Lecture 26: Modules & Packages

Comp 102

Forman Christian University

Your code has

outgrown its room

Remember the weather tracker? 50 lines, one file — it **works**.

But what happens when it **grows**?

Imagine adding. . .

- Multiple cities

Imagine adding. . .

- Multiple cities
- A favorites list

Imagine adding. . .

- Multiple cities
- A favorites list
- A temperature converter

Imagine adding. . .

- Multiple cities
- A favorites list
- A temperature converter
- A history viewer

Imagine adding. . .

- Multiple cities
- A favorites list
- A temperature converter
- A history viewer
- User settings

Imagine adding. . .

- Multiple cities
- A favorites list
- A temperature converter
- A history viewer
- User settings

Now it's **200+ lines** in one file. **Chaos.**

When you were 5, **one room** for everything worked.

Now you need a kitchen, a bedroom, a study. . .

When you were 5, **one room** for everything worked.

Now you need a kitchen, a bedroom, a study. . .

Your code needs the **same thing**:

separate files for separate jobs.

Today's Goal

Modules give your code **rooms**

The **standard library** furnishes them for free

pip lets you order anything the world has ever built

Part 1: What Is a Module?

A Module Is Just a `.py` File

- You've been writing modules all along!

A Module Is Just a `.py` File

- You've been writing modules all along!
- `import math` — `math` is just a `.py` file someone wrote

A Module Is Just a `.py` File

- You've been writing modules all along!
- `import math` — `math` is just a `.py` file someone wrote
- Any `.py` file can be imported by another `.py` file

A Module Is Just a `.py` File

- You've been writing modules all along!
- `import math` — `math` is just a `.py` file someone wrote
- Any `.py` file can be imported by another `.py` file

That's it. A module = a file.

Your First Custom Module

Step 1: Create helpers.py

```python
# helpers.py
def celsius_to_fahrenheit(c):
    return c * 9/5 + 32

def fahrenheit_to_celsius(f):
    return (f - 32) * 5/9
```

Your First Custom Module

Step 2: Create `main.py` in the **same folder**

```python
import helpers

result = helpers.celsius_to_fahrenheit(30)
print(result)  # 86.0
```

Your First Custom Module

Step 2: Create `main.py` in the **same folder**

```python
import helpers

result = helpers.celsius_to_fahrenheit(30)
print(result)  # 86.0
```

Loads the entire `helpers.py` file

Your First Custom Module

Step 2: Create main.py in the **same folder**

```python
import helpers

result = helpers.celsius_to_fahrenheit(30)
print(result)  # 86.0
```

The **dot** means "go into that module and find..."

What Happens at `import helpers`?

What Happens at `import helpers`?

1. Finds `helpers.py` in the same folder

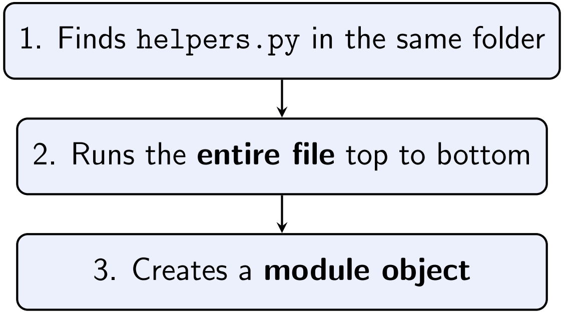
What Happens at `import helpers`?

1. Finds `helpers.py` in the same folder

↓

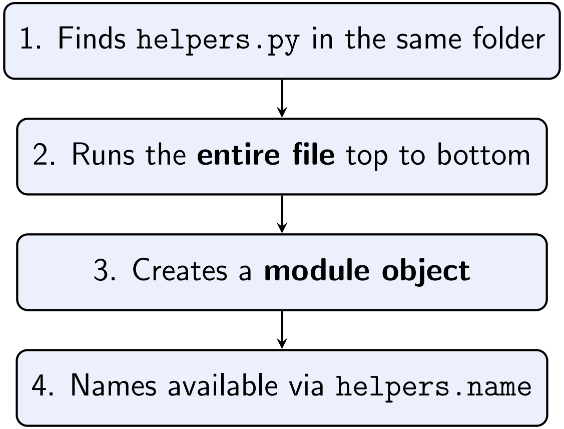
2. Runs the **entire file** top to bottom

What Happens at `import helpers`?



1. Finds `helpers.py` in the same folder
2. Runs the **entire file** top to bottom
3. Creates a **module object**

What Happens at `import helpers`?


1. Finds `helpers.py` in the same folder
2. Runs the **entire file** top to bottom
3. Creates a **module object**
4. Names available via `helpers.name`

You Try!

1. Create `greetings.py`:

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

2. In Thonny's Shell, type:

```
>>> import greetings
>>> print(greetings.hello("Ali"))
```

Part 2: Import Styles

Three Ways to Invite a Friend

- `import math` — **formal**
  “I’d like to speak with the math department”
  → `math.sqrt(16)`

Three Ways to Invite a Friend

- `import math` — **formal**
  "I'd like to speak with the math department"
  → `math.sqrt(16)`
- `from math import sqrt` — **first-name basis**
  "Hey sqrt!"
  → `sqrt(16)`

Three Ways to Invite a Friend

- `import math` — **formal**
  "I'd like to speak with the math department"
  → `math.sqrt(16)`

- `from math import sqrt` — **first-name basis**
  "Hey sqrt!"
  → `sqrt(16)`

- `import math as m` — **nickname**
  → `m.sqrt(16)`

Side by Side: Same Program, Three Ways

```python
# Style 1: import module
import math
print(math.sqrt(16))
print(math.pi)
```

Side by Side: Same Program, Three Ways

```python
# Style 1: import module
import math
print(math.sqrt(16))
print(math.pi)
```

```python
# Style 2: from module import names
from math import sqrt, pi
print(sqrt(16))
print(pi)
```

Side by Side: Same Program, Three Ways

```python
# Style 1: import module
import math
print(math.sqrt(16))
print(math.pi)
```

```python
# Style 2: from module import names
from math import sqrt, pi
print(sqrt(16))
print(pi)
```

```python
# Style 3: import with alias
import math as m
print(m.sqrt(16))
print(m.pi)
```

Danger: `from module import *`

Imports **everything** — sounds convenient, but. . .

```python
from math import *
from cmath import *

print(sqrt(16))  # Which sqrt?!
```

Danger: `from module import *`

Imports **everything** — sounds convenient, but. . .

```
from math import *
from cmath import *

print(sqrt(16))  # Which sqrt?!
```

cmath.sqrt silently replaced math.sqrt!

Danger: `from module import *`

Imports **everything** — sounds convenient, but. . .

```python
from math import *
from cmath import *

print(sqrt(16))  # Which sqrt?!
```

Avoid `import *` — you lose track of where names come from.

Rule of Thumb

Situation	Use
Standard library	`import math`
Use one thing often	`from math import sqrt`
Long module name	`import datetime as dt`
Never	`from module import *`

You Try! — Predict the Output

```python
# Snippet A
import math
print(math.floor(3.7))
```

```python
# Snippet C
import math as m
print(m.pow(2, 10))
```

```python
# Snippet B
from random import randint
print(type(randint(1, 10)))
```

You Try! — Predict the Output

```python
# Snippet A
import math
print(math.floor(3.7))
```

→ **3**

```python
# Snippet B
from random import randint
print(type(randint(1, 10)))
```

```python
# Snippet C
import math as m
print(m.pow(2, 10))
```

You Try! — Predict the Output

```python
# Snippet A
import math
print(math.floor(3.7))
```

→ 3

```python
# Snippet B
from random import randint
print(type(randint(1, 10)))
```

→ <class 'int'>

```python
# Snippet C
import math as m
print(m.pow(2, 10))
```

You Try! — Predict the Output

```
# Snippet A
import math
print(math.floor(3.7))
```

→ **3**

```
# Snippet B
from random import randint
print(type(randint(1, 10)))
```

→ **<class 'int'>**

```
# Snippet C
import math as m
print(m.pow(2, 10))
```

→ **1024.0**

Part 3: Refactoring the Weather Tracker

Remember the weather tracker from Lecture 25?

50 lines, all in **one file**.

Remember the weather tracker from Lecture 25?

50 lines, all in **one file**.

Let's give it **rooms**.

What is Refactoring?

Changing the **structure** of code

without changing what it **does**.

What is Refactoring?

Changing the **structure** of code

without changing what it **does**.

Like reorganising a messy room:

same furniture, same items —
just everything given **its own place**.

What is Refactoring?

Changing the **structure** of code

without changing what it **does**.

Like reorganising a messy room:

same furniture, same items —
just everything given **its own place**.

No new features.   Just better structure.

The Plan: One Job Per File

`weather_tracker.py` currently does **3 jobs**:

The Plan: One Job Per File

`weather_tracker.py` currently does **3 jobs**:

- **Fetch** data from the API

The Plan: One Job Per File

`weather_tracker.py` currently does **3 jobs**:

- **Fetch** data from the API
- **Save/load** JSON files

The Plan: One Job Per File

`weather_tracker.py` currently does **3 jobs**:

- **Fetch** data from the API
- **Save/load** JSON files
- **Run** the main logic

The Plan: One Job Per File

`weather_tracker.py` currently does **3 jobs**:

- **Fetch** data from the API
- **Save/load** JSON files
- **Run** the main logic

Rule: one file = one job.

```
weather_project/
├── weather.py  — API calls
├── storage.py  — file I/O
└── main.py  — the glue
```

File 1: weather.py

The API-fetching logic — its **only job**:

```python
# weather.py
import requests

def get_weather(lat, lon):
    url = "https://api.open-meteo.com/v1/forecast"
    params = {
        "latitude": lat,
        "longitude": lon,
        "current": "temperature_2m,"
                   "wind_speed_10m,"
                   "relative_humidity_2m"
    }
    response = requests.get(url, params=params)
    return response.json()["current"]
```

File 2: storage.py

The JSON read/write logic — its **only job**:

```python
# storage.py
import json

def load_history(filename):
    try:
        with open(filename, "r") as f:
            return json.load(f)
    except FileNotFoundError:
        return []

def save_history(filename, history):
    with open(filename, "w") as f:
        json.dump(history, f, indent=2)
```

File 3: main.py

Clean, **15 lines**, reads like English:

```python
from weather import get_weather
from storage import load_history, save_history

data = get_weather(31.55, 74.35)
print(f"Temperature: {data['temperature_2m']}°C")

history = load_history("weather_history.json")
history.append(data)
save_history("weather_history.json", history)
```

File 3: main.py

Clean, **15 lines**, reads like English:

```python
from weather import get_weather
from storage import load_history, save_history

data = get_weather(31.55, 74.35)
print(f"Temperature: {data['temperature_2m']}°C")

history = load_history("weather_history.json")
history.append(data)
save_history("weather_history.json", history)
```

Each import pulls from a **separate file**

Running the Program

From inside the `weather_project/` folder:

```
$ python main.py
```

Running the Program

From inside the `weather_project/` folder:

```
$ python main.py
```

Python resolves each `from ... import` by finding the file **in the same folder**:

- `weather.py` → provides `get_weather`
- `storage.py` → provides `load_history`, `save_history`

Running the Program

From inside the `weather_project/` folder:

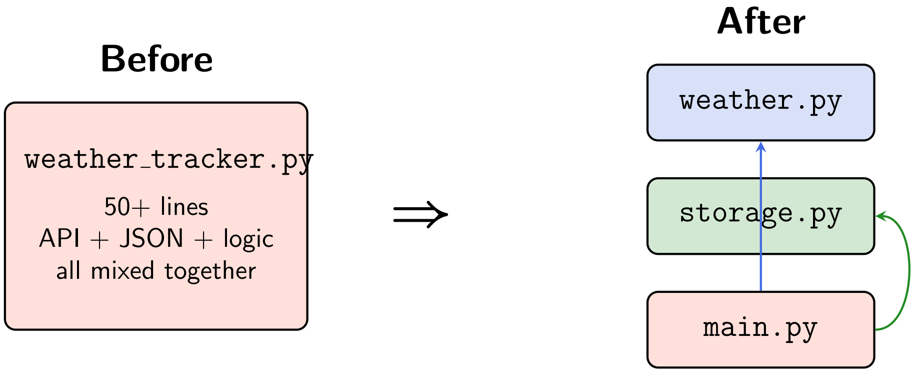
```
$ python main.py
```

Python resolves each `from ... import` by finding the file **in the same folder**:

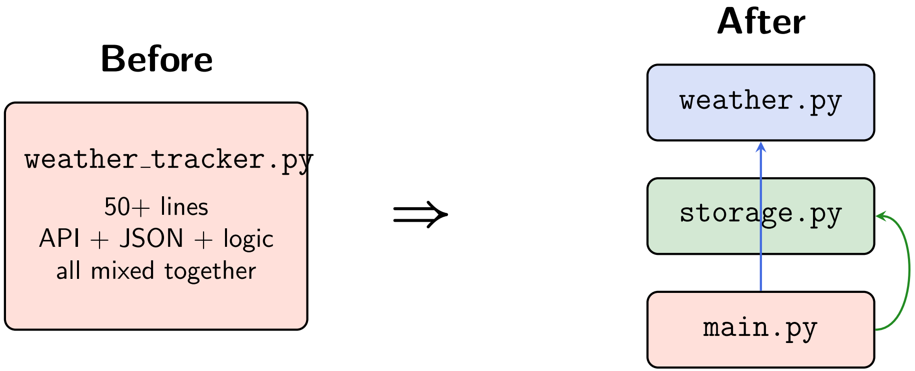
- `weather.py` → provides `get_weather`
- `storage.py` → provides `load_history`, `save_history`

All three files must live in the same folder.

Before vs After



Before vs After



Each file has **one job**. `main.py` reads like a **story**.

The `__name__` Guard

Problem: you add test code to `weather.py`:

```python
# weather.py
import requests

def get_weather(lat, lon):
    ...

# Quick test
print(get_weather(31.55, 74.35))
```

The `__name__` Guard

Problem: you add test code to `weather.py`:

```
# weather.py
import requests

def get_weather(lat, lon):
    ...

# Quick test
print(get_weather(31.55, 74.35))
```

This runs **every time** someone does `import weather`!

The Fix: `if __name__ == "__main__"`

```python
# weather.py
import requests

def get_weather(lat, lon):
    ...

if __name__ == "__main__":
    # Only runs when you execute this file directly
    print(get_weather(31.55, 74.35))
```

The Fix: `if __name__ == "__main__"`

```python
# weather.py
import requests

def get_weather(lat, lon):
    ...

if __name__ == "__main__":
    # Only runs when you execute this file directly
    print(get_weather(31.55, 74.35))
```

Runs only when you click "Run" on this file

How `__name__` Works

Run Directly

Run `weather.py` directly

`__name__ = "__main__"`

Imported

`import weather` from main.py

`__name__ = "weather"`

How `__name__` Works

Run Directly

Run `weather.py` directly

`__name__ = "__main__"`

Test code **runs**

Imported

`import weather` from main.py

`__name__ = "weather"`

Test code **skipped**

Part 4: Python's Standard Library

“Batteries Included”

Python ships with **200+ modules** built in.

It’s like buying a house that comes **fully furnished**.

Why build a table when there’s one in the kitchen?

math — Mind-Blowing Numbers

```python
import math

# Ways to shuffle a deck of cards
print(math.factorial(52))
```

`math` — Mind-Blowing Numbers

```python
import math

# Ways to shuffle a deck of cards
print(math.factorial(52))
```

→ `80658175170943878571660636856403766975...`

That's an **68-digit number**. More than atoms in the Milky Way.

random — Let Python Decide

```python
import random

# Pick a random city
cities = ["Lahore", "Karachi", "Islamabad"]
print(random.choice(cities))

# Shuffle a list
names = ["Ali", "Fatima", "Hassan", "Zara"]
random.shuffle(names)
print(names)
```

datetime — Work with Dates

```python
from datetime import date

today = date.today()
summer = date(2026, 6, 15)

days_left = (summer - today).days
print(f"{days_left} days until summer!")
```

`datetime` — Work with Dates

```python
from datetime import date

today = date.today()
summer = date(2026, 6, 15)

days_left = (summer - today).days
print(f"{days_left} days until summer!")
```

You can **subtract** dates to get a time difference

os — Talk to Your Computer

```python
import os

# List all files in the current folder
print(os.listdir("."))

# Check if a file exists before reading
if os.path.exists("diary.txt"):
    print("Found your diary!")
else:
    print("No diary yet.")
```

string + random = Password Generator

```python
import random
import string

chars = string.ascii_letters + string.digits
password = "".join(
    random.choices(chars, k=12)
)
print(password)  # e.g. "kR7mXp2wLq9N"
```

string + random = **Password Generator**

```python
import random
import string

chars = string.ascii_letters + string.digits
password = "".join(
    random.choices(chars, k=12)
)
print(password)  # e.g. "kR7mXp2wLq9N"
```

Pick 12 random characters from letters + digits

You Try!

In Thonny's Shell, write a **3-line password generator**:

```
>>> import random, string
>>> chars = string.ascii_letters + string.digits
>>> "".join(random.choices(chars, k=16))
```

You Try!

In Thonny's Shell, write a **3-line password generator**:

```
>>> import random, string
>>> chars = string.ascii_letters + string.digits
>>> "".join(random.choices(chars, k=16))
```

Try changing `k=16` to get different lengths!

How to Explore New Modules

- `dir(math)` — list everything in a module

How to Explore New Modules

- `dir(math)` — list everything in a module
- `help(math.sqrt)` — read the documentation

How to Explore New Modules

- `dir(math)` — list everything in a module
- `help(math.sqrt)` — read the documentation
- Google: "python *module name* documentation"

How to Explore New Modules

- `dir(math)` — list everything in a module
- `help(math.sqrt)` — read the documentation
- Google: “python *module name* documentation”

Some modules to explore on your own: `calendar`, `textwrap`, `collections`, `turtle`

Part 5: Third-Party Packages & pip

The App Store for Python

The standard library has 200+ modules.

But the Python **community** has built **500,000+** packages.

Available at **PyPI.org** — the Python Package Index.

Installing Packages with `pip`

In the terminal:

```
pip install requests
pip install pyjokes
pip install emoji
```

Installing Packages with `pip`

In the terminal:

```
pip install requests
pip install pyjokes
pip install emoji
```

In **Thonny**: Tools → Manage Packages → search and install

You already installed `requests` for Lecture 25!

Demo: Fun with Packages

```python
import pyjokes

print(pyjokes.get_joke())
```

Demo: Fun with Packages

```python
import pyjokes

print(pyjokes.get_joke())
```

→ “A programmer puts two glasses on his bedside table before going to sleep. A full one, in case he gets thirsty, and an empty one, in case he doesn’t.”

Demo: Fun with Packages

```python
import pyjokes

print(pyjokes.get_joke())
```

→ "A programmer puts two glasses on his bedside table before going to sleep. A full one, in case he gets thirsty, and an empty one, in case he doesn't."

Popular packages worth knowing:

`requests`	HTTP requests (APIs)
`flask`	Build web apps
`pygame`	Build games
`pillow`	Image processing
`pandas`	Data analysis

Module vs Package vs Library

Term	What it is	Example
Module	One `.py` file	`math`, `random`
Package	A folder of modules	`requests`, `flask`
Library	Informal term for either	"the requests library"

Part 6: Packages — Folders of Modules

When Modules Need Rooms Too

What if you have **many related modules**?

Group them into a **folder** — that's a **package**.

When Modules Need Rooms Too

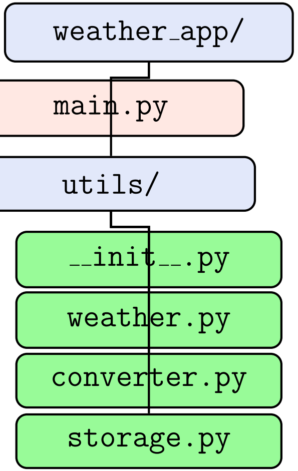
What if you have **many related modules**?

Group them into a **folder** — that's a **package**.

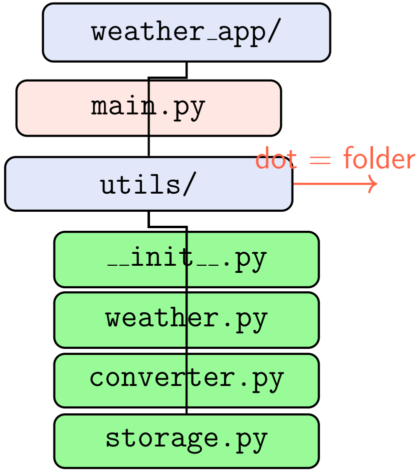
The only rule: the folder must contain an `__init__.py` file.

It can be **empty** — it just tells Python "this folder is a package."

Package Structure

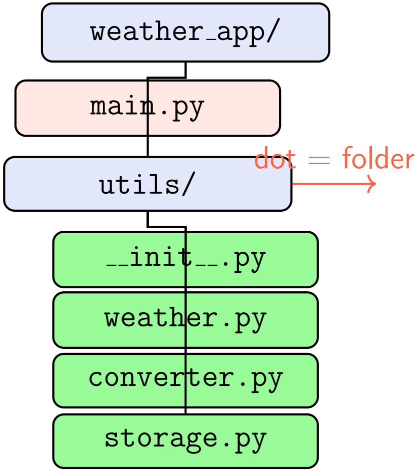


Package Structure



`from utils.weather import get_weather`

Package Structure

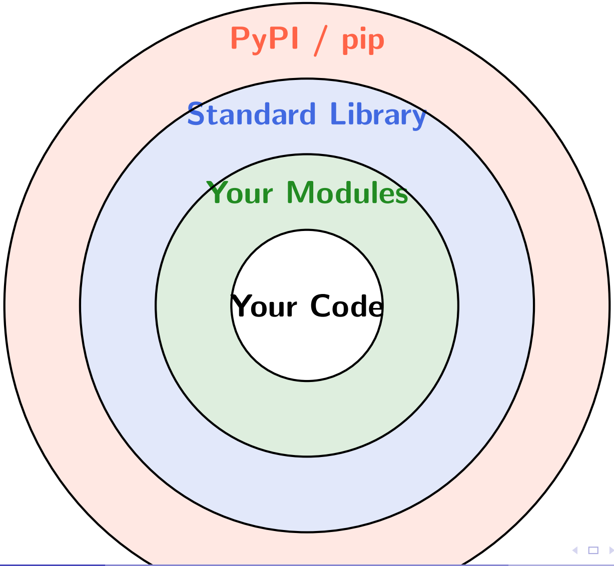


```
from utils.weather import get_weather
```

Dots in imports = navigating folders!

The Big Picture

Where Code Comes From



Import Styles — Cheat Sheet

Style	Usage	Access
`import math`	Import whole module	`math.sqrt(16)`
`from math import sqrt`	Import specific name	`sqrt(16)`
`from math import sqrt, pi`	Import multiple names	`sqrt(16), pi`
`import math as m`	Import with alias	`m.sqrt(16)`
`from math import *`	Import everything	Avoid!

The Story in One Sentence

Modules give your code **rooms**,

the **standard library** furnishes them for free,

and **pip** lets you order anything the world has ever built.

Your weather tracker went from a **messy one-room apartment** to a **well-organized house**.

Summary

Key Concepts

Concept	Key Point
Module	A `.py` file you can import
Package	A folder with `__init__.py`
`import module`	Access via `module.name`
`from m import x`	Access `x` directly
`__name__` guard	Prevents code running on import
Standard library	200+ built-in modules
`pip install`	Install from PyPI (500k+ packages)

Practice Ideas

1. Take any program from earlier in the course and **split it into 2+ modules**
2. Explore 3 standard library modules: try `calendar`, `textwrap`, `collections`
3. Install a fun package from PyPI: `pyjokes`, `emoji`, `art`
4. Build a "utility belt" module with your favorite helper functions from the course

Questions?