows), Windows Terminal (Windows), Konsole (KDE), or a GNOME-terminal on Linux to access the command-line interface.
2. **Check Python Installation**: Verify if Python is installed by typing `python3 --version` in the terminal. This will display the current version of Python if it is installed.
3. **Ensure Updated Version**: If Python is not installed or an older version is present, download the latest stable version from HTTPS://WWW.PYTHON.ORG/DOWNLOADS/ and follow the installation guide provided.
4. **Test Functionality** by completing the following tasks within the terminal:
   - [ ] Print "Hello World" either within the interpreter, or by creating a `hello.py` file where you write the code and run it with python in your terminal.
   - [ ] Display the core philosophy (aka. "Zen of Python"). This is an Easter egg built into Python that you can do a quick search about.
   - [ ] Make some calculations within the Python interpreter.

By following these steps, you will be able to verify and manage Python installations across various operating systems and execute basic commands to test functionality. This exercise serves as a foundational step towards more complex tasks involving Python development and maintenance.

Chapter 2: Understanding Python Libraries, pip, and Virtual Environments

Python provides a robust ecosystem for managing libraries and packages, enabling developers to extend functionality beyond the core language. This system is largely facilitated by `pip`, and the PYTHON PACKAGE INDEX, which serves as the primary repository for Python packages.

1. Introduction to pip: `pip` is a command-line tool that simplifies the installation of Python packages. It allows users to download and install libraries from PyPI (Python Package Index), ensuring that packages are easily accessible and up-to-date. `pip` is managed by PyPA, which has taken over from the original developers, enhancing its reliability and adoption.

2. Installing Python Packages: When a user runs `pip install package_name`, `pip` retrieves the specified package from PyPI and installs it in the current Python environment. This process includes downloading the package code and setting up the necessary files for execution. Popular libraries like `numpy` (https://pypi.org/project/numpy/) are installed this way, making them readily available for use in scripts.

3. Managing Multiple Python Versions: One of the challenges in using Python is managing multiple versions on a single machine. To address this, virtual environments (`venv`) were introduced to isolate different projects with their own set of packages and dependencies. This ensures that updating or reinstalling packages doesn't affect other projects.

4. Virtual Environments: `venv` is the default virtual environment tool for Python 3.3 and later. It creates isolated directories, each containing their own Python interpreter, libraries, and scripts. By activating a specific virtual environment, users can manage dependencies without interfering with other environments or system-wide installations.

5. Global vs Local Installations:

- **Global Installation:** Installing a package globally makes it available to all users on the system. This is useful for shared dependencies but may lead to conflicts if multiple projects require different versions of the same library.
- **Local Installation (Virtual Environment):** Using virtual environments, packages are installed locally, ensuring that each project has control over its own dependencies. This prevents version conflicts and allows for pinning specific package versions as needed.

6. Package Version Management: Within a virtual environment, `pip` allows users to specify exact versions of libraries. This is particularly useful when projects require certain versions of packages for compatibility or functionality. Pinning versions can be done using flags in the `pip install` command, ensuring that specific dependencies are maintained across different environments.

7. Workflow and Best Practices:

- **New Projects:** It's advisable to create a new virtual environment for each project to manage dependencies effectively.
- **Updates:** Use `pip update` within an environment to ensure all packages are up-to-date without affecting other projects.
- **Conflict Resolution:** If version conflicts arise, consider using tools like `poetry` or `pipenv`, which provide more advanced dependency management and isolation. *This is a tip for the future, when projects grow in complexity.*

Understanding these concepts allows developers to manage their Python environments efficiently, ensuring that each project runs smoothly with the correct dependencies. This approach not only enhances productivity but also improves maintainability across different projects.

Exercise #2

- [ ] Ensure pip is installed.
  - [ ] Run `python3 -m pip --version` to verify.
  - [ ] Use `python3 -m ensurepip --upgrade` to install the latest version of pip (if necessary).
- [ ] Create a virtual environment in a new project folder
  - [ ] Change into the directory: `cd project`
  - [ ] Create the virtual environment: `python3 -m venv myenv`
  - [ ] Activate the virtual environment: `source myenv/bin/activate`
- [ ] Fetch a package from PyPI via the website or use `pip search <package-name>`.
  - [ ] Install the package locally in the virtual environment: `pip install <package-name>` within the activated environment.
- [ ] Print out the installed packages: `pip list` or `python3 -m pip list`.

Chapter 3: Understanding Dependencies in Python

Dependencies in programming are essential components that enable functionality beyond the core language features. In Python, external libraries provide additional functionalities needed for complex applications.

This overview explains the concepts of dependencies, tools like pip, and `requirements.txt`, offering a comprehensive understanding of managing these elements effectively.

1. What Are Dependencies? Dependencies are external libraries or packages that your Python application relies on to function. Packages like `numpy` for numerical computations, `pandas` for data manipulation, and `matplotlib` for plotting are common dependencies.

2. Why Are Dependencies Important? Dependencies enrich functionality by extending Python's built-in capabilities with specialized functionalities. They also allow reusability and maintainability of code by separating core logic from external features, making it a modular approach.

3. How To Manage Dependencies? `requirements.txt` is a text file that lists all packages required for a project. This file has a full list of each package that the environment needs to

have installed, along with its specific, minimum, or maximum version, with the following syntax: `numpy==1.21.2`. This allows for us to maintain consistency across environments.

Once this file is created, the listed packages can be installed using the command `pip install -r requirements.txt`, which downloads and installs all the specified packages in one step. You can then use `pip install --upgrade <package>` or `pip freeze | grep <package>` to update or fix the version of specific packages within your environment.

If multiple packages depend on different versions of a common package, pip resolves conflicts based on priority and context.

Dependencies are crucial for building powerful Python applications, providing essential functionalities that enhance the language's capabilities. Using pip and `requirements.txt`, developers can effectively install and manage these dependencies, ensuring projects remain modular, maintainable, and scalable across different environments.

Exercise #3

- [ ] Go back to the project on your computer.
- [ ] Take note of all the packages you have installed manually. *(tip: list installed packages)*
- [ ] Uninstall all packages.
- [ ] Re-install them via pip using `requirements.txt` instead.
- [ ] Print out the installed packages again.
- [ ] Examine the `requirements.txt` file from the GitHub respository of STABLE-DIFFUSION-WEBUI
    - [ ] Note down particularities you can spot.

Chapter 4: Setting Up an AI Framework

Now that the fundamentals of Python environments are covered, let's begin by determining whether you already have a specific AI framework in mind that you wish to utilize or if you are open to exploring new ones.

The following steps are also the required steps for the 4th exercise:

1. Ensure that whatever framework you choose is compatible with and can operate effectively within a Python environment.
2. If you do not have any framework that you are interested in already, consider one of the following recommendations, in order of complexity (in terms of setting up, clarity in documentation, and maintenance):

- ComfyUI: Ideal for those seeking a flowchart and graph-based user interface for AI tasks.
- Stable Diffusion WebUI: A powerful tool for generating synthetic images using Stable Diffusion technology.
- TTS Generation WebUI: Suitable for text-to-speech applications, offering a user-friendly web interface.

③ Obtain detailed instructions on ***manually installing*** the selected framework. This ensures that you can set it up independently, especially if it is not provided through standard package managers.

④ Investigate and document all dependencies required for the chosen framework. This includes additional libraries, APIs, or tools necessary to function optimally.

⑤ Identify the specific version of Python that the framework recommends or requires. If it is different than your currently installed version, find and install it on your OS.

⑥ If all goes well, once installed, launch the framework within your Python environment. Begin experimenting with its features or integrating it into your existing projects.

⑦ Many Python-based frameworks thrive on the modularity of Python and how extensive the ecosystem is. Consider enhancing the framework's capabilities by adding supported "3rd party" extensions, or scripts. This allows you to tailor the AI framework to specific needs, whether for image generation, speech synthesis, or other tasks. Any project that natively supports this will have a dedicated section with instructional write-ups in its documentation.

Chapter 5: Advanced (Bonus) Topics

There are alternatives to `pip` and `python-venv` which we have covered in this workshop that are either considered faster (better performing) implementations, or solutions that bring ease of use.

uv is a `pip` alternative, written in Rust, that is gaining a lot of popularity. Some newer frameworks even opt to use it as their recommended tool for python package management (such as InvokeAI).

conda is another package management tool designed to manage packages and their dependencies, alongside other extended functionality, that is part of the larger anaconda project.

With the conda package and environment manager, you have the capability to manage multiple Python versions without having to install them globally on your system, just like you can with any other Python module.

You can consider these alternatives as they offer additional quality of life improvements when it comes to Python environment management, and in some cases they might also be the best

option for specific frameworks as the recommended way of installation.

Conclusion

By the end of this workshop, participants have had the chance to build a solid understanding of Python environment setups and maintenance to handle complex dependency trees, manage multiple environments, debug common errors, and maintain their workflow in the face of frequent updates.

This guide should be taken as reference to be adapted to participants' specific needs, such as focusing on a particular library or including hands-on exercises with real-world projects. All commands and examples listed here are purely for demonstration, and might require revisions to be applicable to specific project/framework needs.

Vienna, 2025.