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 (pip package manager)
- Multiple branches (virtual environments)

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 (pip): 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   - Uninstalling packages → Removing outdated books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. Virtual Environments: Works 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

While we'll practice some commands here, you'll typically manage packages in your terminal or IDE.

Section 1 - Using Standard Libraries

Before we start using 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 0.40296996463461965!
```

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.

```python
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`.

```python
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 49!
```

Exercise 1.1 - Use the math library

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

```python
# YOUR CODE BELOW
```

```python
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`.

```python
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

Just as a library needs a system to track books (checkout system, catalog, etc.), Python uses pip or conda to manage its packages. Think of pip or conda as your library's management system that:

- 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)

i Miniconda

As we have installed Miniconda in the lecture, we will use conda in this tutorial. If you work with a standard Python installation, you can use pip instead but the commands are slightly different.

Exercise 2.1 - Check your package manager version

Let's start by checking which version of conda we have installed. We do so by running the following command in our terminal. Make sure, you are not working with python yet in the terminal, as this command is not available in the python shell. To open a new terminal in VS Code, you can press `ctrl + shift + p` and then type `Terminal: Create new terminal`. Then, run the following command:

```
{bash}
conda info
```

If you are not using Miniconda, you can use `pip` instead. Here, you can check the version of your package manager by running the following command in the terminal:

```
{bash}
pip --version
```

Section 3 - Installing Packages

Installing packages is like ordering new books for your library branch. Just as a library might:

- Order books from publishers (pip install from PyPI)
- Get specific editions (version specifications)
- Check for space on shelves (system requirements)
- Ensure books don't conflict (dependency management)

Luckily most of that is handled for you and not that important in this basic introduction to python. Installing packages in Python is straightforward using `conda install packagename` if you are using Miniconda. If you are not using Miniconda, you can use `pip install packagename` instead.

Exercise 3.1 - Install pandas

Let's practice by installing pandas, a popular package for data analysis. Execute the following command in your terminal and make sure you are not in the python shell.

```
{bash}
conda install pandas
```

Or, if you are not using Miniconda, you can use the following command.

```
{bash}
pip install pandas
```

You can test if the installation was successful by running the following code.

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

Section 4 - Virtual Environments

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. 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. Still, if you ever want to check them out, you can use the following command to create a new virtual environment with a certain name and Python version.

```
{bash}
conda create -n myenv python=3.10
```

Note

You can replace `myenv` with any name you want and `python=3.10` with the Python version you want.

To activate the environment later on, you can use the following command:

```
{bash}
conda activate myenv
```

Conclusion

Congratulations! You've learned about Python package management through the lens of library caching. Remember, just like libraries use caching to make popular books more accessible, Python uses package management to make useful code more accessible!

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!