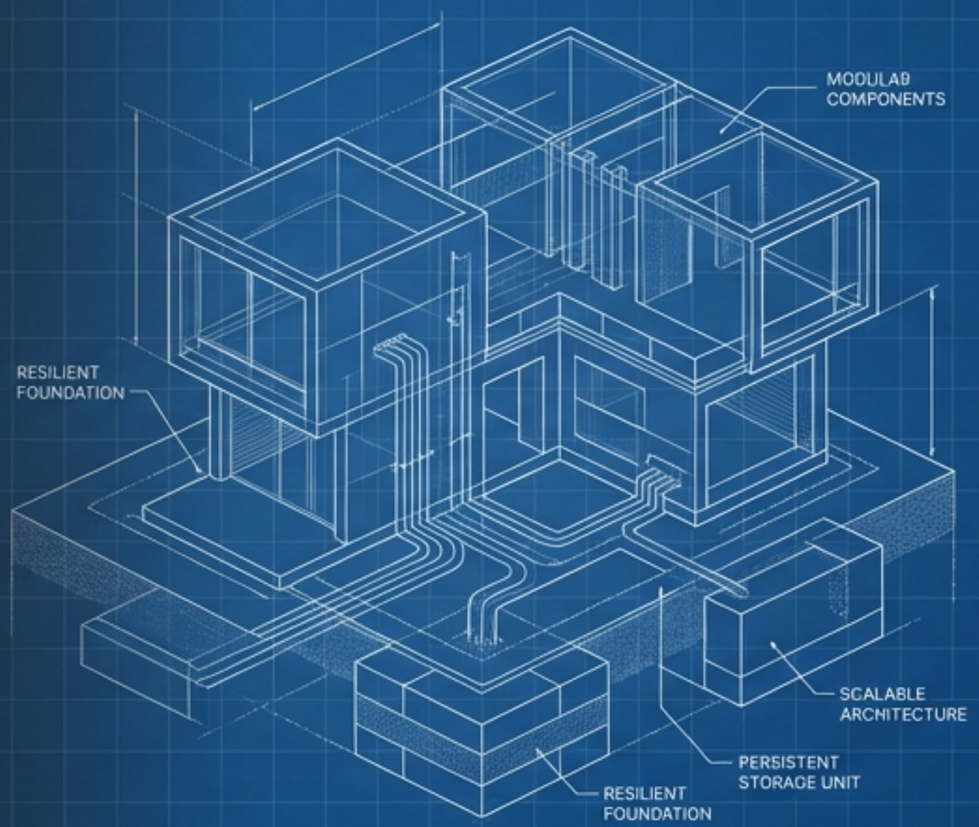
LEVEL UP YOUR PYTHON: FROM SCRIPTS TO SCALABLE APPLICATIONS

Mastering the Four Pillars of Professional Code: Reusability, Resilience, Persistence, and Organization.

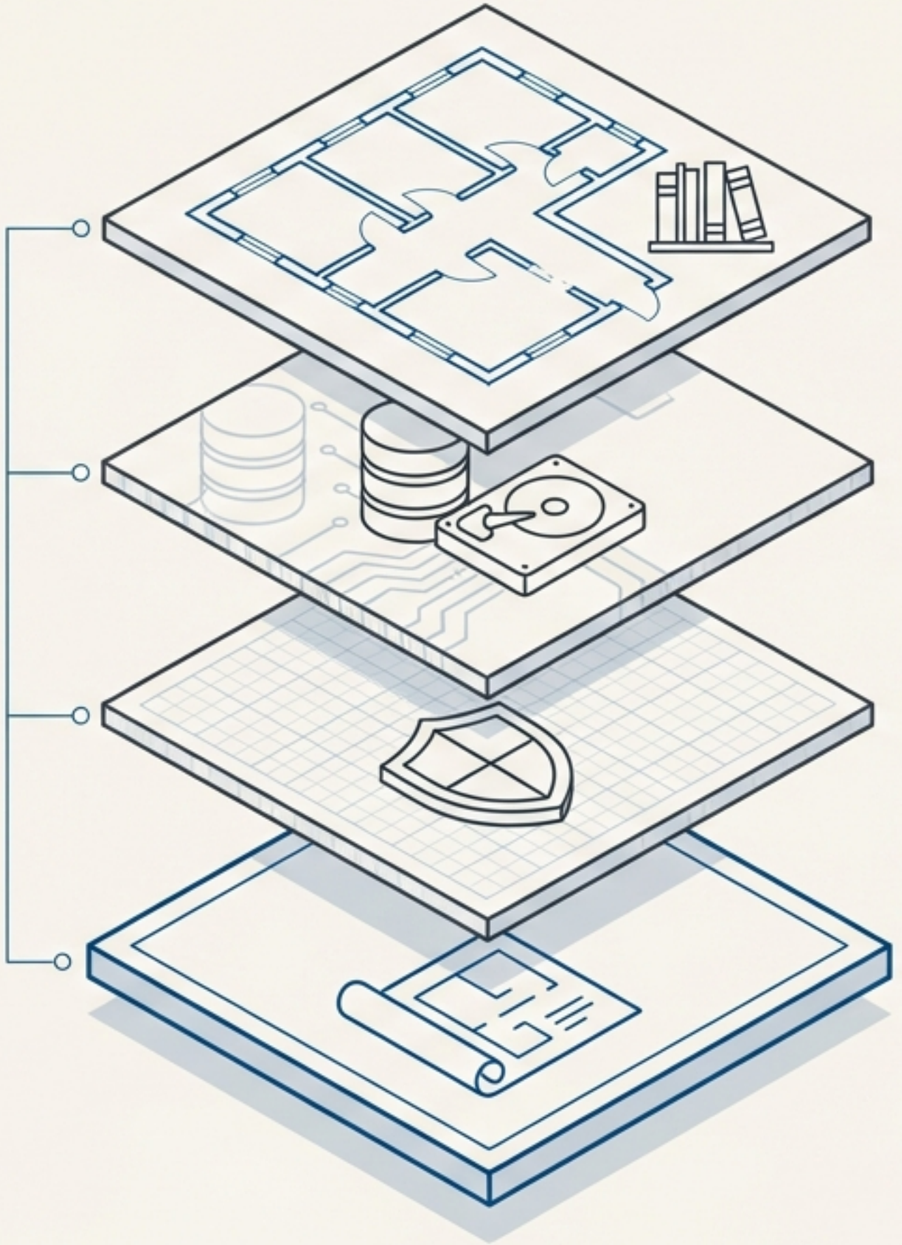
main.py

```python
class Data_processor:

    # Initialize data
    data_processor = "data_processor"
    connection_string = "connection_string"

    # Process the user input
    user input = proces.seuser.input()

    if user Invalid input:
        return "Error: Invalid input"
    else:
        return user


    # Save to database
    def save to database:
        data = peristessety("connection_datao_value")
        return data

    return data.seter()
```



THE ARCHITECT'S JOURNEY

We're moving beyond code that simply *works* to architecting code that *lasts*. This journey will equip you with the four pillars of robust software design.



THE LIBRARY

Designing a Scalable Structure with **Modules**

THE MEMORY

Adding Persistent Data with **Files**

THE FAILSAFE

Engineering Resilient Code with **Exceptions**

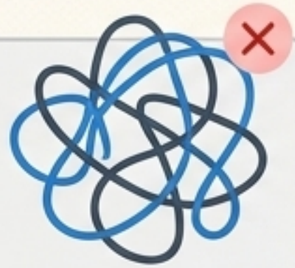
THE BLUEPRINT

Building Reusable Code with **Functions**

PILLAR 1: FUNCTIONS ARE YOUR REUSABLE BLUEPRINTS

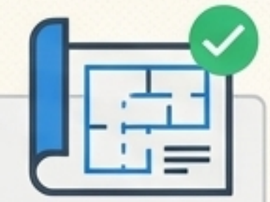
The first rule of smart architecture is to avoid repetition. Functions let you define a piece of logic once and use it everywhere.

THE PROBLEM: REPETITIVE CODE



```
# Calculating for 5 and 3
result1 = 5 + 3
print(f"5 + 3 = {result1}")

# Calculating for 10 and 20
result2 = 10 + 20
print(f"10 + 20 = {result2}")
```

THE SOLUTION: A REUSABLE FUNCTION



The keyword `def` starts the blueprint.

Parameters are the inputs.

```
def add_numbers(a, b):
    result = a + b
    print(f"{a} + {b} = {result}")

add_numbers(5, 3)    # Output: 5 + 3 = 8
add_numbers(10, 20)  # Output: 10 + 20 = 30
```

The body contains the logic.

Call the function to reuse the logic.

Function names should be lowercase with underscores (snake_case) and be descriptive.

A GOOD BLUEPRINT IS FLEXIBLE AND PRODUCTIVE

Parameters are the inputs that customize your function's behavior. The `return` statement is the finished product it delivers.

PARAMETERS: THE INPUTS

Default Parameters: Make your blueprints more versatile. Parameters with defaults must come after those without.


```
def greet(name, greeting="Hello"):
    print(f"{greeting}, {name}!")

greet("Alice")          # Output: Hello, Alice!
greet("Bob", "Hi")      # Output: Hi, Bob!
```

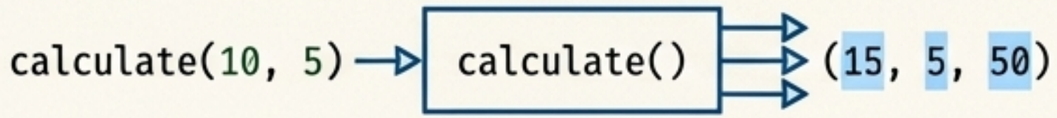
`return`: THE OUTPUT

Returning a Single Value

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

result = add(5, 3) # result is now 8
```

Returning Multiple Values

```
def calculate(a, b):
    return a + b, a - b, a * b

s, d, p = calculate(10, 5)
# s=15, d=5, p=50
```

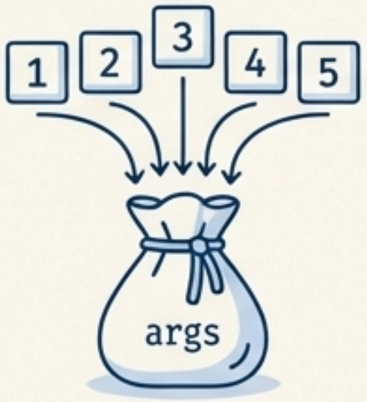


ADVANCED BLUEPRINTS FOR ULTIMATE FLEXIBILITY

Master the tools for handling a variable number of inputs and creating quick, on-the-fly functions.

'*args': The Collector's Bag

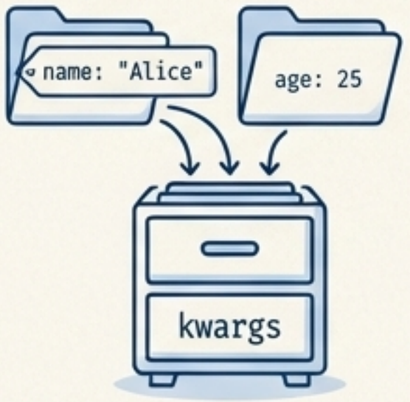
Collects multiple arguments into a tuple. Use it when you don't know how many inputs you'll get.




```python
def sum_all(*args):
    total = 0
    for num in args:
        total += num
    return total
print(sum_all(1, 2, 3, 4, 5)) # Output: 15
```

'**kwargs': The Labeled File Cabinet

Collects keyword arguments into a dictionary. Perfect for handling named options.




```python
def print_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

print_info(name="Alice", age=25)
# Output: name: Alice, age: 25
```

Lambda: The Disposable Tool

Small, single-line, anonymous functions. Ideal for quick, inline tasks like sorting.

```python
students = [{"name": "Bob", "grade": 92}, {"name": "Alice", "grade": 85}]
# Sort students by their grade using a lambda
sorted_students = sorted(students, key=lambda s: s["grade"])
```

PILLAR 2: BUILDING FOR RESILIENCE BY PLANNING FOR FAILURE

Great architects don't just plan for success; they anticipate what can go wrong. Exceptions are errors that occur during execution. Handling them prevents your application from crashing.

Without `try/except`

```
# User enters '0'
number = int(input("Enter a number: "))
number = int(input("Enter a number: "))
result = 100 / number

# CRASH! ZeroDivisionError
```

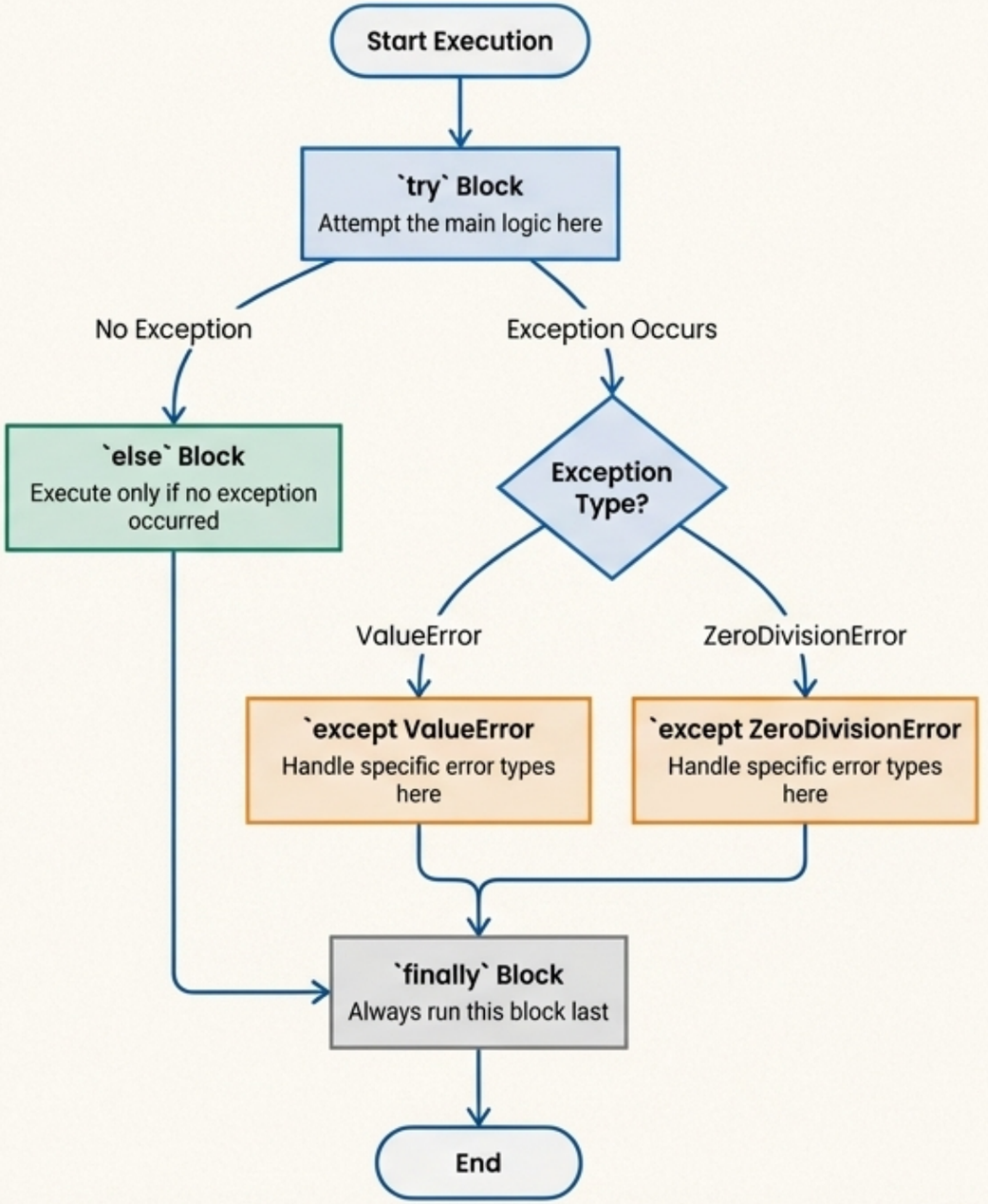
With `try/except`

```
try:
    # User enters '0'
    number = int(input("Enter a number: "))
    result = 100 / number

except ZeroDivisionError:
    print("Cannot divide by zero!")

# Program continues gracefully.
```

THE COMPLETE TOOLKIT FOR HANDLING ANY SITUATION

Use the full `try` block structure to manage success, failure, and cleanup with precision.

```
try:
    # The "Optimistic Path"
    number = int(input("Enter a number: "))
    result = 100 / number
except ValueError:
    # Recovery Plan A
    print("Invalid input!")
except ZeroDivisionError:
    # Recovery Plan B
    print("Cannot divide by zero!")
else:
    # The "Success Path"
    print(f"Result: {result}")
finally:
    # The "Always-On Cleanup Crew"
    print("Operation completed.")
```



KNOWING COMMON FAILURES AND CREATING YOUR OWN ALARMS

Common Built-in Exceptions

A quick-reference table of errors you'll frequently encounter.

Exception Name	Description
ValueError	An operation receives an argument with the right type but an inappropriate value.
TypeError	An operation is performed on an object of an inappropriate type.
FileNotFoundError	A file or directory is requested but doesn't exist.
KeyError	A dictionary key is not found.
IndexError	A sequence subscript is out of range.

Raising Your Own Alarms

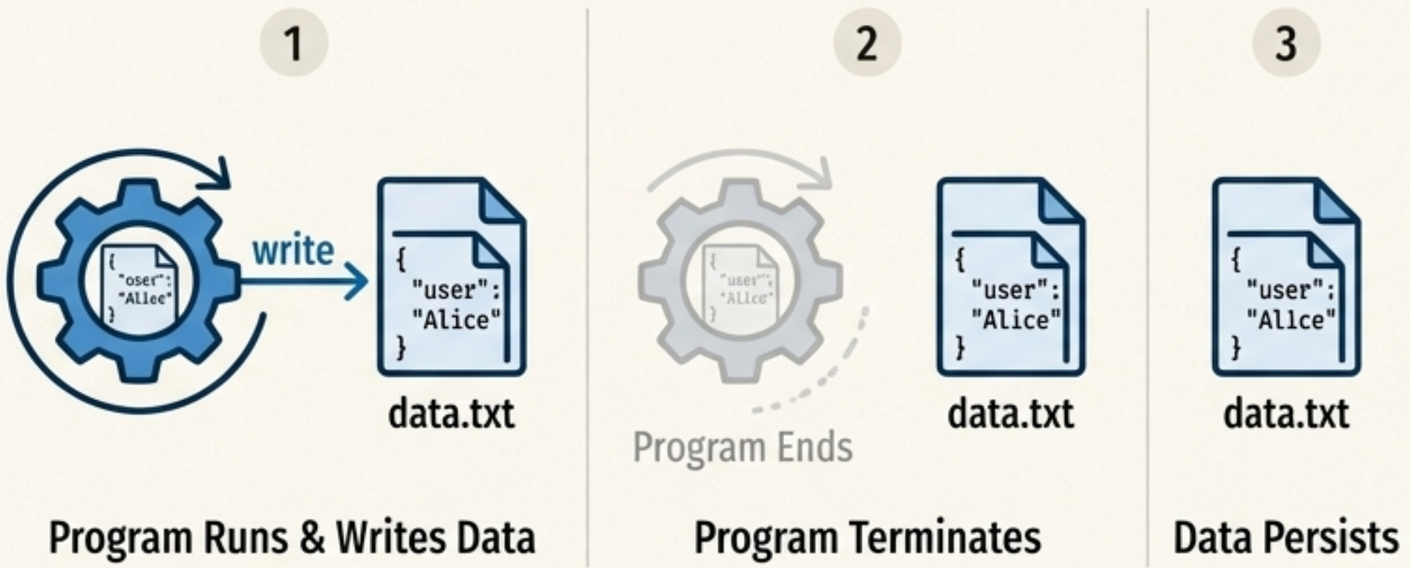
Use the `raise` keyword to signal that an error has occurred according to your program's logic. This is how you enforce your own rules.

```
def validate_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative")
    if age > 150:
        raise ValueError("Age seems unrealistic")
    return True

try:
    validate_age(-5)
except ValueError as e:
    print(f"Validation error: {e}")
```

Enforcing an architectural rule: age must be a positive number.

PILLAR 3: GIVING YOUR APPLICATION A MEMORY WITH FILES

An application that can't remember anything is a temporary tool. File I/O (Input/Output) allows your program to save data and state, making it persistent.



The `data.txt` file and its content remain available for future execution, even after the program has stopped.

The Gold Standard: The `with` statement

GOLD STANDARD

The recommended way to work with files is the `with open(...) as f:` syntax. It's a key resilience feature: it *automatically* handles closing the file for you, even if errors occur.

```
# The safe, modern, and recommended approach
with open("data.txt", "r") as file:
    content = file.read()
    # The file is automatically closed when
    this block is exited.

# Common file modes:
# "r" - Read (default)
# "w" - Write (overwrites existing file)
# "a" - Append (adds to the end of the file)
```

YOUR PRACTICAL TOOLKIT FOR FILE SYSTEM INTERACTION

Reading From Files

- `file.read()`
  - Reads the entire file content into a single string.
- `for line in file:`
  - The most common way to iterate through a file line-by-line.
- `file.readlines()`
  - Reads all lines into a list of strings.

Writing To Files

- `file.write('some string\n')`
  - Writes a string to the file. Remember to add newline characters (`\n`).
- `file.writelines(['line1\n', 'line2\n'])`
  - Writes a list of strings to the file.

Navigating with the `os` Module

- **Check Existence**

```
os.path.exists("data.txt")
```

  - Before reading a file, check if it's there to prevent a `FileNotFoundError`.

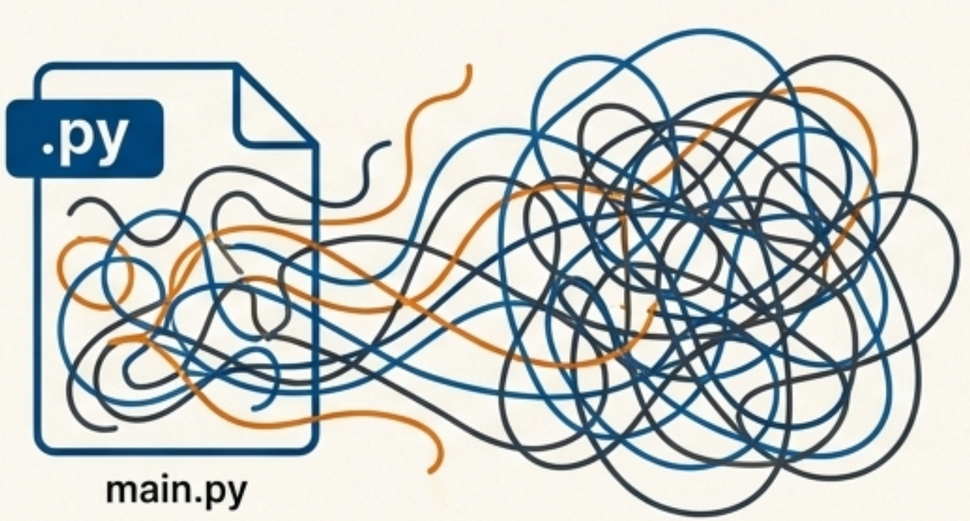
- **Build Paths**

```
os.path.join("folder", "file.txt")
```

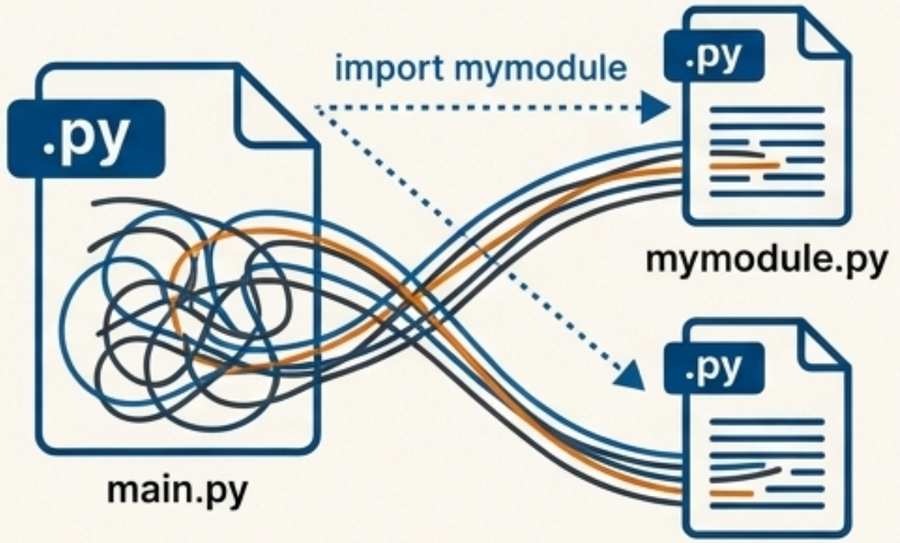
  - The professional, platform-independent way to create file paths. It handles the correct separators (`/` or `\`) for you.

PILLAR 4: ORGANIZING FOR SCALE WITH MODULES

A single, massive blueprint becomes unmanageable. As your project grows, you must organize your code into a library of specialized blueprints called modules. A module is simply a Python file (`.py`) containing functions and variables.



Before



After

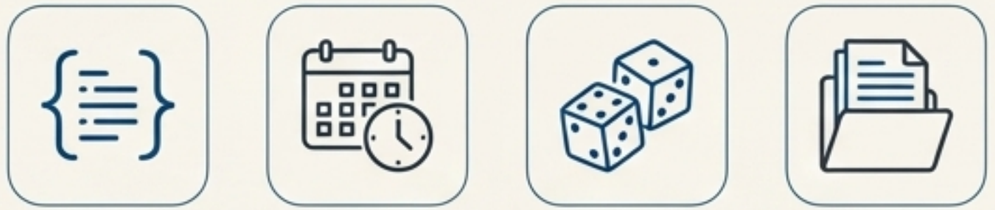
```python
# mymodule.py
def greet(name):
    return f"Hello, {name}!"

PI = 3.14159
```

```python
# main.py
import mymodule

print(mymodule.greet("Alice"))
print(mymodule.PI)
```

USING PRE-BUILT LIBRARIES AND MAKING YOUR MODULES REUSABLE

The Standard Library: Don’t Reinvent the Wheel

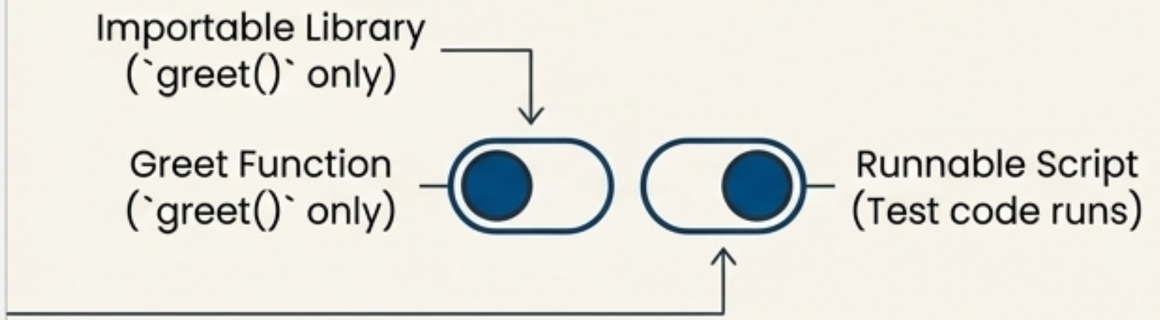
You don’t have to build everything from scratch. Python includes a massive "batteries-included" standard library of modules for common tasks.



The Professional’s Switch: `if __name__ == '__main__'`

This special block of code runs *only* when the file is executed directly, not when it's imported as a module. It's the standard way to include test code or a script's main logic within a module file.

```python
# in mymodule.py
def greet(name):
    return f"Hello, {name}!"

# This code is for testing and only runs when you
# execute 'python mymodule.py' directly.
if __name__ == "__main__":
    print("Testing the greet function...")
    print(greet("World"))
```



THE CAPSTONE: ASSEMBLING THE MASTERPIECE

Let's see how all four pillars come together to build a single, robust application: **a command-line Task Manager.**

- ✓ **Add Tasks**: Create new tasks with a description and priority.
- ✓ **Complete & Delete Tasks**: Manage the task lifecycle.
- ✓ **List Tasks**: View all pending or completed tasks.
- ✓ **Persistence**: Tasks are automatically loaded from and saved to a `tasks.json` file. The application remembers everything.
- ✓ **Robustness**: Handles bad user input and file errors gracefully.

```
=== Task Manager ===

1. Add task
2. List tasks
3. Complete task
4. Delete task
5. Exit

[ ] 1. Design presentation slides (HIGH)
[ ] 2. Write visual prompts (MEDIUM)
[✓] 3. Architect the narrative (MEDIUM)

Enter choice: _
```

DECONSTRUCTING THE TASK MANAGER'S ARCHITECTURE

PILLAR 1: REUSABILITY (FUNCTIONS)

Each core feature is encapsulated in its own reusable method—our blueprints in action.

PILLAR 4: ORGANIZATION (MODULES)

We're not building from scratch. We leverage pre-built libraries for JSON handling and file system interaction.

```
import json, os, from datetime import datetime

class TaskManager:
    # ...
    def load_tasks(self):
        try:
            with open(self.filename, 'r') as file:
                self.tasks = json.load(file)
        except json.JSONDecodeError:
            print("Error: Task file is corrupted.")

    def save_tasks(self):
        with open(self.filename, 'w') as file:
            json.dump(self.tasks, file, indent=2)

    def add_task(self, description):
        # ... logic ...

    def list_tasks(self):
        # ... logic ...
```

PILLAR 2: RESILIENCE (EXCEPTIONS)

This is our failsafe. The app doesn't crash if the save file is corrupted; it recovers.

PILLAR 3: PERSISTENCE (FILES)

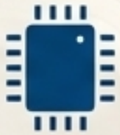
Here is the application's memory. We use the `with` statement and the `json` module to save our task list to disk.

YOUR ARCHITECTURAL JOURNEY CONTINUES

You now have the foundational tools of a Python software architect. By building with these four pillars in mind, you will create code that is not only functional but also clean, robust, and built to last.

Reusability: Build with Functions

Resilience: Plan with Exceptions



Persistence: Remember with Files

Organization: Scale with Modules

Happy Architecting!