📘 1. Step-by-Step Guide: Installing Your Local Python Module Into Your Own Projects

Goal

Install and reuse your own Python module across multiple projects without publishing it to PyPI, using a local editable install.

Step 1 — Create your module structure

Your folder should look like this:

my_module
→ my_module
→ init.py
→ helpers.py
→ setup.py

The outer folder is the project root.

The inner folder is the actual importable module.

Step 2 — Create a setup.py file

Inside the top-level folder, create a file named setup.py with the following content:

```
from setuptools import setup, find_packages
setup(
name="my_module",
version="0.1.0",
packages=find_packages(),
)
```

This tells Python how to install your module.

Step 3 — Install your module locally

Open a terminal and navigate to the folder containing setup.py.

Example:

```
cd /path/to/my_module
```

Install the module in editable mode:

```
pip install -e .
```

Editable mode means:

• You can edit the module's code
• All projects see updates instantly
• No reinstall needed

Step 4 — Import your module in any project

Now you can use it anywhere in your system:

```
import my_module
```

Step 5 — Update your module anytime

Just edit the files inside your module folder.

Because you used "pip install -e", all changes are instantly available in every project.

Step 6 — Optional: Put your module under version control
If you want to track changes:
git init
git add .
git commit -m "Initial version"

📚 2. Full Documentation-Style Version
Below is a polished, professional documentation page suitable for internal use or PDF export.

Using a Local Python Module Across Multiple Projects (Without Publishing to PyPI)
Overview
This document explains how to install and reuse a private Python module across multiple projects using a local installation method. This avoids publishing to PyPI while still providing a clean, maintainable workflow.

1. Directory Structure
A typical module layout looks like this:
my_module
→ setup.py
→ my_module
→ init.py
→ helpers.py
The outer folder is the project root.
The inner folder is the actual importable package.

2. Creating the setup.py File
A minimal but complete setup.py file should contain:
from setuptools import setup, find_packages
setup(
name="my_module",
version="0.1.0",
packages=find_packages(),
)
Explanation of fields:
• name — the import name used by Python
• version — helps track updates
• packages — automatically includes all subpackages

3. Installing the Module Locally
Navigate to the module's root directory:
cd /path/to/my_module
Install the module using pip:
pip install -e .

Editable mode (“-e”) links the module instead of copying it.
This allows immediate updates across all projects.

4. Using the Module in Your Projects
Once installed, import it normally:
import my_module
Your module behaves exactly like a PyPI package — without being published.

5. Updating the Module
Because editable mode is used:
• Modify the source files directly
• No reinstall required
• All dependent projects automatically see updates
This makes the workflow extremely efficient for internal tools and shared utilities.

6. Optional: Version Control
To maintain history and collaborate:
git init
git add .
git commit -m "Initial version"
You may later push this to a private Git server if desired.

7. Summary
Using “pip install -e” provides:
• A clean installation method
• No need for PyPI
• Automatic updates across projects
• Professional packaging structure
• Compatibility with all Python tooling
This is the recommended approach for internal modules, automation libraries, and reusable codebases.

If you want, I can also create:
• A PDF-ready cover page
• A table of contents
• A version of this formatted like official Python documentation
• A multi-module architecture guide
Just tell me how polished you want it.