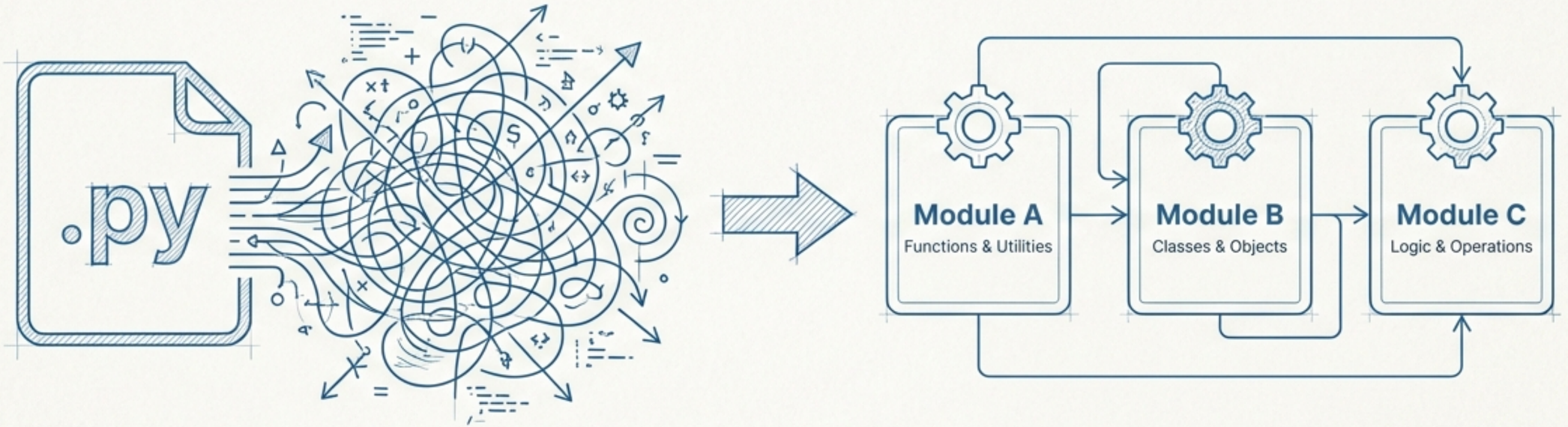
From Code to Craft: Mastering Structure in Python

A Blueprint for Building Organised, Reusable, and Scalable Applications

Every Large Project Starts with a Single File

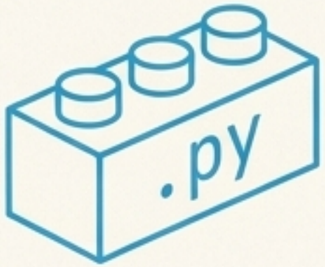
As a project grows, placing all your functions, classes, and logic into one file becomes unmanageable. The code becomes difficult to read, debug, and maintain.

Python's solution is modular programming—a structured approach to organising your code.



The bigger the project, the more crucial its structure.

The Foundational Unit: The Python Module



What is a module?

A module is simply a Python file with a .py extension.

What can it contain?

It can hold a collection of related items: variables, functions, and classes.

What is its purpose?

It allows you to logically group related code, making it clean, reusable, and easy to manage.

File: mymodule.py

```
# mymodule.py

def greet():
    print("Hello from the module")
```

Usage:

```
import mymodule

mymodule.greet()
```

Crafting and Using Your Own Module

The Blueprint

Step 1: Create the File

Create a new file named `mathutils.py`.

Step 2: Add Your Code

Define functions or classes within the file. Here, we'll add a simple `add` function.

Step 3: Import and Use

In a separate Python file, import your new module to access its functionality.

The Implementation

File: mathutils.py

```python
# mathutils.py
def add(a, b):
    return a + b
```

File: main.py

```python
# main.py
import mathutils
result = mathutils.add(5, 7)
print(result) # Output: 12
```

call

The Art of Importing: A Guide to Accessing Modules

1. Standard Import** (`import math`)

Imports the entire module. Access content with `module.function()`.

```
import math
print(math.sqrt(25))
```

2. Specific Import** (`from math import sqrt`)

Imports a specific function or class directly into the current namespace.

```
from math import sqrt
print(sqrt(25))
```

3. Alias Import** (`import math as m`)

Imports the module under a shorter, alternative name to avoid conflicts or save typing.

```
import math as m
print(m.sqrt(25))
```

4. Wildcard Import** (`from math import *`)

Imports all public names from a module.

```
from math import *
print(sqrt(25))
```

Pro-Tip: A Word of Caution

Avoid using wildcard imports (`from module import *`). They can pollute your namespace, lead to naming conflicts, and make your code harder to read and debug. Be explicit.

Inspecting the Toolbox: Discovering a Module's Contents

`dir()

Purpose

The `dir()` function returns a sorted list of names (variables, functions, classes) defined in a module. It's a quick way to see everything a module has to offer.

```
import math

# Shows a list including 'sqrt', 'pi', 'cos', etc.
print(dir(math))
```

`help()

Purpose

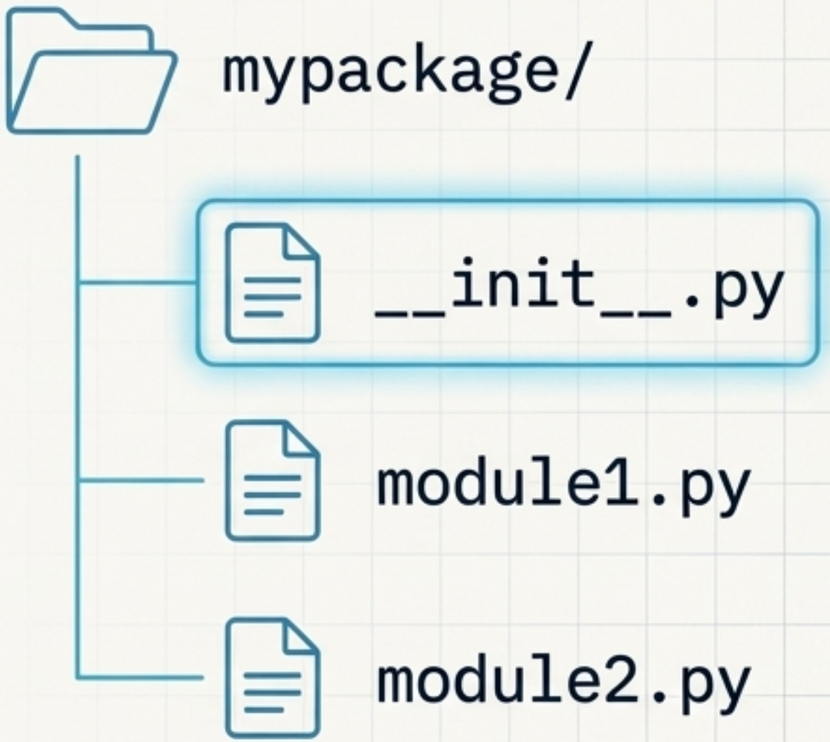
The `help()` function provides the official documentation (docstrings) for a module, giving detailed information on its functions and their usage.

```
import math

# Prints the full help documentation for the math module.
help(math)
```

Scaling the Blueprint: From Modules to Packages

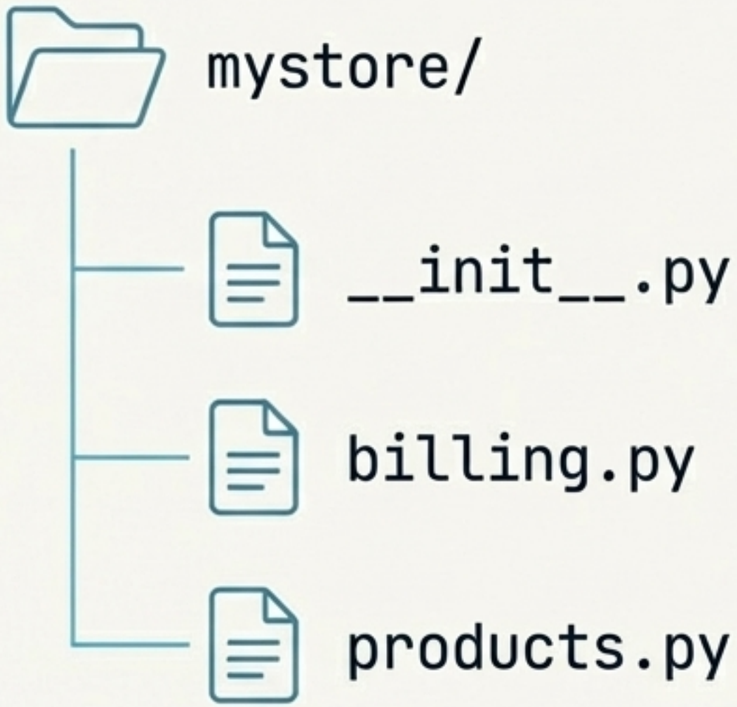
When your project grows to include multiple related modules, you need a way to group them. A package is the answer: it's a folder containing modules and a special `__init__.py` file.

> **Key Concept:** The `__init__.py` file tells Python that the folder should be treated as a package. It can be empty or contain initialisation code for the package.



Constructing Your First Package: A Step-by-Step Guide

Directory Structure



Code Example

File: mystore/billing.py

```
# mystore/billing.py
def calculate_total_price(items):
    # ...imagine complex logic here...
    return 100
```

File: main.py

```
# main.py
# Import the billing module from the mystore package
import mystore.billing
total = mystore.billing.calculate_total_price([])
print(f"Total is: {total}")
```

Accessing Code Within a Package

You can import specific modules or even individual functions directly from a package to make your code more concise.

Module Import

Import the entire module and call the function using dot notation.

```
import mystore.billing

total = mystore.billing.calculate_total_price([])
print(total)
```

Function Import

Import the function directly. This is often cleaner and more readable.

```
from mystore.billing import calculate_total_price

total = calculate_total_price([])
print(total)
```

Defining a Public API with `__all__`

The `__all__` list in a package's `__init__.py` file defines which modules are imported when a user performs a wildcard import (`from package import *`). It's a way to explicitly declare your package's public interface and prevent internal helper modules from being exposed.

File: mystore/__init__.py

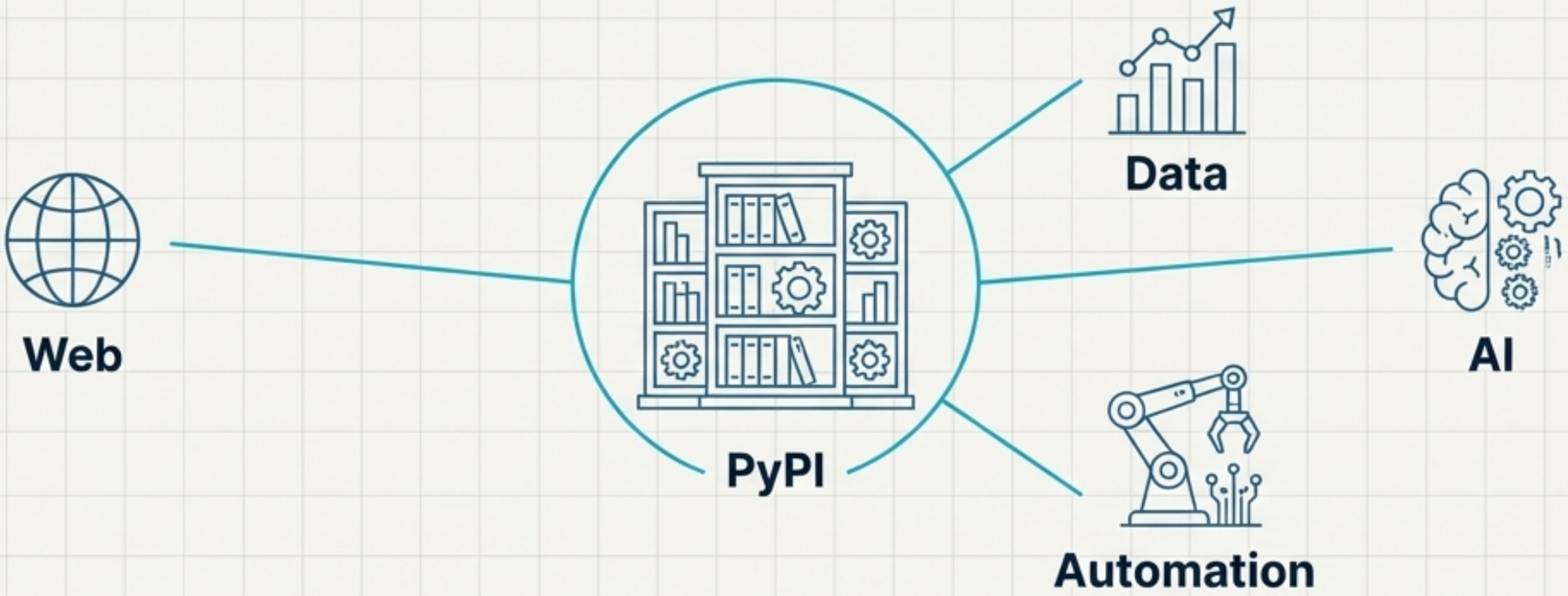
```
# mystore/__init__.py

# This specifies that only 'billing' and 'products'
# should be imported when using 'from mystore import *'

__all__ = ["billing", "products"]
```

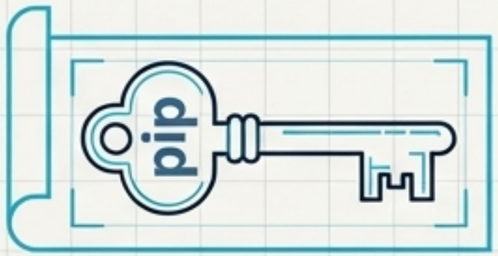
Pro-Tip

Using `__all__` is excellent practice for library authors as it creates a clear and stable API for users.

The Universe of Code: The Python Package Index (PyPI)

You don’t have to build everything from scratch. The Python community has created **hundreds of thousands of packages** for tasks ranging from web development to data science. This ecosystem is called **PyPI** (the Python Package Index).





Key Tool: `pip` is the standard package manager for Python. It is the command-line tool you use to install, update, and remove packages from PyPI.

Your Essential Toolkit: Managing Packages with `pip`

Install a Package

```
pip install <package_name>
```

```
pip install requests
```

```
pip install numpy
```

Uninstall a Package

```
pip uninstall <package_name>
```

```
pip uninstall requests
```

List Installed Packages

```
pip list
```

*(Shows a table of packages and their versions)

Show Package Details

```
pip show <package_name>
```

```
pip show numpy
```

*(Shows version, author, location, etc.)

Putting it all Together: Using a Third-Party Package

Let's use the popular `requests` library to fetch data from a public API on the internet.

Step 1: Installation

```
pip install requests
```

Step 2: The Code

```
# main.py

import requests

# Make a GET request to the GitHub API
response = requests.get("https://api.github.com")

# Print the status code and the JSON content
print(f"Status Code: {response.status_code}")
print(response.json())
```

With just a few lines of code, you can leverage a powerful, professionally built library to perform a complex task.

The Payoff: Why This Craftsmanship Matters

Code Reusability

Write code once and use it across multiple projects.

Project Organisation

A clean, logical structure makes code easier to navigate and understand.

Maintainability

Isolating functionality makes debugging and updating far simpler.

Collaboration

A clear structure allows teams to work on different parts of an application simultaneously.

Ecosystem Power

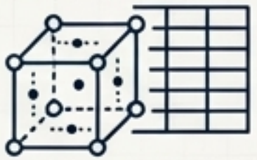
Unlocks access to the vast universe of third-party libraries.

Structure in Action: Real-World Python Architectures

The principles of modules and packages are the foundation upon which the most powerful Python applications and libraries are built.

Web Frameworks

Django and Flask are complex packages that structure entire web applications.



Data Science & ML

Libraries like NumPy, Pandas, and Scikit-learn are meticulously organised packages for scientific computing.

Automation & Scripting

Tools for tasks like API integration or network automation are built as modular scripts and packages.

Security Tools

Ethical hacking tools like Scapy and Requests-based clients are packaged for easy distribution and use.

Mastering structure is the first step to building professional-grade software in Python.