Python Packages:

1. What is a Package in Python?

- A **package** is a way of organizing related Python code into **modules** (files) and **sub-packages** (folders).
- It's essentially a **directory with an `__init__.py` file** (in Python 3.3+, this file can be optional, but its presence makes the folder explicitly a package).

Example structure:

```
my_package/
    __init__.py
    module1.py
    module2.py
    sub_package/
        __init__.py
        sub_module.py
```

---

2. Difference Between Modules and Packages

- **Module** → A single `.py` file with Python code.
- **Package** → A collection of modules in a directory (with `__init__.py`).

Example:

```
Start coding or generate with AI.
```

```
# main.py
import module1
print(module1.greet("Alice"))
```

---

3. Creating a Package

Let's create a package called `math_utils`.

```
math_utils/
    __init__.py
    arithmetic.py
    geometry.py
```

arithmetic.py

```
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b
```

geometry.py

```
import math

def area_circle(radius):
    return math.pi * radius * radius
```

****init**.py**

```
# Expose selected functions at package level
from .arithmetic import add, subtract
from .geometry import area_circle
```

Now you can use:

```
import math_utils

print(math_utils.add(5, 3))       # 8
print(math_utils.area_circle(2)) # 12.566...
```

---

4. Importing from Packages

Different ways to import:

```
import math_utils
print(math_utils.add(2, 3))

from math_utils import add
print(add(2, 3))

from math_utils.arithmetic import subtract
print(subtract(5, 2))

from math_utils import *
print(add(1, 1))  # Only works if __all__ is defined
```

```
Start coding or generate with AI.
```

`__all__` in `__init__.py`

```
__all__ = ["add", "subtract", "area_circle"]
```

This controls what gets imported with `from math_utils import *`.

Great point 👍 Let's expand on the `__all__` concept in the context of Python packages with more **details, examples, and tips**.

🔷 `__all__` in `__init__.py`

The special variable `__all__` is a list of strings that defines the **public API** of a module or package. It explicitly controls what is imported when someone uses:

```
from package_name import *
```

🧩 Example: `math_utils` package

Suppose we have this package structure:

```
math_utils/
│
├── __init__.py
├── basic_ops.py
└── geometry.py
```

basic_ops.py

```
def add(a, b):
    return a + b
```

```
def subtract(a, b):
    return a - b

def _secret_formula(x):   # intended to be private
    return x * 42
```

geometry.py

```
import math

def area_circle(r):
    return math.pi * r * r

def area_square(a):
    return a * a
```

`__init__.py`

```
from .basic_ops import add, subtract
from .geometry import area_circle

__all__ = ["add", "subtract", "area_circle"]
```

🚀 How it works

Now, when you import like this:

```
from math_utils import *
```

✅ Only these names are available:

```
add
subtract
area_circle
```

❌ Hidden:

- `area_square` is not imported.
- `_secret_formula` is not imported.

⚖️ Without `__all__`

If you **don't define** `__all__`, Python will import **all names that do not start with an underscore** (`_`).

So in our example:

```
from math_utils import *
```

would import `add`, `subtract`, `area_circle`, **and** `area_square`.

🎯 Why use `__all__`?

1. **Encapsulation** → hide helper functions (`_secret_formula`).
2. **Clean API** → only expose the important functions.
3. **Documentation** → makes it clear what's public.
4. **Avoid Conflicts** → prevents polluting the namespace with too many imports.

📝 Tips & Tricks

- Use `__all__` at the **package level** (`__init__.py`) to control what the package exports.

- Use it at the **module level** (`.py` files) if you want fine-grained control inside a module.
- Prefix helper functions with `_` and leave them out of `__all__`.
- Tools like **Sphinx** (for docs) respect `__all__` → it defines what shows up as "public".

✅ In short: `__all__` is your way of saying *"these are the official functions and classes I want people to use from my package."*

```
Start coding or generate with AI.
```

5. Nested (Sub) Packages

You can have sub-packages inside packages:

```
data_science/
    __init__.py
    statistics/
        __init__.py
        mean.py
```

statistics/mean.py

```
def mean(values):
    return sum(values) / len(values)
```

Usage:

```
from data_science.statistics.mean import mean
print(mean([1, 2, 3, 4]))  # 2.5
```

```
Start coding or generate with AI.
```

📝 Exercises on Python Module Creation

🔷 Beginner Level

Exercise 1: Create Your First Module

1. Create a file `mymath.py`.
2. Inside, write two functions:

   ```
   def add(a, b): return a + b
   def subtract(a, b): return a - b
   ```

3. In another file `main.py`, import `mymath` and use both functions.

Exercise 2: Import with Aliases

1. Extend `mymath.py` with a function:

   ```
   def multiply(a, b): return a * b
   ```

2. In `main.py`, import it as:

   ```
   import mymath as mm
   ```

3. Use `mm.multiply(3, 4)`.

Exercise 3: Selective Import

1. Add a new function `divide(a, b)` in `mymath.py`.
2. In `main.py`, import only this function with:

```
from mymath import divide
```

3. Test it.

🔷 Intermediate Level

Exercise 4: Use `__name__ == "__main__"`

1. Add the following block in `mymath.py`:

```
if __name__ == "__main__":
    print("Testing add:", add(2, 3))
```

2. Run `mymath.py` directly (what happens?).
3. Import it into `main.py` (does the test run?).

Exercise 5: Organize a Package

1. Create a folder `math_utils/`.
2. Inside, create:
   - `__init__.py`
   - `arithmetic.py` → (add, subtract, multiply, divide)
   - `geometry.py` → (area_circle, area_rectangle)
3. In `main.py`, use:

```
from math_utils.arithmetic import add
from math_utils.geometry import area_circle
```

Exercise 6: Control Exports with `__all__`

1. Inside `math_utils/__init__.py`, add:

```
__all__ = ["arithmetic", "geometry"]
```

2. Try:

```
from math_utils import *
```

→ Which functions can you access?

Exercise 7: Documentation & `__doc__`

1. Add docstrings for every function in your modules.
2. In `main.py`, print:

```
import math_utils.arithmetic as ar
print(ar.add.__doc__)
```

3. Confirm docstrings are accessible.

```
Start coding or generate with AI.
```

🔷 What does **pip** mean?

- **pip** is a **recursive acronym** (the letters refer back to itself).

- It stands for: "**Pip Installs Packages**" (or sometimes jokingly "**Pip Installs Python**").

That means its **only job** is to install and manage **Python packages** (libraries, modules, or frameworks).

🔷 What are Python packages?

A **package** in Python is basically a **collection of Python modules** (code files) bundled together so developers can reuse them easily.

- Example: `requests` → package for making HTTP requests.
- Example: `flask` → package for building web apps.
- Example: `numpy` → package for numerical computing.

Instead of everyone writing their own HTTP client, you just **install** `requests` with pip.

🔷 Where does pip get packages from?

By default, pip installs from **PyPI (Python Package Index)** → the official repository of Python software.

- URL: https://pypi.org
- It has **over 500,000 packages** (as of 2025).

You can also configure pip to install from:

- A **private repository** (like an internal company package index).
- A **local file** (wheel or tarball).
- A **GitHub repo**.

🔷 Why is pip important?

Before pip existed, installing packages in Python was painful:

- You had to download source code manually.
- Compile or copy it into your project.
- Manage versions by yourself.

With pip: ✅ One command installs everything ✅ Handles dependencies (installs other required packages automatically) ✅ Works with virtual environments for isolation

---

🔷 Quick Example

Let's say you want to use Flask for a web app:

```
pip install flask
```

What happens?

1. Pip contacts **PyPI**.
2. Finds the latest version of Flask.
3. Downloads the package files.
4. Installs Flask **and all dependencies** (like `Werkzeug`, `Jinja2`, etc.).
5. Makes it available in your Python environment.

Now you can use it:

```
from flask import Flask

app = Flask(__name__)
```

🔷 Summary

- **pip** = tool for installing/managing Python packages.
- Default source = **PyPI**.
- Simplifies dependency management.

- Essential for almost all Python development.

```
Start coding or generate with AI.
```

6. Installing and Using External Packages

Python has a huge ecosystem of external packages available on **PyPI** (Python Package Index). To install them, use **pip**:

```
pip install requests
```

Usage:

```
import requests
response = requests.get("https://api.github.com")
print(response.status_code)
```

💡 **Tip:** Always use a **virtual environment** to isolate package installations.

```
# Create venv
python -m venv venv

# Activate
source venv/bin/activate   # Linux/macOS
venv\Scripts\activate      # Windows
```

Double-click (or enter) to edit

10. Tips & Tricks

1. **Check installed packages**

```
pip list
```

2. **Show package location**

```
import requests
print(requests.__file__)
```

3. **Reload a package**

```
import importlib
import my_package
importlib.reload(my_package)
```

4. **Inspect a package**

```
import math_utils
print(dir(math_utils))
```

5. **Relative imports (inside a package)**

```
from .arithmetic import add
```

---

11. Summary

- A **module** = single Python file.
- A **package** = collection of modules in a folder (with `__init__.py`).

- Supports **sub-packages**.
- Use **pip** to install external packages.

```
Start coding or generate with AI.
```

📝 Exercises on `pip` and Python Packages

🔷 Beginner Level

Exercise 1: Install and Import a Package

1. Install the `requests` package with pip.
2. Write a Python script that fetches the HTML content of `https://www.python.org` and prints the first 200 characters.

Exercise 2: Check Installed Packages

1. List all installed packages using pip.
2. Find the version of `pip` installed on your system.
3. Show the location where Python packages are installed.

Exercise 3: Freeze Environment

1. Use pip to generate a list of all installed packages (`requirements.txt`).
2. Create a new virtual environment and install all packages from that file.

🔷 Intermediate Level

Exercise 4: Specific Version Installation

1. Install version `2.26.0` of the `requests` library.
2. Verify the installed version inside Python using `requests.__version__`.
3. Upgrade it to the latest version.

Exercise 5: Uninstall a Package

1. Install `numpy` with pip.
2. Import it and confirm it works.
3. Uninstall it with pip.
4. Try importing it again (expecting an error).

Exercise 6: Install from GitHub

1. Use pip to install a package directly from a GitHub repository. Example:

```
pip install git+https://github.com/psf/requests.git
```

2. Verify that the installed package is working.

🔷 Advanced Level

Exercise 7: Work with a Private Package Index

1. Configure pip to install from a custom index URL (simulate with a dummy URL).
2. Explain when and why companies use private PyPI indexes.

Exercise 8: Editable Install (Development Mode)

1. Create a small Python package (`mypkg`) with a `setup.py` file.
2. Install it using:

```
pip install -e .
```

3. Modify the package code and confirm that the changes reflect immediately without reinstallation.

Exercise 9: Dependency Conflicts

1. Try to install two packages that require different versions of the same dependency.
2. Observe the error message pip gives.
3. Use a virtual environment to separate dependencies and solve the conflict.

Exercise 10: Advanced Requirements File

1. Create a `requirements.txt` file with:

```
requests==2.26.0
flask>=2.0.0
numpy
```

2. Install packages from this file.
3. Verify versions installed with `pip show <package>`.

✅ Solutions: Exercises on `pip` and Python Packages

🔷 Beginner Level

Exercise 1: Install and Import a Package

Step 1: Install requests

```
pip install requests
```

Step 2: Python script

```
import requests

url = "https://www.python.org"
response = requests.get(url)

print(response.text[:200])  # print first 200 characters
```

Exercise 2: Check Installed Packages

1. **List all installed packages:**

```
pip list
```

2. **Check pip version:**

```
pip --version
```

Example output:

```
pip 24.2 from /usr/local/lib/python3.10/site-packages/pip (python 3.10)
```

3. **Show package install location:**

```
pip show requests
```

(Look at the `Location:` field).

Exercise 3: Freeze Environment

1. Save installed packages:

```
pip freeze > requirements.txt
```

2. Create new virtual environment:

```
python -m venv myenv
source myenv/bin/activate   # Linux/macOS
myenv\Scripts\activate      # Windows
```

3. Install all packages:

```
pip install -r requirements.txt
```

🔷 Intermediate Level

Exercise 4: Specific Version Installation

1. Install version `2.26.0`:

```
pip install requests==2.26.0
```

2. Check version:

```
import requests
print(requests.__version__)
```

Output: `2.26.0`

3. Upgrade:

```
pip install --upgrade requests
```

Exercise 5: Uninstall a Package

1. Install numpy:

```
pip install numpy
```

2. Confirm:

```
import numpy as np
print(np.__version__)
```

3. Uninstall:

```
pip uninstall numpy -y
```

4. Try importing again:

```
import numpy
```

→ Raises `ModuleNotFoundError`.

Exercise 6: Install from GitHub

```
pip install git+https://github.com/psf/requests.git
```

Verify:

```
import requests
print(requests.__version__)
```

🔷 Advanced Level

Exercise 7: Work with a Private Package Index

Simulated example:

```
pip install --index-url https://example.com/simple/ mypackage
```

👉 Companies use private PyPI indexes to:

- Host internal libraries not public on PyPI.
- Control package versions.
- Improve security by avoiding external dependencies.

Exercise 8: Editable Install (Development Mode)

1. Create `mypkg` folder:

   ```
   mypkg/
       setup.py
       mypkg/
           __init__.py
           hello.py
   ```

   setup.py

   ```
   from setuptools import setup, find_packages

   setup(
       name="mypkg",
       version="0.1",
       packages=find_packages(),
   )
   ```

   mypkg/hello.py

   ```
   def say_hello():
       print("Hello from mypkg!")
   ```

2. Install in editable mode:

   ```
   pip install -e .
   ```

3. Modify `say_hello()` → changes reflect immediately without reinstall.

Exercise 9: Dependency Conflicts

Example:

```
pip install django==2.2
pip install djangorestframework==3.13
```

This may throw an **incompatibility error**.

Solution:

- Use virtual environments for each project:

```
python -m venv env1
python -m venv env2
```

Exercise 10: Advanced Requirements File

`requirements.txt`

```
requests==2.26.0
flask>=2.0.0
numpy
```

Install:

```
pip install -r requirements.txt
```

Check versions:

```
pip show requests
pip show flask
pip show numpy
```

```
Start coding or generate with AI.
```

Double-click (or enter) to edit