Tutorial III.III - Package Management

Programming: Everyday Decision-Making Algorithms

Introduction

Welcome to this guided tutorial on understanding packages and package management in Python! We'll explore this concept through the lens of library management - how libraries organize, maintain, and share their resources.

Think of Python as your local library's main building. Just like a library has:

- A main collection (Python's standard library)
- Different sections (third-party packages)
- A catalog system (a package manager — we use `uv`)
- Multiple branches (virtual environments / project folders)

In more detail

Let's break this down:

1. Standard Library: Like the core collection every library branch has:
   - Python comes with essential tools built-in
   - Data structures (lists, dictionaries, etc.)
   - Mathematical functions (sin, cos, sqrt, etc.)
   - Random numbers (random, choice, etc.)
2. Third-Party Packages: Similar to specialized sections in a library:
   - Data analysis packages (pandas, numpy) → Research section
   - Visualization packages (matplotlib, seaborn) → Art books
   - Web frameworks (Django, Flask) → Technical manuals
   - Machine learning packages (scikit-learn) → Advanced studies section
3. Package Manager (`uv`): Works like the library's catalog:
   - Installing packages → Ordering new books
   - Updating packages → Replacing with newer editions
   - Uninstalling packages → Removing outdated books
   - Dependencies → "You must read Book A before Book B"
4. Project Environments (managed automatically by `uv`): Like different library branches:
   - Each branch can have its own collection of books (packages)
   - Different editions of the same book (package versions)
   - Specialized sections for different purposes (project-specific dependencies)

The best part? Most Python packages are free to use, thanks to the open-source community!

> **i** Note
>
> We use `uv` in this course for Python version management, virtual environments, and packages. See the separate guide for more details: uv setup.

Section 1 - Using Standard Libraries

Before we start using external packages, let's take a look at Python's standard libraries. Python comes with a set of standard libraries that are always available. You can use these libraries by simply importing them. For example, the `math` library provides mathematical functions like `sqrt` and `sin` and the `random` library provides functions to generate random numbers. You can import these libraries using the following syntax:

```
import math
import random
```

You can then use the functions provided by these libraries by calling them with the library name as a prefix. For example, the following code calculates the square root of 16.

```
a_square_root = math.sqrt(16)
print(f"The square root of 16 is {a_square_root}!")
```

```
The square root of 16 is 4.0!
```

The following code generates a random number between 0 and 1.

```
a_random_number = random.random()
print(f"A random number is {a_random_number}!")
```

```
A random number is 0.5686637368231802!
```

We can also use the library to choose a random element from a list. For example, the following code chooses a random element from the list `['book1', 'book2', 'book3']`. Try it out by executing the cell multiple times.

```
random_choice = random.choice(['book1', 'book2', 'book3'])
print(f"We chose {random_choice}!")
```

```
We chose book1!
```

Sometimes, we only need one function from a library. In this case, we can use the `from` keyword to import only the specific function we need. For example, the following code imports only the `randint` function from the `random` library and uses it to generate a random integer between 1 and 10.

```
from random import randint
a_random_integer = randint(1, 10)
print(f"A random integer between 1 and 10 is {a_random_integer}!")
```

```
A random integer between 1 and 10 is 8!
```

Other times, we might need to work with a library that has a rather long name. In this case, we can use the `as` keyword to give the library a shorter name. For example, the following code imports the `random` library and gives it the shorter name `rd`.

```
import random as rd
a_random_integer = rd.randint(1, 100)
print(f"Now, a random integer between 1 and 100 is {a_random_integer}!")
```

```
Now, a random integer between 1 and 100 is 59!
```

Exercise 1.1 - Use the math library

Use the `math` library to calculate the square root of 256. Call the result `square_root`.

```
# YOUR CODE BELOW
```

```
assert square_root == 16
print(f"Great job! The square root of 256 is {square_root}!")
```

Exercise 1.2 - Use the random library

Use the `random` library to generate a random number between 1 and 25. Call the result `random_number`.

```
assert random_number >= 1 and random_number <= 25
print(f"Good! You generated a random integer {random_number} between 1 and
25!")
```

Section 2 - Using a package manager (`uv`)

Just as a library needs a system to track books (checkout system, catalog, etc.), Python projects need a tool to manage versions and dependencies. We use `uv` because it:

- Keeps track of what's available (like searching the catalog)
- Handles "checkouts" (installing packages)
- Manages "returns" (uninstalling packages)
- Ensures you have the right "edition" (version management)

Exercise 2.1 - Check your `uv` installation

Open a terminal (not the Python REPL) and run:

```
{bash}
uv --version
```

You should see a version number.

Then confirm your Python version (after installing one via `uv` if needed):

```
{bash}
uv run python --version
```

Section 3 - Installing Packages

Installing packages is like ordering new books for your library branch. With `uv` you typically:

- Initialize a project once (`uv init`)
- Add dependencies (`uv add packagename`)
- Remove them if not needed (`uv remove packagename`)
- Sync a project copied from elsewhere (`uv sync`)
- Update dependencies (`uv update`)

Under the hood `uv` creates (and reuses) a virtual environment for the project. Luckily most of that is handled for you and not that important in this basic introduction to Python.

Exercise 3.1 - Add `pandas`

Inside a `uv`-initialized project directory, run:

```
{bash}
uv add pandas
```

This: - Resolves dependencies - Adds `pandas` to `pyproject.toml` - Installs it into the project's virtual environment

You can test if the installation was successful by running:

```
# Test your answer
try:
    import pandas
    print("pandas installed successfully!")
except ImportError:
    print("pandas was not installed correctly")
```

(If you see an import error, verify you are inside the project folder you initialized with `uv init`.)

Virtual environments are like having different library branches. Each branch can have:

- Its own collection of books (packages)
- Different editions of the same book (package versions)

- Specialized sections for different purposes (project-specific dependencies)

This separation ensures that:

- Changes in one branch don't affect others (project isolation)
- Each branch can be optimized for its community (project-specific dependencies)
- You can experiment without affecting the main collection (development safety)

For now, you don't need to worry about virtual environments except for the one we created now. This is more advanced and thus not necessary for this tutorial (or lecture). But it's good to know that they exist and that you can use them to manage your packages.

Conclusion

Great work! You learned how Python projects organize and access code via:

- The standard library
- Third-party packages
- A modern package & environment manager (`uv`)
- Isolated project environments for reliability

Treat each project like its own library branch: well-labeled, independent, and easy to maintain.

Solutions

You will likely find solutions to most exercises online. However, we strongly encourage you to work on these exercises independently without searching explicitly for the exact answers to the exercises. Understanding someone else's solution is very different from developing your own. Use the lecture notes and try to solve the exercises on your own. This approach will significantly enhance your learning and problem-solving skills.

Remember, the goal is not just to complete the exercises, but to understand the concepts and improve your programming abilities. If you encounter difficulties, review the lecture materials, experiment with different approaches, and don't hesitate to ask for clarification during class discussions.

Later, you will find the solutions to these exercises online in the associated GitHub repository, but we will also quickly go over them next week. To access the solutions, click on the Github button on the lower right and search for the folder with today's lecture and tutorial. Alternatively, you can ask ChatGPT or Claude to explain them to you. But please remember, the goal is not just to complete the exercises, but to understand the concepts and improve your programming abilities.

That's it for this week! Next week, we'll take a look at how to read and write files in Python and work with tabular data!